Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Chapter 8

Arrays and Their Uses

So far, most of the variables you have used have been scalar variables. That is, each of these variables holds one value, a number, or a string. There may be times when you will encounter a situation in which your program needs to handle a large group of homogeneous data, and it needs to access these data back and forth. In this case, using array variables will be more elegant and/or efficient. An array is a collection of more than one element of data, and is collectively recognized by the same variable name. Each element in the array is indexed with the variable's subscript(s). You can refer to these elements by their indexes.

Although the concept of arrays appears simple, their applications can be fascinating. Their uses can make many complex problems much easier to solve. One interesting problem is sorting, which involves arranging data in order. The goal is simple and well defined, but the algorithms to solving the problem are diverse and intriguing.

An array can have one or more subscripts. The number of subscripts of an array is also referred to as the rank or the number of dimensions; for example, an array with two subscripts is recognized as a two-dimensional array. Although most of this chapter is devoted to discussing uses of one-dimensional arrays, the last section deals with two-dimensional arrays.

After completing this chapter, you should be able to:

- Create and use one-dimensional arrays,
- Illustrate some practical applications of one-dimensional arrays,
- Use the Collection object as a means of handling a list of controls, and
- Create and use two-dimensional arrays.

8.1 One-Dimensional Arrays

This introductory section deals with the technical aspects of handling one-dimensional arrays. Topics include creating one-dimensional arrays, dynamic arrays, arrays as parameters between procedures, and determining array boundaries. This discussion provides the background information required for the remainder of this chapter.

Creating One-Dimensional Arrays

To declare a one-dimensional array, use the Dim statement with the following syntax:

```
Dim Name([upper bound]) As Type
```

where *Name* = any unique identifier that is legitimate as a variable name,
upper bound = an integer representing the upper value of the subscript, and
Type can be any data type, such as Integer, Long, and so on.

Scope of Array Variables

Declaration of array variables can be placed in the general declaration area or in a procedure and has exactly the same scope as scalar variables with the same scope declaration. For example, array variables

declared with a Dim statement in the general declaration area are class-level variables and are recognized by all procedures in the form. Array variables declared in a procedure are local (procedure-level) variables.

Declaration Examples

Here are some examples of valid array declarations:

```
Dim A(5) As Double
Dim Student(50) As String
Dim Balance() As Decimal
```

The first Dim statement declares A to be a variable containing six elements of the Double type with a subscript range of 0 to 5, as shown here:

```
A(0)  A(1)  A(2)  A(3)  A(4)  A(5)
```

The second Dim statement declares the Student string variable to contain 51 elements with a subscript range of 0 to 50, as shown next:

```
Student(0)  Student(2)  ...  Student(49)  Student(50)
```

The third statement declares a Decimal variable, Balance, to contain unknown number of elements. Usually, a variable declared this way is used to receive results from a function that generates varying number of elements. In other cases, the number of elements is not known until run time. The array is then resized when it is needed. Examples will be provided in the next subsection.

Referencing Elements in an Array

After the array is declared, you can refer to individual elements in the array by the use of an index (subscript). For example, Student(0) refers to the first element in the Student array; therefore, you can code the following:

```
    Student(0) = "Angela Allen"
```

This statement will assign the text string (name) "Angela Allen" to the first element of the Student array. You can then use the element just as you would any scalar variable. For example, to display the name, you can code the following:

```
    MsgBox("The first student in the array is " & Student(0))
```

Similarly, A(5) refers to the sixth element in A. Notice that A's smallest subscript value is 0.

Using Variables as the Index

The index in the pair of parentheses does not have to be a constant. It can also be a variable or any expression that results in a numeric value. If you have declared an Amount array variable (say, with an upper bound of 9999), you can code the following:

```
    Amount(N) = 100
```

Depending on the value of N, 100 will be assigned to the corresponding element. That is, if N is 2,000, Amount(2000) will be assigned with a value 100. (Note that N should be an Integer variable.) The ability to handle an expression as the index greatly enhances the flexibility and power of arrays, as you will see in the examples in the next section. Here are some simple uses of this capability:

Example 1. The following code lists all the names of students (assuming that the array has been properly populated with the names):

```
    ' List Student(0) Through Student(50)
    For I = 0 to 50
        Console.WriteLine(Student(I))
    Next I
```

Recall that in the For loop, the counter I will vary from the starting value (0) to the ending value (50) with the specified increment. Because the increment is not specified, its default is 1; therefore, the loop

will execute 51 times, with I taking on 0, 1, 2, 3, ... 49, 50. The following execution table shows the process:

Iteration	I	Element of Student(I) Referenced	Example Result
1	0	Student(0)	Angela Allen
2	1	Student(1)	Bob Bunker
.	.	.	.
.	.	.	.
51	50	Student(50)	Zeff Ziegler

Example 2. The following code increases each element in Scores() by 5 (again, assuming that the array has been properly declared and populated with values).

```
' Add 5 points to Scores(0) through Scores(50)
For I = 0 To 50
    Scores (I) += 5
Next I
```

Dynamic Arrays

When the size of an array cannot be determined until run time, you can use the ReDim statement to resize the array when the information on its size is available. But you must have the variable declared as an array first. The syntax for the ***ReDim statement*** is as follows:

```
ReDim Name1(UpperBoundExpression1) [,Name2(UpperBoundExpression2)] …
```

The ReDim statement can be placed only within procedures; that is, you are not allowed to use the ReDim statement in the general declaration section. The following are examples of valid uses of the ReDim statement.

```
ReDim A(10)
ReDim Employees(EmployeeCount)
ReDim Holder(3 * N)
ReDim Balance(2000), Amount(CustomerCount - 1)
```

If you use a variable as a part of an expression to compute the subscript of an array, the variable must have its proper value before the ReDim statement is executed. In the preceding examples, you assumed proper values have been assigned to the variables EmployeeCount, N, and CustomerCount in lines 2, 3, and 4.

Note again that in the preceding ReDim statements, you assumed that the variables, A, Employees, and Holder have been declared as one-dimensional array variables. The ReDim statement cannot change the data type or rank (number of dimensions) of the variable.

Scope of ReDimed Variables

If the ReDim statement can be used only within procedures, does it mean that all variables sized with the ReDim statement are procedure-level (local) variables? Not necessarily. It depends on where the variable is actually declared (by the Dim statement or other proper modifiers such as Friend or Public). If the variable is declared in a procedure, it is a local variable and can be ReDimed only in that procedure; however, if you declare the variable in the general declaration area, it is a class level variable as shown here:

```
Dim Employees() As String 'Note I don't have to declare the size although I may.
```

In a procedure, you can declare its actual size:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
```

```
  Handles MyBase.Load
    Dim N As Integer
    ' various statements, including one that sets the value of N
    ReDim Employees(N)
    ' other statements
End Sub
```

Declaring Dynamic Arrays with Initial Values

You can declare initial values for a dynamic array by enclosing the constants in a pair of braces. For example, the following code should cause A to contain three elements with the values 12, 24, and 36.

```
    Dim A() As Integer = {12, 24, 36}
```

Assigning Array Values

You can also assign a list of values to an array, using the braces to form the list. Assume you have declared B as an Integer array. You can then use a statement similar to the following to assign values to it:

```
    B = New Integer() {3, N, 5}
```

Notice how the expression on the right is constructed. The New keyword must be used. In addition, the keyword for the data type, such as Integer, must be followed by a pair of parentheses to indicate an array. The data elements in the pair of braces can be either constant or scalar variables of the same type.

Try This

Enter the following code in the form click event.

```
    Dim I As Integer
    Dim N As Integer
    Dim B() As Integer
    N = 5
    B = New Integer() {3, N, 5}
    For I = 0 To 2
        Console.WriteLine(B(I))
    Next I
```

Run the project, click the form, and examine the results. You now know how to construct a list values for an array. You also know that variables can be included in the list, just like constants.

Passing Arrays to Procedures and Returning an Array from a Function.

Sometimes, an entire array needs to be passed to another procedure for computation or data manipulation. To pass an array to a procedure, the called procedure must be written to expect an array as its parameter. The parameter in the header should be followed by a pair of parentheses. The following example illustrates the syntax:

```
    Function Sum(A() As Double) As Double
        '  statements
    End Function
```

To pass an array, such as Salary, in another procedure to the Sum function, you can code the following:

```
    Total = Sum(Salary)
```

Here you assume that the variable Salary has been properly declared as an array of the Double type. Note that Salary as an argument should not be followed by a pair of parentheses.

Can a function be made to return an array? Yes. In the header, add a pair of parentheses after the type specification. For example, the following header indicates that the Sequence function returns an integer array:

```
Function Sequence(N As Integer) As Integer()
```

Again, notice the pair of parentheses following "As Integer." Without the pair, the function is expected to return a scalar value. With the pair, the function is expected to return an array. The code to actually generate the results is left to you as exercise 8-37.

Determining the Boundary of an Array.

If you inspect the Sum function in the preceding example again, you may notice that any Double array can be passed to it. The array parameter passed each time may not have the same size. How does the function know the ending subscript value to perform the summation? The *UBound function* gives the upper boundary of an array. For example, UBound(A) will return the largest subscript of A; therefore, the preceding Sum function can be written as follows:

```
Function Sum(A() As Double) As Double
    ' This function returns the total of a Double type array, A()
    Dim Total As Double
    Dim I as Integer
    For I = 0 To UBound(A)
        Total = Total + A(I)
    Next I
    Return(Total) 'Return total to the caller
End Function
```

Suppose in the Form Click event procedure, you code the following:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim Salary(2) as Double
    Dim CashNeeded As Double
    ' Populate the Salary array with pay data
    ' In real operations, data will be read from a file
    Salary(0)=40000
    Salary(1)=80000
    Salary(2)=55000
    ' Compute total salary to arrive at cash needed for payroll
    CashNeeded = Sum(Salary)
    MsgBox("We need " & CashNeeded & " dollars.")
    ' additional statements follow
End Sub
```

When the Sum function is invoked, the upper bound of the array parameter A will be exactly 2 because Salary's subscript has 2 as its upper bounds. The For loop in the Sum function will be set up equivalently to the following:

```
    For I = 0 To 2
```

All elements in Salary will be involved in the computation. Test the preceding procedures. You should see the total cash needed is 175,000.

Tip

The function to find the lower boundary of an array is LBound. In versions since VB.NET, all arrays have a lower boundary of 0, so the LBound function is not as indispensable as in versions prior to VB.NET.

8.2 Sample Uses of Arrays

In addition to their uses in simplifying repetitive operations (as illustrated in the preceding examples), arrays can be used to solve various programming problems. It is difficult to exhaust the list of their uses. The following examples represent a small sample.

Simplified Selection

What is considered your normal weight? This depends on your sex, height, and frame size. For simplicity, consider only the case of males with a medium frame. The following table gives an abbreviated list.

Height (In Shoes)	Normal Weight
5'2"	118-129
5'3"	121–133
5'4"	124–136
.	
.	
6'3"	167-185
6'4"	172-190

Suppose your program is expected to give an answer when the user enters the height and clicks the Show button. How can this be handled in code? (Assume the height is given in two text boxes: txtFeet and txtInches.)

When the program starts, you can populate the normal weight in an array, with each element holding the weight value for each different height. There should be 15 different heights (from 62 inches to 76 inches); therefore, the array should have 15 elements. In response to repetitive queries, the height entered in the text boxes can be converted to inches, which can then be used to compute the index to obtain the weight stored in the Weight array. The Form Load event procedure may appear as follows:

```
 ' Declare an array to hold weights for heights from 5'2" to 6'4"
Dim Weights(14) As String
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    ' populate the weight array with data
    ' In real application, data will be read from a file
    Weights(0) = "118-129" 'For 5'2"
    Weights(1) = "121-133" 'For 5'3"
    .
    .
    Weights(14) = "172-190" 'For 6'4"
End Sub
```

The preceding code appears tedious. In a real application, the data should be stored in a file. The data will be read and assigned to each element one at a time with a loop. (See the following example for an illustration.)

How can the stored data be used when the user enters the height and clicks the Show button? As explained, the entered height will first be converted to inches. This value can then be used to compute the index to obtain the weight data. The computation should be fairly straightforward. Consider the weight for the height of 62 inches (5'2"). It is stored in the first element (with a 0 index). The weight for 63 inches is stored in the second element. To compute the index, all you have to do is subtract 62 from the height. This relationship is shown in the following table:

Height in inches	Subtract 62 to obtain the index	Look in Weight(Index)	The value is
62	0	Weight(0)	"118-129"
63	1	Weight(1)	"121-133"
.	.		
76	14	Weight(14)	"172-190"

The following code should solve the problem:

```
Private Sub btnShow_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    Dim Inches As Integer
    ' Convert height to inches
    Inches = CInt(Val(txtFeet.Text) * 12 + Val(txtInches.Text))
    ' Display the weight
    MsgBox("Your normal weight should be " & Weights(Inches - 62) _
      & " lbs.")
End Sub
```

Suppose a person's height is 5'3". This height measurement will be converted to 63 inches by executing the line to convert the height, so Weights(Inches - 62) refers to Weights(1), which contains the string "121-133." The message box will display the following:
Your normal weight should be 121-133 lbs.

In Contrast to the Code Using Select Case

Note that the last line of code can also be done by the use of the Select Case structure as follows, assuming a variable, named Weight, has been declared as String:

```
    Select Case Inches
    Case 62
        Weight = "118-129"
    Case 63
        Weight = "121-133"
    Case 64
        .
        .
    Case 76
        Weight = "172-190"
    End Select
    MsgBox("Your normal weight should be " & Weight & " lbs.")
```

The code using this structure not only is much longer but also will have to be revised when the data change— over the years, the normal height and weight relationship may change. When you use the array and the data are stored in the file, however, the data can be updated without the need to revise the program.

Table Look Up

Imagine the information service desk of a midsize company. When a customer calls in to ask for the phone number of a particular employee, the clerk will use your program to find the needed information in response. How should you code this program?

Looking Up the Array

There are, of course, several ways to accomplish this. One possibility is to keep all the employees' names and phone numbers in two separate arrays; for example, Employees() and Phones(). These arrays can be populated as soon as the program starts. When the clerk enters a name in a text box such as txtName and clicks a button such as Search, your program can search through the Employees array, find the match, and give the corresponding phone number. If the arrays have been populated properly, the following code should accomplish the search:

```
Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSearch.Click
    Dim TheName As String, I As Integer
    TheName = txtName.Text 'Get the name entered by the user
    For I = 0 To EmployeeCount -1
        If TheName = Employees(I) then
            ' The ith employee has the name searched for.
            ' Give the ith phone number along with the name.
            MsgBox("The phone number for " & TheName _
              & " is " & Format(Phones(I), "(000)-000-0000"))
            Exit Sub
        End If
    Next I
    MsgBox("Employee name " & TheName & " not found.")
End Sub
```

In this procedure, the name of the employee to look up from the Employees array is taken from the text box txtName. A For loop is then used to compare TheName with each element (name) in the Employees array. The I loop counter will vary from 0 to EmployeeCount – 1 (where *EmployeeCount* is a class-level variable used to represent the number of employees). When I is 0, TheName will be compared with Employees(0), the first employee name in the list. When I is 1, TheName will be compared with Employees(1), and so on. If a match is found, a message box will display the employee's name and phone number (Phones(I)), which is presented with the proper format. The procedure is then terminated with the Exit Sub statement. If no match is found, the loop will continue until all employees in the list are compared. When the loop ends, the list of employees has been exhausted without a match. A message indicating the employee name is not found is thus displayed.

Populating the Array

In the previous procedure, you assumed that the value of the variable EmployeeCount has been set properly. How is the value of this variable set, and how are the two arrays populated? Assume that the employee phone file has been properly created, and the phone number and the employee name are separated with a tab character (vbTab) in each record. The following code should accomplish populating the arrays and setting the value for EmployeeCount:

```
Dim Phones(500) As Long
Dim Employees(500) As String
Dim EmployeeCount As Integer
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
```

```
        ' When the project starts, this routine populates the Phones()
        ' and Employees() arrays
        Dim TheRecord As String
        Dim PhoneNumber As Long
        Dim TheName As String
        Dim P As Integer
        Dim PhoneFile As IO.StreamReader
        ' First, Open the file
        PhoneFile = New IO.StreamReader("C:\Employees\Phones.txt")
        ' Populate the arrays
        Do Until PhoneFile.Peek = -1
            TheRecord = PhoneFile.ReadLine()
            P = InStr(TheRecord, vbTab)
            PhoneNumber = CLng(Microsoft.VisualBasic.Left(TheRecord, P - 1))
            TheName = Mid(TheRecord, P + 1)
            Phones(EmployeeCount) = PhoneNumber
            Employees(EmployeeCount) = TheName
            EmployeeCount += 1
        Loop
        ' Close the file
        PhoneFile.Close()
End Sub
```

You may have noticed that populating an array from a file is basically the same as populating a list box or combo box. You are exactly right. The steps are identical. Only the "objects" that you are populating are different. Also, you use the Items.Add method to add items (elements) to the controls, but you use the assignment statement to add elements to arrays.

Tracking Random Occurrences

Arrays can conveniently be used to keep track of the occurrence of random events; for example, the number of times each customer called during a period, the number of purchase orders issued for each parts inventory, and the number of times each depositor made deposits in a month. All of these involve frequency counting.

Frequency Count

As an illustration of how arrays can simplify frequency count, consider this simple problem. You would like to know that given 1,000 random numbers in the range of 0 to 9, how many times each number appears. The first reaction is probably to code the count routine as follows (assume all variables except Rand have been properly declared):

```
    Dim Rand As New Random()
    For I = 1 to 1000
        Number = Rand.Next(10) 'Generate a random # in range 0 - 9
        If Number = 0 Then Freq0 += 1
        If Number = 1 Then Freq1 += 1
        .
        .
        If Number = 9 Then Freq9 += 1
    Next I
```

Notice the variables Freq0 through Freq9. Each is serving a similar purpose. Why not use an array, such as Freq() instead? If you declare the variable Freq() as follows:

```
    Dim Freq(9) As Integer
```

You can use it in the following manner:

```
    If Number = 0 Then Freq(0) += 1
```

```
    If Number = 1 Then Freq(1) += 1
    .
    .
```

Now, inspect this line carefully:

```
    If Number = 0 Then Freq(0) += 1
```

The If condition tests whether Number is equal to zero. If so, Freq(0) is increased by 1. Note that if Number is zero, Freq(Number) and Freq(0) both refer to the same element. When Number is zero, you can code the following:

```
    Freq(Number) += 1
```

That is, you do not need the If condition to accumulate total count for zero. It turns out that all the lines below this one are the same way. There is no need to test the value of Number! All you have to code to count the frequency is the following:

```
    Freq(Number) += 1
```

This statement has the effect that when Number is zero, Freq(0) is increased by 1, and when Number is 1, Freq(1) is increased by 1, and so on as shown in the following table.

When Number Is	Freq(Number) Refers to	Effect of Freq(Number) += 1
0	Freq(0)	Freq(0) is increased by 1.
1	Freq(1)	Freq(1) is increased by 1.
.		
.		
9	Freq(9)	Freq(9) is increased by 1.

The statement can be used to replace *all* the 10 statements in the previous loop! The previous routine can now be rewritten as follows:

```
    Dim I As Integer
    Dim Rand As New Random()
    Dim Number As Integer
    Dim Freq(9) As Integer
    For I = 1 To 1000
        Number = Rand.Next(10)
        Freq(Number) += 1
    Next I
```

To see the results after the count, you can code the following:

```
    For I = 0 To 9
        Console.WriteLine(I & " " & Freq(I))
    Next I
```

You can test the program by placing the preceding two loops in the Form Click event. Inspect the results. The total of frequency counts for all numbers should be 1,000. Notice that the use of the array simplifies the code not only for counting but also for output.

Tip

Beginners tend to associate the range of the loop counter with the size of the array involved in the loop. Beware of this general tendency (and misconception). The two are not necessarily related. If you inspect the second loop in the previous example, the counter and the size of the Freq variable, do correspond to each other because the counter is used to retrieve the content of each element in the array. In the first loop, however, the counter goes from 1 to 1000, not 0 to 9 (lower and upper bounds of the array). They are completely independent of each other.

Counting Letters

Would you like to know how often each letter of the alphabet is used in daily life? Assume you have a text box named txtText for the user to enter a long text string. When the user is ready, he (she) clicks the Count button.

How should your program proceed to count? Recall that the key code value for A and Z are 65 (Keys.A) and 90 (Keys.Z), respectively. Before any letter is counted, it can be converted to uppercase. Its key code value will then have to fall in the range of Keys.A (65) through Keys.Z (90); therefore, if you declare an array variable LetterCount as follows:

```
    Dim LetterCount(25)
```

you will be able to use it to count the frequency of each letter in a fashion similar to the previous example. The following code represents a sample solution:

```
    Dim LetterCount(25) As Integer
    Dim I As Integer
    Dim Text As String
    Dim Ascii As Integer
    Text = Ucase(txtText.Text) 'Get upper case of the entered text
    For I = 1 To Len(Text)
        ' Obtain the ASCII value of the character being inspected
        Ascii = Asc(Mid(Text, I, 1))
        Select Case Ascii
            Case Keys.A To Keys.Z
                ' alphabet, count its frequency
                LetterCount(Ascii - Keys.A) += 1
            Case Else
                ' non-alphabetic characters, ignore
        End Select
    Next I
```

The For...Next loop is used to inspect each character in the text string (from the first through the last character) one at a time using the Mid function. The key code value of each character is then obtained using the Asc function. If it falls within the range of alphabet values, the key code value is subtracted by the key code value of A and the result is then used in the frequency count, in a similar fashion to the frequency count for random numbers. (Note that if the letter is A, subtracting its key code value, Ascii by Keys.A result in a zero; therefore, its count is tracked by LetterCount(0). By the same token, B's count is tracked by LetterCount(1) and so forth.) If the ASCII value falls out of the range of the alphabet, it is ignored.

Do you want to inspect the results? The following code will display the results in a list box named lstResults:

```
    ' Output the results into a list box
    lstResults.Items.Clear()
    For I = 0 To 25
        lstResults.Items.Add(Chr(I + Keys.A) & vbTab & LetterCount(I))
```

```
    Next I
```
You can test the preceding code by placing both routines in the btnCount Click event (assuming the Count button mentioned previously is named btnCount).

The two frequency counting examples can easily be modified to count the frequency of scores and letter grades for a class when data are available in a file or database table. They can be generalized to handle all the problems mentioned at the beginning of this subsection. Exercises 8-25 and 8-26 provide additional problems of this nature for you to practice.

Simulation

Because of their capability to accumulate frequency count easily, arrays can also be used conveniently to simulate/approximate probabilities in conjunction with the use of random numbers. As an illustration, consider the case of obtaining the number of good output units under uncertainty. Assume the product spoilage rate in a production process is 2%; that is, 98% of the product going through the process will turn out to be good. What is the probability that you will obtain 100, 101, . . . or more good units, when you put 105 raw units in a batch through the process? You can, of course, solve this problem analytically; however, it will take a long time to get the formula right, assuming you have average math skills, and obtain the correct computation thereof.

With the computer, you can use an alternative approach, which involves using random numbers to simulate the event and accumulate the experiences. (This kind of simulation is recognized as Monte Carlo simulation.) You can simulate whether a unit from the process is good or bad by drawing a random number. Because the probability of getting a spoiled unit is 2%, if the random number is higher than .02, you can claim that it is a good unit; otherwise, it is spoiled. If you simulate this inspection of good/bad output units 105 times, you should be able to find how many units turn out to be good. This experiment is equivalent to running 105 raw units through the process, and counting the number of good units. If you run the experiment 1,000 (or even 10,000) times, you should be able to count the number of times good output units equal 100, 101, . . . or more units.

Here is a way to code this simulation process step by step. Assume that the variable GoodUnits is used to accumulate the number of good units in each experiment. When you inspect a unit of output and find it to be good, you add 1 to GoodUnits. (A random number greater than or equal to .02 represents the event of obtaining a good unit.) The code appears as follows:

```
    If Rnd() >=.02 Then
        GoodUnits += 1
    End If
```
Each batch consists of 105 units, so you need to inspect 105 times:

```
    GoodUnits = 0 'Before the process, there is no good unit.
    For I = 1 to 105
        If Rnd() >=.02 Then
            GoodUnits += 1
        End If
    Next I
```
After the inspection, the value of GoodUnits represents the number of good units produced from this batch. This experience can be accumulated by a frequency count. Assume that the array variable GoodUnitCount is declared with a proper range of its subscript. You can place the following line immediately below the loop:

```
    GoodUnitCount(GoodUnits) += 1
```
You have just finished simulating one batch of production. To arrive at a probability distribution, you need to repeat the experiment many times. For the interest of fairly accurate approximation, repeat the

experiment 1,000 times. The code including proper declaration for all the variables used appears as follows:

```
Dim I As Integer
Dim J as Integer
Dim GoodUnits As Integer
Dim GoodUnitCount(105) As Integer
' Randomize to get a different sequence of random numbers each time
Randomize()
For J = 1 to 1000
    ' Set good units to 0 at start of each production run
    GoodUnits = 0
    'Inspect a batch and count the number of good units
    For I = 1 to 105
        If Rnd() >=.02 Then
            GoodUnits += 1
        End If
    Next I
    ' Accumulate the frequency for good units
    GoodUnitCount(GoodUnits) += 1
Next J
```

Notice that GoodUnitCount is declared with an upper bound of the subscript of 105. This allows the possibilities that in a production run, the entire batch can be either all bad or all good. Notice also that before the inspection of a batch, the variable GoodUnits is assigned a value of 0. This must be done; otherwise, the count from the previous batch will be carried over, resulting in an extremely unreasonable large number (much greater than 105 units). Note also that you place the Randomize statement before the simulation starts so that the random number sequence will be different each time the program is run. You can place the results in a list box named lstResults and view the results:

```
For I = 0 To 105
    lstResults.Items.Add(I & vbTab & GoodUnitCount(I))
Next I
```

You can place these statements along with the preceding code in a button Click event. Alternatively, you can place them in a Sub procedure and then call the procedure from a button Click event. The following table is a sample result from one simulation.

Number of Good Units	Frequency	Approximate Probability of Good Units+
98	3	100.0%
99	14	99.7%
100	33	98.3%
101	97	95.0%
102	194	85.3%
103	278	65.9%
104	262	38.1%
105	119	11.9%
+ The probability that all 105 units are good is 119/1000; the probability to obtain at least 104 units is (199 + 262)/1000; etc.		

Because the random number sequences will differ in each simulation, the results will vary slightly; however, the results should give a pretty good approximation of the probability of obtaining a specified number of good units. (The preceding results show that there is approximately a 98.3%

probability that you will get 100 or more good units if you place a batch of 105 raw units into production, given a spoilage rate of 2%.) If you are interested in more stable or accurate results, you can increase the number of experiments. Of course, it will then take longer to compute the results.

Random Sampling Without Replacement

Consider the drawing of a lottery. There are 50 balls, numbered from 1 to 50. Six balls will be drawn. None of the drawn balls will be placed back in the bin; therefore, none of the six numbers will be repeated. (This type of problem is recognized as random sampling without replacement.) You want your program to emulate the drawing; that is, your program will display six random numbers in the range of 1 to 50, without repetition when the user clicks the Draw button. How do you proceed?

Drawing a Ball

Drawing a random number in the range of 1 to 50 can be done with the following statements:

```
    Dim Rand As New Random()
    BallNumber = Rand.Next(1, 51)
```

The first statement declares and associates the Rand variable with a random object. Recall that the Next method of the Random object generates a random number in the range of the two specified integers (at least equal to the lower specified number but less than the higher specified number). Rand.Next should generate a random integer in the range of 1 and 50.

Avoiding Repetition

The preceding statement does not guarantee that some of the six numbers drawn will not be the same. You need to find a way to verify that a new number has not already been drawn. There are two possible ways to accomplish this. Consider an *intuitive approach* first. As soon as you draw a number, you can compare it with all those numbers previously drawn. If a match is found, you should draw another number (discard the current number) and compare again until the new number has no match with the previously drawn numbers. This number is then included in the numbers already drawn. When the total number of numbers drawn is equal to six, your program has done the job, can display the results, and quit.

You can use a variable, for example, NumbersDrawn, to count the number of numbers drawn. To keep track of the numbers drawn, you can set up an array—for example, Balls(5). Each time a number is drawn, you can use a For loop, varying the counter from 0 to NumbersDrawn- 1 to compare this new number with the ones drawn to determine whether the number is already drawn. So that you can easily determine whether a match is found at the end of the loop, you can use another variable such as NotFound. Before the loop, you can assume the new number has no match; therefore, set NotFound to True. If a match is found within the loop, the variable is set to False and the loop is terminated.
The description can be translated into the following code:

```
    Dim Rand As New Random()
    Dim BallNumber As Integer
    Dim NumbersDrawn As Integer
    Dim NotFound As Boolean

    BallNumber = Rand.Next(1, 51) 'Draw a random number
    NotFound = True 'Assume this number is not already drawn
    For I = 0 To NumbersDrawn - 1
        If BallNumber = Balls(I) Then
            'A match is found; so NotFound is false
            NotFound = False
```

```
            Exit For 'and terminate the search
        End If
    Next I
    If NotFound Then
        ' The current ball number is indeed new
        Balls(NumbersDrawn) = BallNumber 'Keep the ball number
       NumbersDrawn += 1 'Increase the number count
    End If
```

Notice how a new number is added to the "already drawn" list. The current value of NumbersDrawn is used as the subscript to identify the position in the array, Balls where the ball number is stored because the index starts with a lower bound of 0. That variable is then increased by 1.

The routine should be repeated as long as NumbersDrawn is less than six. You should enclose the preceding code within a Do loop as follows:

```
    Do
        ' Place all the above statement here
    Loop While NumbersDrawn < 6
```

Complete Code for the Intuitive Approach

Assume the routine will be triggered by the user's click on a button named btnDraw. The complete procedure can appear as follows:

```
Private Sub btnDraw_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnDraw.Click
    Dim I As Integer
    Dim NumbersDrawn As Integer
    Dim BallNumber As Integer
    Dim Balls(5) As Integer
    Dim NotFound As Boolean
    Dim Rand As New Random()
    Do
        BallNumber = Rand.Next(1, 51)
        NotFound = True
        For I = 1 To NumbersDrawn
            If BallNumber = Balls(I) Then
                NotFound = False
                Exit For
            End If
        Next I
        If NotFound Then
            Balls(NumbersDrawn) = BallNumber
            NumbersDrawn += 1
        End If
    Loop While NumbersDrawn < 6
    lstNumbers.Items.Clear() 'Clear the list box
    ' Show results
    For I = 0 To 5
        lstNumbers.Items.Add(Balls(I))
    Next I
End Sub
```

Notice that in this code, you have included proper declarations for all variables used in the procedure. A For... Next loop is also used at the end to display the results by adding the drawn numbers to a list box named lstNumbers.

Another Approach

The preceding routine keeps track of the numbers that have been drawn. Alternatively, you can focus on each number in the entire range. That is, when a number is drawn, you can ask whether the ball has been drawn, not by comparing the numbers that have been drawn but by checking a "marker" for the particular number. If the marker is not turned on, the ball has not yet been drawn and you can display the number and turn its marker on; otherwise, it has been drawn, and the number should be discarded.
If you want to set up a marker for each ball, you will need 50 markers because there are 50 balls. The marker should be an array with 50 elements. Let the array name be AlreadyPicked. You can then declare it as follows:

```
    Dim AlreadyPicked(49) As Boolean
```

Now imagine you have drawn the number 3. You can test the value of AlreadyPicked(2) (the third element) to determine whether it has already been drawn. If AlreadyPicked(2) is False, you can set it to True and display 3; otherwise, you can ignore the number and repeat the drawing. Notice that 3 is just a particular number. Any number you draw (BallNumber) should work the same way. The routine to obtain a legitimate (non-repeated) ball number can be coded as follows:

```
    Do
        BallNumber = Rand.Next(1, 51) ‘Draw a number
        ‘ If the number has been drawn, repeat the drawing.
    Loop While AlreadyPicked(BallNumber - 1)
    Dim AlreadyPicked(49) As Boolean
    AlreadyPicked(BallNumber - 1) = True
    lstNumbers.Items.Add(BallNumber) ‘display the number currently drawn
```

You need six numbers, so the routine should be repeated six times. You can accomplish this by enclosing the preceding routine in a For...Next loop as follows:

```
    For I = 1 to 6
        ‘ Place the above drawing routine here
    Next I
```

Complete Code for the Second Approach

To test this approach, you can draw another button on the same form. Name it btnDrawAlt and set its Text property to "Draw, Alternative Way". The complete code should appear as follows:

```
Private Sub btnDrawAlt_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnDrawAlt.Click
    Dim BallNumber As Integer
    Dim I As Integer
    Dim Rand As New Random()
    lstNumbers.Items.Clear() 'Erase previous output
    For I = 1 To 6
        Do
            BallNumber = Rand.Next(1, 51) 'Draw a ball
            ' If it is already picked, then repeat the drawing
        Loop While AlreadyPicked(BallNumber - 1)
        AlreadyPicked(BallNumber - 1) = True 'Mark the ball drawn
        lstNumbers.Items.Add(BallNumber)  'Display the number
    Next I
End Sub
```

When you run the project, you should be able to verify that each time you click the button, six unique numbers in the range of 1 to 50 are displayed in the list box. These numbers should vary randomly from one round to the next.

Notice that with this approach, there is no need to have another array, Balls(), which keeps track of all the ball numbers drawn. Each number is added to the list box when it is determined to be

legitimate. These numbers are also implicitly kept in the AlreadyPicked array. The position of the element that has been set to True represents the ball number that has been drawn; that is, if AlreadyPicked(4) is True, 5 is a number that has been drawn. The following loop should also give the numbers that have been drawn:

```
For I = 0 To 49
    If AlreadyPicked(I) Then
        ' I+1 has been picked
        Console.WriteLine(I + 1) 'Print the ball number
    End If
Next I
```

Comparing the Two Approaches

When you inspect the two procedures, you should notice that the first approach requires longer code but takes a smaller array, whereas the second approach requires shorter code but takes a longer array. The second approach should be faster because the loop to determine whether a ball (number) has been drawn requires a test on one element of the array; that is, AlreadyPicked(BallNumber - 1). In contrast, the first approach requires the execution of a loop. Only after the loop is completely executed can the routine conclude that the number has not been drawn; thus, there is a trade-off between speed and space (storage) used.

You will encounter many problems of similar nature and, therefore, be faced with the same design issue as to which approach to use given a situation. If speed of execution is the most important consideration, you should use the second approach. You should use the first approach only when the number of elements to keep track of (balls to draw) is relatively small. When this number approaches the total potential number (50 in this example), you will always be better off using the second approach.

Sorting and Searching

Another use of arrays is in the area of sorting and searching. Sorting entails arranging data in either ascending or descending order, while searching involves locating an item in any array of data. Although the goals of sorting and searching are fairly simple, they are important in practical applications. Decades ago, it was claimed that nearly 20%of the computer resources were used in sorting. Interestingly, while the goals of sorting and searching are simple, their algorithms are fascinating; and disparities in the performance of these algorithms are great. As a result, volumes of books were devoted to studying algorithms for sorting and searching.

Writing efficient sorting and searching routines for practical applications is no longer an important job of the software engineer because most software systems provide support for sorting and searching. For example, VB 2008 provides Sort and BinarySearch methods for array objects. These methods are discussed in Section 8.5. Even so, you should find algorithms for sorting and searching intriguing. Learning these algorithms can inspire you to design better algorithms in other areas of applications. An introduction to sorting and searching is presented in Appendix C for the inquisitive reader.

8.3 Control Arrays

This section discusses issues pertaining to handling controls as an array. You may encounter situations in which the visual interface design of your application requires the flexibility of control arrays. For example, consider the data entry screen for a program that evaluates the merits of an investment project.

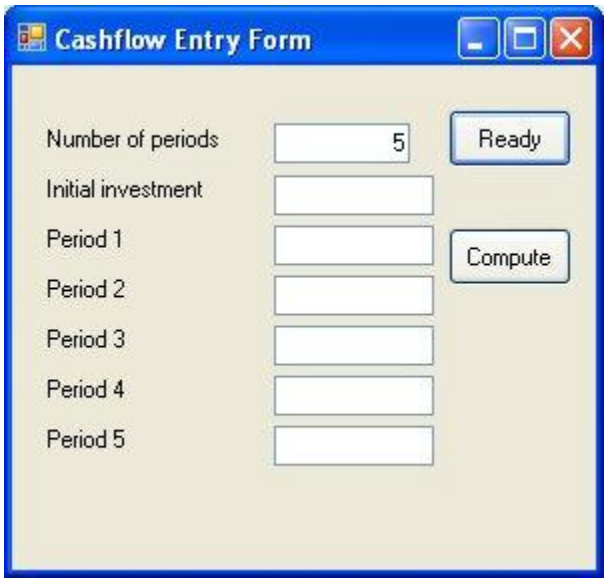
The user will enter the project life in years and then enter the cash flow for each year. The cash flows should be entered in text boxes, the number of which should vary based on the project life. A sample visual interface is shown in Figure 8-1. As you can see, the number of text boxes for the cash flows and their related labels depend on the number of years specified in the text box for the project life. The text boxes for cash flows should appear when the user clicks the Ready button or presses the Enter key in the text box for the project life ("how many years"). There are at least two challenges:

- The controls (labels and text boxes) should be added to the form during run time, and should depend on the number specified by the user.
- To handle the user input flexibly, the text boxes must be handled in a way similar to an array.

Before considering these challenges, take a look at the controls drawn to the form at design time in the following table first:

Control	Name	Remark
Label	Label1	To display the text "How many periods"
Text box	txtPeriods	To enter number of periods (project life in years)
Button	btnReady	To invoke the procedure to add labels and text boxes
Button	btnCompute	To invoke the computation routine

Figure 8-1
Visual interface for the Cashflow Entry Form



In this application, the text boxes for cash flow amounts are created after the user has specified the number of periods and either pressed the Return key or clicked the Ready button.

Adding Controls to the Form by Code

VB controls are objects. They can be added to the form similar to other objects such as the stream reader. The main difference is that controls should be visible in the form; therefore, in the code, you must specify those properties that pertain to the visual elements—for example, size, location, and text properties. Also, the form has a Controls collection, as explained in Chapter 7. A newly added control must also be added to that collection, using the *Add method*. The following code should add a label to a form.

```
    Dim lblSample As Label
    LblSample = New Label
```

```
With lblSample
   .Text = "This is a sample"
   .Location = New Point(5, 20) 'Set Left at 5 and Top at 20
   .AutoSize = True 'So that I don't have to specify its size
End With
Me.Controls.Add(lblSample)
```

The Dim statement declares that lblSample is of the Label type. The next statement creates a new label. The statements within the With … End With block set up the visual properties for the label. Its Location property is assigned a Point object at (5, 20), which is five pixels from the left margin, and 20 pixels from the top margin of its container (the form). This statement is equivalent to assigning 5 to the label's Left property, and 20 to its Top property. The last statement adds lblSample to the form's Controls collection.

Tip

If you have more than one control to add to the form, you can use the AddRange method that expects a collection of controls as its parameter. A collection of controls can be created with the following syntax:

```
New Control(){Control1, Control2, …}
```

The following statement will add Label1 and Label2 to the form.

```
Me.Controls.AddRange(New Control() {Label1, Label2})
```

You may wonder why you use a separate statement to create a new label, instead of using the New keyword in the Dim statement. In this case, either way will produce the same result; however, keep in mind that there is some difference between the two approaches. When a new label is created using an assignment statement with the New keyword, as in you code, each time this statement is executed, a new label is created. If it is placed in a loop, many labels can be created. If you use the New keyword in the Dim statement, you create the label at that time. Only when that statement is invoked again will another label be created.

Creating a Specified Number of Text Boxes

Back to the cash flow project, assume that the number of periods specified is N. You will then need to generate N + 1 text boxes (period 0 for initial investment). You can use a For loop in which a text box is created. The code can appear as follows:

```
Dim txtCashflow() As TextBox
Dim I as Integer
Dim N as Integer

For I = 0 To N
   txtCashflow(I) = New TextBox 'Create a new text box
    With txtCashflow(I)
        .Size = New Size(80, 20) 'Width=80; Height=20
        .Location = New Point(95, 50 + I * 25)
        .TabIndex = 1 + I 'Set the tabindex
       .TextAlign = HorizontalAlignment.Right 'Align the text on the right
   End With
  Me.Controls.Add(txtCashflow(I))
Next I
```

With this code, you are able to generate the desired number of text boxes. Note that each element of the text box array is created by the first statement within the For loop. This is the way to create a control array in VB 2008, which does not support the use of the New keyword to create a control array.

Complete Code to Create the Project Interface

You are now ready to develop the code to create the visual interface for your cash flow project. You will place the code in a sub procedure for the reason to be explained later. The procedure appears as follows:

```
Dim txtCashflow() As TextBox
Dim lblCashflow As Label
Private Sub ShowBoxes()
    Dim I As Integer
    Dim N As Integer
    Dim LL, LT As Integer
    Dim Top As Integer

    N = CInt(Val(txtPeriods.Text))
    If N = 0 Then Exit Sub 'Do nothing if N is zero
    ReDim txtCashflow(N)
    LL = lblPeriods.Left
    LT = txtPeriods.Left
    Top = lblPeriods.Top + 25
    For I = 0 To N
        lblCashflow = New Label
        With lblCashflow
            If I = 0 Then
                .Text = "Initial investment"
            Else
                .Text = "Period " & I
            End If
            .AutoSize = True
            .Location = New Point(LL, Top + I * 25)
        End With
        txtCashflow(I) = New TextBox 'Create a new textbox
        With txtCashflow(I)
            .Size = New Size(80, 20) 'Width=80; Height=20
            .Location = New Point(LT, Top + I * 25)
            .TabIndex = 1 + I 'Set the tabindex
            .TextAlign = HorizontalAlignment.Right 'Align the text on the right
        End With
        Me.Controls.AddRange(New Control() {lblCashflow, txtCashflow(I)})
    Next I
    btnReady.TabIndex = N + 2
    btnCompute.TabIndex = N + 3
End Sub
```

Notice the last statement within the loop. The two newly created controls (the label and text box) are now added to the form's Controls collection, using the *AddRange method* instead of the Add method. The AddRange method expects a control array as its parameter. The control array is created with exactly the same syntax as creating any data array as explained in Section 8.1. Notice also the two statements below the loop. They set the TabIndex properties for the two buttons that were drawn on the form at design time so that the user can tab through the text boxes before running into the buttons.

Invoking the ShowBoxes Sub

The ShowBoxes sub can be invoked in two different events: when the user clicks on the Ready button, or when the user presses the Enter key in the text box named txtPeriods. These two event procedures can be coded as follows:

```
Private Sub btnReady_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnReady.Click
```

```
    ShowBoxes()
End Sub
Private Sub txtPeriods_KeyPress(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles txtPeriods.KeyPress
    If e.KeyChar = ControlChars.Cr Then
        ShowBoxes()
        e.Handled = True 'No further process needed
    End If
End Sub
```

Referencing Elements in a Control Array

As you may have already noticed in the preceding example, elements in a control array are referenced by an index, just like any variables. For example, txtCashFlow(I) refers to the $i+1^{th}$ element in the array. Suppose the user clicks the Compute button after entering relevant data in the cash flow boxes. You obtain the values entered in the text boxes and then assign them to an array with the following code:

```
Private Sub btnCompute_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCompute.Click
    Dim I As Integer
    Dim CashFlow() As Double
    Dim N As Integer
    N = cllCashflow.Count - 1
    ReDim CashFlow(N)
    For I = 0 To N
        CashFlow(I) = Val(txtCashflow.Text)
    Next I
    'Other statements
End Sub
```

Additional Note

The labels and text boxes that you created in the preceding examples are not capable of handling events. This capability can be important. For example, you may want to use the KeyPress event to ensure that keys entered in the text boxes are numeric only. With minor modifications, you can add this capability. This topic is discussed in Chapter 12, "Object-Based Programming."

8.4 Two-Dimensional Arrays

As stated at the beginning of this chapter, array variables can have more than one dimension. Consider the case of two-dimensional variables. The syntax to declare a two-dimensional variable is as follows:

```
    Dim Name(UB1, UB2) As Type
```

where *Name* = any legitimate variable name,

UB_n = an integer representing the upper bound for the n^{th} subscript, and

Type = any valid data type.

As you can see, the only difference between a two-dimensional array and one-dimensional array is the number of subscript(s). When a variable has more than one subscript, use commas to separate the subscripts. The following are examples of valid declarations of two-dimensional arrays:

```
    Dim A(9, 9) As Integer
    Dim Weights(Row, Col) As String
```

The first line declares a 10 × 10 Integer array, A, depicted as follows:

```
A(0, 0)     A(0, 1)     A(0, 2)     ...   A(0, 9)
A(1, 0)     .     .     .                 A(1, 9)
```

```
.     .     .     .     .
A(9, 0)                                   A(9, 9)
```

The second line declares a Weights String table whose size depends on the values of the variables, Row and Col. These two variables should exist and have been assigned proper values before the Dim statement is executed.

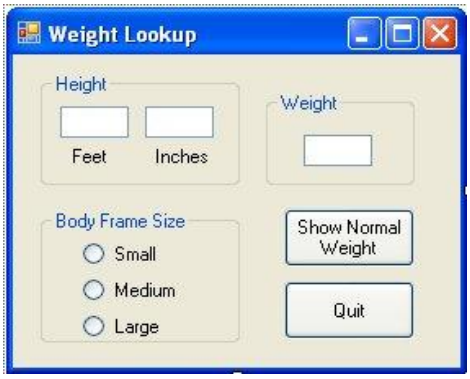
Two-dimensional arrays can be used to handle various kinds of two-dimensional data. The following discussion explains how they can be used to represent tables, matrices, and even game boards.

Tables

Revisit the weight lookup example from the first section. The weight table shown there was only for males with a medium frame. What happens to the small- and large-frame males? Actually, the complete weight table appears as follows:

Height (in Shoes)	Small Frame	Medium Frame	Large Frame
5'2"	112–120	118–129	126–141
5'3"	115–123	121–133	129–144
5'4"	118–126	124–136	132–148
5'5"	121–129	127–139	135–152
5'6"	124–133	130–143	138–156
5'7"	128–137	134–147	138–156
5'8"	132–141	138–152	147–166
5'9"	136–145	142–156	151–170
5'10"	140–150	146–160	155–174
5'11"	144–154	150–165	159–179
6'	148–158	154–170	164–184
6'1"	152–162	158–175	168–189
6'2"	156–167	162–180	173–194
6'3"	160–171	167–185	178–199
6'4"	164–175	172–190	182–204

Figure 8-2
Visual interface for the Weight Lookup project



Refining the Weight Lookup Project

You can design a project similar to the previous one with this more complete set of data to respond to any normal-weight queries. The weights shown in the preceding table can be read into a two-dimensional array. The user can specify the height and the frame size and then click a Show Normal Weight button. The program can then retrieve a cell (an element) from the table, and display it in a text box. The visual interface appears as in Figure 8-2.

The following table lists the controls and properties used in the code.

Control	Property	Setting	Remarks
Text box	Name	txtFeet	To specify height
Text box	Name	txtInches	
Text box	Name	txtNormalWeight	To display normal weight for the given height
Radio button	Name	rdbBodyFrameS	To specify body frame
	Text	Small	
Radio button	Name	rdbBodyFrameM	
	Text	Medium	
Radio button	Name	rdbBodyFrameL	
	Text	Large	
Button	Name	btnShow	To initiate the computation and display of normal weight
	Text	Show normal weight	
Button	Name	btnQuit	To end the program
	Text	Quit	

Code Requirements

The program should satisfy the following requirements:

- As soon as the program starts, the weight table should be populated with data read in from a file.
- When the user clicks the Show Normal Weight button, the program should convert the height into inches; identify the radio button clicked for the frame size; and use the numbers obtained in the preceding bullet point to retrieve and display the weight data in the label.

Coding the Project

Now consider the code for the project. The first step is to populate the Weights(,) table as the project starts. Assume the data file appears as follows:

```
62, 112-120, 118-129, 126-141
63, ...
.
.
76, ...                 182-204
```

The table can then be populated as follows:

```
Dim Weights(14, 2) As String
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
    Dim Col As Integer
    Dim Row As Integer
    Dim TheRecord As String
    Dim HeightNWeight() As String
    Dim WeightFile As IO.StreamReader
```

```
    ' Open weight file
    WeightFile = New IO.StreamReader("MaleWeights.txt") 'file in the default folder
    ' Populate the weight table
    Do Until WeightFile.Peek() = -1
        TheRecord = WeightFile.ReadLine()
        HeightNWeight = Split(TheRecord, ",")
        Row = CInt(HeightNWeight(0)) - 62
        For Col = 0 To 2
            Weights(Row, Col) = HeightNWeight(Col + 1)
        Next Col
    Loop
End Sub
```

The array variable, Weights is declared as a 15 × 3 table. The 15 rows will accommodate the heights from 5'2" to 6'4" and the three columns will accommodate the three body frame sizes. The StreamReader, WeightFile reads in the data one record (line) at a time. The record is assigned to the TheRecord variable, which is parsed into a string array using the *Split function*. The Split function has the following syntax:

```
    Split(StringToParse, Delimiter)
```

It breaks the string into sub-strings based on the specified delimiter, and returns the sub-strings as a one-dimensional array. For example, the expression Split("AB,C,DEF", ",") will return a string array with "AB," "C," and "DEF." The expression Split(TheRecord, ",") should return an array with the height as the first element followed by three weight ranges for different frame sizes. The result is assigned to the HeightNWeight variable. The row position to store the weight data is computed by subtracting 62 from the height (in inches). In the For loop, the weights (from the record) are then assigned to the Weights table in the calculated row position. Notice that the first element in HeightNWeight is the height, and its second element holds the normal weight range for the small body frame. Thus, its position Col + 1 holds the data that should be assigned to the Col of Weights.

When the Show button is clicked, the program needs to convert the height to inches and identify which of the frame radio buttons is clicked. Converting the height is easy, as shown in the previous example:

```
    Inches = CInt(txtFeet.Text) * 12 + CInt(txtInches.Text)
```

To compute the row position in the Weights table, you can subtract 62 from Inches; that is,

```
    Row = Inches - 62
```

But how do you identify which radio button was clicked last, and therefore the column position? You will use the approach explained in the Explore and Discover exercise 5-11. After you identify the row and column, the element (cell), Weights(Row, Col) should give the weight that corresponds to the height and body frame computed previously. The code to handle the event when the user clicks the Show button should appear as follows:

```
Private Sub btnShow_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    Dim Inches As Integer
    Dim Row As Integer
    Dim Col As Integer
    Inches = CInt(txtFeet.Text) * 12 + CInt(txtInches.Text)
    Row = Inches - 62
    'The following Select Case structure identifies the column position
    Select Case True
        Case rdbBodyFrameS.Checked
            Col = 0 'Column for small frame
        Case rdbBodyFrameM.Checked
            Col = 1 'Column for medium frame
        Case rdbBodyFrameL.Checked
```

```
            Col = 2 'Column for large frame
    End Select
    txtNormalWeight.Text = Weights(Row, Col) 'Show normal weight range
End Sub
```

Can you see the benefits of using a table (array) for selection? Without the Weights table as an array, the alternative will be to code the project using the Select Case structure. Compare this solution with the solution to the tuition problem illustrated in Chapter 5, "Decision." Using that approach, you will also need to nest the Select Case structure because the selection involves two factors, the height and the body frame size. The code will be much longer and not as neat, elegant, or flexible.

Matrices

A matrix consists of a rectangular array of numeric elements in rows and columns. It is mainly used in matrix algebra to handle various linear algebraic problems. The two-dimensional array fits exactly the definition of a matrix, and can be used to perform any matrix operations. For example, by definition, adding two matrices calls for the addition of the corresponding elements in the two matrix operands. Suppose the two matrices as represented by two two-dimensional arrays are A and B. The matrix operation A + B will mean performing A(I, J) + B(I, J) for the entire range of both subscripts. A and B must have the same dimensions. The following function procedure adds the two matrices and returns the results to the caller.

```
Function MatrixAdd(ByVal A(,) As Double, ByVal B(,) As Double) As Double(,)
    Dim I As Integer
    Dim J As Integer
    Dim C(,) As Double
    ReDim C(UBound(A, 1), UBound(A, 2))
    For I = 0 To UBound(A, 1)
        For J = 0 To UBound(A, 2)
            C(I, J) = A(I, J) + B(I, J)
        Next J
    Next I
    Return (C)
End Function
```

In this procedure, the loop counters I and J are set to start from 0 to the upper bound of their respective subscripts.

Notice how the header is specified. The two parameters (A and B) and As Double are each followed by a pair of parentheses. A comma is placed in each pair to indicate the *rank* (number of dimensions) of the array. Without the comma, each is presumed to be a one-dimensional array. The As Double(,) clause in the end of the header indicates that the function will return a two-dimensional array. Notice also the arguments used in the UBound functions. When the array has more than one subscript, you can specify the subscript number for which you want the boundary. UBound(A, 1) tells the function to return the upper bound of the first subscript; and UBound(A, 2) tells the function to return the upper bound of the second subscript. When the optional second parameter (the subscript number) is omitted, the first subscript is the default.

Game Boards

Two-dimensional arrays can also be used to represent boards used in games that call for the placement of different markers or stones on the board. For example, in a tic-tac-toe game, two players will mark alternately on a 3 × 3 board with X's and O's. The first player who can place three of the same marks in a straight line (vertically, horizontally, or diagonally) wins the game. You can use a two-dimensional

variable such as Status to keep track of the marks that the players place on the board while your program draws the marks on the screen. For example, when the first player places an X mark on position (1,1), you can code the following:

```
    Status(1, 1) = 1
```

You can use any nonzero value to represent one side and another value (for example, -1) to represent the other side. A zero value in a position will indicate that the position is not yet marked (occupied) and is available; otherwise, an attempt to mark the position will be considered illegitimate. Each time a player marks a position, your program will need to determine whether your program can declare the winner.

In addition to keeping track of the status internally, you will also need to consider how the game should be represented externally; that is, what will be used to represent the board and the marks onscreen. One way is to actually draw the X and O marks on a picture box, which can be used to represent the board. You should be able to do this after you study Appendix B, "Graphics, Animation, Drag and Drop." Another way, which uses more computer resources, is to use nine labels to represent the nine positions on the board. As illustrated previously, you can create labels and any controls at run time; however, in such a case, you will need the capability to handle events for these labels; for example, when the user clicks on a label that is not yet marked, the program needs to draw a mark. You will revisit this programming challenge in Project 12-25 in Chapter 12. Suffice it to say that the two-dimensional array can be handily used to represent the state of the board.

8.5 Additional Notes on Arrays

The preceding discussion focuses on the practical uses of arrays. This section presents several additional aspects and issues concerning arrays that deserve your attention.

Preserving Data in ReDim

Each time you use the ReDim statement to change the size of an array, the previous content of all elements will be reinitialized (to 0 for numeric variables and to zero-length string for string variables). You can preserve their previous contents, however, by using the keyword Preserve in the ReDim statement. For example, assume an array variable Students() has been previously populated with text strings, and you would like to expand its size by doubling its upper bound without losing its previous contents. You should be able to accomplish this by the following statement:

```
    ReDim Preserve Students(2 * UBound(Students))
```

Displaying Array Contents in the List Box or Combo box

It is easy to display the content of a one-dimensional array in a list box or combo box. All you have to do is to set the control's *DataSource property* to the array. For example, assume you have a list box named lstX on your form, and a one-dimensional array named X. To display the content of X in lstX, you can code:

```
    lstX.DataSource = X
```

Note that when the content of X changes, the contents of the list box remains the same. To update the list box, you will need to reset the data source to Nothing (listX.DataSource = Nothing) and execute the same statement above. Assigning a control's DataSource property with an object is referred to as to bind the data to the control. More about data binding will be discussed in Chapter 9, "Database and the ADO.NET."

Releasing an Array: The Erase Statement

You can release the resources used by an array by using the Erase statement. For example, suppose the array variable Employees has been populated with a department's employee names, and you no longer need it. You can use the following statement to release the memory used by the variable.

```
Erase Employees
```

The array will become Nothing. All its previous contents are gone. It does not exist any longer.

Array Properties and Methods

VB 2008 treats all arrays as objects. As such, they are given built-in properties and methods. For example, each array has the *Length property* that gives the length of the array, and the *Rank property* that gives the number of dimensions of the array. To illustrate, assume X is an array. The following expression will return the number of elements in X:

```
X.Length
```

The CopyTo Method

The CopyTo method allows a convenient way to copy data from an entire array into another. For example, assume X and Y have the same length, and you want to copy the contents of X to Y. You can code the following:

```
X.CopyTo(Y, 0)
```

The second parameter specifies the index of the target array (Y) into which to start copying data. If you specify 1 instead of 0, the copy operation begins with the second element in Y; that is, the first element of X will go into the second element of Y, and the second element of X will go into the third element of Y, and so forth. In such a case, Y will have to have at least one more element than X; otherwise, an error will result.

Tip

Beware of the following statement when both are declared as array variables:

```
Y = X
```

It does not produce any syntax error, but it may not give what you want. The statement copies the pointer of X to Y. After the statement, both X and Y refer to the same array (memory address). Whatever changes you make to one of the two variables will be reflected in the other variable.

Shared Array Methods

The preceding discussion pertains to properties and method for specific arrays. In addition to these interfaces, the *Array object* (type) also has many shared (static) methods that can operate on any arrays. The following lists a few selected methods of particular interest.

The Clear Method

The Clear method allows you to reinitialize a portion of an array, and has the following syntax:

```
Array.Clear(Name, BeginningIndex, Length)
```

The following statement will reinitialize 10 elements in the array named A beginning from the second element:

```
Array.Clear(A, 1, 10)
```

Tip

Notice that in the statement the reference is made to Array, not A, the array variable. Although the statement A.Clear(A, 1, 10) will work correctly in VB.NET, the same code will not work in VB 2005 or VB 2008. Clear is a Shared method of the Array object (type). The newer versions will handle a shared member through the "type" reference but not through the "object" reference. Shared members of objects are briefly discussed in chapter 12.

The Sort Method

The Sort method sorts an array in order. For example, assume that you have an array, X. You can sort it by the following statement:

```
Array.Sort(X)
```

You can even specify the range to sort. For example, the following statement will sort five elements in X starting from the second position.

```
Array.Sort(X, 1, 5)
```

The Reverse Method

The Reverse method will reverse the order of the elements in an array. For example, assume X has {1, 2, 3} as its contents. The following statement will result in X containing {3, 2, 1}

```
Array.Reverse(X)
```

The Binary Search Method

If you have a sorted array, the Binary Search method can be used to search the array more efficiently. Assume X is a sorted Integer array. The following expression will return the position in which 5 is found. If the search value is not found, the method returns a negative value.

```
P = Array.BinarySearch(X, 5)
```

Look It Up

Use the "array class" keyword to search the Index tab of the Help file; then click All Members under the found list to browse for all available array properties and methods for their uses. In most cases, each method can perform different tasks depending on the parameters specified.

Appropriate Use of Arrays

As you can see from the preceding examples, arrays can be useful and powerful. They are often used in conjunction with loops. For this reason, arrays tend to be overused by the beginner when the program calls for the use of loops. As an illustration, suppose you would like to read a file and list its content. Simple? Assume the StreamReader named DataFile has been associated with the physical file. After learning about arrays, a beginner might code as follows:

```
Dim DataRec(5000) As String
Dim Counter As Integer
Dim I as Integer
Do Until DataFile.Peek() = -1
    DataRec(Counter) = DataFile.ReadLIne()
   Counter = Counter + 1
Loop
For I = 1 To Counter
    1stRecords.Items.Add(DataRec(I))
Next I
```

That is, the entire file is read in and kept in the memory before each record is added to the list box 1stRecords for display. If the data in a file can be handled in one single pass (reading from the beginning through the end), there is actually no need to use an array.

The same goal can indeed be accomplished with the following code:

```
Dim DataRec As string
Do Until DataFile.Peek() = -1
    DataRec =DataFile.ReadLine
    1stRecords.Items.Add(DataRec)
Loop
```

Arrays use much more memory than scalar variables. When there is no need to use arrays, they should be avoided. In general, you will need an array if the group of data will need to be worked on back and forth (usually more than one time) or randomly such that the order cannot be predicted. Hopefully, the examples in this chapter have shown you the proper context where arrays are called for.

8.6 The Sales by Product Project: An Application Example

This chapter concludes with an application example that requires the accumulation of total sales by product. The project shows how arrays can be used to solve a practical business problem.

The Sales by Product Project

Your company sells 25 different products. The sales transactions are kept in a text file. Each record contains a field representing the product (product code) and another field for the sales amount. These records are kept in ascending order by the transaction date, not by the product code. You are interested in obtaining the total sales by product. The results should be displayed in a list box.

Algorithm

How can the sales be accumulated by product when the file is not sorted in the order of product code? There are several ways, each with variations in details. This example will follow an approach that uses two arrays: one to keep track of the product codes already encountered and the other to accumulate total sales for the corresponding products. This can be depicted as follows:

Product Codes	D	A	C	...	K
Sales by Product	Sales for D	Sales for A	Sales for C	...	Sales for K

How do you keep track of product codes? As each record is read, its product code is compared with the ones already in the product code array. The number of codes already in the array can be tracked by a counter variable; for example, ProductCount. If the current product code is not found, the current (new) product code should be placed in the position pointed by ProductCount, which should then be increased by 1. (Recall that the first element of the array is indexed as zero.) This approach is similar to the first method of generating a random lottery number discussed previously in this chapter.

How do you accumulate the sales for the corresponding product? The preceding search for the product code should give the position of the product code in the array. Call the position ProductPos. This value can then be used as the subscript for the array SalesByProduct to add the current sales; that is, the total sales by product can be accumulated with the following statement:

```
SalesByProduct(ProductPos) += Sales 'Increase the array element by Sales
```

Coding the Project

Suppose the routine to perform the computation is invoked when the user clicks the btnCompute button. This routine should do the following:

- Read the sales records.
- Build the product code array, and accumulate the total sales by product.
- Display the results in a list box.

Because the accumulation of sales hinges on the successful creation of the product code array, let's consider this aspect in more detail. To show how this array can be built, assume the first few records in the file are those shown in the following table. For simplicity, pay attention only to product code.

Record No.	Product Code	Sales Amount (Not Shown)
1	D	
2	A	
3	D	
4	C	

When the first record is read, the product code array is empty and the product count is zero. The current product code in record one (that is, D) will be compared with all elements in the product code array, which is still empty; therefore, D is not found. You will place D in the product code array at the position pointed by the product count and then increase the product count by 1. The state in the memory can be represented as in the following table.

Record No.	Product Code	Content of Product Codes (Array)	Product Count
1 (before processing)	D	(Empty)	0
1 (after processing)	D	D	1

After the second record is read and before it is processed, the memory state will be the same as after processing the first record, except for the current product code from the record, which is now A. The same steps just outlined will be repeated. Again, A is not found in the array, so the product code A is added to the array, and the product count is increased by 1. The states of memory should be as shown in the following table.

Record No.	Product Code	Content of product Codes (Array)	Product Count
1 (before processing)	D	(Empty)	0
1 (after processing)	D	D	1
2 (after processing)	A	D, A	2

When the third record is read, the current product code will be D, which is already in the array. There will be no change to the product codes array. The product code position will be identified and used to accumulate the sales amount.

Because the product codes can appear in any order, they will be searched sequentially. The search routine appears as follows:

```
    For I = 0 To ProductCount - 1
```

```
       If ProductCode = ProductCodes(I) Then
            ' A match is found, set the product position and
            ' terminate the loop
            ProductPos = I
            Exit For
        End If
    Next I
```

If the product code is not found, you will need to add it to the array. As shown in the lottery number example in Section 8.2, you will need an indicator to determine whether the product code is in the array. Here, you will use the value of ProductPos for this purpose.

How? As shown in the preceding code, if the product code is found, ProductPos will be set to its position in the array. The position has to be in the range of 0 to ProductCount - 1. If you set ProductPos to ProductCount before the loop, you can check whether this value has changed. If it has not changed, the product code is not found and you can add the current product code to the array. The code can appear as follows:

```
    ProductPos = ProductCount
    For I = 0 To ProductCount - 1
        If ProductCode = ProductCodes(I) Then
            ' Product code is found, get this position and exit the loop
            ProductPos = I
            Exit For
        End If
    Next I
    If ProductPos = ProductCount Then
        ' No change in product pos as initially set.
        ' This means the product code is not found
        ' Update count and add the code to array
        ProductCount += 1 'Increase count by 1
        ' and keep the current code in the array
        ProductCodes(ProductPos) = ProductCode
    End If
```

The preceding code deals with a product code read from a record. Of course, you will need to open and read the file. Reading records from the file will take another loop, which should be the outer loop of the preceding routine. For simplicity, assume that each transaction in the file has only two fields: product code and sales. Further assume that the two fields in each record are separated by a Tab character (vbTab). The following code should satisfy the project's requirements:

```
Private Sub btnCompute_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCompute.Click
    Dim I As Integer
    Dim SalesFile As IO.StreamReader
    Dim ProductPos As Integer
    Dim ProductCount As Integer
    Dim CodeNSales As String
    Dim CNS() As String
    Dim ProductCode As String
    Dim Sales As Double
    Dim ProductCodes(24) As String
    Dim SalesByProduct(24) As Double
    ' Step 1: Find the file in the default folder
    SalesFile = New IO.StreamReader("Sales.txt")
    ' Step 2: read and accumulate sales by product
    Do Until SalesFile.Peek() = -1
        CodeNSales = SalesFile.ReadLine()
        CNS = Split(CodeNSales, vbTab) 'Parse the record into two fields
```

```
        ProductCode = CNS(0) 'The first field is product code
        Sales = CDbl(CNS(1)) 'The second field is sales
        ' Search for the product code in Product codes array
        ' Assume this product code is not in array;
        ' in that case the product position will be the same as the current count
        ProductPos = ProductCount
        For I = 0 To ProductCount - 1
            If ProductCode = ProductCodes(I) Then
                ' Product code is found, get this position and exit the loop
                ProductPos = I
                Exit For
            End If
        Next I
        If ProductPos = ProductCount Then
            ' No change in product pos as initially set;
            ' this means the product code is not found
            ' Update count and add the code to array
            ProductCount += 1 'Increase count by 1
            ' and keep the current code in the array
            ProductCodes(ProductPos) = ProductCode
        End If
        ' Add current sales to corresponding total
        SalesByProduct(ProductPos) += Sales
    Loop
    ' Step 3: Show results in the list box
    For I = 0 To ProductCount - 1
        lstSales.Items.Add(ProductCodes(I) & vbTab & SalesByProduct(I))
    Next I
End Sub
```

The procedure begins with associating the sales file with the StreamReader, SalesFile; then, in the outer loop, the StreamReader reads a line into the variable, CodeNSales, which is parsed into an array CNS with the delimiter, vbTab. The first element is assigned to ProductCode; the second element is converted to Sales. (Of course, you can use the old trick of using the InStr function to find the position for vbTab in CodeNSales and then obtain the product code and sales. Here you use a different approach with an interest to show the use of the Split function.) Next comes the routine to search for the product code and set the value for the product position, including handling the addition of a new product code; then the statement:

```
    SalesByProduct(ProductPos) += Sales
```

adds the current sales to the total sales at the position ProductPos. If the current product code is D (refer to the illustration at the beginning of this example), ProductPos should have a value of 0. Thus, the statement will in effect appear as follows:

```
    SalesByProduct(0) + = Sales
```

Consequently, the current sales will be added to the first element, which corresponds to the position of Product D, of the array SalesByProduct. In sum, this statement tells the computer to add current sales to the total sales of the current product code.

Finally, step 3 in the code uses a For loop to populate the list box with the accumulated sales by product using the Items.Add method. If the Sorted property of the list box is set to True, the list box will show the sales in ascending order by the product code.

As stated at the beginning of this example, other algorithms can be used to solve the same problem. For example, you can sort the file by product code first; then, the accumulation of sales by product basically involves going through the sorted data arrays in sequential order, and adding the sales with the same product code together. The implementation of this algorithm is left to you.

Summary

- You can declare a one-dimensional array variable with a Dim statement or any scope modifier such as Private, Friend, or Public. The variable should be followed by a pair of parentheses. If an expression is placed inside the pair of parentheses, it must be evaluated to a positive integer. The value inside the parentheses gives the upper bound of the array.
- All array indices begin with zero.
- An element of an array can be referenced by an integer expression in the pair of parentheses following the variable.
- The scope and lifetime of an array are determined by exactly the same rules as the scalar variable.
- An array's length can be changed by the used of the ReDim statement.
- An array of values can be constructed by enclosing the elements with a pair of braces. Commas must be inserted between elements.
- When a procedure expects an array parameter, the parameter name must be followed by a pair of parentheses. If the parameter is expected to be two dimensional, the pair of parentheses should also enclose a comma.
- The UBound function gives the upper bound of the array specified in the parameter.
- Arrays have many uses. This chapter gives examples that show the following:
    - Simplified selection
    - Table lookup
    - Tracking random occurrence
    - Simulation
    - Random sampling without replacement
- You can create control arrays by code at runtime. However, you can only declare their types with the Dim statement or any scope modifiers. Each element in the control array can only be created with the New keyword in an assignment.
- Two-dimensional arrays can be used to handle tables, matrices, and game boards.
- To preserve the contents of an array while it is being re-dimensioned (with the ReDim statement), use the Preserve keyword. To release the content of an array so that it becomes Nothing, use the Erase statement.
- Arrays are objects. As such, they have inherent properties and methods. The Rank and Length properties as well as the CopyTo, Clear, Sort, Reverse, and BinarySearch methods were discussed.
- Although arrays have many uses, beginners should be aware that not all problems call for the use of arrays. Arrays use more memory than scalar variables. Be sure that you use an array only when its use is called for.

Explore and Discover

8-1. Relative Position of a Named Constant for Use. Try each of the following pairs of statements separately in the general declaration section in a new form; then run the project:

```
1.    Const HUNDRED As Integer = 100
      Dim A(HUNDRED) As Integer
2.    Dim A(HUNDRED) As Integer
      Const HUNDRED As Integer = 100
```

Does the computer accept both? (Yes. Also, use the UBound function to check the upper bound of A for both.) What do you infer as the rule concerning the use of a named constant in the declaration of an array?

8-2. Use of a Variable for Array Declaration. Place the following code in the Form Click event:

```
    Dim N as Integer
    N = 10
    Dim A(N) As Integer
    Console.WriteLine(LBound(A) & " " & UBound(A))
```

Run the project and click the form. What result do you see? As long as the value of a variable to define the size of an array is known before the Dim statement, the statement is executed properly.

8-3. Static Arrays. Can you declare a static array in a procedure? If so, how is it different? You can try the following code to find out.

```
Private Sub btnTest_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
    Dim I As Integer
    Dim A(5) As Integer
    Static B(5) As Integer
    For I = 0 To 5
        A(I) += + 1
        B(I) +=  + 1
    Next I
    Console.Write("A: ")
    For I = 0 To 5
        Console.Write(A(I))
    Next I
    Console.WriteLine()
    Console.Write("B: ")
    For I = 0 To 5
        Console.Write(B(I))
    Next I
    Console.WriteLine()
End Sub
```

Run the project, and click the btnTest button several times. Do you see any differences between the values of elements in arrays A and in B?

8-4. Use of ReDim. Create a new project. Type the following code in the Form Click event procedure:

```
    Dim N As Integer
    Dim A() As Integer
    N = 10
    ReDim A(N) As Integer
    Console.writeline(LBound(A) & " " & UBound(A))
```

Is any line highlighted for syntax error? The ReDim statement does not allow any redefinition of data type, even if there is no change. Remove As Integer from the ReDim statement. Run the project and then click the form. Is it working properly now?

8-5. Use of ReDim. (continued from exercise 8-4). Comment out the Dim A statement from the event procedure in exercise 8-4. Is any line highlighted for syntax error? The variable in a ReDim statement must be declared in a Dim statement first.

8-6. Using Functions to Declare Subscript Range. Place the following code in the Form Click event.

```
    Dim A(100) As Integer
    Dim B() As Long
    ReDim B(UBound(A))
    sole.WriteLine(UBound(B))
```

Run the project and then click the form. What do you see in the immediate window? What do you learn from this exercise? The range of an array subscript in a ReDim statement can be any expression as long as it can be resolved to an integer.

8-7. Content of an Array After ReDim. Draw a button on a new form. Name it btnReDim and set its Text property to Re Dim; then enter the following code:

```
Dim A() As Integer
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim I As Integer
    ReDim A(10)
    For I = 0 To 10
        A(I) = CInt(Rnd * 100)
    Next I
    ShowMe(A)
End Sub
Private Sub btnReDim_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnReDim.Click
    ReDim A(12)
    ShowMe(A)
End Sub
Sub ShowMe(X() As Integer)
    Dim I As Integer
    For I = 0 To UBound(X)
        Console.Write(X(I) & " ")
    Next I
    Console.WriteLine()
End Sub
```

Run the project, and click the form. What numbers do you see in the immediate window? Now click the ReDim button. What numbers do you see in the immediate window?

8-8. Preserving Content of an Array After ReDim. Change the btnReDim Click event procedure in exercise 8-7 as shown in the following code. Add the Preserve key word to the ReDim statement and run the project again. Click the form. What numbers do you see? Click the Re Dim button. What numbers do you see? What difference does the keyword Preserve make?

```
Private Sub btnReDim_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnReDim.Click
    ReDim Preserve A(12)
    ShowMe(A)
End Sub
```

8-9. Changing the Number of Subscripts and the Preserve Keyword. Change the event procedure in exercise 8-8 to the following:

```
Private Sub btnReDim_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnReDim.Click
    ReDim Preserve A(4, 4)
End Sub
```

What do you see? The variable A as initially declared is expected to be a one-dimensional array, but the ReDim statement now is trying to change its rank, which is not allowed. To make this statement work, you need to change the declaration for A to:

```
    Dim A(,) As Integer
```

(Notice the comma inside the pair of parentheses.) This will cause errors in all other procedures in which A was coded as a one-dimensional array. The following exercise explores the use of two-dimensional arrays.

8-10. ReDim and Preserve with a Multidimensional Array. Draw a button on a new form. Name it btnReDim and then set its Text property to Re Dim. Enter the following code:

```
Dim A(,) As Integer 'Note the presence of a  comma
Private Sub Form1_Click(ByVal sender As System.Object,  ByVal e As
  System.EventArgs) Handles MyBase.Click
    ReDim A(10, 10)
End Sub
Private Sub btnReDim_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnReDim.Click
    ReDim A( 4, 10)
End Sub
```

Run the project. Click the form and then click the Re Dim button. Did you get an error message? Now change the ReDim statement in the btnReDim Click event procedure to the following:

```
    ReDim Preserve A(4, 10)
```

Run the project. Click the form and then click the Re Dim button. Again, did you get an error message? Now change this ReDim statement to:

```
    ReDim Preserve A(10, 4)
```

Repeat the experiment: Run the project, click the form, and click the Re Dim button. Did you experience any problems this time? You can change the second (last) subscript, but not the first (others) in a ReDim statement with the Preserve keyword.

8-11. Returning an Array from a Function. Can a function return an array? Yes. The following function will return a sequence of 1, 2, 3, ... N in an array.

```
Function Sequence(ByVal N As Integer) As Integer()
    Dim A() As Integer
    Dim I As Integer
    ReDim A(N)
    For I = 0 To UBound(A)
        A(I) = I
    Next I
    Return(A)
End Function
```

Note how the function is declared. Note also the last line within the procedure. The entire array A is returned with the Return statement.

To test the function, enter the preceding code in the code window. Also enter the following in Form Click event procedure.

```
    Dim B() As Integer
    Dim I As Integer
    B = Sequence(5) 'Generate a sequence of 1, 2, ...5
    For I = 0 To UBound(B)
        Console.WriteLine(B(I))
    Next I
```

Note how B is declared and assigned. Run the project, and click the form. What do you see?

8-12. What do you really get? Place the following code in the Form click procedure. Run the project, click the form, and examine the results. Do you see anything puzzling? Why do both X(2) and Y(2) have 1000? Because arrays are objects, when one array is assigned another array, its pointer is replaced by that of the other array. The two arrays in effect are pointing to the same area and thus are the same.

```
Dim X() As Integer
Dim Y() As Integer
Dim I As Integer
Y = New Integer() {1, 2, 3, 4, 5}
X = Y
X(2) = 1000
For I = 0 To 4
    Console.Write(X(I) & " ")
Next I
Console.WriteLine()
For I = 0 To 4
    Console.Write(Y(I) & " ")
Next I
Console.WriteLine()
```

8-13. Another Way to Copy. In the text, elements in one array can be copied to another, using the CopyTo method. You can also use the Copy method. Change the assignment statement in exercise 8-12 as follows. Run the program, click the form, and examine the results. Are they different from those in exercise 8-12?

```
'X = Y 'Comment out this statement
Dim X(UBound(Y))
Array.Copy(Y, X, 5) 'This is the way to copy
```

8-14. Why Add? Enter the following code in the Form Click event:

```
Dim MyLabel As New Label()
With MyLabel
    .Location = New Point(10, 20)
    .Size = New Size(90, 25)
    .Text = "This is my label"
End With
```

Run the project and click the form. Do you see the label? Now add the following statement at the end:

```
Me.Controls.Add(MyLabel)
```

Run the project and click the form again. This time, do you see the label? A control must be added to the form's Controls collection before it can appear in the form.

8-15. Rank of An Array. Enter the following code:

```
Dim A() As Integer ={1,3,5,7}
```

Do you see any hint of error by the compiler? Everything should be fine. Now, change A() to A(,). What do you see? The compiler is expecting a two-dimensional array, but the pair of braces initializes a one-dimensional array.

8-16. Rank of An Array. (continued from exercise 8-15) Enter the following code:

```
Dim A(,) As Integer ={{1,3},{5,7}}
```

Do you see any hint of error? To construct a higher rank array, use inner braces. The innermost pair of braces contains the values for the rightmost dimension. Verify the assertion with the following routine:

```
Dim I, J As Integer
For I = 0 To 1
    For J = 0 To 1
```

```
            Console.Write(A(I, J))
        Next J
        Console.WriteLine()
    Next I
```

Exercises

8-17. Random Roll Calls with Repetition. Create a text file with a class roll of 15 names, one line each. Use Notepad, and save the result. Draw a button on a new form. Name the button btnCall and set its Text property to Call. Develop the code to perform the following:

- As soon as the program starts, populate an array named Students with the names in the file.
- When the user clicks the Call button, a student's name drawn randomly from the array will be displayed with the MsgBox. The name of each student can be repeated.

8-18. Random Roll Calls Without Repetition. Modify the program in exercise 8-17 so that the following can be handled properly:

- The number of students in the file may change, and the array Students will accommodate a huge class of 400 students.
- When the user clicks the Call button, a student's name drawn randomly from the array will be displayed with the MsgBox. The name of each student cannot be repeated.

(*Hint:* This is a sampling-without-replacement problem.)

8-19. Computing the Value of a Polynomial Function. Write a function to compute the value of a polynomial function of any degree. The header should appear as follows:

```
Function Polynomial(A() As Double, X As Double) as Double
```

where A() is an array containing the coefficients of the polynomial function and X is the value of the variable in the polynomial function.
For example, given the following function:

$$f(x) = 3x^4 + 10x^3 - 12x^2 + 5x - 20$$

the array parameter A should contain 3, 10, -12, 5, and -20.

The function should return the value of the formula given any value of x. Test the result using 1 and 2 individually as the value of x.

8-20. Computing Average, Max, and Min. Draw two buttons on a new form. Name them btnPopulate and btnCompute. Give each its proper text. Provide the code to handle the following:

- When the Populate button is clicked, the array Scores is populated with random integers ranging from 65 to 100. The array should have 60 elements.
- When the Compute button is clicked, the average, high, and low scores are computed and displayed in a message box.

8-21. Finding the Two Largest Numbers. Modify exercise 8-20 so that when the Compute button is clicked, the two highest scores are displayed in a message box.

8-22. Finding the n (n <=5) Largest Numbers. Modify the program in exercise8-21 so that when the Compute button is clicked, the n highest scores are displayed in a message box. The number n is specified by the user in a text box named txtNumber; add this control to the form. (*Hint:* Use the Array object's Sort and Reverse methods.)

8-23. Which Checks Are Still Outstanding? Suppose you have written 100 checks, numbered 1001 through 1100, in a month. When you receive the monthly statement from the bank, you keep in a file the check numbers that have been returned. Design a project that will show you the checks that are still outstanding after your program reads through the checks returned file.

8-24. Who Has Not Yet Checked In? In an exclusive club meeting, all 200 members are all expected to attend. As each member checks in, the member reports his/her membership number (501 to 700). Assume the check-in procedure is computerized. The user is to enter a membership number and then click the Check In button. At any time, you can request the list of members (membership numbers) who are still outstanding. The outstanding list should be populated in a list box. Note that a member is not allowed to check in twice. (Your program should detect this error.)

8-25. Counting Frequency of Random Numbers Ranging from 0 to 99. Modify the frequency-counting example in the subsection, "Tracking Random Occurrences" of section 8.2 so that the random numbers involved are 0 through 99, instead of 0 through 9. Display the results in a list box.

8-26. How Many Times Did Each Patient Visit the Clinic Last Year? A clinic maintains a file recording all patient visits. The file is arranged in ascending order by date (first field of each record). Other fields in each record include the patient number (second field) and the diagnosis code. The physician would like to know how many times each patient visited the clinic last year. The patient number ranges from 20,001 to 25,000.

Develop a project that can provide this information. Display the results in a list box. Each line in the list box will give the patient number and the number of times the patient visited the clinic. (*Note:* For simplicity, instead of reading data from a file, use random numbers in the range of 20,001 and 25,000 to represent the patient numbers. Assume the file contains 40,000 patient visit records.)

8-27. Searching for a Number in an Array and Returning the Position. Draw a text box and a button on a new form. Name the text box txtNumber and the button btnSearch. Set the button's Text property to Search; then provide the code to accomplish the following:

- When the project starts, populate an array (named EmployeeNos) of 10,000 elements with random numbers ranging from 10,000 to 50,000. Each number in the array should be unique.
- Write a function to search for a number in an array. It should return the position of the array in which the number is found; otherwise, it should return a value of –1. The header of the function should appear as follows:

```
Function FindPos(Numbers() As Integer, SearchKey As Integer) As Integer
```

- When the user clicks the button, the computer searches for the number entered in the text box in the array by using the FindPos function, and displays a message similar to the following if the number is found:

```
Employee no. 10538 is found in position 92 of the array.
```

- If the number is not found, the message should appear as follows:

```
Employee no. 30348 is not found in the array.
```

8-28. Lookup: Your Grade Given a Score. Suppose a teacher has the following grading system:

Score Range	Grade
88 and over	A
75–87	B

60–74	C
Below 60	D

Instead of using the Select Case structure, you would rather use two arrays to handle this problem. Develop a project that will display the grade in a message box when the user enters a score in a text box named txtScore and clicks a button named btnGrade. (*Hint:* You need to keep only the lower score of each grade to decide the grade.)

8-29. Probability of at Least Two Persons with the Same Birthday in a Gathering with 34 People. What is the probability that in a group of 34 people, at least two people will have the same birthday? Develop a project to simulate (approximate) this probability. You can set up an array with 365 elements, each representing a day in the year. Generate 34 random days in the year (with a value range from 1 to 365) and then use the array to keep track of how many birthdays fall in each day. If any day (element in the array) has more than 1 count, you have found at least one incident of the same birthday in this experiment.

If you do this experiment 1,000 times, and count the number of times you find the occurrence of the same birthday, you can approximate the probability by dividing the count by 1,000.

8-30. Probability of at Least n Persons with the Same Birthday in a Gathering of p People. Modify the program in exercise 8-29 so that the number of people having the same birthday and the number of people in a party can be any number specified by the user.

8-31. Probability of Exactly n (n = 2 to p) Persons with the Same Birthday in a Gathering of p People. Consider the experiment in exercise 8-30. How many times do 2, 3, 4, ... p people have the same birthdays? (*Hint:* Use another array to count the frequency of the counts that are greater than 1.)

8-32. Chances of Winning. Assuming that as a blackjack player, you have 49% probability of winning a hand. Develop a project to allow yourself to approximate the probability of winning exactly 30, 31, 32, ... or more hands out of 50. Use the Monte Carlo simulation method illustrated in the "Simulation" subsection of 8.2. (The results should be returned as an array.)

8-33. Good Product Units from a Sequential Process. Suppose a product takes two processes to complete. A batch of raw materials is placed at the beginning of the first process. At the end, units are inspected for quality. Good units are then placed into the second process. At the end, another inspection is performed. Assuming the probabilities of obtaining good units are 97% and 98% for processes 1 and 2, respectively, develop a project to compute the probability of obtaining at least g (such as 1,000) good units when r (such as 1,100) units of raw materials are placed in production at the beginning of process 1. (Show separate probability for each good unit count; such as 1000, 1001, ... and so on.)

8-34. Parse a Text String into Words. Assume that a text string contains many words, all separated by blank spaces. There can be one or more spaces between two words. Write a Function procedure that will parse the string and extract all words in the string and then return them in an array. The text string will be passed to this function as a parameter. See exercise 8-11 for a function returning an array. (*Note:* There is a function, Split, that can perform this requirement. Do not use that function. Create one from scratch for yourself.)

8-35. List All Prime Numbers up to 60,000. Write a Sub procedure that will list all the prime numbers from 1 to 60,000. A prime number is one that can be evenly divided only by 1 or itself. The first smallest prime numbers are 1, 2, 3, 5, and 7. (*Hint:* Identify these numbers from the smallest. Use an array to store all the identified prime numbers. Test whether a number is a prime number by checking the remainder of this number divided by all the identified prime numbers. If a number cannot be evenly divided by any prime numbers that are less than the square root of this number, this number is a prime number.)
Test your program as follows:

- Draw a button and a list box on a new form.
- When the user clicks the button, the event procedure should call your prime number procedure to obtain the results. Populate the results in the list box.

8-36. The Sequence Function. Write a function that will return an integer array representing the specified sequence of numbers. The function should allow the following different kinds of specifications (all parameters are of the integer type):

Specification	Returns
Sequence(n)	{1, 2, 3,… N}
Sequence(3, 2, 4)	{3, 5, 7, 9}

(Hint: Modify exercise 8-11. Use optional parameters or overload the function.)

8-37. Load Text Boxes, Obtain Cash Flow, and Compute the Net Present Value of an Investment Project. Develop a project to calculate the net present value of an investment project. When your program starts, the form has two text boxes with proper labels and one button. The text boxes expect the user to enter the required rate of return for the project and the expected life of the project. As soon as the user finishes entering the expected life by pressing the Enter key, your program should show additional text boxes with proper labels for the user to enter the cash inflow for each period of the expected life. Note the total number of text boxes should be one more than the number of periods so that the user can also enter the amount of initial investment. When the user clicks the Compute button, your program computes and displays the net present value in a text box, whose ReadOnly property should be set to True.

8-38. Computing the Internal Rate of Return for a Project. For a typical simple project that has a negative initial cash flow (investment) followed by a series of positive cash flows, there is a rate that makes the net present value of the project equal to zero. This rate is called the internal rate of return. You are to design a project that will compute the internal rate of return for a project based on the same setting as given in exercise 8-37. (Change the label ”Required rate of return” to ”Initial guess”; or delete this label and the related text box.)

8-39. Revisiting the Tuition Computation Problem. Consider the tuition computation problem presented in Chapter 5 again. Design a project to carry out the “computation” using an array to hold the tuition data. The interface should appear the same as in the second approach.

8-40. Weight Lookup Program Continued. The following table shows the normal weight for women. Modify the weight lookup program presented in Section 8.2 so that you can also inquire about women’s normal weight, given the height. You will need to add two radio buttons for the user to indicate the gender of the person of interest. (*Hint:* This can be handled several ways. One way is to use two tables

to store the weights: one for men and another for women. The table to look up the weight will then depend on which of the radio buttons for gender is clicked.)

Height	Small Frame	Medium Frame	Large Frame
4’10”	92–98	96–101	104–119
4’11”	94–101	98–110	106–122
5’	96–104	101–113	109–125
5’1”	99–107	104–116	112–128
5’2”	102–110	107–119	115–131
5’3”	105–113	110–122	118–134
5’4”	108–116	113–126	121–138
5’5”	111–119	116–130	125–142
5’6”	114–123	120–135	129–146
5’7”	118–127	124–139	133–150
5’8”	122–131	128–143	137–154
5’9”	126–135	132–147	141–158
5’10”	130–140	136–151	145–163
5’11”	134–144	140–155	149–168
6”	138–148	144–159	153–173

8-41. Binary Search for a Match. It is well known that when searching a sorted array, the binary search algorithm is much more efficient than the sequential search. Write a binary search function with the following header:

```
Function BinarySearch(X() As Long, SK As Long) as Integer
```

where X() is a sorted array and SK is the value (search key) to search for.
The function will return the position of the array at which SK is found. If SK is not found in the array, the function will return a value of –1. Compare your results with the BinarySearch method of the Array object.

Projects

8-42. Demand During Lead Time with Random Lead Time Days and Random Daily Demand. The demand for a product during a lead time period (between the day you place a purchase order and the day you actually receive the goods) is determined by two factors: the lead time in days, and the demand in each of these days. The lead time period (in days) is subject to a random distribution, as is the demand each day during this period. Suppose the lead time period has a probability distribution as follows:

Lead Time (No. of Days)	Probability	Cumulative Probability
13	.3	.3
14	.5	.8
15	.2	1.0

Also assume the daily demand for the product is subject to the following probability distribution:

Daily Demand in Units	Probability	Cumulative Probability
30	.2	.2
31	.6	.8

32	.2	1.0

If you inspect the two tables, you should see that the total demand during each lead time period can vary from 390 (13 x 30) to 480 (15 x 32), subject to a joint probability distribution. Develop a project to simulate (approximate) the probability of demand during lead time for this company. Your output should appear as follows:

Demand During a Lead Time Period	Probability
390	0
391	.01
392	.
.	.
.	.
480	0

(*Hint:* Consider the simulation for the actual demand for a day. If you draw a random number, and it turns out to be .5, you can assert that the demand for that day is 31. You draw this conclusion because the number falls in the cumulative probability that includes 31. Similarly, if the random number is .9, you can claim the demand for that day is 32. To simulate daily demand, you can use two arrays: one holding the cumulative probability, and another holding the daily demand. After you obtain a random number, you can search the cumulative probability array and then identify the position, which is used to obtain the actual daily demand from the daily demand array. If you repeat this simulation for a lead time period and add all the daily demand, you will come up with the demand for a lead time period. But how do you know the days of a lead time period? You use the same simulation method, but a different pair of arrays. Refer to the first table for needed data. If you repeat the simulation for 1,000 lead time periods, you will be able to count the number of times each demand quantity occurs.)

8-43. Finding Connected Groups in a Go Game Board. A Go game board has a dimension of 19 x 19 lines. Two players take turns in placing their stones (black versus white) on the intersections. The objective is to occupy as big a territory as possible. A group of isolated stones of one player is captured if it is completely surrounded by the opponent's stones without empty intersections inside the group. One aspect of programming the game is to identify the groups of stones that are connected. Two stones of the same side (color) are connected if both are on the same line (vertical or horizontal; that is, on the row or column) and next to each other on the other dimension. For example, the two stones at 9,10 and 9,11 are connected because they are on the same row and are next to each other by column. Many stones placed on the board can be connected to one another vertically and horizontally to form a group.

The Challenge: When your program starts, the user will enter the number of stones to be placed on the board. Use a text box for this purpose. Your program will proceed to simulate where these stones are placed on the board. The next step is the key challenge: to identify the number of connected groups (as described in the preceding paragraph) and the stones belonging to each group. To simplify the problem, assume all stones on the board are of the same color and played by the same player.

Internal Representation: Various schemes can be used to represent the status of the game board internally. For example, an intersection, Board(Row, Col) has a value zero if it is not occupied by either side and has a value of –1 if occupied by the Black side and a value of 1 if occupied by the White side. Because you are assuming only one side is playing, you can use either –1 or 1 to indicate that a stone has been placed at the intersection.

Simulating the Intersection Points Occupied (Played): You can use a pair of random numbers to simulate the intersection point at which the player places the stones. More specifically, if you declare the

Board as Board(18, 18), you will need to generate a pair of random number Row and Col, each within the range of 0 to 18. Notice that an intersection may have been occupied already. In that case, another pair of Row and Col must be generated until an empty intersection is found. Refer to the "Sampling Without Replacement" subsection in Section 8.2 for hints to identify whether a number has already been selected.

(Hint: Note that a stone may be connected to as many as four other stones, which can in turn be connected to other stones. It would be much easier to use a recursive procedure to identify the group; however, you will need to avoid double counting, which involves including an intersection that has already been included/connected. This will result in an endless connection loop. Use another array with the same dimension as the board to identify the group to which a particular intersection belongs.)

Displaying Results: Use a list box to show the result. One way to show the results is as follows:

Group 1:
1, 1
1, 2
1, 3
2, 1
3, 1
Group 2:
10, 12
11, 12
12, 12

where each pair of numbers represents the coordinate of the intersection. Enjoy your challenge!