CHAPTER 10
WORKING WITH ARRAYS AND COLLECTIONS

Imagine a mail-order company that sells products to customers from all 50 states in the United States. This company is required to charge sales tax on sales to customers in many states, and the sales tax rate varies from state to state. How do we represent these sales tax rates in a program? If we store them in variables of the kind we have seen so far, we will need 50 variables with names like NYTaxRate, NJTaxRate, and CATaxRate. Then the program will require a Select…Case or If…Then…Else statement to identify a customer's state of residence and determine the corresponding tax rate variable to use in the calculation of the sales tax. Because of the large number of variables, this statement will be very long.

Storing a set of related values, such as 50 sales tax rates, in separate variables is tedious at best. However, such situations are common. To resolve this problem, the VB programming language provides another kind of variable called an array. **An *array* is a variable with a single symbolic name that represents many different data items.** This chapter describes how to declare arrays and how to use them to solve some common processing tasks.

In additional to traditional arrays, Visual Basic .NET also provides a number of classes that manage collections of objects. A ***collection* of objects is like an array in that one collection is associated with many objects. These classes "manage" the objects in a number of different ways**. These management techniques differentiate the collection classes. This chapter covers three of these classes – the ArrayList, the Hashtable, and the SortedList collections.

Objectives

After studying this chapter you should be able to

- Explain why arrays are needed to solve many types of problems.
- Construct an array to store multiple related data values.
- Use ArrayList, Hashtable, and SortedList collections to store and process data values.

10.1 SOLVING PROBLEMS WITH ARRAYS

Since we are now introducing arrays, we need a term to describe the ordinary variables we have used prior to this point. We will use the term ***simple variable* to mean a variable that can store only one value.** We first examine a problem whose solution is very inefficient when we use simple variables. We then show how arrays improve our solution. As we solve the problem, we introduce the structure of arrays.

The Problem

A financial analyst has calculated last year's rate of return[1] for 10 different companies and would like a program to display the names of those companies whose rate of return exceeded the average of the group. What processing steps must we perform to display this information?

The Solution Using Simple Variables

Using the programming techniques we've employed in earlier programs, we might begin to solve this problem by writing the following pseudocode.

[1] *Rate of return* is a measure of profitability.

Compute the average rate of return for the group of ten companies
Display the names of companies whose rate of return exceeds the average

We first have to compute the average. This requires accessing each individual company's rate of return and adding it to a cumulative total. We can refine the pseudocode for this step as follows:

For each company
    get a rate of return and a company name
    add the rate of return to the sum
Next company
Compute the average by dividing the sum by the number of companies

Once we have the average, we must go back through the rates of return, comparing each to the average. If a rate of return exceeds the average, we display the company name. The pseudocode for this step might be

For each company
    If the company's rate of return is greater than the average Then
        display the company's name
    End If
Next company

This problem requires two passes through the data: one pass to compute the average and a second to compare the average with each rate of return. There is no other way to solve this problem.

Now think about how you would convert this pseudocode to actual code. You would have to store each rate of return in a variable because each is used twice: first to compute the sum, then later to compare to the average. If the company names are input into the program at the same time as the rate of return values, then the company names will also have to be stored in variables. Thus, if we had 10 companies, we'd need 20 variables.

Let's look at some code, assuming that we have only three companies instead of ten. The following code might be found in a Click event procedure for a button captioned "Find Best Companies".

```
Private Sub cmdFindBestCompanies_Click(…)
    Dim CompAReturn As Decimal, CompAName As String
    Dim CompBReturn As Decimal, CompBName As String
    Dim CompCReturn As Decimal, CompCName As String
    Dim Average As Decimal
    Dim Sum As Decimal
    CompAReturn = InputBox("Enter Rate of Return for Company A")
    CompAName = InputBox("Enter Company A Name")
    Sum = Sum + CompAReturn
    CompBReturn = InputBox("Enter Rate of Return for Company B")
    CompBName = InputBox("Enter Company B Name")
    Sum = Sum + CompBReturn
    CompCReturn = InputBox("Enter Rate of Return for Company C")
    CompCName = InputBox("Enter Company C Name")
    Sum = Sum + CompCReturn
```

```
        Average = Sum / 3
        If CompAReturn > Average Then
            MsgBox CompAName & " exceeds the average rate of return"
        End If
        If CompBReturn > Average Then
            MsgBox CompBName & " exceeds the average rate of return"
        End If
        If CompCReturn > Average Then
            MsgBox CompCName & " exceeds the average rate of return"
        End If
    End Sub
```

This solution approach works, but it is not a good general solution. Imagine if you had 30, or 300, or 30,000 companies. You would be writing code for the rest of your life. There must be a better way.

You can spot two places in the code where statement blocks are "nearly repeated" three times. We would like to use loops but are prevented from doing so because each value is stored in a different variable and each variable has a different name. For example, we'd like to have something like the following loop:

```
    For K = 1 To 3
        CompAReturn = InputBox("Enter Rate of Return for Company A")
        CompAName = InputBox("Enter Company A Name")
        Sum = Sum + CompAReturn
    Next K
```

Although the sum would be computed correctly, the problem with this loop is that there is no way to store and preserve the individual rate of return and name for each company. Each iteration of the loop updates the value in `CompAReturn`, destroying the previous rate of return as it obtains the next value needed to accumulate the sum.

The Structure of an Array

What we need is a different kind of variable—one that has a single name but can store more than a single value. This new kind of variable is known as an array. Figure 10.1 depicts an array named `RateOfReturn` that stores five values. **Each *element* of an array is like a simple variable**. We can store a different company's rate of return in each element. The entire set of elements shares the name `RateOfReturn`.

<insert old figure 9.1 – start numbering at 0>

Figure 10.1 An array

The elements in an array are numbered, starting at zero, as shown in Figure 10.1. To access an individual element, we append its number to the array name. For example, in the array depicted in Figure 10.1, the third element is referred to as `RateOfReturn(2)`[2], and it currently holds the value 0.10. The **element number is called the *subscript***. **An alternative term for an element number is the *index value***. For this reason, **another term that is sometimes used for an array is *subscripted* or *indexed variable***. The array depicted in Figure 10.1 is a ***one-dimensional array*** **because each element is referenced**

[2] Note that since the array numbering system starts at zero, when we speak of the n^{th} element of the array, it will be numbered (n-1), e.g., the 4^{th} element of the array is element number 3.

using a single subscript value. You can also create **two-, three-, and higher-dimensional arrays, called *multidimensional arrays***.

The syntax for referencing a specific element in a one-dimensional array is

ArrayName(SubscriptValue)

where *SubscriptValue* is a numeric expression.

The Solution Using Arrays

If we store the rates of return in an array like the one shown in Figure 10.1, we can design a better solution to the problem. A procedure can process the contents of an array element just as it processes any other variable. For example, the following statement adds the values in elements 0 and 1 of the array.

```
Sum = RateOfReturn(0) + RateOfReturn(1)
```

The result stored in the variable `Sum` is 0.20. However, using constants as subscripts this way is almost never done. The reason is simple: using constants as subscripts in your code is no different than using simple variables. That is, the constant fixes the reference to a specific element just as the name of a simple variable refers to a specific storage location.

If we don't use constants as subscripts, then what do we use? The answer is a numeric variable, or a numeric expression containing a variable. For example, the following statement, in which `K` is an Integer variable, is typical of statements that refer to array elements.

```
Sum = Sum + RateOfReturn(K)
```

Which element in the array does this refer to? You don't know unless you know the value of the subscript `K`. This is very important. If `K` holds the value 2, then the statement above refers to the third element. If K holds `3`, then the very same statement refers to the fourth element. Thus, we can write a For…Next loop that processes each array element in turn by using the loop's counter variable as the array subscript, as follows:

```
For K = 0 To 4
        Sum = Sum + RateOfReturn(K)
Next K
```

Figure 10.2 illustrates the execution of this code.

If the array had 5000 elements instead of 5, what change would you have to make to this code? You would simply have to change the For statement so that it starts at 0 and ends at 4999.

```
For K = 0 To 4999
```

A drawback of our first solution to this problem was that the number of lines of code would increase proportionately if the number of companies that we needed to analyze increased. However, with the array approach shown in Figure 10.2, the number of lines of code is independent of the number of companies.

<insert old figure 10.2 – change indexes 0 to 4>

Figure 10.2 Summing the elements of an array using a For…Next loop

The number of lines of code you write should not have to be increased in proportion to the number of items to be processed. If you find that this is happening, you probably need to use an array instead of simple variables.

To make the loop

```
For K = 0 To 4
```

```
        Sum = Sum + RateOfReturn(K)
    Next K
```

even more useful, it would be a good idea to not restrict it to processing only five elements. How can we remove this restriction? Consider the following:

```
    For K = 0 To NumOfElements - 1
        Sum = Sum + RateOfReturn(K)
    Next K
```

Now we can control the number of elements included in the summing process simply by setting the value of the variable NumOfElements prior to the For statement. Again note that since the subscripts start at zero, the largest subscript value will be one less than the number of elements, e.g., if there are 10 elements, the range of the subscripts is 0…9.

Let's return to our problem of finding the average rate of return and displaying the company names that exceed the average. We'll add one more array to store the company names, as shown in Figure 10.3.

<insert of figure 9.3 – change subscripts 0…4>

Figure 10.3 Arrays for rate of return problem

Observe that the elements of the two arrays are coordinated. That is, the rate of return in element number 0 of the RateOfReturn array corresponds to the company name in element number 0 of the CompanyName array. This technique is often used when the data types of the two arrays are different; in the present case, RateOfReturn holds numeric data (Decimal, most likely) and CompanyName holds string data. When the data types of the arrays are the same, an alternative approach is to use a two-dimensional array, which we will discuss shortly.

Consider the following code segment for the rate of return problem. It assumes that the two arrays already exist and the number of companies has already been stored in the variable NumberOfCcompanies.

```
    For K = 1 To NumberOfCcompanies - 1
        Sum = Sum + RateOfReturn(K)
    Next K
    Average = Sum / NumberOfCompanies
    For K = 1 To NumberOfCompanies - 1
        If RateOfReturn(K) > Average Then
            MsgBox CompanyName(K) & " exceeds the average"
        End If
    Next K
```

Convince yourself that this solution will work for any number of companies as long as the arrays have been created and correctly initialized with values.

Storing Data in Arrays versus Databases

Could you solve this problem without arrays if the data were stored in a database? If so, what is the advantage of using arrays?

Reading the data from a database is an option. If Rate of Return and Company Name were two fields in a database table, then you could read every record from the table and compute the sum of the rates of return. You could then compute the average. Finally, you could move back to the first record of the table and read every record again to compare each rate of return to the average.

Arrays are variables and, like all variables, they exist in random access memory (RAM). Data stored in RAM are accessed quickly, typically in nanoseconds (billionths of a second). But storage for arrays is limited because RAM is relatively expensive and is needed for other uses such as storage of the operating system.

Databases are stored on disk. The computer must transfer database records to RAM in order to process them and, as a result, access to database records is relatively slow—it is typically measured in milliseconds (thousandths of a second). But storage for databases is less constrained because disk storage is considerably less expensive and a computer typically has much more disk storage than RAM storage.

When the same data items are accessed multiple times during processing, using arrays can produce a superior program. The factors that must be considered when deciding which approach to use include (1) execution speed, (2) the amount of data that needs to be processed, and (3) the clarity of the code that accomplishes the task. If the amount of data were very large (too large for RAM), then directly accessing the database would be the only feasible solution even though it might execute more slowly. If the amount of data is relatively small, then reading the data from the database and storing it in an array for processing can improve execution speed. The speed advantage of the array solution will be greater the more times the data are accessed during processing.

Finally, as RAM gets less expensive, the distinction between databases and arrays gets less clear. Database technology tries to store as much of the data being processed in RAM as it can so as RAM capacity increases, the database technology uses high-speed RAM just as arrays do. If speed is a critical factor, you would store data in RAM regardless of whether it is managed by a database management system or is stored in an array.

Multidimensional Arrays

The number of dimensions an array has is called its *dimensionality*. A one-dimensional array uses one subscript to refer to an element. The array `RateOfReturn` is a one-dimensional array. One-dimensional arrays are useful when the data to be stored in the array are similar to a list of numbers or a list of names.

A two-dimensional array uses two subscripts to refer to a single element and is sometimes called a *matrix* or *table*. Figure 10.4 shows a two-dimensional array. All 12 elements are collectively referred to by the single name `Quantity`. Supplying the values of the specific row and column, the subscripts, identifies an individual element. For example, the reference Quantity(3, 1) refers to the element in row number 3 and column number 1 (which holds the number 62 in this case). The first value between the parentheses is the row number and the second is the column number. This is easy to get mixed up; for example, does `Quantity(1, 2)` refer to the value 13 or the value 21? It is row number 1, column number 2, not column number 1, row number 2 so it refers to the value 13.

The general syntax for referencing a specific element in a two-dimensional array is

ArrayName(RowSubscriptValue, ColumnSubscriptValue)

where *RowSubscriptValue* and *ColumnSubscriptValue* are numeric expressions.

For what purpose might we use a two-dimensional array like the one in Figure 10.4? Well, the rows might represent products (four products) and the columns might represent warehouses (three warehouses). Then the values stored in the array could

represent the inventory count of a particular product in a particular warehouse (the quantity of product number 3 in warehouse number 1 is 62).

<insert old figure 9.4 – change the index values>

Figure 10.4 A two-dimensional array

You are not limited to two dimensions. You can create three, four, or more. However, using subscripts slows down the execution of any program. The more subscripts an array has, the slower the execution, so you should always use the minimum number of subscripts necessary to solve the problem.

Exercise 10.1. Suppose you are writing a program that will continually retrieve and update inventory counts for four products at three warehouses. Your program contains three simple Integer variables named `Stock`, `ProdNo`, and `WareNo`. `ProdNo` currently holds a valid product number (0, 1, 2, or 3) and `WareNo` currently holds a valid warehouse number (0, 1, or 2). Assume you have the array depicted in Figure 10.4. Write the statement that stores in the variable `Stock` the inventory count of the product specified by `ProdNo` at the warehouse specified by `WareNo`.

10.2 DECLARING ARRAYS

Arrays can help you solve many kinds of problems like the one discussed in Section 10.1. As with simple variables, in order to use arrays in a program you must declare them. In this section we examine how to declare arrays, and we discuss subscript bounds.

Arrays can store Integers, Strings, Decimal, or any other data type just as simple variables can. However, every element of an array must have the same data type[3]. That is, if an array is declared as type Integer, all of its elements store integers. You cannot create an array that has some elements of type Integer and some elements of type Decimal.

Use the Dim (or the Static or Public) statement to declare arrays just as you do for simple variables. But for an array, in addition to specifying the variable name and type, you must also specify the number of subscripts (number of dimensions) and their maximum values. The general syntax of the Dim statement for an array is

Dim ArrayName(subscripts) As Type

The *subscripts* portion of the definition is simply a constant for each dimension, separated by commas if there is more than one dimension. These constants define the maximum value of each subscript. For example, you can write

```
Dim RateOfReturn(100) As Decimal
Dim InventoryCount(3, 4) As Integer
```

to create the arrays depicted in Figure 10.5.

<insert old figure 9.5>

Figure 10.5 Array dimensions

The first Dim statement creates a one-dimensional array with 101 elements numbered 0 through 100. The second Dim statement creates a two-dimensional array with four rows and five columns (a total of 20 elements). Notice that the minimum value for each subscript is zero (0). Some people find it odd that the first element number is 0, not 1. After all, if you want to store information on 100 rates of return, it would make the most sense to start at element 1 and end at element 100, not start at element 0 and end at

[3] If an array is declared as type Object, then each cell can store different types, e.g., Strings and numbers. This is because the type Object can include any other type.

element 99. But with Visual Basic .NET, as well as many other languages, the first element is numbered zero.

Visual Basic .NET has a very nice feature: it will tell you when you use a subscript value that is **outside the range—called the *subscript bounds*—specified by the Dim statement**. For example, suppose the RateOfReturn array is declared with maximum subscript value 100. Suppose also that K and NumberOfCompanies are simple variables of type Integer. If the value stored in NumberOfCompanies is greater than 100, executing the loop

```
For K = 0 To NumberOfCompanies
    Sum = Sum + RateOfReturn(K)
Next K
```

will cause a run time error—a System.IndexOutOfRangeException—when K equals 101. Figure 10.6 shows the message that VISUAL BASIC .NET displays when your program exceeds the bounds of an array. You can write a Try/Catch block to handle a situation like this.

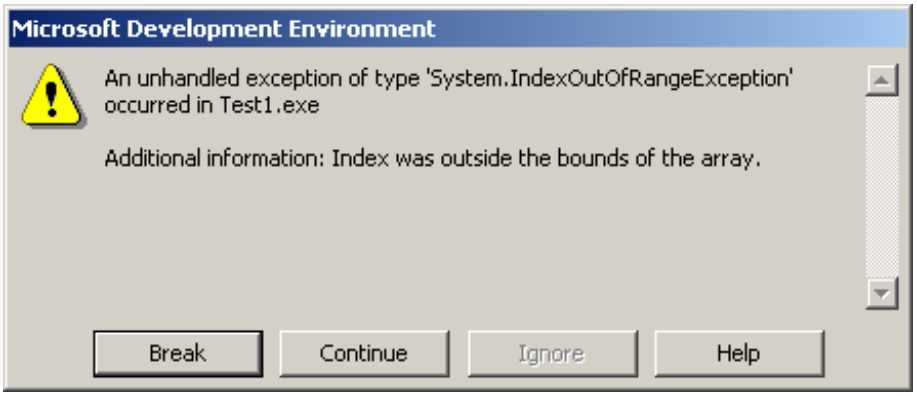

Figure 10.6 Error message for illegal subscript value

Finally, variable scope is the same for arrays as for simple variables.

Exercise 10.2. For each of the following cases, write the declaration statement that creates the specified array.

1. A one-dimensional array of type String named Cities, with 500 elements.
2. A one-dimensional of type Decimal named Price, with a maximum subscript value 98910.
3. A two-dimensional array of type Decimal named Sales, with twenty rows and six columns.

Exercise 10.3. When the user clicks on the button named btnArrayQuiz, the following code displays three message boxes. Hand-check this code, and show the numbers that appear in each message box.

```
Dim A(6) As Integer

Private Sub btnArrayQuiz_Click(...) Handles btnArrayQuiz.Click
    Dim J As Integer
    A(0) = 10
    A(1) = 7
    A(2) = -3
```

```
    A(3) = 4
    A(4) = 1
    A(5) = -4
    A(6) = 2
    DisplayArray()
    A(3) = A(2)
    A(2) = A(3)
    A(6) = A(5)
    A(5) = A(4)
    A(0) = A(1)
    DisplayArray()
    For J = 1 To 6
       A(J - 1) = A(J)
    Next J
    DisplayArray()
  End Sub

  Private Sub DisplayArray()
    Dim K As Integer
    Dim Message As String
    For K = 0 To 6
       Message = Message & " " & A(K)
    Next K
    MsgBox(Message)
  End Sub
```

Example 10.1 Populating An Array From a Database

Arrays are capable of holding large quantities of data, and it is unlikely that the user will be willing (or able) to type in a large amount of data every time a program is executed. In actual business applications the data for populating an array are typically obtained from a database instead of from the user.

In this example, the Form_Load event procedure performs the task of populating two class-level arrays named `CompanyName` and `RateOfReturn`. There is a third class-level variable named `NumRows` that stores the number of rows in the DataSet. A DataSet named DsCompInfo1 and OleDataBaseAdapter named odaCompInfo establishes a connection to a table named CompInfo in the Access database named Comp.mdb(remember to use the Server Explorer to make the connection to the Comp.mdb database). The two fields in the table are named CompName and RateOfReturn. The code for this example is in Figure 10.7. Figure 10.8 shows the user interface after the user has clicked the Review Array Contents button.

```
Dim NumRows As Integer
Dim CompName(99) As String
Dim RateOfReturn(99) As Decimal

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
   Dim Row As Integer
   DsCompInfo1.Clear() ' clear and fill the DataSet
   odaCompInfo.Fill(DsCompInfo1)
   NumRows = DsCompInfo1.CompInfo.Rows.Count ' determine number of rows
   ' test for array overflow
   If NumRows > 100 Then
      MsgBox("Not enought array storage for all rows - first 100 will be used", _
                MsgBoxStyle.Information, "Storage Problem")
      NumRows = 100
   End If
   For Row = 0 To NumRows - 1 ' fill the array
      CompName(Row) = DsCompInfo1.CompInfo.Rows.Item(Row).Item("CompName")
      RateOfReturn(Row) = DsCompInfo1.CompInfo.Rows.Item(Row).Item("RateOfReturn")
   Next
End Sub

Private Sub btnReviewArrayContents_Click(ByVal sender As System.Object, ByVal e As System.
   Dim Row As Integer, NewRow As String
   For Row = 0 To NumRows - 1
      NewRow = CompName(Row) & "  -  " & RateOfReturn(Row)
      lstArrayContents.Items.Add(NewRow)
   Next
End Sub
```

Figure 10.7 Code for Example 10.1

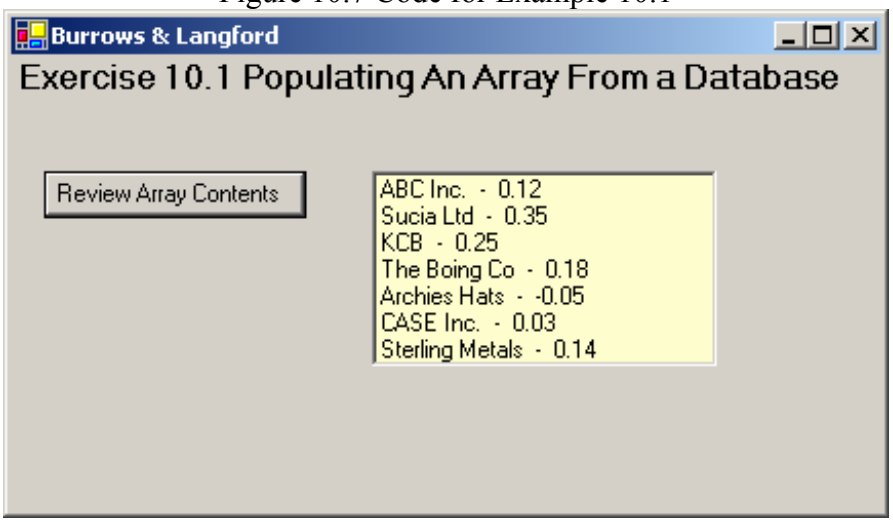

Figure 10.8 Array contents displayed when the user clicks on Review Array Contents button

Note that the arrays have fixed upper bounds: the Dim statements fix the maximum subscript value at 99 (100 total values). Since the database table could conceivably have more than 100 records, in Form1_Load the value for the number of rows is checked and, if it is too large, the user is informed and the number of rows is adjusted to the array maximum. We will show how to overcome this fixed-size restriction later in the chapter when we introduce collections.

Within the Form1_Load event, we see the loop that actually transfers the data from the DataSet into the arrays. This loop includes the following code:

```
For Row = 0 To NumRows - 1 ' fill the array
  CompName(Row) = DsCompInfo1.CompInfo.Rows.Item(Row).Item("CompName")
  RateOfReturn(Row) = DsCompInfo1.CompInfo.Rows.Item(Row).Item("RateOfReturn")
Next
```

Here we see how to access a specific value within a given row and column in the DataSet. We start by getting the DataTable associated with the DataSet (DsCompInfo1.CompInfo). Using this, we get a specific row from the Rows collection (Rows.Item(Row)). Now that we have a row, we need to get the specific value from one of the valid columns in each row (Item("CompName") or Item("RateOfReturn")). This is a long statement but you can clearly see how the various objects are used in the decomposition of the DataSet. Once the value from the DataSet is identified, it is stored in the array in the element indicated by the variable Row.

Exercise 10.4. Modify Example 10.1 so that it computes the sum of the rates of return for all companies and then displays the average (sum/count) rate of return using a MsgBox.
Exercise 10.5. Modify Example 10.1 so that it displays the name of the company with the largest rate of return value.

Example 10.2 Sequentially Searching An Unordered Array

Searching an array involves checking the values of the array's elements to see if a certain value, called the target, exists within the array. The way we go through the elements, that is, the order in which they are searched, determines the type of the search.

A *Sequential search* works by examining each element of the array individually and in turn. The search begins at the first element, proceeds to the second element, then the third, and so on, continuing until the program finds the target or comes to the end of the array. **Sequential search is sometimes also called *linear search***.

For this sequential search example we use the two-dimensional array created by the following statement:

```
Dim PriceTable(4,1) As Decimal
```

We populate the array with the values shown in Figure 10.9 in the form's load event. The values in the first column represent unique item numbers, and the values in the second column represent corresponding prices. Thus, we interpret the values in row 0 to mean that item number 324 has a price of $2.34.

Let us use our search procedure in the following way: given an item number, we want to know the corresponding price. The target for our search will be an item number, and once we locate the target we can easily retrieve its price.

When the values in an array are in ascending (smallest to largest) or descending (largest to smallest) order, we say the array is *ordered*. Otherwise, we say the array is *unordered*. The values in Figure 10.9 are unordered.

<insert old figure 9.14 – adjust index values to 0>

Figure 10.9 A two-dimensional array for testing search techniques

Our sequential search uses a simple loop.

```
For rows 0 To the last row
    If the item number in column zero of the current row equals the target Then
        we found a match; remember the row number and exit the loop
    Else
        do nothing; let loop continue with next iteration
    End If
Next row
If we reached the end of the array Then
    conclude that the item we are searching for does not exist
End If
```

We write our solution in the form of click event. The search begins in row 0 of the array. If it finds the target, it sets `FoundLoc` equal to the row number where the target is located; if the target does not exist, `FoundLoc` will still be equal to its initial value -1.

Figure 10.10 shows the code for the form's load event and the button's click event.

```
Dim PriceTable(4, 1) As Decimal

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.Event
   ' this "brute force" method of populating an array is
   ' generally only valid for demonstration purposes
   PriceTable(3, 0) = 324 : PriceTable(3, 1) = 2.34
   PriceTable(4, 0) = 423 : PriceTable(4, 1) = 3.23
   PriceTable(1, 0) = 254 : PriceTable(1, 1) = 1.95
   PriceTable(2, 0) = 321 : PriceTable(2, 1) = 0.34
   PriceTable(0, 0) = 132 : PriceTable(0, 1) = 2.25
End Sub

Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As System.E
   Dim Row As Integer
   Dim FoundLoc As Integer = -1
   Dim Target As Integer = txtItemNumber.Text
   For Row = 0 To 4
      If PriceTable(Row, 0) = Target Then
         FoundLoc = Row
         Exit For
      End If
   Next
   If FoundLoc <> -1 Then
      MsgBox("Price for item #" & Target & " is $" & PriceTable(FoundLoc, 1))
   Else
      MsgBox("Item #" & Target & " not found in table")
   End If
End Sub
```

Figure 10.10 Code for the sequential search in Example 10.2

Finally, it is possible to define and initialize an array just as one can do with simple variables. For example, consider the following two Visual Basic .NET statements:

```
Dim a() As Integer = {1, 2, 3, 4, 5}
Dim b(,) As Integer = {{1, 2}, {3, 4}, {5, 6}}
```

In the first statement, the array "a" is declared as an Integer type and stores the five values shown in the braces. The array's legal index values range from a(0) to a(4) (a total of 5 values.) In the second statement, a two-dimensional array is declared and initialized to the values shown in the braces. The inner set of braces each represents a new row in the array "b". Thus the array has three rows and two columns. Sample values from this array include b(0,0) = 1, b(1,1) = 4, and b(2,0) = 5. Note that in both examples, the actual size of the array is determined by the number of values shown after the equal sign. It is not possible to use this technique to declare an array that has more cells than the number of given values.

As you can see, arrays provide a powerful way of representing data. There are numerous algorithms that can be programmed using arrays including more complex (and faster) searches, and sorting data into ascending or descending order. In the past the developer would often write this complex and hard to debug code. However, Visual Basic .NET provides a number of specialized classes that make the job of working with array-like structures much easier and more effective for the typical developer.

In the sections that follow we look at three of these classes, called collection classes, because they are all members of the System.Collections namespace. We will cover the ArrayList, SortedList, and Hashtable collections. In addition to these three classes, there are a number of additional collection classes available within Visual Basic .NET.

Exercise 10.6. Modify the data in Example 10.2 so that they are stored in the array in order by product number. Then, using this new ordered data, modify the search so that it will be more efficient if the user enters a nonexistent product number. By "more efficient", we mean it potentially goes through the loop fewer times.

10.3 THE ARRAYLIST COLLECTION

The ***ArrayList*** **collection class is like a one-dimensional array in that it is indexed using a zero-based indexing system**. You can reference individual elements by using a subscript just like with an array. What is different is the capacity of the ArrayList – it provides a dynamic resizing ability. Its capacity (size) grows automatically as items are added. Although it is limited to one dimension, each element can store references to other objects so you could, for example, store a reference to another ArrayList object in each element. This would give you the ability to simulation a multidimensional array.

Properties

The most common properties available to objects of the ArrayList class are shown in Table 10.1

Table 10.1 Common Properties of the ArrayList Class

Property	**Action**
Capacity	*Gets or sets the number of elements that the ArrayList can contain.*
Count	*Gets the number of elements actually contained in the ArrayList.*
Item	*Gets or sets the element at the specified index.*

It is generally a good idea to leave the Capacity property alone. The ArrayList object can manage this property very well and probably better than the developer. The Count property tells you how many elements are actually being used in the ArrayList. Because the indexing starts at zero, the legal index values are 0 … (Count-1). The Item property is used to reference a specific element. For example, if you have an ArrayList named `CityName`, you would reference element numbered 2 by saying:

```
CityName.Item(2)
```

Methods

The most common methods available to objects of the ArrayList class are shown in Table 10.2.

Table 10.2 Common Methods of the ArrayList Class

Method	**Behavior**

Add	*Adds an object to the end of the ArrayList.*
BinarySearch	*Uses a binary search algorithm to locate a specific element in the* ***sorted*** *ArrayList.*
Clear	*Removes all elements from the ArrayList.*
Contains	*Determines whether an element is in the ArrayList. Returns True or False.*
IndexOf	*Returns the zero-based index of the first occurrence of a value in the ArrayList.*
Insert	*Inserts an element into the ArrayList at the specified index.*
Remove	*Removes the first occurrence of a specific object from the ArrayList.*
RemoveAt	*Removes the element at the specified index of the ArrayList.*
Reverse	*Reverses the order of the elements in the ArrayList.*
Sort	*Sorts the elements in the ArrayList.*
ToArray	*Copies the elements of the ArrayList to a new array.*
TrimToSize	*Sets the capacity to the actual number of elements in the ArrayList.*

The Add and Insert method are used to place new values into the ArrayList. Add places elements at the end of the ArrayList while Insert places a new element at a specific index value. When inserting, the element at the current index location and the elements after this one are shifted down one location to make room for the inserted element.

Remove and RemoveAt remove elements from the ArrayList. RemoveAt takes an index value and removes the element at that index location. Remove takes an object and removes the first element storing that object. In Figure 10.11 you see a sample ArrayList named `Words`. If you say `Words.Remove("Hello")`, then the value at item 3 ("Hello") would be removed. Using the RemoveAt method, you would say `Words.RemoveAt(3)`.

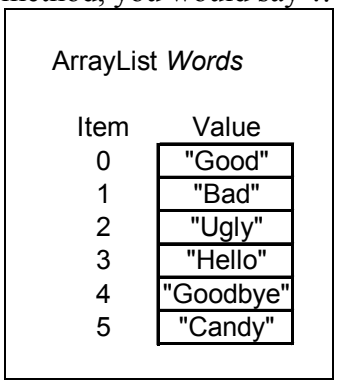

Figure 10.11 A sample ArrayList named *Words*

Contains and IndexOf are similar to Remove and RemoveAt. Contains searches for an occurrence of a value. `Words.Contains("Hello")` would return True. IndexOf returns the actual index value. `Words.IndexOf("Hello")` returns 3.

The BinarySearch method performs a search of the ArrayList (that must first be sorted using the Sort method). A ***binary search*** **is a very fast search method** (much faster than the sequential search we saw in the previous section). In a binary search, you start your search by checking the middle value (if you had 11 values, you would start by

checking element 5 assuming the first element is numbered zero). If you find what you are looking for you are done. However, if you do not find what you are looking for you only need to search the top or bottom half of the ArrayList. This is because the data are sorted. If your values are integers and they are sorted, and you are looking for the value 104, if you find the value 110 at location 5, you know that all values after location 5 are greater than 110. Thus, the value you are searching for cannot be in locations 6 through 10 and you would eliminate those elements from any further searching. In the remaining locations (0 through 4), you would again try the middle value and so on. The approach is called a binary search because you keep reducing the search space by one half. This gets you very quickly to the target location. If you have 1000 values, numbered 1 … 1000, the locations you would search might be: 500, 250, 125, 62, 31, 15, etc. Note that after only 6 tries, you have limited the search area to only 7 locations out of 1,000.

Exercise 10.7. Explain why a binary search requires the data to be ordered (sorted) for it to work?

Exercise 10.8. You are given the following list of numbers: 10, 20, 22, 34, 45, 46, 50, 66, 70, 78, 79, 80, 85, 92, and 98. Show how a binary search would work if you were searching for the value 80. Show how a binary search would work if you were searching for the value 21. How can you determine when to stop a binary search?

Example 10.3 Populating An ArrayList From a Database

In this example, we redo Example 10.1 except we use two ArrayLists instead of arrays to store the data from the database table. This example demonstrates the dynamic nature of the length of the ArrayList. Figure 10.12 shows the code for this example.

```
Dim NumRows As Integer
Dim CompName As New ArrayList()
Dim RateOfReturn As New ArrayList()

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) H
   Dim Row As Integer
   DsCompInfo1.Clear() ' clear and fill the DataSet
   odaCompInfo.Fill(DsCompInfo1)
   NumRows = DsCompInfo1.CompInfo.Rows.Count ' determine number of rows
   For Row = 0 To NumRows - 1 ' fill the array
      CompName.Add(DsCompInfo1.CompInfo.Rows.Item(Row).Item("CompName"))
      RateOfReturn.Add(DsCompInfo1.CompInfo.Rows.Item(Row).Item("RateOfReturn"))
   Next
End Sub

Private Sub btnReviewArrayContents_Click(ByVal sender As System.Object, ByVal e As S
   Dim Row As Integer, NewRow As String
   For Row = 0 To NumRows - 1
      NewRow = CompName.Item(Row) & "  -  " & RateOfReturn.Item(Row)
      lstArrayContents.Items.Add(NewRow)
   Next
End Sub
```

Figure 10.12 Code for Example 10.3

If you compare Figure 10.7 with Figure 10.12, you can first see that the variables `CompName` and `RateOfReturn` are declared as new ArrayList objects in Figure 10.12. Then, in Form1's Load event, you see no need to check on the number of rows because the capacity of the ArrayList is dynamic and will grow as needed. You also see the use of the ArrayList's Add method used within the loop that processes rows of the DataSet.

In the btnReviewArrayContents's click event, you see that use of the Item property to get access to the values stored within the ArrayList so that they may be transferred to the ListBox component.

Example 10.4 Performing a Binary Search

As discussed previously, the binary search is a very fast search method, especially when the number of items to be searched is large. In this example, we fill an ArrayList with an unordered set of fruit names and then provide a search function. Figure 10.13 shows the application at run time. Notice that the original values displayed in the ListBox component are not ordered, that the "Sort" button is enabled, and that the "Search" button is not enabled. Since the binary search only works on an ordered list, the "Sort" button must first be clicked, which causes the items in the ArrayList to be sorted and then used to refill the ListBox. In addition, the Search button is enabled. Finally, as you can see in the figure, a successful search indicates where the item was found (remember that we start at item number zero).

<combine into one figure>

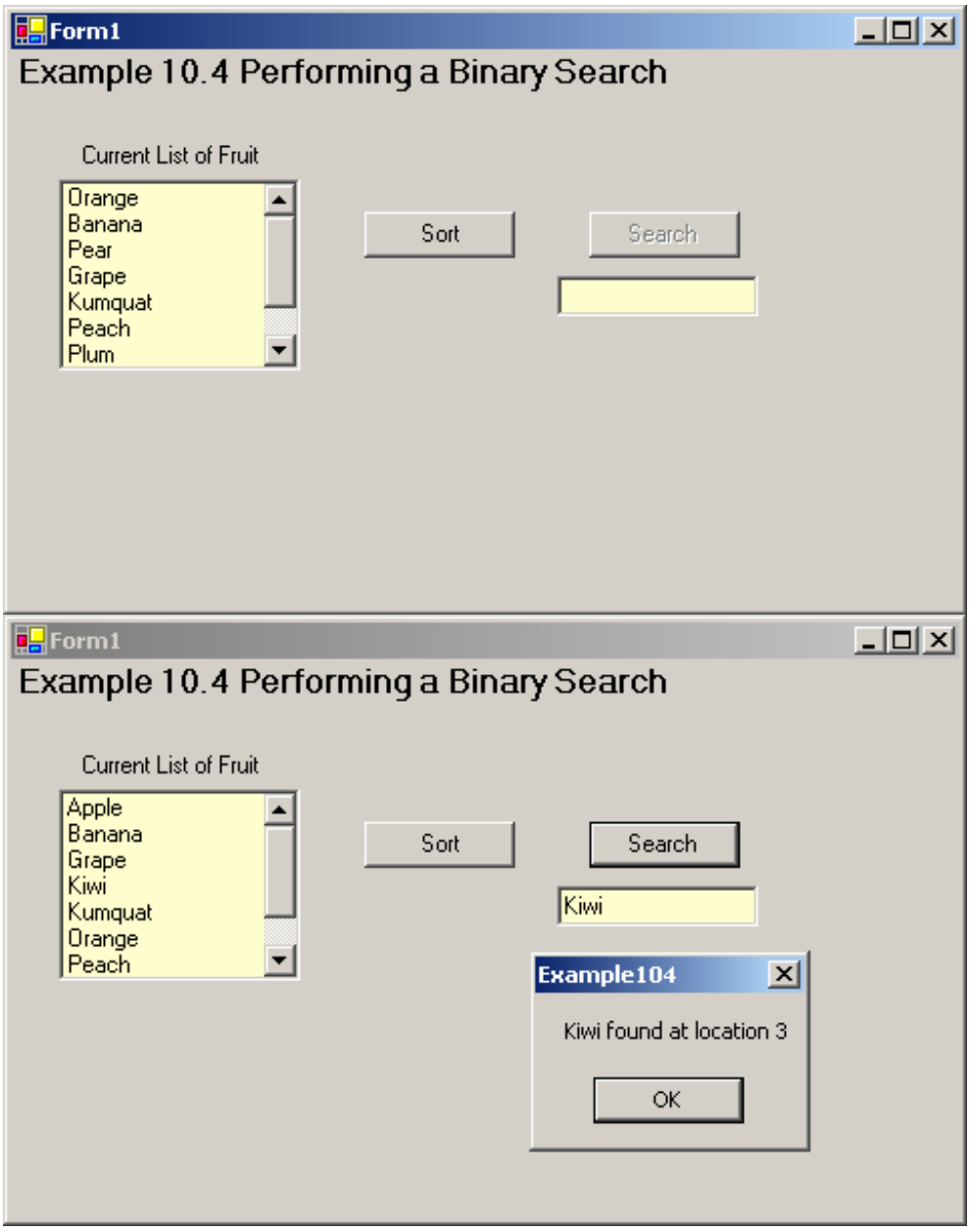

Figure 10.13 Example 10.4 at run time

The code for the form's Load event is shown in Figure 10.14. In this code, we fill the ArrayList by just entering some fruit names. In reality the data would more likely come from a database. Once the ArrayList is filled, the ListBox component is filled.

```
Dim FruitList As New ArrayList()

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) H
   Dim i As Integer
   ' manually add some fruit names - note not sorted
   FruitList.Add("Orange")
   FruitList.Add("Banana")
   FruitList.Add("Pear")
   FruitList.Add("Grape")
   FruitList.Add("Kumquat")
   FruitList.Add("Peach")
   FruitList.Add("Plum")
   FruitList.Add("Apple")
   FruitList.Add("Kiwi")
   ' fill the ListBox
   For i = 0 To FruitList.Count - 1
      lstFruitNames.Items.Add(FruitList.Item(i))
   Next
End Sub
```

Figure 10.14 The form's Load event for Example 10.4

The code for the sort is very simple. The Sort method for the ListArray is executed, the ListBox is cleared and then refilled from the now sorted ArrayList, and the Search button is enabled. The search code is also very straightforward. The BinarySearch method is executed and returns an integer value indicating where it found the target value. If this return value is less than zero, the search failed (the actual value that is returned changes but a failed search will always return a negative number). Figure 10.15 shows the code for these two click events.

```
Private Sub btnSort_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Dim i As Integer
    FruitList.Sort()
    ' refill the ListBox
    lstFruitNames.Items.Clear()
    For i = 0 To FruitList.Count - 1
        lstFruitNames.Items.Add(FruitList.Item(i))
    Next
    btnSearch.Enabled = True
End Sub

Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As System.EventAr
    Dim target As String, FoundLoc As Integer
    target = txtTarget.Text
    FoundLoc = FruitList.BinarySearch(target)
    If FoundLoc < 0 Then 'not found
        MsgBox("Sorry, " & target & " not found.")
    Else
        MsgBox(target & " found at location " & FoundLoc)
    End If
End Sub
```

Figure 10.15 The click event code for Example 10.4

Exercise 10.9. Modify Example 10.4 so that the user can perform a search without first sorting the array. What happens when you do this?

Example 10.5 Performing Multiple Result Search

In this example, we use the IndexOf method to find and count the number of times a specific word exists in an ArrayList of words. Figure 10.16 shows the example at run time.

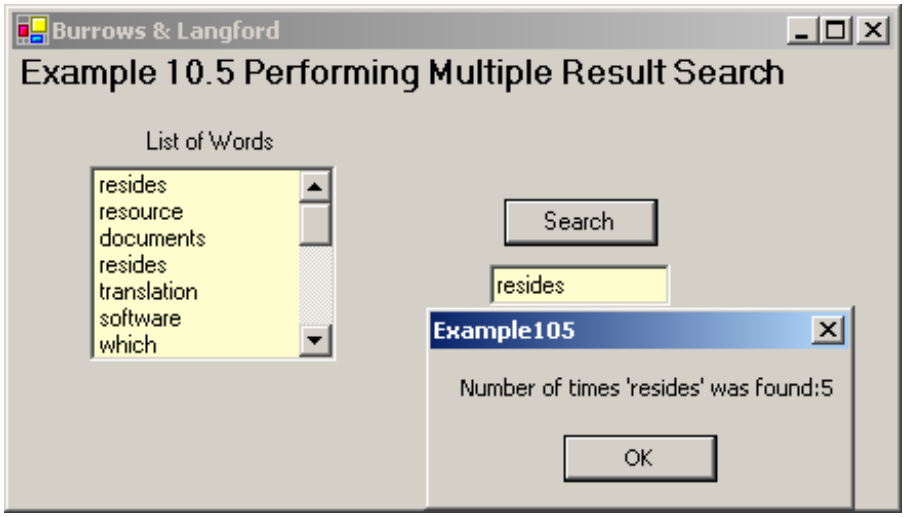

Figure 10.16 Example 10.5 at run time

The ArrayList is filled and then used to fill the ListBox component in the form's Load event like the previous examples. The code for the "Search" button is shown in Figure 10.17. This code uses the IndexOf method very much like you might use the InStr() function. It begins by searching for the target word starting in item zero. If this search fails, it returns a value of –1. The searching continues as long as it succeeds (return values >= 0). Each new search starts in the next item after where the previous search succeeded.

```
Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As S
   Dim target As String = txtTarget.Text
   Dim count As Integer
   Dim loc = WordList.IndexOf(target, 0)
   Do While loc >= 0
      count = count + 1
      loc = WordList.IndexOf(target, loc + 1)
   Loop
   MsgBox("Number of times '" & target & "' was found:" & count)
End Sub
```

Figure 10.17 The search code for Example 10.5

10.4 THE HASHTABLE COLLECTION

A ***Hashtable*** **is a data structure that supports very fast access to information based on a key field**. For example, if you want to store information on students, you could use the student number as the key field. Later, when you wanted to find information on a particular student, you would specify the specific student number and the Hashtable would find the corresponding information based on the value of the student number.

The details of how a Hashtable works are rather complex and beyond the scope of this text. However, the general concepts are fairly easy and worth discussing because they help explain when a Hashtable makes sense to use and when it does not make sense. The secret behind a Hashtable is something called the hash function. The ***hash function*** **takes a unique key field (part number, student number, social security number, etc.) and transforms it into a new value called a bucket number**. We can describe this symbolically as:

f(keyField) ➔ bucketNo

where *f* represents the hash function. There are a number of different hash functions but a commonly used one uses the modulus (remainder) function. In this case, the key field is divided by a constant and the remainder becomes the bucket number.

Once a bucket number is generated, the information related to the key field (like student name, address, etc.) is stored in that bucket. Later, when you want to get the information related to the key, you specify the key field, it is transformed into a bucket number, and then you go to that bucket to find the information. Figure 10.18 shows information on six students including their student numbers and names. The student numbers are the key fields. The bucket number is calculated by dividing the student

number by 11 and using the remainder. Note that if you divide a number by 11, the remainders range from 0 to 10.

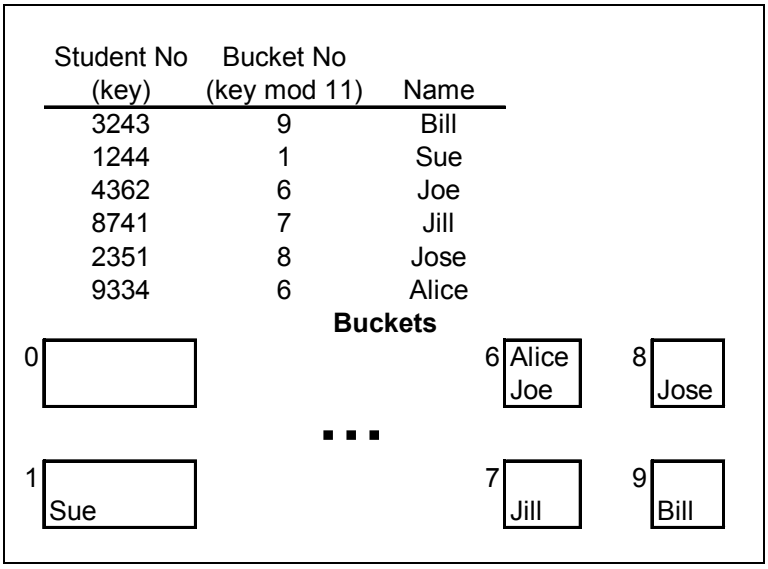

Figure 10.18 An example of hashing 6 students into buckets

You can see from Figure 10.18 that the values "hash" into buckets in a random fashion. That is, there is no relationship between the order in the buckets and either the original order of the student records or the value of the student numbers. This is an important characteristic of the ***hashing process*** **– it results in storing information in a random order**. Also note that two different keys can end up being hashed to the same bucket (see bucket 6). The Hashtable has the ability to handle this.

Once the values are "hashed" to their buckets, performing the hashing process again later provides accesses to the information. If you want to know the name of student number 2351, you would apply the hash function to 2351, get bucket number 8, and go directly to bucket 8 to find the name. Unlike search methods such as the binary search, hashing can generally find the correct value in one or two tries.

As developers, we can use the Hashtable class available in Visual Basic .NET that handles all the details for us. The important things we need to remember regarding the use of Hashtables are:

* They support very fast access to a record given the key field.
* They store the records randomly in the buckets.
* It is difficult to access the entire set of records one after the other.
* You must use the exact key to access the value as you used when the value was first stored. This means that searches for partial key matches, such as all names that start with "Smi", are not possible.

The last point above is a result of the randomization process. You can go through the buckets sequentially but if you do, the order of the records will be random. For example, the order of the records (by name) using the data in Figure 2.18 would be Sue, Alice, Joe, Jill, Jose, and Bill.

Now that we have some understanding of the Hashtable and its operation, we turn our attention to the common properties and methods.

Properties

Recall that a Hashtable is based on the key field and value(s) associated with it. Microsoft calls this the key/value pair. You can see the most common properties of the Hashtable class in Table 10.3.

Table 10.3 Common Properties of the Hashtable Class

Property	**Action**
Count	*Gets the number of key/value pairs contained in the Hashtable.*
Item	*Gets or sets the value associated with the specified key.*
Keys	*Gets a collection containing the keys in the Hashtable.*
Values	*Gets a collection containing the values in the Hashtable.*

The most commonly used property is the Item property. This property gets the value associated with a key. Its syntax is:

.Item(key)

For example, if you had a Hashtable named Students, and you wanted to get the value associated with student number 1234, you would say:

```
String Name = Students.Item("1234")
```

If there is a record in the Hashtable with a key matching "1234", then the student's name would be returned (assuming that is what was stored in the first place). If there was no record associated with the key "1234", then the special value Nothing would be returned. Typically you follow a reference to the Items property with an If statement to see what happened as follows:

```
String Name = Students.Item("1234")
If Name = Nothing Then
        ' no match
Else
        ' match
End If
```

Methods

The common methods associated with the Hashtable class are described in Table 10.4.

Table 10.4 Common Methods of the Hashtable Class

Method	**Behavior**
Add	*Adds an element with the specified key and value into the Hashtable.*
Clear	*Removes all elements from the Hashtable.*
Contains	*Determines whether the Hashtable contains a specific key. Returns True or False*
ContainsKey	*Determines whether the Hashtable contains a specific key. Returns True or False*
ContainsValue	*Determines whether the Hashtable contains a specific value. Returns True or False*
GetEnumerator	*Returns an enumerator that can iterate through the Hashtable.*
Remove	*Removes the element with the specified key from the Hashtable.*

The syntax of the Add method is:

MyHashtable.Add(key, value)

where *key* is used in the hash function to find the bucket number and *value* is what you want to associate with the key. The key cannot be anything – it must be from a class that implements a method called the GetHashCode. Fortunately the String class implements this method so we can use Strings for keys. Like the ArrayList, the capacity of the Hashtable is dynamic – its capacity grows as needed as new elements are added.

If you want to get all the values from a Hashtable, you must use the GetEnumerator method. This method returns an IDictionaryEnumerator object. The term "to enumerate" can be defined as "to specify one after another". An ***enumeration*** **therefore is a list of items one after the other**. The IDictionaryEnumerator object is a list of all values one after the other. The IDictionaryEnumerator class has a method called MoveNext. When the enumeration is first created, the "current" item is pointing before the first item so you have to execute the MoveNext once to get to the first item. After that, MoveNext moves to the next items one after the other. The MoveNext method returns a Boolean value True if the process of moving next finds a next record. It returns a Boolean value false if the processing of moving next comes to the end of the list.

The IDictionaryEnumerator class has two properties, Key and Value that store the key and the value of the current item. Assuming you have a Hashtable named Students, you could get and process an enumeration with the following code:

```
Dim AnEnumerator As IDictionaryEnumerator = Students.GetEnumerator
While AnEnumerator.MoveNext
   ' here you could reference AnEnumerator.Key and AnEnumerator.Value
End While
```

We now turn our attention to two examples that work with Hashtables.

Exercise 10.10. You are given the following set of student numbers: 2132, 4365, 3864,1649, 9342, 5477, 1992, 1032, 7493, and 2299. Create a set of buckets numbered 0 to 10 and store the student numbers into their proper bucket using the modulus-11 method (bucket number = remainder after dividing by 11). Then, using the value 5477, show how you would find the student number in the set of buckets.

Example 10.6 Populating a Hashtable From a Database Table

We again return to our example that processes a database table and stores the records in a Hashtable. The example also creates and displays an enumeration from the Hashtable. Figure 10.19 shows Example 10.6 at run time. Note that the original order of the records is different that the order shown in the enumeration. This should not surprise you because of the randomization process associated with converting the key values into bucket numbers.

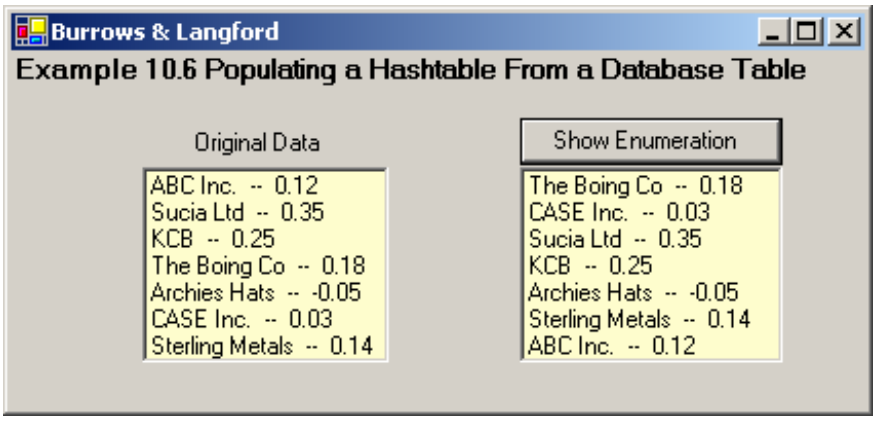

Figure 10.19 Example 10.6 at run time

The code for the example is shown in Figure 10.20. In the Form's Load event, you can see the Add method using the company name as the key and rate of return as the value. The loop that processes the DataSet also adds items to the ListBox lstOriginal. This list box shows the items in the order they were stored in the DataSet.

```
Dim NumRows As Integer
Dim ROR As New Hashtable()

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs
   Dim Row As Integer
   Dim Key As String, RateOfReturn As Decimal
   DsCompInfo1.Clear() ' clear and fill the DataSet
   odaCompInfo.Fill(DsCompInfo1)
   NumRows = DsCompInfo1.CompInfo.Rows.Count ' determine number of rows
   For Row = 0 To NumRows - 1 ' fill the array
      Key = DsCompInfo1.CompInfo.Rows.Item(Row).Item("CompName")
      RateOfReturn = DsCompInfo1.CompInfo.Rows.Item(Row).Item("RateOfReturn")
      ROR.Add(Key, RateOfReturn)
      lstOriginal.Items.Add(Key & "  --  " & RateOfReturn)
   Next
End Sub

Private Sub btnShowEnumeration_Click(ByVal sender As System.Object, ByVal e As Sy
   Dim AnEnumerator As IDictionaryEnumerator = ROR.GetEnumerator
   While AnEnumerator.MoveNext
      lstEnumeration.Items.Add(AnEnumerator.Key & "  --  " & AnEnumerator.Value)
   End While
End Sub
```

Figure 10.20 Code for Example 10.6

The click event creates an enumerator and then goes through it item by item, accessing the enumerator's Key and Value properties to get and display the company name (key) and rate of return (value).

Exercise 10.11. Modify the code in Example 10.6 so that the values for the enumeration are shown in alphabetic order based on the company name. The ListBox component should just show the company names, not the rates of return. Consider using an ArrayList object in your solution.

Example 10.7 Searching a Hashtable

The primary value of the Hashtable is its ability to find a record quickly given a key value. This example demonstrates this feature. Figure 10.21 shows the example at run time.

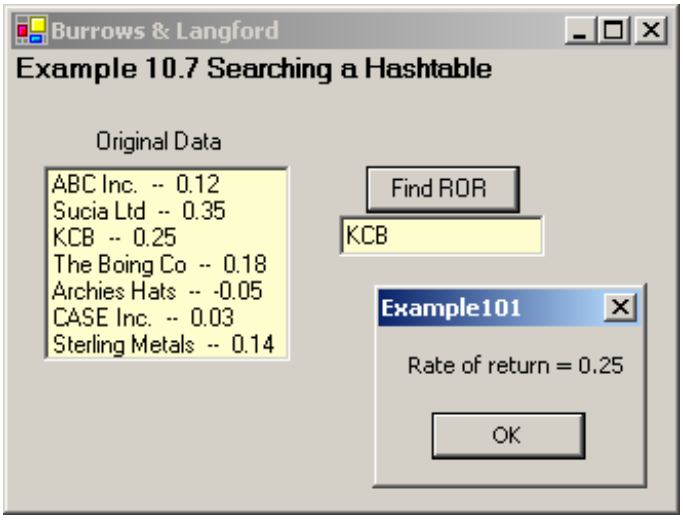

Figure 10.21 Example 10.7 at run time

The code for the click event for this example is shown in Figure 10.22 (the Form's Load event is the same as that in Example 10.6). You can see that this event gets the value entered by the user in the TextBox component and uses this to access the Item property to get the value associated with the key. The value returned is tested to see if it is equal to "Nothing" to determine if the value entered by the user exists as a key in the Hashtable.

```
Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As S
   Dim key As String = txtCompNo.Text
   Dim RateOfReturn As Decimal = ROR.Item(key)
   If RateOfReturn = Nothing Then
      MsgBox("Sorry, company " & key & " not found.")
   Else
      MsgBox("Rate of return = " & RateOfReturn)
   End If
End Sub
```

Figure 10.22 Code for the click event for Example 10.7

Exercise 10.12. Modify Example 10.7 so that it computes the average rate of return (sum of rates of return divided by the number of rates of return). Place this code in a new Button's click event.

Exercise 10.13. Modify Example 10.7 so that it finds and prints the company name for the company with the largest rate of return. Place this code in a new Button's click event.

ArrayList versus Hashtable

It is important to understand the main differences between the ArrayList and the Hashtable. The ArrayList is an ***indexed-based data structure*****, that is, it uses indices (subscripts) to access information it holds (just like a regular array).** On the other hand, the Hashtable is not index based; it stores information based on the value of a key field.

With index-based data structures the developer has control of the order in which the information is stored. It is also possible to go directly to an item based on the subscript value. For example, you can go directly to item 10. The difficulty is you cannot go directly to the item that stores information on product 3000 (unless product 3000 is stored in element number 3000). This is why the ArrayList class includes the BinarySearch method. You can, however, access items one after the other by using index values such as 0, 1, 2, ….

The ***key-based data structures*** **like the Hashtable provide fast access to information based on the key value**. However, the developer has no control over the order in which the values are stored. In addition, one must use an enumerator to access all the values one after the other.

You need to keep these factors in mind when deciding what type of collection you should use for your applications.

We now turn our attention to a third type of collection known as the SortedList. As you will see, this data structure provides some of the advantages of both the index- and key-based data structures.

10.5 THE SORTEDLIST COLLECTION

A ***SortedList*** **is a hybrid between a Hashtable and an Array. When an element is accessed by its key using the Item property, it behaves like a Hashtable**. When an element is accessed by its index using GetByIndex or SetByIndex, it behaves like an Array.

A SortedList internally maintains two arrays to store the elements of the list; that is, one array for the keys and another array for the associated values.

The elements of a SortedList are ordered by the value of the keys. A SortedList does not allow duplicate keys. It is also not possible to sort the items of a SortedList by its values.

Operations on a SortedList tend to be slower than operations on a Hashtable because of the sorting. However, the SortedList offers more flexibility by allowing access to the values either through the associated keys or through the indexes.

Properties

The common properties of the SortedList class are shown in Table 10.5.

Table 10.5 Common Properties of the SortedList Class

Property	**Action**
Count	*Gets the number of elements contained in the SortedList.*
Item	*Gets and sets the value associated with a specific key in the SortedList.*
Keys	*Gets the keys in the SortedList as an ICollection.*

Values	*Gets the values in the SortedList as an ICollection.*

As with the Hashtable, the Item property can be used to get a specific value based on value of the key. The syntax is

.Item(key)

If there is an item matching the *key*, the value associated with that item is returned, otherwise Nothing is returned.

Methods

The commonly used methods of the SortedList class are shown in Table 10.6.

Table 10.6 Common Methods of the SortedList Class

Method	**Behavior**
Add	*Adds an element with the specified key and value to the SortedList.*
Clear	*Removes all elements from the SortedList.*
ContainsKey	*Determines whether the SortedList contains a specific key.*
ContainsValue	*Determines whether the SortedList contains a specific value.*
GetByIndex	*Gets the value at the specified index of the SortedList.*
GetEnumerator	*Returns an IDictionaryEnumerator that can iterate through the SortedList.*
GetKey	*Gets the key at the specified index of the SortedList.*
IndexOfKey	*Returns the zero-based index of the specified key in the SortedList.*
IndexOfValue	*Returns the zero-based index of the first occurrence of the specified value in the SortedList.*
Remove	*Removes the element with the specified key from SortedList.*
RemoveAt	*Removes the element at the specified index of SortedList.*
SetByIndex	*Replaces the value at a specific index in the SortedList.*
TrimToSize	*Sets the capacity to the actual number of elements in the SortedList.*

As you can see from the list of methods, there are some that are index-based, like those found in Table 10.2 for the ArrayList and others that are key-based, like those found in Table 10.4 for the Hashtable. This reinforces the idea that the SortedList is both an index- and key-based data structure.

The GetKey and GetByIndex are the primary methods used to get the key and value at a particular index location. Both require you to supply an index value.

The Add method is similar to the Hashtable's Add method, using both a key and a value. However, unlike the Hashtable's Add method, the key does not have to belong to a class that implements the GetHashCode method. This means that it is somewhat more flexible as far as keys are concerned. Like both the ArrayList and Hashtable, the capacity of the SortedList is dynamic and grows as new elements are added.

The SortedList class also includes a GetEnumerator method. Once you get the enumeration, you process it like was described in Section 10.4.

Example 10.8 Populating a SortedList From a Database Table

We return a final time to our example that populates a collection; in this case we populate a SortedList. Figure 10.23 shows the example at run time. You can see that in addition to populating the SortedList, an enumeration is created and displayed and the SortedList is displayed by index. You can see that both enumeration and indexed display are ordered by the key, which is the company name in this case.

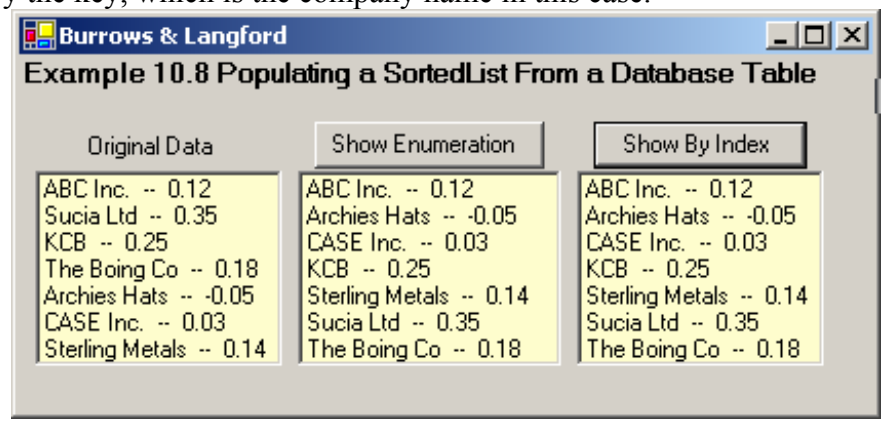

Figure 10.23 Example 10.8 at run time

The code for the Form Load event is shown in Figure 10.24. Except for the declaration of the SortedList:

```
Dim ROR As New SortedList()
```

this code is identical to the Form Load event for the Hashtable in Example 10.6. That is, the Add method looks the same; it is only different in its internal operation.

```
Dim NumRows As Integer
Dim ROR As New SortedList()

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs
   Dim Row As Integer
   Dim Key As String, RateOfReturn As Decimal
   DsCompInfo1.Clear() ' clear and fill the DataSet
   odaCompInfo.Fill(DsCompInfo1)
   NumRows = DsCompInfo1.CompInfo.Rows.Count ' determine number of rows
   For Row = 0 To NumRows - 1 ' fill the array
      Key = DsCompInfo1.CompInfo.Rows.Item(Row).Item("CompName")
      RateOfReturn = DsCompInfo1.CompInfo.Rows.Item(Row).Item("RateOfReturn")
      ROR.Add(Key, RateOfReturn)
      lstOriginal.Items.Add(Key & "  --  " & RateOfReturn)
   Next
End Sub
```

Figure 10.24 Code for Example 10.8's Form Load event

The code for the two "display" methods is shown in Figure 10.25. The code that shows the enumeration is unchanged from the Hashtable example. This is because both are based on an enumerator and once you have an enumerator, it is processed the same way regardless of where it came from. The only difference is the order of the items in the enumerator due to the random nature of the Hashtable and sorted nature of the SortedList.

```
Private Sub btnShowEnumeration_Click(ByVal sender As System.Object, ByVal e As Sy
    Dim AnEnumerator As IDictionaryEnumerator = ROR.GetEnumerator
    lstEnumeration.Items.Clear()
    While AnEnumerator.MoveNext
        lstEnumeration.Items.Add(AnEnumerator.Key & " -- " & AnEnumerator.Value)
    End While
End Sub

Private Sub btnShowByIndex_Click(ByVal sender As System.Object, ByVal e As Syste
    Dim r As Integer
    lstShowByIndex.Items.Clear()
    For r = 0 To ROR.Count - 1
        lstShowByIndex.Items.Add(ROR.GetKey(r) & " -- " & ROR.GetByIndex(r))
    Next
End Sub
```

Figure 10.25 Code for the two click events in Example 10.8

In Figure 10.25, the ShowByIndex code uses an index `r` to go through the SortedList based on the index value. This code uses `GetKey(r)` and `GetByIndex(r)` methods to access the key and value pair for element `r`.

Example 10.9 Finding Average Rate of Return

In this example we read the company rate of return data from the database and store it into a SortedList. We then go through the SortedList and compute the average rate of return. Finally we go back through the SortedList and place the company names of the companyies whose rate of return exceeds the average into a ListBox. Figure 10.26 shows Example 10.9 at run time.

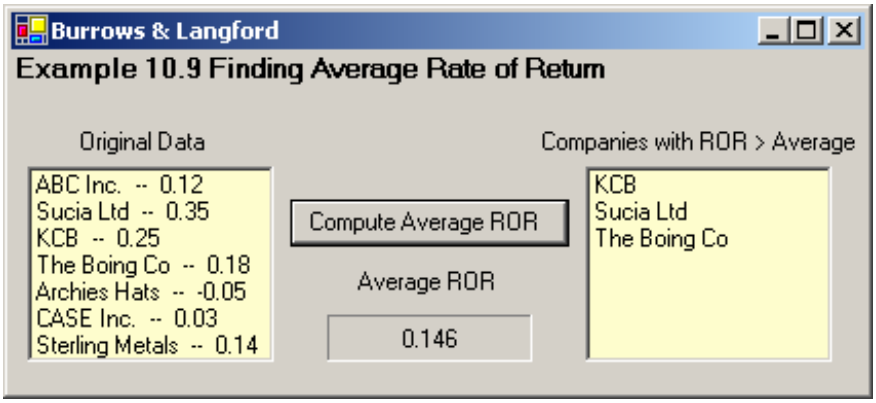

Figure 10.26 Example 10.9 at run time

The Form's Load event is identical to that in Example 10.8. The click event that computes the average and fills the list box is shown in Figure 10.27. Note that this code implements the pseudocode discussed in Section 10.1. It first goes through the SortedList and finds the sum of the ROR values. It then calculates the average ROR. Finally, it goes through the SortedList a second time comparing each company's ROR with the average and adding the company name to a ListBox for those whose ROR exceeds the average.

```
Private Sub btnComputeAvgROR_Click(ByVal sender As System.Object, ByV
   Dim Sum As Decimal, Item As Integer, Average As Decimal
   ' sum the ROR values
   For Item = 0 To ROR.Count - 1
      Sum = Sum + ROR.GetByIndex(Item)
   Next
   Average = Sum / ROR.Count
   lblAverageROR.Text = Format(Average, "0.###")
   ' now fill list box with companies whose have above average ROR
   lstAboveAverage.Items.Clear()
   For Item = 0 To ROR.Count - 1
      If ROR.GetByIndex(Item) > Average Then
         lstAboveAverage.Items.Add(ROR.GetKey(Item))
      End If
   Next
End Sub
```

Figure 10.27 Code for the Example 10.9's click event

Exercise 10.14. Modify Example 10.9 so that it finds and prints the company name for the company with the largest rate of return. Place this code in a new Button's click event.

10.6 PROJECT 11: ORDER ENTRY APPLICATION

Description of the Application

Note: In addition to using arrays, this project also introduces additional features associated with the DataSet and DataGrid controls.

This project simulates an order entry system. It displays a list of products and provides the user with the ability to build an order by choosing products from the list. The user can display the contents of their order at any time.

The main user interface for the project is shown in Figure 10.28. The form shows a list of products. The user may select any product in this list and then click on the Buy button to add the product to the order.

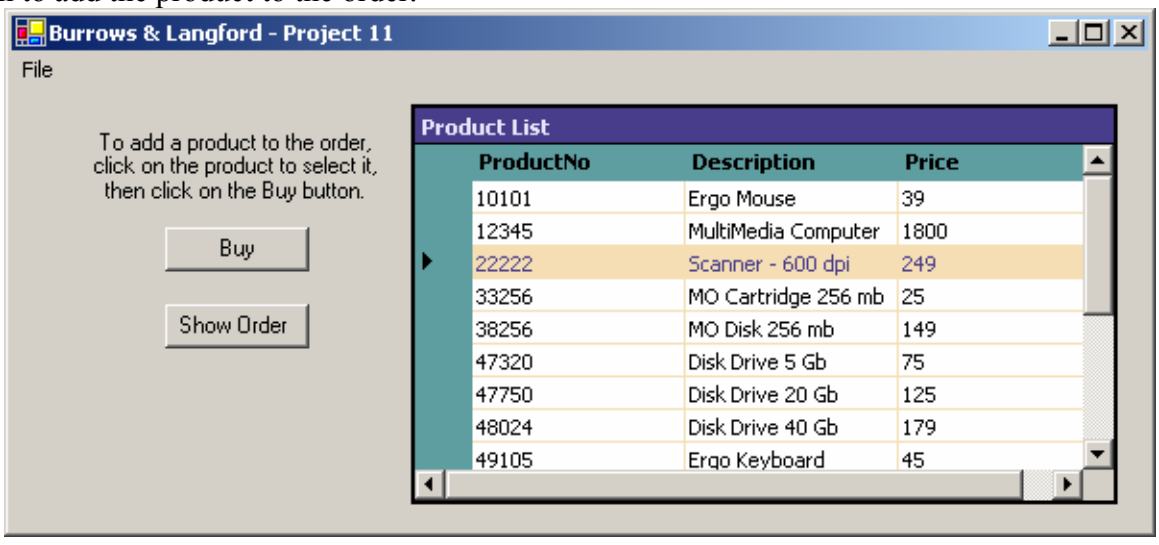

Figure 10.28 Main form for Project 11

When the user clicks on the Buy button, a new dialog box is displayed that asks the user for a quantity. This dialog box is shown in Figure 10.29. The user must enter a value number for the quantity. If they do not do this, then an error dialog is displayed. This error dialog is also shown in Figure 10.29.

<combine into one figure>

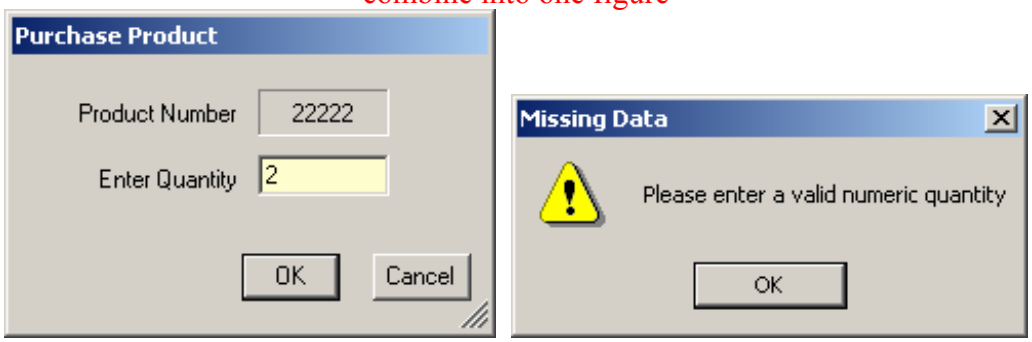

Figure 10.29 Purchase dialog box and error dialog if an invalid quantity value entered by user

In this version of the project (there is a more complex version in the Comprehensive Projects Section), the user cannot order the same product more than one time. If they try to do so, an error message, shown in Figure 10.30, is displayed.

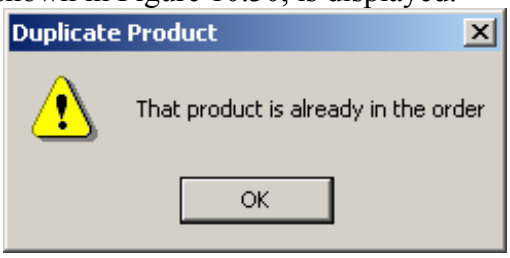

Figure 10.30 Error dialog if a product is already in the order

The user can click on the "Show Order" button at any time. When clicked, the current products that have been ordered are displayed. Figure 10.31 shows an example of this display.

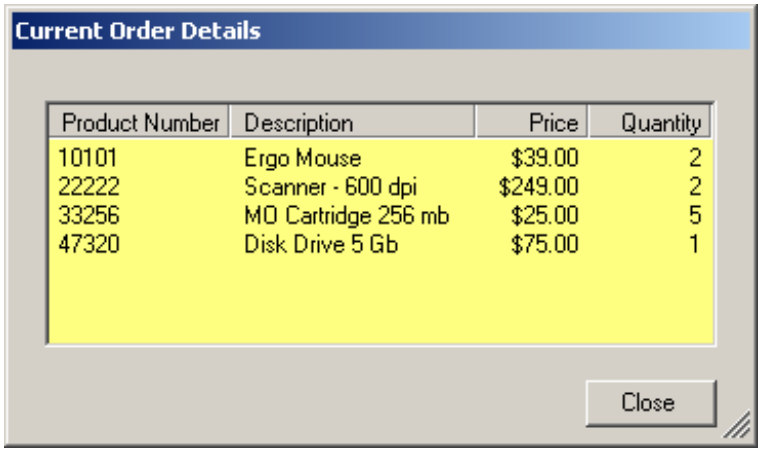

Figure 10.31 The order details dialog box

Design of the Application

This is a challenging project because it incorporates material from Chapter 7 and 8 as well as some new material. References to relevant material in prior chapters are made throughout the following discussion. It is likely that you will have to review that material.

The overall design of this project involves a database containing product information that is used to populate the DataGrid component on the main form. When the "Buy" button is clicked, there needs to be a way of recording the product and its quantity. This is necessary because the product information for ordered products must be saved for display purposes on the Order Details dialog box. The information on this dialog box is displayed using a ListView component.

The database for this project is an Access database named Proj11.mdb. This database contains a single table named Product. The design information for this table is shown in Figure 10.32.

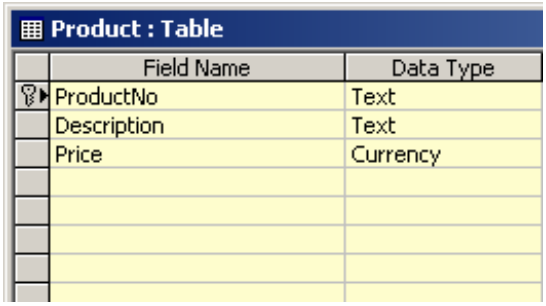

Figure 10.32 The Product table in the Proj11.mdb database

This table is used to populate the DataGrid component. A connection, data adaptor, and dataset must be created in order to do this. The actual code should be placed in the Form's Load event so that the product information is displayed when the application first starts.

When the user clicks on the "Buy" button, there must be a way to determine which product was selected at that time. This can be done by first finding the row number of the current cell and then using that information to get the value of the Product Number for the product in this row. You can find the current row by using the DataGrid's CurrentCell and RowNumber properties. Assuming you have a DataGrid named grdProducts, the code would be:

```
Dim Row As Integer = grdProducts.CurrentCell.RowNumber
```

Once you have the row number of the current row, you can get the value of any cell within that row using the DataGrid's Item property. Since the Product Number is in the first column of the DataGrid, its column number is zero. The code to get the value out of the row identified by the variable `Row`, column zero, is:

```
Dim ProdNo As String = grdProducts.Item(Row, 0)
```

At this point the user needs to be asked for the quantity of the product he wants to order. A new form must be displayed to do this (see Figure 10.29). Before the form is displayed, the Product Number must be stored in a Label component on the "Buy" form. This can be done by creating a new method in the "Buy" form named something like `SetProdNo`, which stores a String value into the label. The following is a sample of what might be created:

```
Public Sub SetProdNo(ByVal ProdNo As String)
  lblProdNo.Text = ProdNo
```

```
End Sub
```

This method can first be called, and then the "Buy" form can be shown using the ShowDialog method. This should be done from the main form code using statements such as:

```
objBuyForm.SetProdNo(ProdNo)
objBuyForm.ShowDialog()
```

Within the "Buy" dialog box, code must be written that provides access to the quantity the user entered as well as which button, OK or Cancel, was pressed. You should review Project 9 where this type of communication is discussed in detail.

When the OK button on the "Buy" dialog box is clicked, the Product Number and Quantity Ordered should be known. The question to address is how to store this information for later use when the user wants to show the current order details. There are several options. One option would be to write the information to a database table (not the Product table in the Proj11 database). Another option would be to add the information to an array. The later option is the one we will use here.

The next questions are: what should be stored in that array and what type of array should be used? To answer the first question, we will just store the Product Number and Quantity in the array. If we have the Product Number, we can go to the database and get the other information (Description and Price). This will simplify what needs to be stored in the array. We have several choices for the type of array to use. One choice is a two-dimensional array such as the one shown in Figure 10.33. This array has two columns; the first column stores the Product Number and the second column stores the Quantity. The problem with this option is determining how many rows to dimension. If we create too many, we waste storage because many rows will be unused. On the other hand, if we create too few, then the user will be limited on the number of products that can be ordered. This is certainly not good.

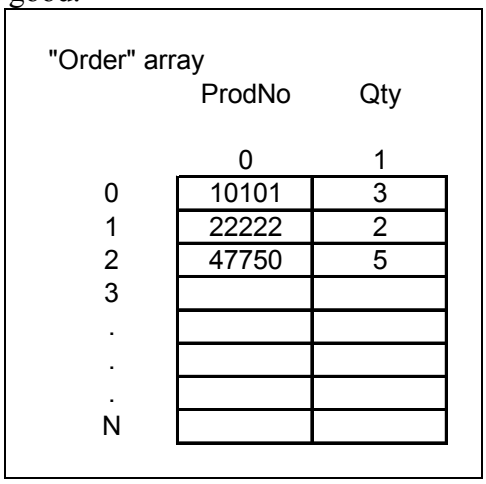

Figure 10.33 Using a traditional two-dimensional array

An alternative is to use one of the array collection classes. The advantage of this is the fact that their capacity is unbounded and can be as large as needed. For this project we will use a SortedList collection. This allows us to use the Product Number as the key and Quantity as the data item or value. We will be able to access an entry in the SortedList by using either the Product Number or by index. In this way we can access by

index to get each value, one at a time. Later, if we need to access a specific product, for example, to update a quantity value, we can use key access.

Assume we have a SortedList named ShoppingCart. Once the user has finished entering the quantity value in the "Buy" dialog box, we can first check to see if the OK button was clicked and, if it was, add a new entry to the SortedList that includes the Product Number and Quantity. The code below shows how this might be done.

```
If objBuyForm.GetBtnStatus = objBuyForm.OK Then
   Qty = objBuyForm.GetQty
   ShoppingCart.Add(ProdNo, Qty)
Else
   Exit Sub
End If
```

This code makes a reference to the symbolic constant objBuyForm.OK. This symbolic constant is defined in the "Buy" form using a Public statement:

```
Public Const OK = 1
```

Since we are using a SortedList, we cannot add more than one entry that has a specific Product Number. As we saw earlier, the code needs to detect this and handle the exception that would be thrown if one tried to add a duplicate Product Number.

We turn our attention to displaying the current order status. A separate form is used to display the information (see Figure 10.31). This form includes a ListView component that must be populated. The easiest way to do this is to do it from the main form since the main form has access to both the shopping cart SortedList as well as the DataSet component. The code segment below shows code for the "Show Order" button:

```
Dim objOrderForm As New frmOrder()
populateOrderFormListView(objOrderForm)
objOrderForm.ShowDialog()
objOrderForm.Close()
```

The populateOrderFormListView method is part of the main form's code. It uses the ShoppingCart collection, plus information from the Product table in the DataSet, to add new rows to the ListView component that is stored on the order form.

The populateOrderFormListView method needs to perform the following steps:

1. Set up the ListView (see Chapter 7).
    a. Set the view type.
    b. Get the ColumnHeaderCollection.
    c. Set up the column headings.
2. Perform the steps below for each item in the ShoppingCart collection (For N = 0 To ShoppingCart.Count – 1).
    a. Get the product number from location N of the ShoppingCart collection.
    b. Add a new row with the current product number in the first cell to the ListView component on the order form.
    c. Get the row from the DataSet that matches the current product number.
    d. Extract the description and price from the row retrieved in step c.
    e. Add subitems to the current ListView row including the description, price, and quantity (from the ShoppingCart).

Step 1 above should be fairly straightforward – review the material in Section 7.7 on the ListView component.

Step 2 needs some explanation. Getting the product number for item N of the ShoppingCart is done using the GetKey method. Using this value to add a new row to the ListView component is done using the Add method for the ListView component's Items property. Remember that if you want to access a component on another form, you need to prefix the component's name with the form name such as:

```
Row = objfrmOrder.lvwOrder.Items.Add(ProdNo)
```

where Row is declared as a ListViewItem object.

Step c requires accessing the row from the DataSet component that matches the current product number. The DataSet class's Table property has a Select method. The Select method selects a specific record or records based on a **"*filter*", which defines the condition that must be met to choose a row**. The following code segment shows how this is done.

```
Dim Filter As String = "ProductNo = " & ShoppingCart.GetKey(N)
Dim DBRow As DataRow() = DsProducts.Tables("Product").Select(Filter)
```

Since product numbers are unique in the DataSet, there will only be one row (row number 0) returned in this case. Step d says to extract a value from the row. This can be done using Item property of the DataRow class.

```
Dim Desc As String = DBRow(0).Item("Description")
```

Finally the values from the DataRow, as well as the quantity from the ShoppingCart, must be added as subitems to the row of the ListView using the Add method for the SubItems class.

```
Row.SubItems.Add(Desc)
```

Construction of the Application

After creating a new Windows Application, you need to add two new forms – one for obtaining the quantity the user wants to buy and another to display the current order status. A connection to the Access database needs to be set up using the Server Explorer. An OleDbDataAdapter with access to all the records in the Product table needs to be added to the main form. A DataSet also needs to be added to the main form. Once this is done, a DataGrid can be added to the main form and connected to the data adapter and data set.

The properties that should be set for the DataGrid (besides the DataSource and DataMember properties) include setting ReadOnly to True and Changing PreferredColumnWidth and PreferredHeaderWidth to appropriate values. The sample solution shown here used 111 and 30 respectively.

In Figure 10.28 you can see that the DataGrid is formatted with a variety of colors and fonts. The DataGrid provides an Auto Format feature that gives you some predefined options. Figure 10.34 shows where this feature is located in the Properties Window

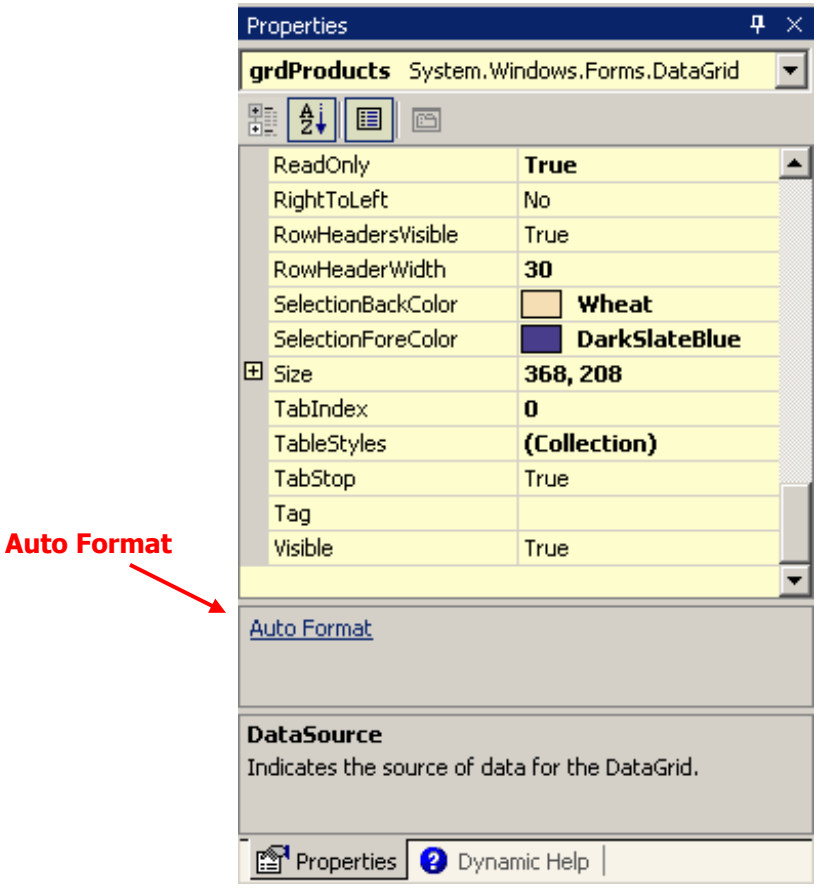

Figure 10.34 The Auto Format link in the Properties Window

When you click on this link, an Add Format dialog box like the one shown in Figure 10.35 is displayed. The solution shown here used the "Colorful 4" format.

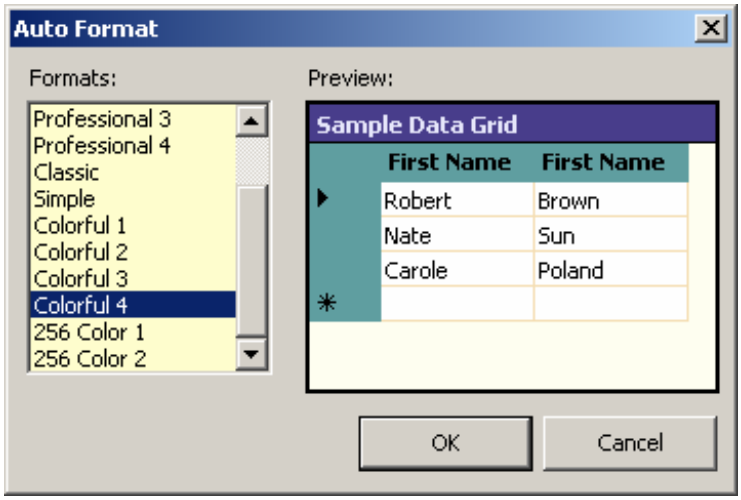

Figure 10.35 The Formats defined for the DataGrid

There is not much left besides the code. Remember that the main form needs to communicate with both the "Buy" form and the "Order Details" form. The main form needs to tell the Buy form the product number and the Buy form needs to make available

to the main form the quantity entered and the button clicked. The main form has to tell the Order Details form the products ordered (`ShoppingCart`) so that it knows what to add to the ListView component. Project 9 includes details on this type of communication.

Chapter Summary

1. An array is a variable that stores more than a single value. Arrays make it possible to solve many problems that cannot be solved with simple variables.
2. Like simple variables, arrays need to be declared using a Dim, Static, or Public statement. However, in addition to establishing the variable's name, type, and scope, for arrays the declaration statement also establishes the number of dimensions (the dimensionality) and the upper limits on the subscript for each dimension.
3. To access an element of an array, the programmer specifies both the array name and specific subscript values. Typically, subscripts are specified using variables or more complex expressions containing variables. For example, in the reference `PayRate(K)` the current value of `K` determines which element of the array `PayRate` is being accessed.
4. As with simple variables, the declaration statement that creates an array initializes all of its elements to zero or the zero-length string. Thus, it is typical for a program to first store data in an array, and then use the array to process the data. Storing data in an array is referred to as populating the array. Business applications typically obtain the data to populate an array from a database. The data sometimes come from another, previously populated array.
5. An ArrayList collection is an index-based data structure like an array. However, it does not have a fixed capacity. The capacity of an ArrayList is dynamic and grows as new items are added. The ArrayList class includes methods to sort the elements of the ArrayList (Sort) and search for a specific element (BinarySearch).
6. A Hashtable collection is a key-based data structure that, unlike an array, uses a key field to store and retrieve elements. This provides the ability to access a specific element very quickly without having to do a slow search operation. To access all the elements of the Hashtable, the class provides a method to generate an enumeration. The enumeration allows processing of the element one by one but the order of the elements is random (due to the randomizing effect of hashing).
7. A SortedList provides both index-based and key-based access to its elements. Thus, it shares the characteristics of both the ArrayList and Hashtable. Accessing an element via an index requires use of the GetByIndex method. Accessing an element by key is done using the Item property. The sorted list also provides a method to generate an enumeration. The order of the elements in the enumeration is based on the sorted order of the key.

Key Terms

array	Hashtable	sequential search
ArrayList	index value	simple variable
binary search	indexed variable	SortedList
collection	indexed-based data structure	subscript
dimensionality	key-based data structure	subscript bounds
element	linear search	subscripted variable
enumeration	matrix	table
filter	multidimensional array	unordered array
hash function	one-dimensional array	
hashing process	ordered array	

End-of-Chapter Problems

1. Why would an array be used instead of a database? When is it better to use a database than an array? Can they be used together? Why?
2. Explain the difference between an array and a collection. Which structures can hold references to other objects: an array, a collection, or both?
3. What are the advantages of using an array, an ArrayList, or a SortedList? Which of the three types are index-based data structures, key-based data structures, or both? Give an example where each array type (an array, an ArrayList, and a SortedList) would best be used to solve a problem.
4. What is the purpose of a hash table? If you wanted to step through a hash table one element at a time, how would you do it?
5. If a multi-dimensional array was defined with the "Dim myArray(2,30) As Integer" statement, how many total elements would the array contain? How many rows? How many columns? What are the index numbers of the first row and the first column of this array?
6. If you had to store the ordering information for football jerseys (price, jersey number, and player position), how could three arrays be coordinated to solve the problem? Why not use one array?
7. Describe how a binary search can be used on an ArrayList to locate a value. Why is using a binary search must faster than using index numbers to move through the list testing one element at a time?
8. How are new values added to an existing ArrayList? Provide an example of adding an item to an ArrayList and one of inserting an item before the first element. How can you determine the number of elements in an ArrayList?
9. In an ArrayList, SortedList, or Hashtable, can you use the available search functions to locate a partial string such as all of the string values ending with the ".edu" suffix? Can a database be searched to look for a partial value in a field?
10. What is the function of a database filter? How is a filter used with a database? Is there a SQL language statement that you learned in chapter 8 that could perform a similar function?

Programming Problems

1. Write an application that creates a one dimensional array and then copies the items contained in the array into a SortedList object and a Hashtable. Start by placing

three list boxes to display the primary array, the ArrayList, and the Hashtable. Then create two buttons: one that fills the list boxes with the current contents of the arrays and the second to copy the items from the first array into the second two.

The form load event should populate a list of items (such as states, names, foreign language terms, etc.) into the primary array. Transfer all of the values from the array into both the SortedList and the Hashtable.

2. Each telephone keypad has letters that correspond to the various numbers. While advertisers use these letters extensively, you can use an array to enter text to figure out a when you're getting a new phone number. Create an application that converts a text string input in a text box into a telephone number. Define two arrays: a letterArray (as an ArrayList so you can use the IndexOf() function) to hold the twenty-four letters on the phone (the letters 'Q' and 'Z' aren't included) and a numberArray that relates the keypad number to each element in the letterArray. Use a TextBox for input, a convert button to activate the conversion, and a label to receive the converted number. For the conversion algorithm you will need to use the Mid() function to retrieve the individual letters from the string and the IndexOf() function on the ArrayList to determine which element that letter represents. Use the following keypad table as a reference for defining the letter/number relation in your arrays:

Letters	Number
ABC	2
DEF	3
GHI	4
JKL	5
MNO	6
PRS	7
TUV	8
WXY	9

3. In this chapter, you learned how to use the Filter option on a DataSet to return just the data that you want. Create an application that uses a filter to access the stores table of the pubs database and select only the stores that are located in Washington (Dim Filter As String = "state = 'WA'"). Fill a ListBox using the DataRow object with the 'stor_name' field of all the stores that match the filter criteria.
4. One of the common uses of multi-dimensional arrays in a program is for internal calculation to perform the same types of functions needed in spreadsheets. Tabulating rows and columns of figures is simple and fast using arrays to store the data. Create an application that generates two simple reports: Total education spending by year (in thousands) and Average education spending by year (in thousands). You will need to define a two-dimension array (4 x 11) to hold the figures shown in the table below. They are the educational budget amounts from the U.S. Office of Management and Budget for the even years of the 1980s. The figures are broken down into 11 divisions. Provide the users with two ComboBox controls so they may select the year and the report to be displayed. Then allow the user to click a button that will run the computations and display the results in a label. You

can use the SelectedIndex of the year ComboBox to select the correct column in the array.

	1980	1982	1984	1986
Elementary and secondary	$4,239,022	$3,802,234	$4,294,269	$4,447,153
School assistance	$812,873	$457,227	$608,791	$677,055
Handicapped education	$1,555,253	$2,023,536	$2,416,799	$2,573,399
Vocational/adult programs	$1,153,743	$751,118	$954,320	$1,016,302
Postsecondary assistance	$5,108,534	$6,584,012	$7,478,401	$8,932,803
Direct postsecondary aid	$277,068	$284,467	$311,221	$294,681
Higher education facilities	$268,493	$449,191	$216,893	$206,017
Other higher education programs	$34,927	$38,226	$82,410	$64,032
Public library services	$101,218	$80,074	$107,895	$117,998
Special institutions	$273,860	$251,570	$249,610	$255,297
Department accounts	$277,174	$347,943	$352,089	$355,944

5. One of the important features of arrays is their ability to store lists of other objects. As mentioned in the chapter, these objects can even be other arrays. Create three ArrayList objects each that contain three elements for a total of nine items (Orange, Apple, Peach, Beef, Fish, Chicken, Water, Milk, and Soda). Create a summary ArrayList with three elements: the three other ArrayLists. Make an application that uses a loop to move through the summary ArrayList and a nested loop to place all of the text of the individual array items from the secondary array to display to a list box. Hint: It is probably easiest to create a temporary ArrayList inside the main loop that holds the current array and is used by the nested loop (tempArray = myALSummary(i)).
6. Create an application with a hash table of a list of individuals (social security number and name). Use the string containing social security numbers as the key fields. Allow the user to enter a key field and click the search button. If the entered value is found, display the name of the user with that SSN. Otherwise, show a display that the key field was not found. Hint: You can use the Masked Edit control to enforce the standard SSN format to prevent entry errors. The control is available

by right-clicking the Toolbox, selecting the Customized Toolbox option, and adding the Microsoft Masked Edit Control.