

## TOPICS

- |  |  |
|--|--|
| 7.1 Value Types and Reference Types        | 7.6 Advanced Algorithms for Sorting and Searching Arrays |
| 7.2 Array Basics                           | 7.7 Two-Dimensional Arrays                               |
| 7.3 Working with Files and Arrays          | 7.8 Jagged Arrays  |
| 7.4 Passing Arrays as Arguments to Methods | 7.9 The <code>List</code> Collection                     |
| 7.5 Some Useful Array Algorithms           |  |

## 7.1

## Value Types and Reference Types

**CONCEPT:** The data types in C# and the .NET Framework fall into two categories: value types and reference types.

In this chapter, you will gain more experience working with objects. Specifically, you will work with arrays and collections, which are objects that store groups of data. Before we go into the details of creating and working with those objects, it will be helpful for you to understand how objects are stored in memory. In this section, we discuss the ways that different types of objects are internally stored by the .NET Framework. As a result, you will better understand the concepts presented in this chapter, and chapters to come.

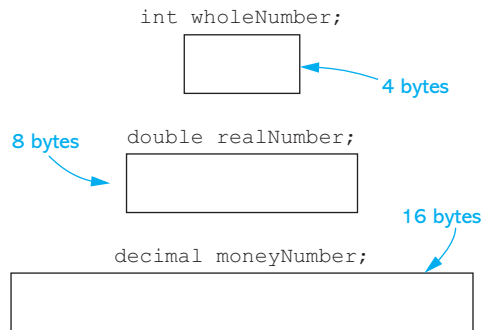
All the data types in C#—and the underlying .NET Framework—fall into two categories: **value types** and **reference types**. Of the C# data types that you have used so far, the following are value types: `int`, `double`, `decimal`, and `bool`. (There are other value types in addition to these, but these are the ones we focus on in this book.)

When you declare a value type variable, the compiler sets aside, or allocates, a chunk of memory that is big enough for that variable. For example, look at the following variable declarations:

```
int wholeNumber;  
double realNumber;  
decimal moneyNumber;
```

Recall from Chapter 3 that an `int` uses 32 bits of memory (4 bytes), a `double` uses 64 bits of memory (8 bytes), and a `decimal` uses 128 bits of memory (16 bytes). These declaration statements cause memory to be allocated as shown in Figure 7-1.

**Figure 7-1** Memory allocated

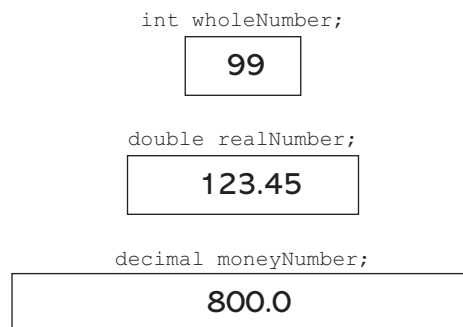


The memory that is allocated for a value type variable is the actual location that will hold any value that is assigned to that variable. For example, suppose we use the following statements to assign values to the variables shown in Figure 7-1:

```
wholeNumber = 99;
realNumber = 123.45;
moneyNumber = 800.0m;
```

Figure 7-2 shows how the assigned values are stored in each variable's memory location.

**Figure 7-2** Values assigned to the variables



As you can see from these illustrations, value types are very straightforward. When you are working with a value type, you are using a variable that holds a piece of data.

This is different from the way that reference types work. When you are working with a reference type, you are using two things:

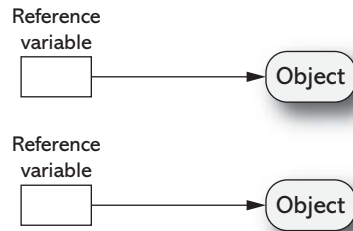
- An object that is created in memory
- A variable that references the object

The object that is created in memory holds data of some sort and performs operations of some sort. (Exactly what the data and operations are depends on what kind of object it is.) In order to work with the object in code, you need some way to refer to it. That's where the variable comes in. The variable does not hold an actual piece of data with which your program will work. Instead, it holds a special value known as a **reference**, which links the variable to the object.<sup>1</sup> When you want to work with the object, you use the variable that references it.

<sup>1</sup>A reference is similar to a memory address. It is a value that identifies the object's memory location.

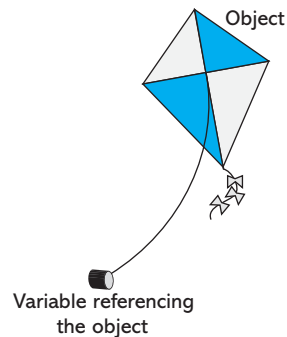
A variable that is used to reference an object is commonly called a **reference variable**. Reference variables can be used only to reference objects. Figure 7-3 illustrates two objects that have been created in memory, each referenced by a variable.

**Figure 7-3** Two objects referenced by variables



To understand how reference variables and objects work together, think about flying a kite. In order to fly a kite, you need a spool of string attached to it. When the kite is airborne, you use the spool of string to hold onto the kite and control it. This is similar to the relationship between an object and the variable that references the object. As shown in Figure 7-4, the object is like the kite, and the variable that references the object is like the spool of string.

**Figure 7-4** The kite and string metaphor



Creating a reference type object typically requires the following two steps:

1. You declare a reference variable.
2. You create the object and associate it with the reference variable.

After you have performed these steps, you can use the reference variable to work with the object. Let's look at an example that you have already learned about: creating objects of the `Random` class. Recall from Chapter 5 that the `Random` class allows your program to generate random numbers. Here is an example of how you create an object from the `Random` class:

```
Random rand = new Random();
```

Let's look at the different parts of this statement:

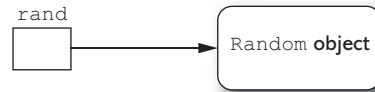
- The first part of the statement, appearing on the left side of the `=` operator, reads `Random rand`. This declares a variable named `rand` that can be used to reference an object of the `Random` type.
- The second part of the statement, appearing on the right side of the `=` operator, reads `new Random()`. The `new` operator creates an object in memory and returns a

reference to that object. So, the expression `new Random()` creates an object from the `Random` class and returns a reference to that object.

- The `=` operator assigns the reference that was returned from the `new` operator to the `rand` variable.

After this statement executes, the `rand` variable references a `Random` object, as shown in Figure 7-5. The `rand` variable can then be used to perform operations with the object, such as generating random numbers.

**Figure 7-5** The `rand` variable referencing a `Random` object



## Checkpoint

- 7.1 Into what two categories do the data types in C# and the underlying .NET Framework fall?
- 7.2 What is the difference in the way you work with value types and reference types?
- 7.3 How is the relationship between an object and a reference variable similar to a kite and a spool of string?
- 7.4 Is a variable of the `Random` class a reference type or a value type?

## 7.2 Array Basics

**CONCEPT:** An array allows you to store a group of items of the same data type together in memory. Processing a large number of items in an array is usually easier than processing a large number of items stored in separate variables.

In the programs you have written so far, you have used variables to store data in memory. The simplest way to store a value in memory is to store it in a variable. Variables work well in many situations, but they have limitations. For example, they can hold only one value at a time. Consider the following variable declaration:

```
int number = 99;
```

This statement declares an `int` variable named `number`, initialized with the value 99. Consider what happens if the following statement appears later in the program:

```
number = 5;
```

This statement assigns the value 5 to `number`, replacing the value 99 that was previously stored there. Because `number` is an ordinary variable, it can hold only one value at a time.

Because variables hold only a single value, they can be cumbersome in programs that process lists of data. For example, suppose you are asked to write a program that holds the names of 50 employees. Imagine declaring 50 variables to hold all those names:

```
string employee1;
string employee2;
string employee3;
```

*and so forth . . .*

```
string employee50
```

Then, imagine writing the code to process all 50 names. For example, if you wanted to display the contents of the variables in a `ListBox`, you would write code such as this:

```
employeeListBox.Items.Add(employee1); // Display employee 1
employeeListBox.Items.Add(employee2); // Display employee 2
employeeListBox.Items.Add(employee3); // Display employee 3
```

*and so forth . . .*

```
employeeListBox.Items.Add(employee50); // Display employee 50
```

As you can see, variables are not well suited for storing and processing lists of data. Each variable is a separate item that must be declared and individually processed.

Fortunately, you can use an array as an alternative to a group of variables. An **array** is an object that can hold a group of values that are all the same data type. You can have an array of `int` values, an array of `double` values, and array of `decimal` values, or an array of `string` values, but you cannot store a mixture of data types in an array. Once you create an array, you can write simple and efficient code to process the values that are stored in it.

Arrays are reference type objects. Recall from Section 7.1 that two steps are required to create and use a reference type object:

1. You declare a reference variable.
2. You create the object and associate it with the reference variable.

Suppose you want to create an array that can hold `int` values. Here is an example of how you might declare a reference variable for the array:

```
int[] numbersArray;
```

This statement declares a reference variable named `numbersArray`. Notice that this statement looks like a regular `int` variable declaration except for the set of brackets ( `[]` ) that appear after the keyword `int`. The expression `int[]` indicates that this variable is a reference to an `int` array. So, we cannot use this variable to store an `int` value. Rather, we can use it to reference an `int` array.

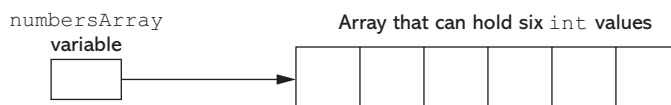
The next step in the process is to create the array object and associate it with the `numbersArray` variable. The following statement shows an example:

```
numbersArray = new int[6];
```

As previously mentioned, the `new` keyword creates an object in memory. The expression that appears after the `new` keyword specifies what type of object to create. In this case, the expression `int[6]` specifies that the object should be an array large enough to hold six `int` values. The number inside the brackets is the array's **size declarator**. It indicates the number of values that the array should be able to hold.

The `new` keyword also returns a reference to the object that it creates. In the previously shown statement, the `new` keyword creates an `int` array and returns a reference to that array. The `=` operator assigns the reference to the `numbersArray` variable. After this statement executes, the `numbersArray` variable will reference an `int` array that can hold six values. This is shown in Figure 7-6.

**Figure 7-6** The `numbersArray` variable referencing an `int` array



In the previous example, we used two statements to (1) declare a reference variable and (2) create an array object. These two steps can be combined into one statement, as shown here:

```
int[] numbersArray = new int[6];
```

You can create arrays of any data. The following are all valid array declarations:

```
double[] temperatures = new double[100];
decimal[] prices = new decimal[50];
string[] nameArray = new string[1200];
```

An array's size declarator must be a nonnegative integer expression. It can be a literal value, as shown in the previous examples, or a variable. It is a preferred practice to use a named constant as a size declarator, however. Here is an example:

```
const int SIZE = 6;
int[] numbersArray = new int[SIZE];
```

This practice can make programs easier to maintain. As you will see later in this chapter, many array-processing techniques require you to refer to the array's size. When you use a named constant as an array's size declarator, you can use the constant to refer to the size of the array in your algorithms. If you ever need to modify the program so the array is a different size, you need only change the value of the named constant.

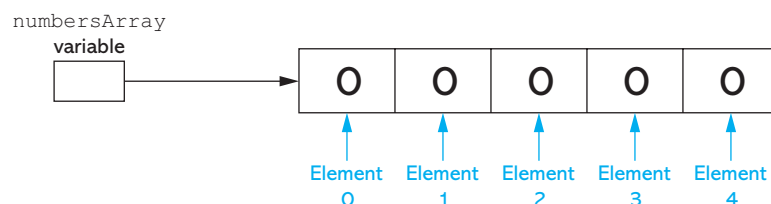
## Array Elements and Subscripts

The storage locations in an array are known as **elements**. In memory, an array's elements are located in consecutive memory locations. Each element in an array is assigned a unique number known as a **subscript**. Subscripts are used to identify specific elements in an array. The first element is assigned the subscript 0, the second element is assigned the subscript 1, and so forth. For example, suppose a program has the following declarations:

```
const int SIZE = 5;
int[] numbersArray = new int[SIZE];
```

As shown in Figure 7-7, the array referenced by `numbersArray` has five elements. The elements are assigned the subscripts 0–4. (Because subscript numbering starts at 0, the subscript of the last element in an array is 1 less than the total number of elements in the array.)

**Figure 7-7** Array subscripts



## Array Element Default Values

Notice that Figure 7-7 shows each element of the array containing the value 0. When you create a numeric array in C#, its elements are set to the value 0 by default.

Remember, you can create an array to hold any type of value. It is possible to create an array of reference type objects. If you create an array of reference type objects, each element of the array acts as a reference variable. By default, the elements of an array of reference type objects are set to the special value `null`. The value `null` indicates that a reference variable is not set to a valid object and cannot be used for any meaningful purpose.



**NOTE:** As you will see in Chapter 8, strings are actually reference types, so the default value of a `string` array's elements is `null`.

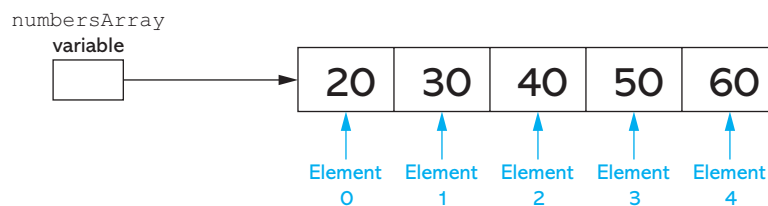
## Working with Array Elements

You access the individual elements in an array by using their subscripts. For example, the following code creates an `int` array with five elements and assigns values to each of its elements.

```
const int SIZE = 5;
int[] numbersArray = new int[SIZE];
numbersArray[0] = 20;
numbersArray[1] = 30;
numbersArray[2] = 40;
numbersArray[3] = 50;
numbersArray[4] = 60;
```

This code assigns the value 20 to element 0, the value 30 to element 1, and so forth. Figure 7-8 shows the contents of the array after these statements execute.

**Figure 7-8** Values assigned to each element



**NOTE:** The expression `numbersArray[0]` is pronounced “numbersArray sub zero.”

The following code shows another example. It creates a `string` array with three elements and assigns strings to each of its elements.

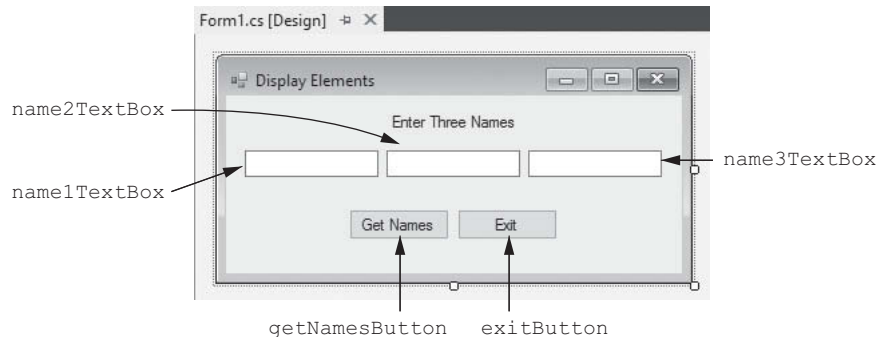
```
const int SIZE = 3;
string[] names = new string[SIZE];
names[0] = "Chris";
names[1] = "Laurie";
names[2] = "Joe";
```

The following code sample shows how values can be assigned from `TextBox` controls to array elements. Assume that an application's form has three `TextBox` controls named `amount1TextBox`, `amount2TextBox`, and `amount3TextBox` and that the user has entered a numeric value into each one. The following code creates a `decimal` array named `amounts` and assigns each of the `TextBox` control's input value to an array element.

```
const int SIZE = 3;
decimal[] amounts = new decimal[SIZE];
amounts[0] = decimal.Parse(amount1TextBox.Text);
amounts[1] = decimal.Parse(amount2TextBox.Text);
amounts[2] = decimal.Parse(amount3TextBox.Text);
```

Let's look at a complete program that demonstrates how to assign values to an array and then display the values in the array. In the *Chap07* folder of this book's Student Sample Programs, you will find a project named *Display Elements*. Figure 7-9 shows the application's form.

**Figure 7-9** The *Display Elements* application's form



Here is the code for the `getNamesButton_Click` event handler:

```

1 private void getNamesButton_Click(object sender, EventArgs e)
2 {
3     // Create an array to hold three strings.
4     const int SIZE = 3;
5     string[] names = new string[SIZE];
6
7     // Get the names.
8     names[0] = name1TextBox.Text;
9     names[1] = name2TextBox.Text;
10    names[2] = name3TextBox.Text;
11
12    // Display the names.
13    MessageBox.Show(names[0]);
14    MessageBox.Show(names[1]);
15    MessageBox.Show(names[2]);
16 }

```

Run the application, enter a name into each of the `TextBox` controls, and then click the *Get Names* button. The following actions take place:

- In line 5, an array to hold three strings is created.
- In lines 8–10, the names that you entered into the `TextBox` controls are assigned to the array elements.
- In lines 13–15, each element of the array is displayed in a message box.

The *Display Elements* application displays the contents of a `string` array. Because the array's elements are strings, we can pass each element directly to the `MessageBox.Show` method without performing a data type conversion. If you want to pass a numeric array element to the `MessageBox.Show` method, however, you will have to call the element's `ToString` method. The following code sample demonstrates:

```

1 // Create an array to hold three integers.
2 const int SIZE = 3;
3 int[] myValues = new int[SIZE];
4
5 // Assign some values to the array elements.
6 myValues[0] = 10;
7 myValues[1] = 20;
8 myValues[2] = 30;
9
10 // Display the array elements.

```



```
11 MessageBox.Show(myValues[0].ToString());
12 MessageBox.Show(myValues[1].ToString());
13 MessageBox.Show(myValues[2].ToString());
```

## Array Initialization

When you create an array, you can optionally initialize it with a group of values. Here is an example:

```
const int SIZE = 5;
int[] numbersArray = new int[SIZE] { 10, 20, 30, 40, 50 };
```

The series of values inside the braces and separated with commas is called an **initialization list**. These values are stored in the array elements in the order they appear in the list. (The first value, 10, is stored in `numbersArray[0]`, the second value, 20, is stored in `numbersArray[1]`, and so forth.)

When you provide an initialization list, the size declarator can be left out. The compiler determines the size of the array from the number of items in the initialization list. Here is an example:

```
int[] numbersArray = new int[] { 10, 20, 30, 40, 50 };
```

In this example, the compiler determines that the array should have five elements because five values appear in the initialization list.

You can also leave out the `new` operator and its subsequent expression when an initialization list is provided. Here is an example:

```
int[] numbersArray = { 10, 20, 30, 40, 50 };
```

Here are three separate examples that declare and initialize a `string` array named `days`. Each of these examples results in the same array:

```
// Example 1
const int SIZE = 7;
string[] days = new string[SIZE] = { "Sunday", "Monday",
                                     "Tuesday", "Wednesday", "Thursday",
                                     "Friday", "Saturday" };

// Example 2
string[] days = new string[] = { "Sunday", "Monday",
                                 "Tuesday", "Wednesday", "Thursday",
                                 "Friday", "Saturday" };

// Example 3
string[] days = { "Sunday", "Monday", "Tuesday",
                 "Wednesday", "Thursday", "Friday",
                 "Saturday" };
```

## Using a Loop to Step through an Array

You can store a number in an `int` variable and then use that variable as a subscript. This makes it possible to use a loop to step through an array, performing the same operation on each element. For example, look at the following code sample:

```
1 // Create an array to hold three integers.
2 const int SIZE = 3;
3 int[] myValues = new int[SIZE];
4
5 // Assign 99 to each array element.
6 for (int index = 0; index < SIZE; index++)
7 {
8     myValues[index] = 99;
9 }
```

Line 3 creates an `int` array named `myValues` with three elements. The `for` loop that starts in line 6 uses an `int` variable named `index` as its counter. The `index` variable is initialized with the value 0 and is incremented after each loop iteration. The loop iterates as long as `index` is less than 3. So, the loop will iterate three times. As it iterates, the `index` variable is assigned the values 0, 1, and 2.

Inside the loop, the statement in line 8 assigns the value 99 to an array element, using the `index` variable as the subscript. This is what happens as the loop iterates:

- The first time the loop iterates, `index` is set to 0, so 99 is assigned to `myValues[0]`.
- The second time the loop iterates, `index` is set to 1, so 99 is assigned to `myValues[1]`.
- The third time the loop iterates, `index` is set to 2, so 99 is assigned to `myValues[2]`.

## Invalid Subscripts

When working with an array, it is important that you do not use an invalid subscript. You cannot use a subscript that is less than 0 or greater than the size of the array minus 1. For example, suppose an array has 100 elements. The valid subscripts for the array are the integers 0 through 99. If you try to use any value outside this range, an exception will be thrown at runtime. The following code sample demonstrates how a loop that is not carefully written can cause such an exception to be thrown:

```
1 // Create an array to hold three integers.
2 const int SIZE = 3;
3 int[] myValues = new int[SIZE];
4
5 // Will this loop cause an exception?
6 for (int index = 0; index <= SIZE; index++)
7 {
8     myValues[index] = 99;
9 }
```

Notice that the `for` loop iterates as long as `index` is less than *or equal to* 3. During the loop's last iteration, `index` is set to 3, so the statement in line 8 attempts to make an assignment to `myValues[3]`. There is no element in the array with the subscript 3, so an exception will be thrown.

## The Length Property

In C#, all arrays have a **Length** property that is set to the number of elements in the array. For example, consider an array created by the following statement:

```
double[] temperatures = new double[25];
```

The `temperatures` array's `Length` property will be set to 25. If we executed the following statement, it would display the message "The temperatures array has 25 elements."

```
MessageBox.Show("The temperatures array has " +
    temperatures.Length + " elements.");
```

The `Length` property can be useful when processing the entire contents of an array with a loop. The subscript of the last element is always 1 less than the array's `Length` property. Here is an example:

```
for (int index = 0; index < temperatures.Length; index++)
{
    MessageBox.Show(temperatures[index].ToString());
}
```



**NOTE:** An array's `Length` property is read only, so you cannot change its value.

In Tutorial 7-1, you complete an application that generates a set of random numbers similar to those used in lotteries. The numbers will be stored in an array.



## Tutorial 7-1:

### Using an Array to Hold a List of Random Lottery Numbers

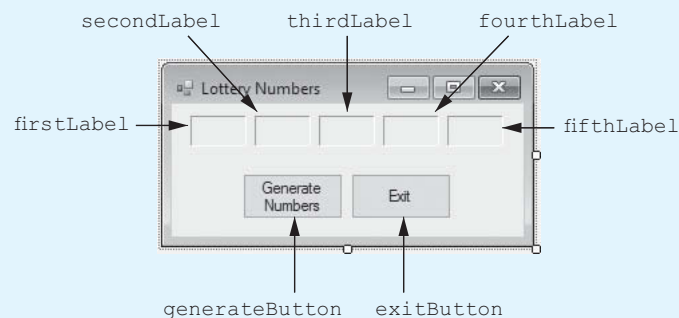


VideoNote

**Tutorial 7-1:**  
Using an  
Array to  
Hold a List  
of Random  
Lottery  
Numbers

In this tutorial, you complete an application that randomly generates lottery numbers. The application's form is shown in Figure 7-10. When the *Generate Numbers* button is clicked, the application will generate five two-digit integer numbers and store them in an array. The contents of the array will then be displayed in Label controls.

**Figure 7-10** The *Lottery Numbers* application's form



**Step 1:** Start Visual Studio. Open the project named *Lottery Numbers* in the *Chap07* folder of this book's Student Sample Programs.

**Step 2:** Open the *Form1* form in the *Designer*. Double-click the `generateButton` control. This will open the code editor, and you will see an empty event handler named `generateButton_Click`. Complete the `generateButton_Click` event handler by typing the code shown in lines 22–41 in Program 7-1. Let's take a closer look at the code:

**Line 23:** This statement declares an `int` constant named `SIZE`, set to the value 5. This is used as an array size declarator.

**Line 24:** This statement creates an `int` array named `lotteryNumbers` with five elements.

**Line 27:** This statement creates a `Random` object, referenced by a variable named `rand`.

**Line 31:** This `for` loop uses an `int` variable named `index` as its counter. The `index` variable is initialized with the value 0 and is incremented after each loop iteration. The loop iterates as long as `index` is less than `lotteryNumbers.Length` (which is 5). So, the loop will iterate five times. As it iterates, the `index` variable is assigned the values 0, 1, 2, 3, and 4.

**Line 33:** This statement gets a random number in the range of 0 through 99 and assigns it to `lotteryNumbers[index]`. The first time the loop iterates, this statement assigns a random number to `lotteryNumbers[0]`. The second time the loop iterates, this statement assigns a random number to `lotteryNumbers[1]`.

This continues until the loop is finished. At that time, each element in the array is assigned a random number.

**Lines 37–41:** These statements display the array elements in the `firstLabel`, `secondLabel`, `thirdLabel`, `fourthLabel`, and `fifthLabel` controls.

**Step 3:** Switch your view back to the *Designer* and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the `exitButton_Click` event handler by typing the code shown in lines 46–47 in Program 7-1.

**Step 4:** Save the project. Then, press `F5` on the keyboard, or click the *Start Debugging* button (`F5`) on the toolbar to compile and run the application. When the application runs, click the *Generate Numbers* button. The application should display a set of random numbers in the Label controls. Click the *Generate Numbers* button several more times to see different sets of random numbers. When you are finished, click the *Exit* button to exit the application.

---

### Program 7-1 Completed code for Form1 in the *Lottery Numbers* application

```

1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10
11 namespace Lottery_Numbers
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void generateButton_Click(object sender, EventArgs e)
21         {
22             // Create an array to hold the numbers.
23             const int SIZE = 5;
24             int[] lotteryNumbers = new int[SIZE];
25
26             // Create a Random object.
27             Random rand = new Random();
28
29             // Fill the array with random numbers, in the range
30             // of 0 through 99.
31             for (int index = 0; index < lotteryNumbers.Length; index++)
32             {
33                 lotteryNumbers[index] = rand.Next(100);
34             }
35
36             // Display the array elements in the Label controls.
37             firstLabel.Text = lotteryNumbers[0].ToString();
38             secondLabel.Text = lotteryNumbers[1].ToString();

```

```
39         thirdLabel.Text = lotteryNumbers[2].ToString();
40         fourthLabel.Text = lotteryNumbers[3].ToString();
41         fifthLabel.Text = lotteryNumbers[4].ToString();
42     }
43
44     private void exitButton_Click(object sender, EventArgs e)
45     {
46         // Close the form.
47         this.Close();
48     }
49 }
50 }
```

## Watching for Off-by-One Errors

Because array subscripts start at 0 rather than 1, you have to be careful not to perform an off-by-one error. An **off-by-one error** occurs when a loop iterates one time too many or one time too few. For example, look at the following code sample:

```
1 // Create an array to hold three integers.
2 const int SIZE = 100;
3 int[] myValues = new int[SIZE];
4
5 // Assign 99 to each array element.
6 for (int index = 1; index < myValues.Length; index++)
7 {
8     myValues[index] = 99;
9 }
```

The intent of this code is to create an `int` array with 100 elements and assign the value 99 to each element; however, this code has an off-by-one error. During the loop's execution, the `index` variable is assigned the values 1 through 99 when it should be assigned the values 0 through 99. As a result, the first element, which is at subscript 0, is skipped.

## Using the `foreach` Loop with Arrays

C# provides a special loop that, in many circumstances, simplifies array processing. It is known as the **foreach loop**. When you use the `foreach` loop with an array, the loop automatically iterates once for each element in the array. For example, if you use the `foreach` loop with an eight-element array, the loop will iterate eight times. Because the `foreach` loop automatically knows the number of elements in an array, you do not have to use a counter variable to control its iterations, as with a regular `for` loop.

The `foreach` loop is designed to work with a temporary, read-only variable known as the **iteration variable**. Each time the `foreach` loop iterates, it copies an array element to the iteration variable. For example, the first time the loop iterates, the iteration variable will contain the value of element 0, the second time the loop iterates, the iteration variable will contain the value of element 1, and so forth.

Here is the general format of the `foreach` loop:

```
foreach (Type VariableName in ArrayName)
{
    statement;
    statement;
    etc.
}
```

The statements that appear inside the curly braces are the body of the loop. These are the statements executed each time the loop iterates. As with other control structures, the curly braces are optional if the body of the loop contains only one statement, as shown in the following general format:

```
foreach(Type VariableName in ArrayName)
    statement;
```

Let's take a closer look at the items appearing inside the parentheses:

- *Type* is the data type of the values in the array.
- *VariableName* is the name of the iteration variable.
- *in* is a keyword that must appear after the *VariableName*.
- *ArrayName* is the name of an array.

Suppose we have the following array declaration:

```
int[] numbers = { 3, 6, 9 };
```

We can use the following `foreach` loop to display the contents of the `numbers` array:

```
foreach (int val in numbers)
{
    MessageBox.Show(val.ToString());
}
```

Because the `numbers` array has three elements, this loop will iterate three times. The first time it iterates, `val` will contain the value of `numbers[0]`, so a message box will display the value 3. During the second iteration, `val` will contain the value of `numbers[1]`, so a message box will display the value 6. During the third iteration, `val` will contain the value of `numbers[2]`, so a message box will display the value 9.

### The `foreach` Loop versus the `for` Loop

When you need to read the values that are stored in an array from the first element to the last element, the `foreach` loop is simpler to use than the `for` loop. With the `foreach` loop, you do not have to be concerned about the size of the array, and you do not have to create a counter variable to hold subscripts; however, because the iteration variable is read only, there are circumstances in which the `foreach` loop is not adequate. You cannot use the `foreach` loop if you need to do any of the following:

- Change the contents of an array element
- Work through the array elements in reverse order
- Access some, but not all, of the array elements
- Simultaneously work with two or more arrays within the loop

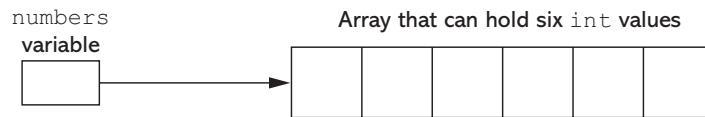
In any of these circumstances, you should use the `for` loop to process the array.

## Reassigning an Array Reference Variable

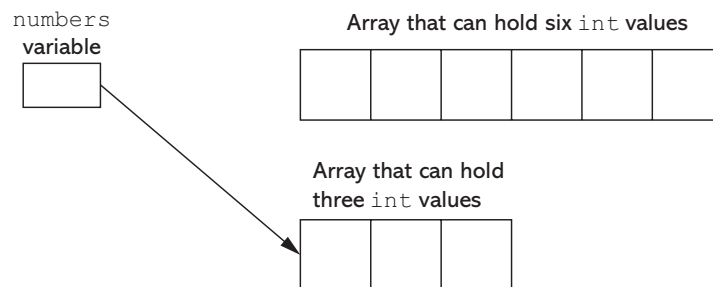
It is possible to reassign an array reference variable to a different array, as demonstrated by the following code sample:

```
1 // Create an array referenced by the numbers variable.
2 int[] numbers = new int[6];
3
4 // Reassign the numbers variable to a new array.
5 numbers = new int[3];
```

The statement in line 2 creates a six-element `int` array. A reference to the array is assigned to the `numbers` variable. Figure 7-11 shows how the `numbers` variable references the six-element array after this statement executes.

**Figure 7-11** The `numbers` variable referencing a six-element array

Then, the statement in line 5 creates a new, three-element `int` array. A reference to the new array is assigned to the `numbers` variable. When line 5 executes, the reference that is currently stored in the `numbers` variable will be replaced by a reference to the three-element array. After this statement executes, the `numbers` variable will reference the three-element array instead of the six-element array. This is illustrated in Figure 7-12.

**Figure 7-12** The `numbers` variable referencing a three-element array

Notice in Figure 7-12 that the six-element array still exists in memory, but it is no longer referenced by any variables. Because it is no longer referenced, it cannot be accessed. When an object is no longer referenced, it becomes eligible for garbage collection. **Garbage collection** is a process that periodically runs, removing all unreferenced objects from memory.



### Checkpoint

- 7.5 Write a statement that declares a reference variable named `monthlyPay` for an array that can hold `decimal` values.
- 7.6 Write a statement so that the `monthlyPay` variable from Checkpoint 7.5 references a `decimal` array that can hold 12 values.
- 7.7 Combine the statements from Checkpoints 7.5 and 7.6 into a single statement, and use a named constant for a size declarator.
- 7.8 Write a statement that creates an array of 3 `string` values referenced by a variable named `fullName`. Provide an initialization list for the array using string values for a first, middle, and last name.
- 7.9 Under what circumstances should you use a `for` loop rather than a `foreach` loop to process data stored in an array?
- 7.10 What happens when an object such as an array is no longer referenced by a variable?

## 7.3 Working with Files and Arrays

**CONCEPT:** For some problems, files and arrays can be used together effectively. You can easily write a loop that saves the contents of an array to a file, and vice versa.

Some tasks may require you to save the contents of an array to a file so the data can be used at a later time. Likewise, some situations may require you to read the data from a file into an array. For example, suppose you have a file that contains a set of values and you want to reverse the order of the values. One technique for doing this is to read the file's values into an array and then write the values in the array back to the file from the end of the array to the beginning.

### Writing an Array's Contents to a File

Writing the contents of an array to a file is a straightforward procedure: Open the file and use a loop to step through each element of the array, writing its contents to the file. For example, in the *Chap07* folder of the Student Sample Programs, you will find a project named *Array To File*. When you click the OK button, the application writes the contents of an `int` array to a file. The following code shows the Click event handler for the OK button.

```

1 private void okButton_Click(object sender, EventArgs e)
2 {
3     try
4     {
5         // Create an array with some values.
6         int[] numbers = { 10, 20, 30, 40, 50 };
7
8         // Declare a StreamWriter variable.
9         StreamWriter outputFile;
10
11        // Create the file and get a StreamWriter object.
12        outputFile = File.CreateText("Values.txt");
13
14        // Write the array's contents to the file.
15        for (int index =0; index < numbers.Length; index++)
16        {
17            outputFile.WriteLine(numbers[index]);
18        }
19
20        // Close the file.
21        outputFile.Close();
22
23        // Let the user know it's done.
24        MessageBox.Show("Done");
25    }
26    catch (Exception ex)
27    {
28        // Display an error message.
29        MessageBox.Show(ex.Message);
30    }
31 }

```

The `try-catch` statement handles any file-related errors. Here is a summary of the code inside the `try` block:

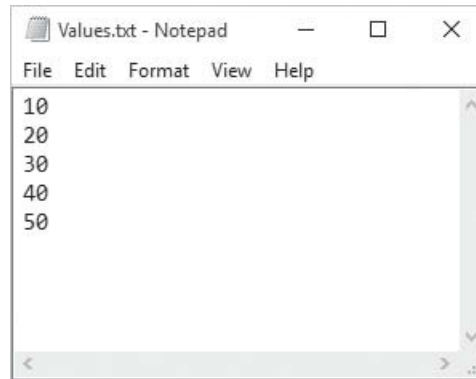
- Line 6 creates an `int` array with five elements, initialized to the values 10, 20, 30, 40, and 50.



- Line 9 declares a `StreamWriter` variable named `outputFile`. (You do not see it in this code sample, but the directive using `System.IO;` appears at the top of the file. This is required for the `StreamWriter` declaration in line 9.)
- Line 12 creates a file named `Values.txt` for writing. After this statement executes, the `outputFile` variable will reference a `StreamWriter` object that is associated with the file.
- Line 15 is the beginning of a `for` loop. The loop iterates once for each element of the array. During the loop's iterations, the `index` variable will be assigned the values 1, 2, 3, 4 and 5.
- Inside the loop, line 17 writes the array element `numbers[index]` to the file.
- Line 21 closes the file.
- Line 24 displays a message box letting the user know the operation is done.

Figure 7-13 shows the contents of the `Values.txt` file, opened in Notepad, after the `OK` button has been clicked.

**Figure 7-13** Contents of the `Values.txt` file



## Reading Values from a File and Storing Them in an Array

Reading the contents of a file into an array is also straightforward: Open the file and use a loop to read each item from the file, storing each item in an array element. The loop should iterate until either the array is filled or the end of the file is reached. For example, in the `Chap07` folder of the Student Sample Programs, you will find a project named *File To Array*. When you click the *Get Values* button, the application reads values from a file named `Values.txt` into an `int` array. The contents of the array are then displayed in a list box. The following code shows the Click event handler for the *Get Values* button.

```
1 private void getValuesButton_Click(object sender, EventArgs e)
2 {
3     try
4     {
5         // Create an array to hold items read from the file.
6         const int SIZE = 5;
7         int[] numbers = new int[SIZE];
8
9         // Counter variable to use in the loop
10        int index = 0;
```

```

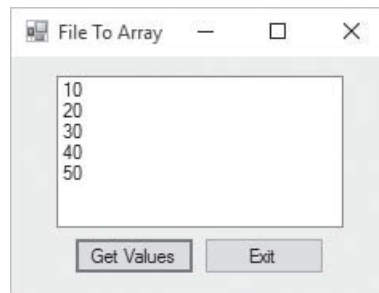
11
12     // Declare a StreamReader variable.
13     StreamReader inputFile;
14
15     // Open the file and get a StreamReader object.
16     inputFile = File.OpenText("Values.txt");
17
18     // Read the file's contents into the array.
19     while (index < numbers.Length && !inputFile.EndOfStream)
20     {
21         numbers[index] = int.Parse(inputFile.ReadLine());
22         index++;
23     }
24
25     // Close the file.
26     inputFile.Close();
27
28     // Display the array elements in the list box.
29     foreach (int value in numbers)
30     {
31         outputListBox.Items.Add(value);
32     }
33 }
34 catch (Exception ex)
35 {
36     // Display an error message.
37     MessageBox.Show(ex.Message);
38 }
39 }

```

The try-catch statement handles any file-related errors. Here is a summary of the code inside the try block:

- Lines 6 and 7 create an `int` array with five elements.
- Line 10 declares an `int` variable named `index`, initialized with the value 0. This variable will be used in a loop to hold subscript values.
- Line 13 declares a `StreamReader` variable named `inputFile`. (You do not see it in this code sample, but the directive using `System.IO;` appears at the top of the file. This is required for the `StreamReader` declaration in line 13.)
- Line 16 opens a file named `Values.txt` for reading. After this statement executes, the `inputFile` variable references a `StreamReader` object that is associated with the file.
- Line 19 is the beginning of a `while` loop that reads items from the file and assigns them to elements of the `numbers` array. Notice that the loop tests two Boolean expressions connected by the `&&` operator. The first expression is `index < numbers.Length`. The purpose of this expression is to prevent the loop from writing beyond the end of the array. When the array is full, the loop stops. The second expression is `!inputFile.EndOfStream`. The purpose of this expression is to prevent the loop from reading beyond the end of the file. When there are no more values to read from the file, the loop stops.
- Inside the loop, line 21 reads a line of text from the file, converts it to an `int`, and assigns the `int` to `numbers[index]`. Then, line 22 increments `index`.
- Line 26 closes the file.
- The `foreach` loop in lines 29–32 displays the array elements in the `outputListBox` control.

Figure 7-14 shows the application's form after the *Get Values* button has been clicked.

**Figure 7-14** The *File To Array* form

## 7.4 Passing Arrays as Arguments to Methods

**CONCEPT:** An array can be passed as an argument to a method. To pass an array, you pass the variable that references the array.

Sometimes you will want to write a method that accepts an entire array as an argument and performs an operation on the array. For example, the following code shows a method named `ShowArray`. The method accepts an array of strings as an argument and displays each element in a message box.

```

1 private void ShowArray(string[] strArray)
2 {
3     foreach (string str in strArray)
4     {
5         MessageBox.Show(str);
6     }
7 }

```

Notice in line 1 that the method has a parameter variable named `strArray` and that the parameter's data type is `string[]`. The expression `string[]` indicates that this parameter variable is a reference to a `string` array. When you call this method, you must pass a `string` array as an argument.

When you call a method and pass an array as an argument, you simply pass the variable that references the array. The following code shows an example of how the `ShowArray` method (previously shown) might be called:

```

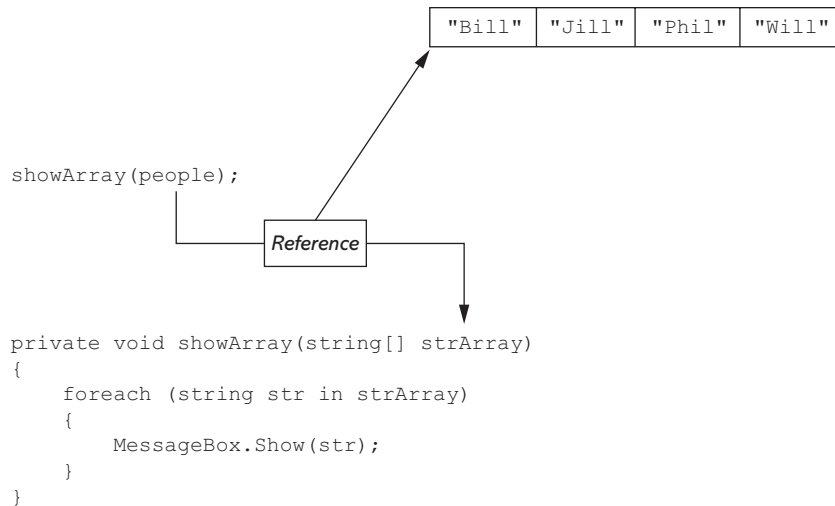
1 // Create an array of strings.
2 string[] people = { "Bill", "Jill", "Phil", "Will" };
3
4 // Pass the array to the ShowArray method.
5 ShowArray(people);

```

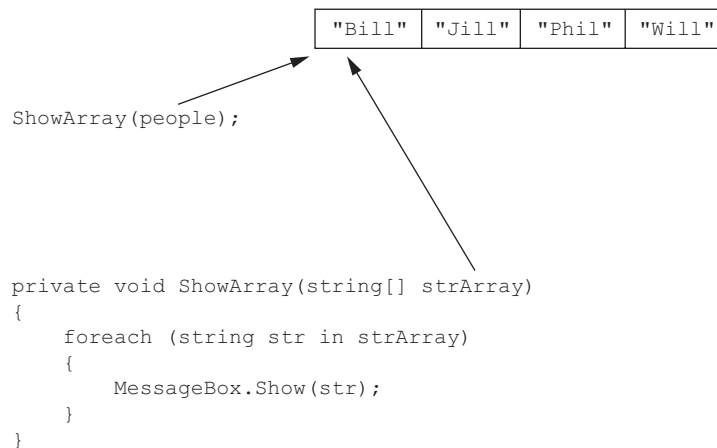
Line 2 creates an array of strings named `people` and initializes it with four strings. Line 5 calls the `ShowArray` method passing the `people` array as an argument.

Keep in mind that arrays are *always* passed by reference. When you pass an array as an argument, the thing that is passed into the parameter variable is a reference to the array. This is illustrated in Figure 7-15. As shown in the figure, the `people` variable contains a reference to an array. When the `people` variable is passed to the `ShowArray` method, the reference to the array is passed into the `strArray` parameter variable. Figure 7-16 shows that while the `ShowArray` method is executing, the `people` variable and the `strArray` parameter variable reference the same array in memory.

**Figure 7-15** An array passed as an argument

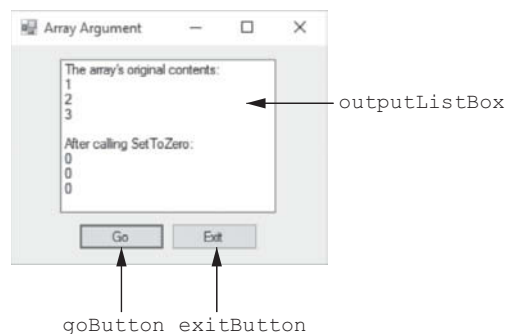


**Figure 7-16** The `people` and `strArray` variables referencing the same array



Because arrays are always passed by reference, a method that receives an array as an argument has access to the actual array (not a copy of the array). For example, in the *Chap07* folder of the Student Sample Programs, you will find a project named *Array Argument*. Figure 7-17 shows the application’s form just after the user has clicked the *Go* button.

**Figure 7-17** The *Array Argument* application



The following code shows the Click event handler for the *Go* button, and a method named `SetToZero`:

```
1 // Click event handler for the goButton control.
2 private void goButton_Click(object sender, EventArgs e)
3 {
4     // Create an int array.
5     int[] numbers = { 1, 2, 3 };
6
7     // Display the array in the list box.
8     outputListBox.Items.Add("The array's original contents:");
9     foreach (int number in numbers)
10    {
11        outputListBox.Items.Add(number);
12    }
13
14    // Pass the array to the SetToZero method.
15    SetToZero(numbers);
16
17    // Display the array in the list box again.
18    outputListBox.Items.Add("");
19    outputListBox.Items.Add("After calling SetToZero:");
20    foreach (int number in numbers)
21    {
22        outputListBox.Items.Add(number);
23    }
24 }
25
26 // The SetToZero method accepts an int array as an
27 // argument and sets its elements to 0.
28 private void SetToZero(int[] iArray)
29 {
30     for (int index = 0; index < iArray.Length; index++)
31     {
32         iArray[index] = 0;
33     }
34 }
```

Let's take a closer look at the `goButton_Click` event handler:

- Line 5 creates an `int` array named `numbers`, initialized with the values 1, 2, and 3.
- Line 8 displays the string "The array's original contents:" in the `outputListBox` control.
- The `foreach` loop in lines 9–12 displays the contents of the `numbers` array in the `outputListBox` control. Look at Figure 7-17 and notice that the array's values are 1, 2, and 3.
- Line 15 calls the `SetToZero` method, passing the `numbers` array as an argument.
- Line 18 displays a blank line in the `outputListBox` control, and line 19 displays the string "After calling `SetToZero`:".
- The `foreach` loop in lines 20–23 displays the contents of the `numbers` array in the `outputListBox` control. Look at Figure 7-17 and notice that the array's values are now 0, 0, and 0.

As you can see from Figure 7-17, the `SetToZero` method changed the values stored in the `numbers` array. Let's look at the `SetToZero` method:

- Notice in line 28 that the method accepts an `int` array as an argument. The parameter variable's name is `iArray`.

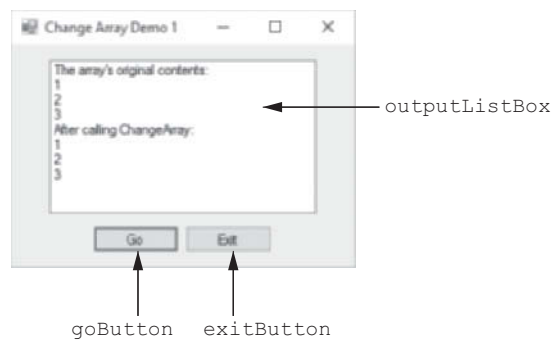
- Line 30 is the beginning of a `for` loop that steps through the array. As the loop iterates, the `index` variable is assigned the values 0, 1, 2, and so forth. The loop iterates as long as `index` is less than `iArray.Length`.
- The statement in line 32 assigns 0 to the array element `iArray[index]`.

Because the `iArray` parameter is a reference to the array that was passed as an argument, the statement in line 32 assigns 0 to an element of the `numbers` array.

## Using `ref` and `out` with Array Parameters

You saw in the previous example that arrays are always passed by reference. When you pass an array as an argument to a method, the method has direct access to the array through its parameter variable. However, the method cannot access the original reference variable that was used to pass the array. For example, in the *Chap07* folder of the Student Sample Programs, you will find a project named *Change Array 1*. Figure 7-18 shows the application's form just after the user has clicked the *Go* button. The following code shows the Click event handler for the *Go* button and a method named `ChangeArray`:

**Figure 7-18** The *Change Array Demo 1* application



```

1 private void goButton_Click(object sender, EventArgs e)
2 {
3     // Create an int array.
4     int[] numbers = { 1, 2, 3 };
5
6     // Display the numbers array's contents.
7     outputListBox.Items.Add("The array's original contents:");
8     foreach (int value in numbers)
9     {
10        outputListBox.Items.Add(value);
11    }
12
13    // Pass the numbers array to the ChangeArray method.
14    ChangeArray(numbers);
15
16    // Display the numbers array's contents.
17    outputListBox.Items.Add("After calling ChangeArray:");
18    foreach (int value in numbers)
19    {
20        outputListBox.Items.Add(value);
21    }
22 }
23
24 private void ChangeArray(int[] iArray)

```

```

25 {
26     const int NEW_SIZE = 5;
27
28     // Make iArray reference a different array.
29     iArray = new int[NEW_SIZE];
30
31     // Set the new array's elements to 99.
32     for (int index = 0; index < iArray.Length; index++)
33     {
34         iArray[index] = 99;
35     }
36 }

```

Let's take a closer look at the `goButton_Click` event handler:

- Line 4 creates an `int` array named `numbers`, initialized with the values 1, 2, and 3.
- Lines 7–11 display the array's contents in the `outputListBox` control. Look at Figure 7-18 and notice that the array's values are 1, 2, and 3.
- Line 14 calls the `ChangeArray` method, passing the `numbers` array as an argument.
- Lines 17–21 display the contents of the `numbers` array in the `outputListBox` control after the `ChangeArray` method has executed. Look at Figure 7-18 and notice that the array's values are still 1, 2, and 3. Apparently, the method did not change the array.

Let's look at the `ChangeArray` method:

- Notice in line 24 that the method accepts an `int` array as an argument. The parameter variable's name is `iArray`. Keep in mind that when we call this method in line 14, passing `numbers` as an argument, the `iArray` parameter and the `numbers` variable reference the same array in memory.
- Line 26 declares an `int` constant named `NEW_SIZE`, set to the value 5.
- Line 29 creates a new `int` array in memory with five elements. A reference to the array is assigned to the `iArray` parameter variable. As shown in Figure 7-19,

**Figure 7-19** After line 29 executes

```

private void goButton_Click(object sender, EventArgs e)
{
    // Create an int array.
    int[] numbers = { 1, 2, 3 };
    and so forth ...

    ChangeArray(numbers);

    and so forth ...
}

private void ChangeArray(int[] iArray)
{
    const int NEW_SIZE = 5;

    // Make iArray reference a different array.
    iArray = new int[NEW_SIZE];
    and so forth ...

    // Set the new array's elements to 99.
    for (int index = 0; index < iArray.Length; index++)
    {
        iArray[index] = 99;
    }
}

```

this causes the `iArray` parameter variable to no longer reference the array that was passed as an argument. Instead, the `iArray` parameter references the new array.

- The `for` loop in lines 32–35 assigns the value 99 to each element of array referenced by `iArray`. This does not affect the `numbers` array.

When you use either the `ref` or `out` keywords with an array parameter, the receiving method not only has access to the array, but it also has access to the reference variable that was used to pass the array. For example, the *Change Array 2* project, in the *Chap07* folder of the Student Sample Programs, is identical to the *Change Array 1* project, except that the `iArray` parameter is declared with the `ref` keyword in the `ChangeArray` method. The following code shows the Click event handler for the *Go* button, and the `ChangeArray` method.

```

1 private void goButton_Click(object sender, EventArgs e)
2 {
3     // Create an int array.
4     int[] numbers = { 1, 2, 3 };
5
6     // Display the number array's contents.
7     outputListBox.Items.Add("The array's original contents:");
8     foreach (int value in numbers)
9     {
10        outputListBox.Items.Add(value);
11    }
12
13    // Pass the number array to the ChangeArray method.
14    ChangeArray(ref numbers);
15
16    // Display the number array's contents.
17    outputListBox.Items.Add("After calling ChangeArray:");
18    foreach (int value in numbers)
19    {
20        outputListBox.Items.Add(value);
21    }
22 }
23
24 private void ChangeArray(ref int[] iArray)
25 {
26     const int NEW_SIZE = 5;
27
28     // Make iArray reference a different array.
29     iArray = new int[NEW_SIZE];
30
31     // Set the new array's elements to 99.
32     for (int index = 0; index < iArray.Length; index++)
33     {
34         iArray[index] = 99;
35     }
36 }

```

Notice that in line 24 the `iArray` parameter is declared with the `ref` keyword, and in line 14 the `ref` keyword is used to pass `numbers` as an argument to the `ChangeArray` method. In this code, the `iArray` parameter refers to the `numbers` variable. Anything that is done to the `iArray` parameter is actually done to the `numbers` variable. Figure 7-20 shows how line 29 causes the `numbers` variable to reference the new five-element array.



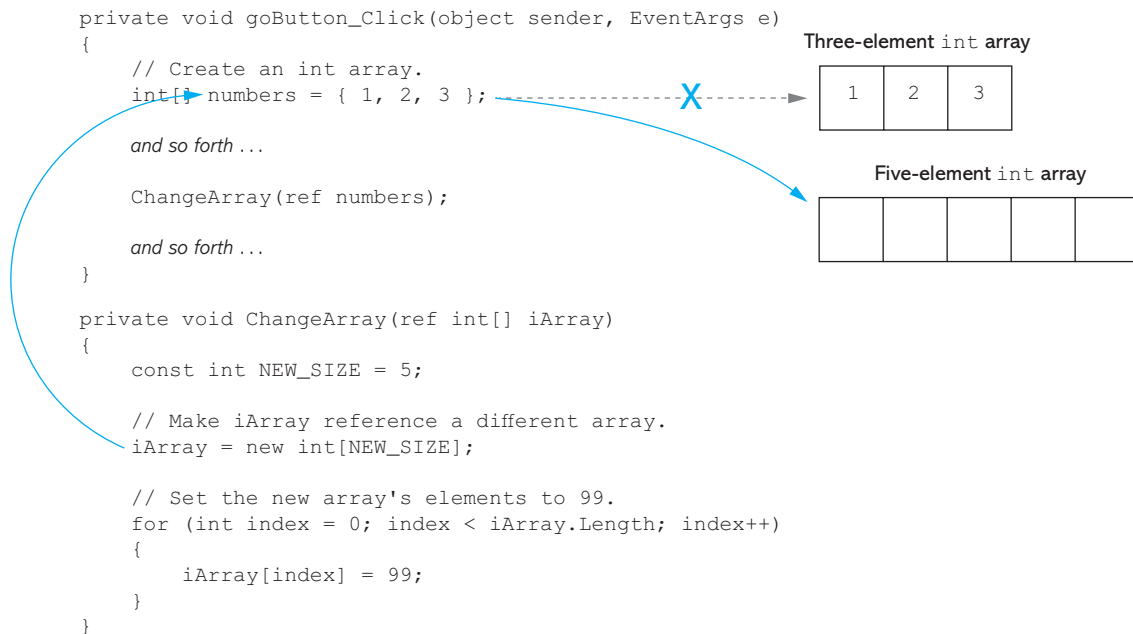
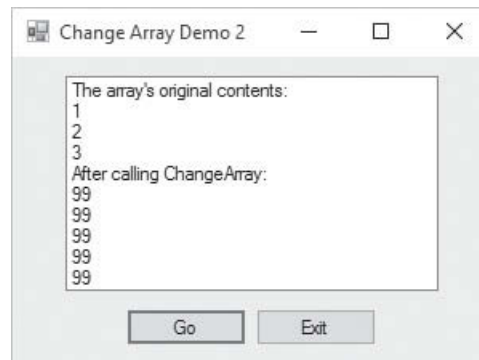
**Figure 7-20** After line 29 executes in the *Change Array 2* application

Figure 7-21 shows the application's form just after the user has clicked the *Go* button. Notice from the program's output that after the `ChangeArray` method has been called, the `numbers` variable references a five-element array, and each element's value is 99.

**Figure 7-21** The *Change Array 1* application

## Checkpoint

- 7.11 When you pass an array as an argument, what is passed into the parameter variable?
- 7.12 Does a method that receives an array as an argument have access to the actual array or only a copy of the array?
- 7.13 What is the result when you use either the `ref` or `out` keyword with an array parameter?

## 7.5

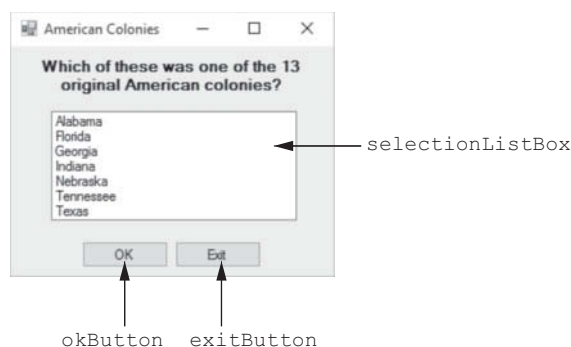
## Some Useful Array Algorithms

## The Sequential Search

Programs commonly need to search for data that is stored in an array. Various techniques known as **search algorithms** have been developed to locate a specific item in a larger collection of data, such as an array. In this section, we discuss the simplest of all search algorithms—the sequential search. The **sequential search algorithm** uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for and stops when the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm unsuccessfully searches to the end of the array.

Let's look at an example. In the *Chap07* folder of the Student Sample Programs, you will find a project named *American Colonies*. The application is a game that tests your knowledge of U.S. history. As shown in Figure 7-22, the application's form displays a list of states in a ListBox control. Only one of the states shown in the ListBox was an original American colony. You select the state that you believe was a colony and click the OK button to see if you were correct.

**Figure 7-22** The *American Colonies* application



The following code is taken from the application. It shows a method named `SequentialSearch` and the Click event handler for the OK button.

```

1 // The SequentialSearch method searches a string array
2 // for a specified value. If the value is found, its
3 // position is returned. Otherwise, -1 is returned.
4 private int SequentialSearch(string[] sArray, string value)
5 {
6     bool found = false; // Flag indicating search results
7     int index = 0;      // Used to step through the array
8     int position = -1;  // Position of value, if found
9
10    // Search the array.
11    while (!found && index < sArray.Length)
12    {
13        if (sArray[index] == value)
14        {
15            found = true;
16            position = index;
17        }
18
19        index++;
20    }

```

```

21
22     // Return
23     return position;
24 }
25
26 private void okButton_Click(object sender, EventArgs e)
27 {
28     string selection;    // To hold the user's selection
29
30     // Create an array with the colony names.
31     string[] colonies = { "Delaware", "Pennsylvania", "New Jersey",
32                          "Georgia", "Connecticut", "Massachusetts",
33                          "Maryland", "South Carolina", "New Hampshire",
34                          "Virginia", "New York", "North Carolina",
35                          "Rhode Island" };
36
37     if (selectionListBox.SelectedIndex != -1)
38     {
39         // Get the selected item.
40         selection = selectionListBox.SelectedItem.ToString();
41
42         // Determine if the item is in the array.
43         if (SequentialSearch(colonies, selection) != -1)
44         {
45             MessageBox.Show("Yes, that was one of the colonies.");
46         }
47         else
48         {
49             MessageBox.Show("No, that was not one of the colonies.");
50         }
51     }
52 }

```

The `SequentialSearch` method, which begins in line 4, searches a `string` array for a specified value. It accepts a `string` array and a `string` search value as arguments. If the search value is found in the array, the method returns the value's subscript. If the search value is not found in the array, the method returns `-1`. Let's take a closer look at the method:

- Line 6 declares a `bool` variable named `found`. The `found` variable is used as a flag. Setting `found` to `false` indicates that the search value has not been found. Setting `found` to `true` indicates that the search value has been found. Notice that `found` is initialized with `false`.
- Line 7 declares an `int` variable named `index` that will be used to step through the elements of the array. Notice that `index` is initialized with the value 0.
- Line 8 declares an `int` variable named `position`. If the search value is found in the array, we save its subscript in the `position` variable. Notice that the `position` variable is initialized with the value `-1`.
- The `while` loop that begins in line 11 searches the array for the specified value. It iterates as long as `found` is not `true` and `index` is less than the array's length.
- The `if` statement in line 13 determines whether `sArray[index]` is equal to `value`. If this is true, then the search value has been found in the array. In that case, line 15 sets `found` to `true`, and line 16 assigns `index` to `position`.
- Line 19 increments `index`.
- When the loop finishes, line 23 returns the value of the `position` variable. If the search value was found in the array, the `position` variable will contain the value's subscript. If the search value was not found in the array, the `position` variable will still be set to `-1`.

The Click event handler for the OK button begins in line 26. Let's take a closer look at the event handler's code:

- Line 28 declares a `string` variable named `selection`. This variable will hold the item that is selected from the `ListBox` control.
- Lines 31–35 declare a `string` array named `colonies`. The array is initialized with the names of the U.S. colonies.
- The `if` statement that begins in line 37 determines whether an item has been selected in the `selectionListBox` control. If an item has been selected, the following actions take place:
  - Line 40 gets the selected item and assigns it to the `selection` variable.
  - The `if` statement in line 43 calls the `SequentialSearch` method, passing the `colonies` array and the `selection` variable as arguments. If the value of the `selection` variable is found in the `colonies` array, the method returns a value other than `-1`, and line 45 displays a message box informing the user that the selected item was one of the colonies. However, if the value of the `selection` variable is not found in the `colonies` array, the method will return `-1`, and line 49 displays a message box informing the user that the selected item was not one of the colonies.

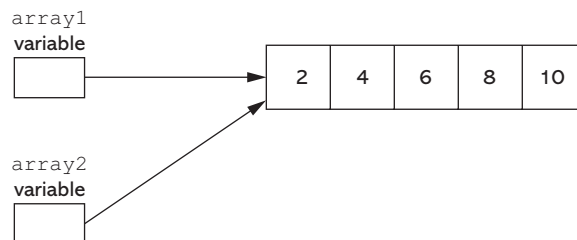
## Copying an Array

Because an array is an object, there is a distinction between an array and the variable that references it. The array and the reference variable are two separate entities. This is important to remember when you wish to copy the contents of one array to another. You might be tempted to write something like the following code, thinking that you are copying an array:

```
int[] array1 = { 2, 4, 6, 8, 10 };
int[] array2 = array1; // This does not copy array1.
```

The first statement creates an array referenced by the `array1` variable. The second statement assigns `array1` to `array2`. This does not make a copy of the array referenced by `array1`. Rather, it assigns the reference that is in `array1` to `array2`. After this statement executes, both the `array1` and `array2` variables will reference the same array. This type of assignment operation is called a **reference copy**. Only a reference to the array object is copied, not the contents of the array object. This is illustrated in Figure 7-23.

**Figure 7-23** Both `array1` and `array2` referencing the same array



If you want to make a copy of an array, you must create the second array in memory and then copy the individual elements of the first array to the second. This is usually best done with a loop, such as the following:

```
1 const int SIZE = 5;
2 int[] firstArray = { 5, 10, 15, 20, 25 };
3 int[] secondArray = new int[SIZE];
4
```

```
5 for (int index = 0; index < firstArray.length; index++)
6 {
7     secondArray[index] = firstArray[index];
8 }
```

The loop in this code copies each element of `firstArray` to the corresponding element of `secondArray`.

## Comparing Arrays

You cannot use the `==` operator to compare two array reference variables and determine whether the arrays are equal. For example, the following code appears to compare two arrays, but in reality it does not:

```
1 int[] firstArray = { 5, 10, 15, 20, 25 };
2 int[] secondArray = { 5, 10, 15, 20, 25 };
3
4 if (firstArray == secondArray) // This is a mistake.
5 {
6     MessageBox.Show("The arrays are the same.");
7 }
8 else
9 {
10    MessageBox.Show("The arrays are not the same.");
11 }
```

When you use the `==` operator with reference variables, the operator compares the references that the variables contain, not the contents of the objects referenced by the variables. Because the `firstArray` and `secondArray` variables in this example reference different objects in memory, the result of the Boolean expression `firstArray == secondArray` is `false`, and the code reports that the arrays are not the same.

To compare the contents of two arrays, you must compare the elements of the two arrays. For example, look at the following code:

```
1 int[] firstArray = { 2, 4, 6, 8, 10 };
2 int[] secondArray = { 2, 4, 6, 8, 10 };
3 boolean arraysEqual = true; // Flag variable
4 int index = 0; // To hold array subscripts
5
6 // First determine whether the arrays are the same size.
7 if (firstArray.length != secondArray.length)
8 {
9     arraysEqual = false;
10 }
11
12 // Next determine whether the elements contain the same data.
13 while (arraysEqual && index < firstArray.length)
14 {
15     if (firstArray[index] != secondArray[index])
16     {
17         arraysEqual = false;
18     }
19     index++;
20 }
21
22 if (arraysEqual)
23 {
24     MessageBox.Show("The arrays are equal.");
25 }
26 else
```

```

27 {
28     MessageBox.Show("The arrays are not equal.");
29 }

```

This code determines whether `firstArray` and `secondArray` (declared in lines 1 and 2) contain the same values. A Boolean flag variable, `arraysEqual`, is declared and initialized to `true` in line 3. The `arraysEqual` variable is used to signal whether the arrays are equal. Another variable, `index`, is declared and initialized to 0 in line 4. The `index` variable is used in a loop to step through the arrays.

First, the `if` statement in line 7 determines whether the two arrays are the same length. If they are not the same length, then the arrays cannot be equal, so the flag variable `arraysEqual` is set to `false` in line 9. Then a `while` loop begins in line 13. The loop executes as long as `arraysEqual` is `true` and the `index` variable is less than `firstArray.Length`. During each iteration, it compares a different set of corresponding elements in the arrays. When it finds two corresponding elements that have different values, the flag variable `arraysEqual` is set to `false`.

After the loop finishes, an `if` statement examines the `arraysEqual` variable in line 22. If the variable is `true`, then the arrays are equal and a message indicating so is displayed in line 24. Otherwise, they are not equal, so a different message is displayed in line 28.

## Totaling the Values in an Array

To calculate the total of the values in a numeric array, you use a loop with an accumulator variable. First, the accumulator is initialized with 0. Then, the loop steps through the array, adding the value of each array element to the accumulator.

```

1 // Create an int array.
2 int[] units = { 2, 4, 6, 8, 10 };
3
4 // Declare and initialize an accumulator variable.
5 int total = 0;
6
7 // Step through the array, adding each element to
8 // the accumulator.
9 for (int index = 0; index < units.Length; index++)
10 {
11     total += units[index];
12 }
13
14 // Display the total.
15 MessageBox.Show("The total is " + total);

```

## Averaging the Values in an Array

The first step in calculating the average of all the values in a numeric array is to get the total of the values. The second step is to divide the total by the number of elements in the array. The following code shows an example:

```

1 // Create an array.
2 double[] scores = { 92.5, 81.6, 65.7, 72.8 }
3
4 // Declare and initialize an accumulator variable.
5 double total = 0.0;
6
7 // Declare a variable to hold the average.

```

```
8 double average;
9
10 // Step through the array, adding each element to
11 // the accumulator.
12 for (int index = 0; index < scores.Length; index++)
13 {
14     total += scores[index];
15 }
16
17 // Calculate the average.
18 average = total / scores.Length;
19
20 // Display the average.
21 MessageBox.Show("The average is " + average);
```

When this code finishes, the `average` variable will contain the average of the values in the `scores` array. Notice that the last statement, which divides `total` by `scores.Length`, is not inside the loop. This statement should execute only once, after the loop has finished its iterations.

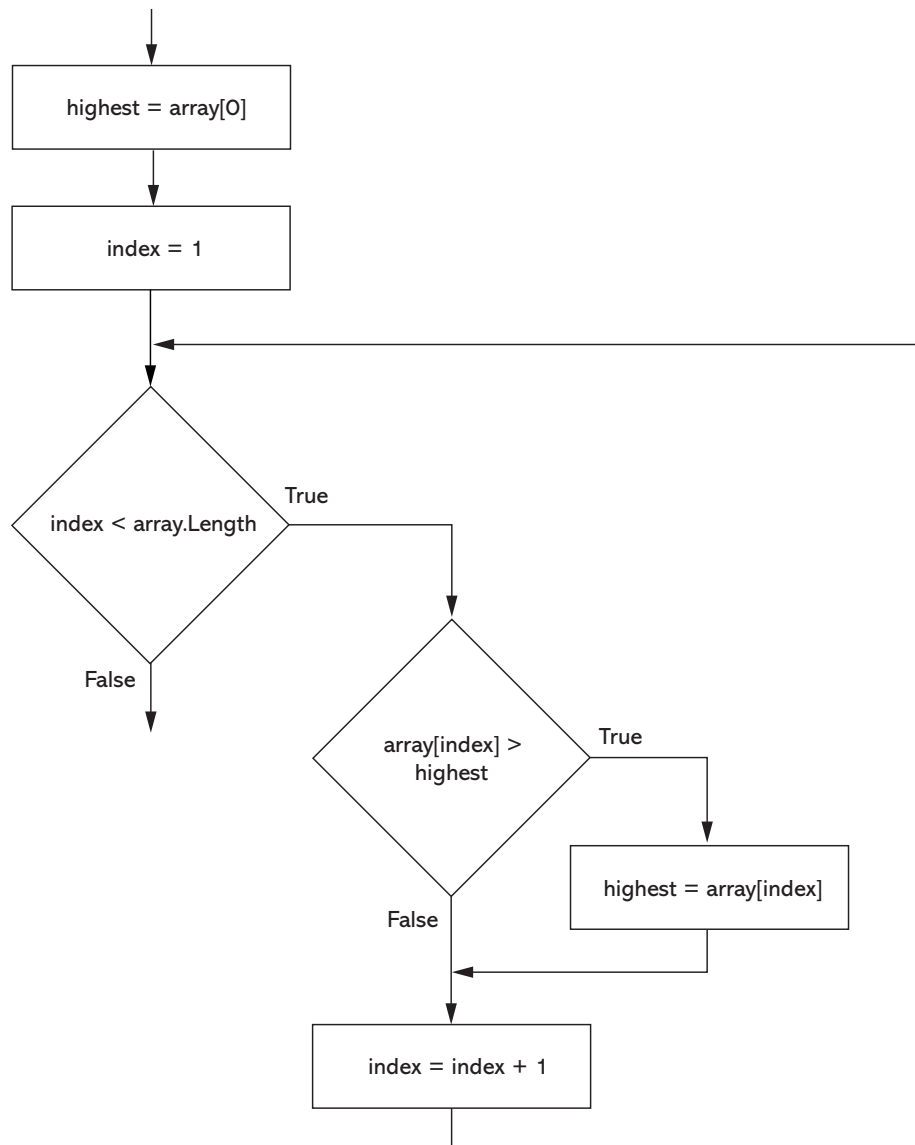
## Finding the Highest and Lowest Values in an Array

Some programming tasks require you to find the highest value in a set of data. Examples include programs that report the highest sales amount for a given time period, the highest test score in a set of test scores, the highest temperature for a given set of days, and so forth.

The algorithm for finding the highest value in an array works like this: You create a variable to hold the highest value (the following example names this variable `highest`). Then, you assign the value at element 0 to the `highest` variable. Next, you use a loop to step through the rest of the array elements, beginning at element 1. Each time the loop iterates, it compares an array element to the `highest` variable. If the array element is greater than the `highest` variable, then the value in the array element is assigned to the `highest` variable. When the loop finishes, the `highest` variable will contain the highest value in the array. The flowchart in Figure 7-24 illustrates this logic.

The following code demonstrates this algorithm:

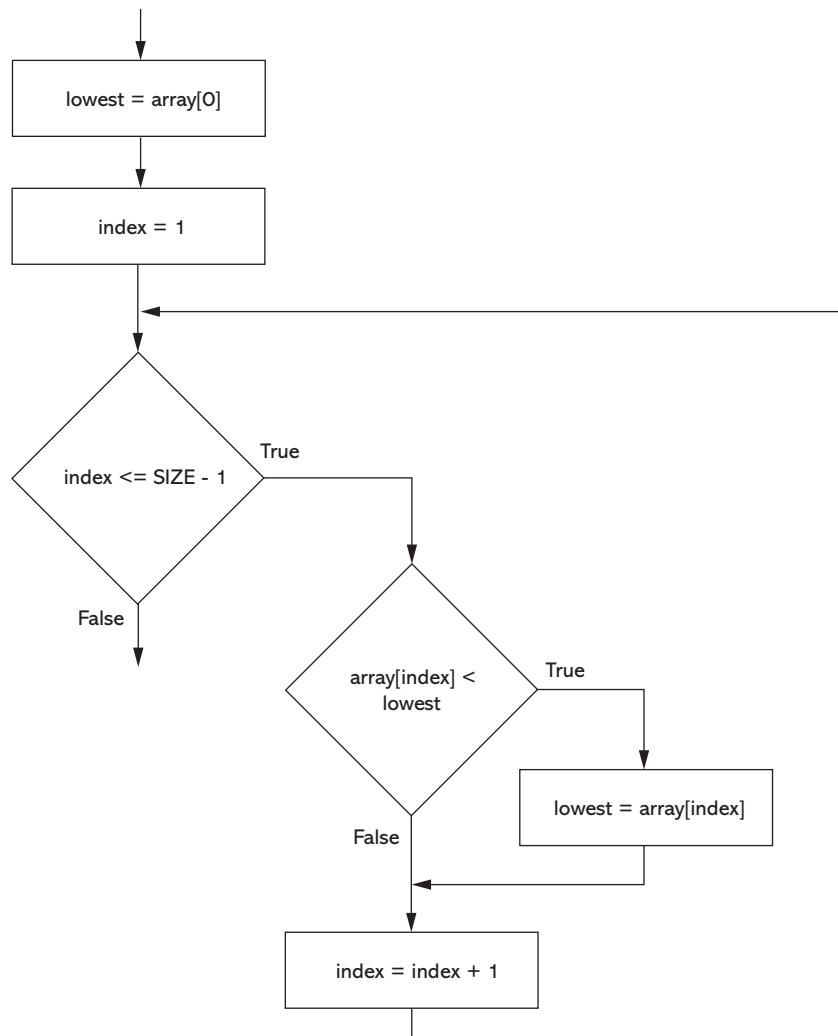
```
1 // Create an array.
2 int[] numbers = { 8, 1, 12, 6, 2 };
3
4 // Declare a variable to hold the highest value, and
5 // initialize it with the first value in the array.
6 int highest = numbers[0];
7
8 // Step through the rest of the array, beginning at
9 // element 1. When a value greater than highest is found,
10 // assign that value to highest.
11 for (int index = 1; index < numbers.Length; index++)
12 {
13     if (numbers[index] > highest)
14     {
15         highest = numbers[index];
16     }
17 }
18
19 // Display the highest value.
20 MessageBox.Show("The highest value is " + highest);
```

**Figure 7-24** Flowchart for finding the highest value in an array

In some programs, you are more interested in finding the lowest value than the highest value in a set of data. For example, suppose you are writing a program that stores several players' golf scores in an array and you need to find the best score. In golf, the lower the score the better, so you need an algorithm that finds the lowest value in the array.

The algorithm for finding the lowest value in an array is very similar to the algorithm for finding the highest score. It works like this: You create a variable to hold the lowest value (the following example names this variable `lowest`). Then, you assign the value at element 0 to the `lowest` variable. Next, you use a loop to step through the rest of the array elements, beginning at element 1. Each time the loop iterates, it compares an array element to the `lowest` variable. If the array element is less than the `lowest` variable, then the value in the array element is assigned to the `lowest` variable. When the loop finishes, the `lowest` variable contains the lowest value in the array. The flowchart in Figure 7-25 illustrates this logic.



**Figure 7-25** Flowchart for finding the lowest value in an array

The following code demonstrates this algorithm:

```

1 // Create an array.
2 int[] numbers = { 8, 1, 12, 6, 2 };
3
4 // Declare a variable to hold the lowest value, and
5 // initialize it with the first value in the array.
6 int lowest = numbers[0];
7
8 // Step through the rest of the array, beginning at
9 // element 1. When a value less than lowest is found,
10 // assign that value to lowest.
11 for (int index = 1; index < numbers.Length; index++)
12 {
13     if (numbers[index] < lowest)
14     {
15         lowest = numbers[index];
16     }
17 }
18
19 // Display the lowest value.
20 MessageBox.Show("The lowest value is " + lowest);

```

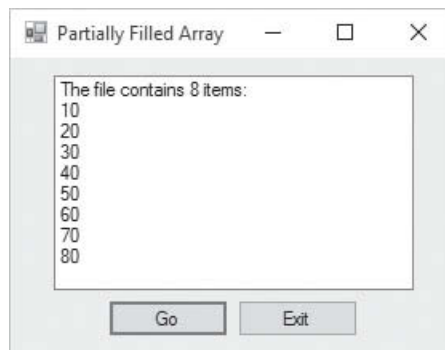
## Partially Filled Arrays

Sometimes you need to store a series of items in an array, but you do not know the number of items in the series. As a result, you do not know the exact number of elements needed for the array. One solution is to make the array large enough to hold the largest possible number of items. This can lead to another problem, however. If the actual number of items stored in the array is less than the number of elements, the array will be only partially filled. When you process a partially filled array, you must process only the elements that contain valid data items.

A partially filled array is normally used with an accompanying integer variable that holds the number of items that are actually stored in the array. If the array is empty, then 0 is stored in this variable because there are no items in the array. Each time an item is added to the array, the variable is incremented. When code steps through the array's elements, the value of this variable is used instead of the array's size to determine the maximum subscript.

For example, in the *Chap07* folder of the Student Sample Programs, you will find a project named *Partially Filled Array*. When you click the *Go* button, the application reads up to 100 values from a file named *Values.txt* and stores them in a 100-element `int` array. If the file contains fewer than 100 values, the application will partially fill the array. The contents of the array are then displayed in a list box. Figure 7-26 shows the application's form just after the user has clicked the *Go* button. The following code shows the Click event handler for the *Get Values* button.

**Figure 7-26** The *Partially Filled Array* application



```

1 private void goButton_Click(object sender, EventArgs e)
2 {
3     try
4     {
5         // Create an array to hold items read from the file.
6         const int SIZE = 100;
7         int[] numbers = new int[SIZE];
8
9         // Variable to hold the number of items stored in
10        // the array
11        int count = 0;
12
13        // Declare a StreamReader variable.
14        StreamReader inputFile;
15
16        // Open the file and get a StreamReader object.
17        inputFile = File.OpenText("Values.txt");
18
19        // Read the file's contents into the array until the

```

```
20         // end of the file is reached, or the array is full.
21         while (!inputFile.EndOfStream && count < numbers.Length)
22         {
23             // Read the next item from the file.
24             numbers[count] = int.Parse(inputFile.ReadLine());
25
26             // Increment count.
27             count++;
28         }
29
30         // Close the file.
31         inputFile.Close();
32
33         // Display the array elements in the list box.
34         outputListBox.Items.Add("The file contains " + count +
35             " items:");
36
37         for (int index = 0; index < count; index++)
38         {
39             outputListBox.Items.Add(numbers[index]);
40         }
41     }
42     catch (Exception ex)
43     {
44         // Display an error message.
45         MessageBox.Show(ex.Message);
46     }
47 }
```

Let's examine the code in detail:

- Line 3 is the beginning of a `try-catch` statement that handles any errors that might result while reading data from the file.
- Line 6 declares a constant, `SIZE`, initialized with the value 100.
- Line 7 declares an `int` array named `numbers` using `SIZE` as the size declarator. As a result, the values array has 100 elements.
- Line 11 declares an `int` variable named `count`, which holds the number of items that are stored in the `numbers` array. Notice that `count` is initialized with 0 because there are no values stored in the array.
- Line 14 declares a `StreamReader` variable named `inputFile`. (You do not see it in this code sample, but the directive using `System.IO`; appears at the top of the file. This is required for the `StreamReader` declaration in line 14.)
- Line 17 opens a file named `Values.txt` for reading. After this statement executes, the `inputFile` variable references a `StreamReader` object that is associated with the file.
- Line 21 is the beginning of a `while` loop that reads items from the file and assigns them to elements of the `numbers` array. Notice that the loop tests two Boolean expressions connected by the `&&` operator. The first expression is `!inputFile.EndOfStream`. The purpose of this expression is to prevent the loop from reading beyond the end of the file. When there are no more values to read from the file, the loop stops. The second expression is `count < numbers.Length`. The purpose of this expression is to prevent the loop from writing beyond the end of the array. When the array is full, the loop will stop.
- Inside the loop, line 24 reads a line of text from the file, converts it to an `int`, and assigns the `int` to `numbers[index]`.
- Then, line 27 increments the `count` variable. Each time a number is assigned to an array element, the `count` variable is incremented. As a result, the `count` variable holds the number of items that are stored in the array.

- Line 31 closes the file.
- The `for` loop in lines 37–40 displays the array elements in the `outputListBox` control. Rather than stepping through all the elements in the array, however, the loop steps through only the elements that contain values. Notice that the loop iterates as long as `index` is less than `count`. Because `count` contains the number of items stored in the array, the loop stops when the element containing the last valid value has been displayed.

Now that you’ve seen several algorithms for processing the contents of an array, you should practice writing some of them yourself. Tutorial 7-2 takes you through the process of writing an application that reads data from a file into an `int` array and then determines the highest, lowest, and average values in the array.



## Tutorial 7-2: Processing an Array

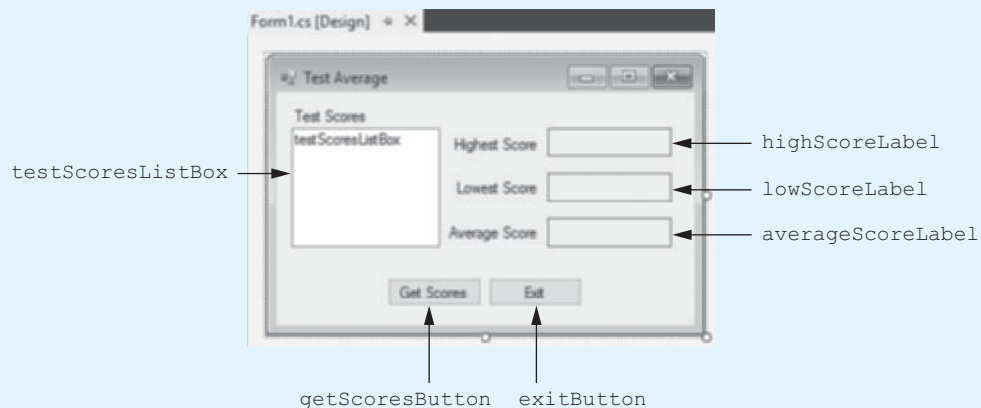


VideoNote

### Tutorial 7-2: Processing an Array

In this tutorial, you complete an application that reads five test scores from a file and stores the test scores in an array. The application displays the test scores as well as the highest score, the lowest score, and the average score. Figure 7-27 shows the application’s form, which has already been created for you. A set of five test scores is stored in a file named `TestScores.txt`, which has also been created for you.

**Figure 7-27** The *Test Average* application’s form



**Step 1:** Start Visual Studio. Open the project named *Test Average* in the *Chap07* folder of the Student Sample Programs.

**Step 2:** Open the `Form1` form’s code in the code editor. Insert the `using System.IO;` directive shown in line 10 of Program 7-2 at the end of this tutorial. This statement is necessary because we will be using the `StreamReader` class, and it is part of the `System.IO` namespace in the .NET Framework.

**Step 3:** With the code editor still open, type the comments and code for the `Average` method, shown in lines 21–40 of Program 7-2. The purpose of the `Average` method is to accept an `int` array as an argument and return the average of the values in the array. This method uses an algorithm similar to the array averaging you saw earlier in this chapter.

**Step 4:** Type the comments and code for the `Highest` method, shown in lines 42–63 of Program 7-2. The purpose of the `Highest` method is to accept an `int` array as

an argument and return the highest value in the array. This method uses an algorithm similar to the algorithm that you saw earlier in this chapter for finding the highest value in an array.

**Step 5:** Type the comments and code for the `Lowest` method, shown in lines 65–86 of Program 7-2. The purpose of the `Lowest` method is to accept an `int` array as an argument and return the lowest value in the array. This method uses an algorithm similar to the algorithm that you saw earlier in this chapter for finding the lowest value in an array.

**Step 6:** Now you create the Click event handlers for the Button controls. Switch back to the *Designer* and double-click the `getScoresButton` control. This opens the code editor, and you will see an empty event handler named `getScoresButton_Click`. Complete the `getScoresButton_Click` event handler by typing the code shown in lines 90–134 in Program 7-2. Let's review this code:

**Line 90:** This is the beginning of a `try-catch` statement that handles any exceptions that are thrown while reading and processing data from the file. If an exception occurs in the `try` block (lines 92–128), the program jumps to the `catch` block, and line 133 displays an error message.

**Lines 93–99:** The following declarations appear in these lines:

- `SIZE`—a constant, set to 5, for the number of test scores
- `scores`—an `int` array that holds the test scores
- `index`—an `int` variable, initialized to 0, that is used in a loop to step through the elements of the scores array
- `highestScore`—an `int` that holds the highest score
- `lowestScore`—an `int` that holds the lowest score
- `averageScore`—a `double` that holds the average score
- `inputFile`—a variable that references the `StreamReader` object that is used to read data from the file

**Line 102:** After this statement executes, the `TestScores.txt` file will be opened for reading, and the `inputFile` variable will reference a `StreamReader` object that is associated with the file.

**Line 105:** This is the beginning of a `while` loop that iterates as long as the end of the `TestScores.txt` file has not been reached and as long as `index` is less than `scores.Length`. (Recall that `index` starts with the value 0.)

**Line 107:** This statement reads a line of text from the file and assigns it to the array element `scores[index]`.

**Line 108:** This statement increments the `index` variable.

**Line 112:** This statement closes the `TestScores.txt` file.

**Lines 115–118:** This `foreach` loop displays the contents of the `scores` array in the `testScoresListBox` control.


**Line 121:** This statement calls the `Highest` method, passing the `scores` array as an argument. The method returns the highest value in the array, which is assigned to the `highestScore` variable.

**Line 122:** This statement calls the `Lowest` method, passing the `scores` array as an argument. The method returns the lowest value in the array, which is assigned to the `lowestScore` variable.

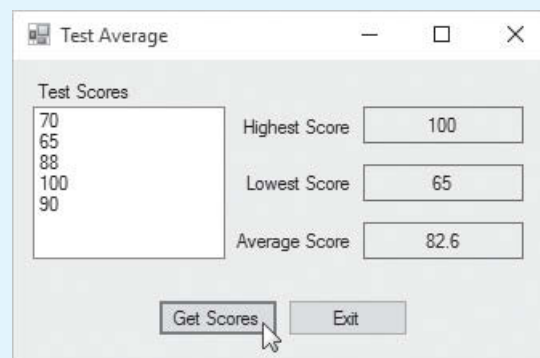
**Line 123:** This statement calls the `Average` method, passing the `scores` array as an argument. The method returns the average of the values in the array, which is assigned to the `averageScore` variable.

**Lines 126–128:** These statements display the highest score, lowest score, and average score.

**Step 7:** Switch your view back to the *Designer* and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the `exitButton_Click` event handler by typing the code shown in lines 139–140 in Program 7-2.

**Step 8:** Save the project. Then, press `F5` on the keyboard or click the *Start Debugging* button (  ) on the toolbar to compile and run the application. When the application runs, click the *Get Scores* button. This should display a set of test scores in `ListBox` as well as the highest, lowest, and average of the test scores, as shown in Figure 7-28. Click the *Exit* button to exit the application.

**Figure 7-28** The *Test Average* application



**Program 7-2** Completed code for `Form1` in the *Test Average* application

```

1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10 using System.IO;
11
12 namespace Test_Average
13 {
14     public partial class Form1 : Form
15     {
16         public Form1()
17         {
18             InitializeComponent();
19         }
20
21         // The Average method accepts an int array argument
22         // and returns the Average of the values in the array.
23         private double Average(int[] iArray)
24         {
25             int total = 0; // Accumulator, initialized to 0
26             double average; // To hold the average

```

```
27
28     // Step through the array, adding each element to
29     // the accumulator.
30     for (int index = 0; index < iArray.Length; index++)
31     {
32         total += iArray[index];
33     }
34
35     // Calculate the average.
36     average = (double) total / iArray.Length;
37
38     // Return the average.
39     return average;
40 }
41
42 // The Highest method accepts an int array argument
43 // and returns the highest value in that array.
44 private int Highest(int[] iArray)
45 {
46     // Declare a variable to hold the highest value, and
47     // initialize it with the first value in the array.
48     int highest = iArray[0];
49
50     // Step through the rest of the array, beginning at
51     // element 1. When a value greater than highest is found,
52     // assign that value to highest.
53     for (int index = 1; index < iArray.Length; index++)
54     {
55         if (iArray[index] > highest)
56         {
57             highest = iArray[index];
58         }
59     }
60
61     // Return the highest value.
62     return highest;
63 }
64
65 // The Lowest method accepts an int array argument
66 // and returns the lowest value in that array.
67 private int Lowest(int[] iArray)
68 {
69     // Declare a variable to hold the lowest value, and
70     // initialize it with the first value in the array.
71     int lowest = iArray[0];
72
73     // Step through the rest of the array, beginning at
74     // element 1. When a value less than lowest is found,
75     // assign that value to lowest.
76     for (int index = 1; index < iArray.Length; index++)
77     {
78         if (iArray[index] < lowest)
79         {
80             lowest = iArray[index];
81         }
82     }
83
84     // Return the lowest value.
85     return lowest;
```

```

86     }
87
88     private void getScoresButton_Click(object sender, EventArgs e)
89     {
90         try
91         {
92             // Local variables
93             const int SIZE = 5;           // Number of tests
94             int[] scores = new int[SIZE]; // Array of test scores
95             int index = 0;                // Loop counter
96             int highestScore;            // To hold the highest score
97             int lowestScore;             // To hold the lowest score
98             double averageScore;         // To hold the average score
99             StreamReader inputFile;       // For file input
100
101             // Open the file and get a StreamReader object.
102             inputFile = File.OpenText("TestScores.txt");
103
104             // Read the test scores into the array.
105             while (!inputFile.EndOfStream && index < scores.Length)
106             {
107                 scores[index] = int.Parse(inputFile.ReadLine());
108                 index++;
109             }
110
111             // Close the file.
112             inputFile.Close();
113
114             // Display the test scores.
115             foreach (int value in scores)
116             {
117                 testScoresListBox.Items.Add(value);
118             }
119
120             // Get the highest, lowest, and average scores.
121             highestScore = Highest(scores);
122             lowestScore = Lowest(scores);
123             averageScore = Average(scores);
124
125             // Display the values.
126             highScoreLabel.Text = highestScore.ToString();
127             lowScoreLabel.Text = lowestScore.ToString();
128             averageScoreLabel.Text = averageScore.ToString("n1");
129         }
130         catch (Exception ex)
131         {
132             // Display an error message.
133             MessageBox.Show(ex.Message);
134         }
135     }
136
137     private void exitButton_Click(object sender, EventArgs e)
138     {
139         // Close the form.
140         this.Close();
141     }
142 }
143 }

```



## 7.6

## Advanced Algorithms for Sorting and Searching Arrays

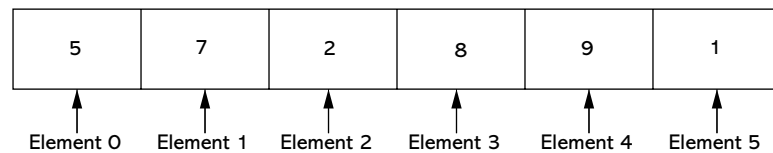
**CONCEPT:** A sorting algorithm is used to arrange data into some order. A search algorithm is a method of locating a specific item in a larger collection of data. The selection sort and the binary search are popular sorting and searching algorithms.

### The Selection Sort Algorithm

Often the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Product codes could be sorted so all the products of the same color are stored together. In this section, we explore how to write your own sorting algorithm. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order. The algorithm that we explore is called the selection sort.

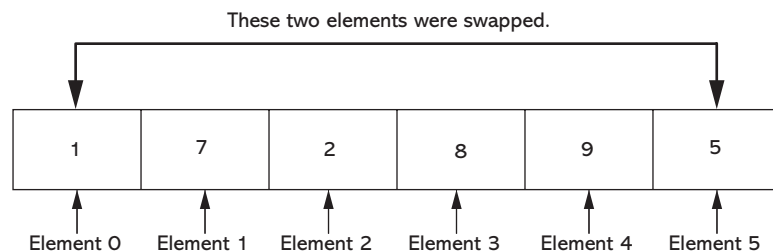
The **selection sort** works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all the elements have been placed in their proper order. Let's see how the selection sort works when arranging the elements of the array in Figure 7-29.

**Figure 7-29** Values in an array

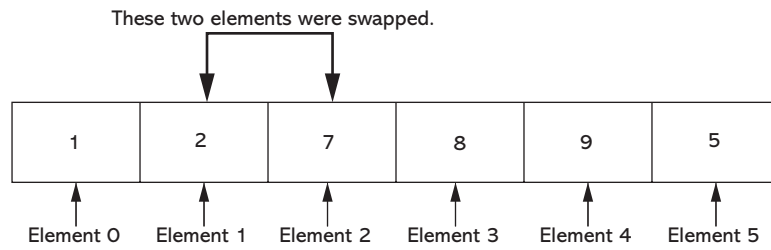


The selection sort scans the array, starting at element 0, and locates the element with the smallest value. Then, the contents of this element are swapped with the contents of element 0. In this example, the 1 stored in element 5 is swapped with the 5 stored in element 0. After the swap, the array appears as shown in Figure 7-30.

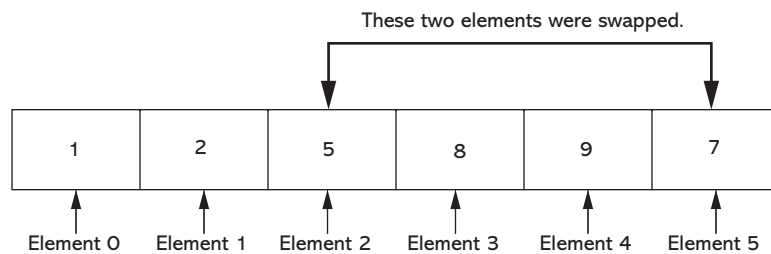
**Figure 7-30** Values in the array after the first swap



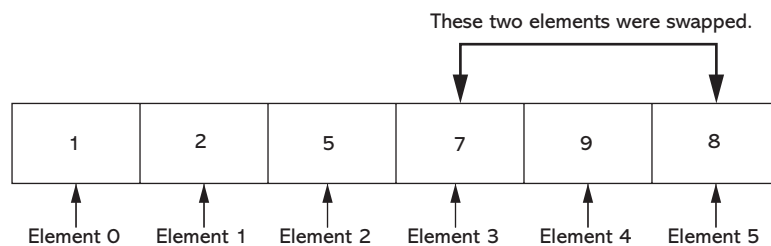
Then, the algorithm repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. This time, the algorithm begins the scan at element 1. In this example, the value in element 2 is swapped with the value in element 1. Then, the array appears as shown in Figure 7-31.

**Figure 7-31** Values in the array after the second swap

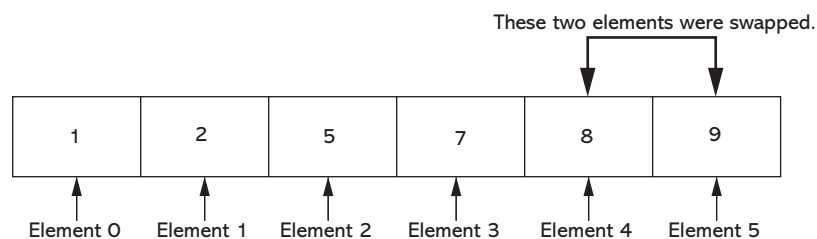
Once again, the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value. This element's value is swapped with that of element 2, causing the array to appear as shown in Figure 7-32.

**Figure 7-32** Values in the array after the third swap

Next, the scanning begins at element 3. Its value is swapped with that of element 5, causing the array to appear as shown in Figure 7-33.

**Figure 7-33** Values in the array after the fourth swap

At this point, there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement, as shown in Figure 7-34.

**Figure 7-34** Values in the array after the fifth swap

## Swapping Array Elements

As you saw in the description of the selection sort algorithm, certain elements are swapped as the algorithm steps through the array. Let's briefly discuss the process of swapping two items in computer memory. Assume we have the following variable declarations:

```
int a = 1;
int b = 9;
```

Suppose we want to swap the values in these variables so the variable `a` contains 9 and the variable `b` contains 1. At first, you might think that we need only assign the variables to each other, like this:

```
// ERROR! The following does NOT swap the variables.
a = b;
b = a;
```

To understand why this does not work, let's step through the code. The first statement is `a = b`. This causes the value 9 to be assigned to `a`. But, what happens to the value 1 that was previously stored in `a`? Remember, when you assign a new value to a variable, the new value replaces any value that was previously stored in the variable. So, the old value, 1, is thrown away. Then the next statement is `b = a`. Since the variable `a` contains 9, this assigns 9 to `b`. After these statements execute, the variables `a` and `b` both contain the value 9.

To successfully swap the contents of two variables, we need a third variable that can serve as a temporary storage location:

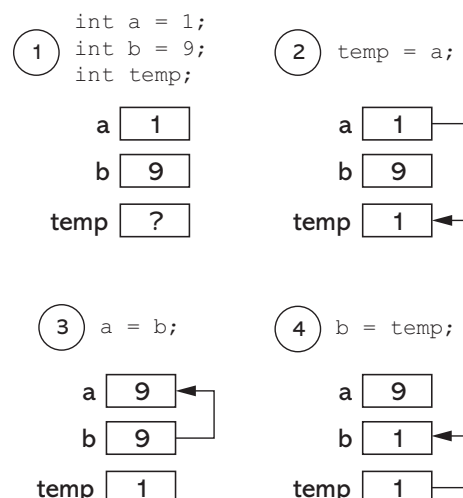
```
int temp;
```

Then we can perform the following steps to swap the values in the variables `a` and `b`:

- Assign the value of `a` to `temp`.
- Assign the value of `b` to `a`.
- Assign the value of `temp` to `b`.

Figure 7-35 shows the contents of these variables as we perform each of these steps. Notice that after the steps are finished, the values in `a` and `b` are swapped.

**Figure 7-35** Swapping the values of `a` and `b`



Here is the code for a `Swap` method that we can use to swap `int` values:

```

1 private void Swap(ref int a, ref int b)
2 {
3     int temp = a;
4     a = b;
5     b = temp;
6 }

```



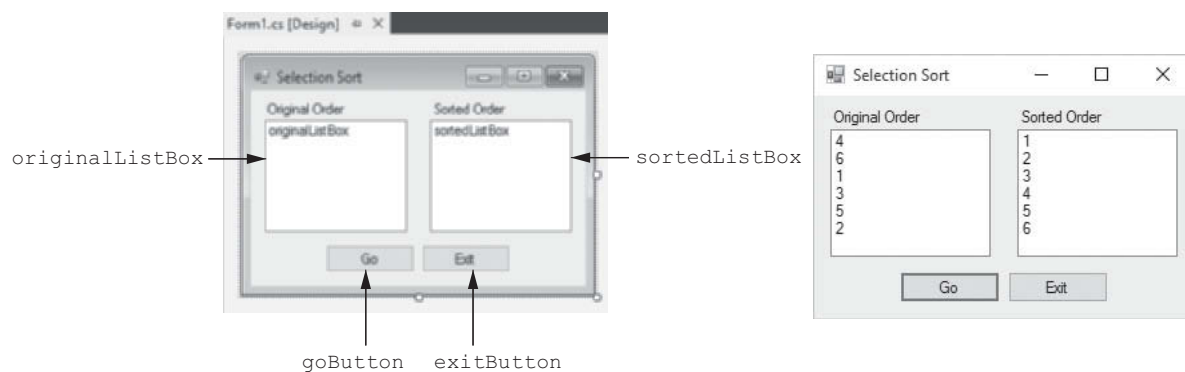
**NOTE:** It is critical that we use reference parameters in the `Swap` method because the method must be able to change the values of the items that are passed to it as arguments.

Let's look at a complete program that demonstrates the Selection Sort algorithm. In the *Chap07* folder of the Student Sample Programs, you will find a project named *Selection Sort*. Figure 7-36 shows the application's form. On the left, you see the form in the *Designer*, with the names of various controls. On the right, you see the form at run time, after the *Go* button has been clicked. When you click the *Go* button, the following actions take place:

- An `int` array is created, initialized with unsorted values.
- The contents of the array are displayed in the `originalListBox` control.
- The array is passed as an argument to the `SelectionSort` method. The method uses the Selection Sort algorithm to sort the array.
- The contents of the array are displayed in the `sortedListBox` control.

Program 7-3 shows the complete code for the *Selection Sort* application.

**Figure 7-36** The *Selection Sort* application's form



**Program 7-3** Complete code for `Form1` in the *Selection Sort* application

```

1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10

```

```
11 namespace Selection_Sort
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         // The SelectionSort method accepts an int array as an argument.
21         // It uses the Selection Sort algorithm to sort the array.
22         private void SelectionSort(int[] iArray)
23         {
24             int minIndex; // Subscript of smallest value in scanned area
25             int minValue; // Smallest value in the scanned area
26
27             // The outer loop steps through all the array elements,
28             // except the last one. The startScan variable marks the
29             // position where the scan should begin.
30             for (int startScan = 0; startScan < iArray.Length - 1; startScan++)
31             {
32                 // Assume the first element in the scannable area
33                 // is the smallest value.
34                 minIndex = startScan;
35                 minValue = iArray[startScan];
36
37                 // Scan the array, starting at the 2nd element in the
38                 // scannable area, looking for the smallest value.
39                 for (int index = startScan + 1; index < iArray.Length; index++)
40                 {
41                     if (iArray[index] < minValue)
42                     {
43                         minValue = iArray[index];
44                         minIndex = index;
45                     }
46                 }
47
48                 // Swap the element with the smallest value with the
49                 // first element in the scannable area.
50                 Swap(ref iArray[minIndex], ref iArray[startScan]);
51             }
52         }
53
54         // The Swap method accepts two integer arguments, by reference,
55         // and swaps their contents.
56         private void Swap(ref int a, ref int b)
57         {
58             int temp = a;
59             a = b;
60             b = temp;
61         }
62
63         private void goButton_Click(object sender, EventArgs e)
64         {
65             // Create an array of integers.
66             int[] numbers = { 4, 6, 1, 3, 5, 2 };
67
68             // Display the array in original order.
69             foreach (int value in numbers)
70             {
71                 originalListBox.Items.Add(value);

```

```

72         }
73
74         // Sort the array.
75         SelectionSort(numbers);
76
77         // Display the array in sorted order.
78         foreach (int value in numbers)
79         {
80             sortedListBox.Items.Add(value);
81         }
82     }
83
84     private void exitButton_Click(object sender, EventArgs e)
85     {
86         // Close the form.
87         this.Close();
88     }
89 }
90 }

```

---

## The Binary Search Algorithm

Previously in this chapter, we discussed the sequential search algorithm, which uses a loop to step sequentially through an array, starting with the first element. It compares each element with the value being searched for and stops when the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm unsuccessfully searches to the end of the array.

The advantage of the sequential search is its simplicity: It is very easy to understand and implement. Furthermore, it does not require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contains 20,000 elements, the algorithm has to look at all 20,000 elements in order to find a value stored in the last element.

In an average case, an item is just as likely to be found near the beginning of an array as near the end. Typically, for an array of  $n$  items, the sequential search locates an item in  $n/2$  attempts. If an array has 50,000 elements, the sequential search makes a comparison with 25,000 of them in a typical case. This is assuming, of course, that the search item is consistently found in the array. ( $n/2$  is the average number of comparisons. The maximum number of comparisons is always  $n$ .)

When the sequential search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Although the sequential search algorithm is adequate for small arrays, it should not be used on large arrays if speed is important.

The **binary search** is a clever algorithm that is much more efficient than the sequential search. Its only requirement is that the values in the array must be sorted in ascending order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value is not found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm tests *its* middle element. If the desired value is not

found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found or there are no more elements to test.

Here is the pseudocode for a method that performs a binary search on an array:

```

Method BinarySearch(array, searchValue)
  Set first to 0
  Set last to the last subscript in the array
  Set position to -1
  Set found to false

  While found is not true and first is less than or equal to last
    Set middle to the subscript half way between array[first] and array[last]
    If array[middle] equals searchValue
      Set found to true
      Set position to middle
    Else If array[middle] is greater than searchValue
      Set last to middle - 1
    Else
      Set first to middle + 1
    End If
  End While

  Return position
End Method

```

This algorithm uses three variables to mark positions within the array: *first*, *last*, and *middle*. The *first* and *last* variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's first and last elements. The subscript of the element halfway between first and last is calculated and stored in the *middle* variable. If the element in the middle of the array does not contain the search value, the *first* or *last* variable is adjusted so that only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The following C# method performs a binary search on an integer array. The first parameter, `iArray`, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, `-1` is returned, indicating the value did not appear in the array.

```

1 private int BinarySearch(int[] iArray, int value)
2 {
3     int first = 0;           // First array element
4     int last = iArray.Length - 1; // Last array element
5     int middle;             // Midpoint of search
6     int position = -1;      // Position of search value
7     bool found = false;    // Flag
8
9     // Search for the value.
10    while (!found && first <= last)
11    {
12        // Calculate the midpoint.
13        middle = (first + last) / 2;
14
15        // If value is found at midpoint ...
16        if (iArray[middle] == value)
17        {
18            found = true;
19            position = middle;

```

```

20     }
21     // else if value is in lower half ...
22     else if (iArray[middle] > value)
23     {
24         last = middle - 1;
25     }
26     // else if value is in upper half ...
27     else
28     {
29         first = middle + 1;
30     }
31 }
32
33 // Return the position of the item, or -1
34 // if it was not found.
35 return position;
36 }

```

If you want to see a complete application that uses the binary search algorithm, look at the *Binary Search* project, located in the *Chap07* folder of the Student Sample Programs. It loads a list of names from a file into an array and then performs a binary search to find a specific name in the array.



## Checkpoint

- 7.14 What is a search algorithm?
- 7.15 What is the purpose of a sorting algorithm?
- 7.16 What is the only requirement of the binary search algorithm?

## 7.7

## Two-Dimensional Arrays

**CONCEPT:** A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

The arrays that you have studied so far are known as one-dimensional arrays. They are called **one-dimensional** arrays because they can hold only one set of data. **Two-dimensional** arrays, which are also called *2D arrays*, can hold multiple sets of data. Think of a two-dimensional array as having rows and columns of elements, as shown in Figure 7-37. This figure shows a two-dimensional array having three rows and four columns. Notice that the rows are numbered 0, 1, and 2, and the columns are numbered 0, 1, 2, and 3. There is a total of 12 elements in the array.

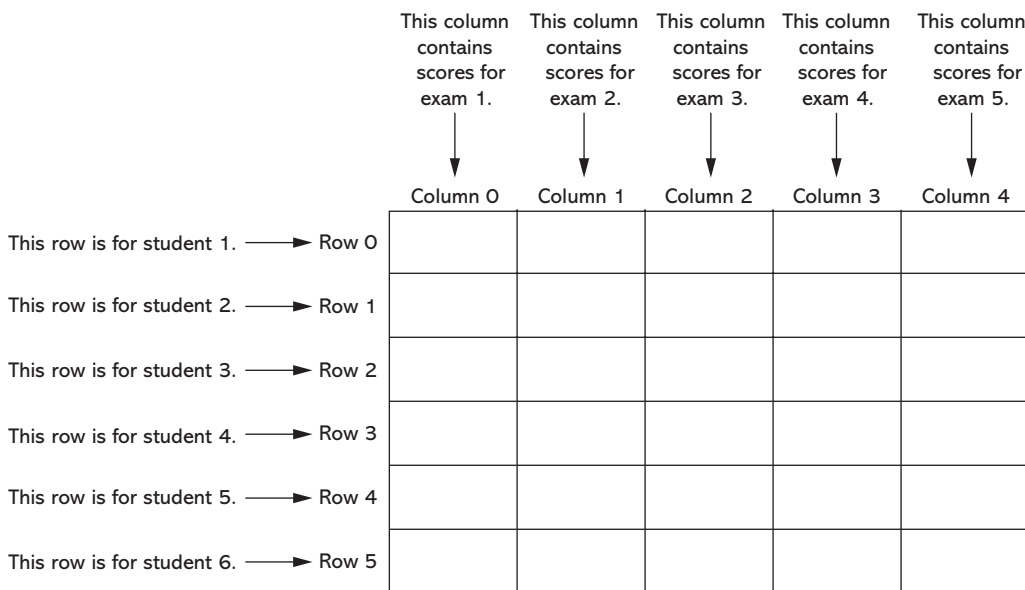
**Figure 7-37** A two-dimensional array

	Column 0	Column 1	Column 2	Column 3
Row 0				
Row 1				
Row 2				



Two-dimensional arrays are useful for working with multiple sets of data. For example, suppose you are designing a grade-averaging program for a teacher. The teacher has six students, and each student takes five exams during the semester. One approach would be to create six one-dimensional arrays, one for each student. Each of these arrays would have five elements, one for each exam score. This approach would be cumbersome, however, because you would have to separately process each of the arrays. A better approach would be to use a two-dimensional array with six rows (one for each student) and five columns (one for each exam score), as shown in Figure 7-38.

**Figure 7-38** Two-dimensional array with six rows and five columns



## Declaring a Two-Dimensional Array

To declare a two-dimensional array, two size declarators are required: The first one is for the number of rows, and the second one is for the number of columns. Here is an example declaration of a two-dimensional array with three rows and four columns:

```
double[,] scores = new double[3, 4];
```

Notice the comma that appears inside the first set of brackets. This indicates that the `scores` variable references a two-dimensional array. The numbers 3 and 4 are size declarators. The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice that the size declarators are separated by a comma.

As with one-dimensional arrays, it is best to use named constants as the size declarators. Here is an example:

```
const int ROWS = 3;
const int COLS = 4;
int[,] scores = new int[ROWS, COLS];
```

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `scores` array, the elements in row 0 are referenced as follows:

```
scores[0,0]
scores[0,1]
scores[0,2]
scores[0,3]
```

The elements in row 1 are referenced as follows:

```
scores[1,0]
scores[1,1]
scores[1,2]
scores[1,3]
```

And, the elements in row 2 are referenced as follows:

```
scores[2,0]
scores[2,1]
scores[2,2]
scores[2,3]
```

Figure 7-39 illustrates the array with the subscripts shown for each element.

**Figure 7-39** Subscripts for each element of the `scores` array

	Column 0	Column 1	Column 2	Column 3
Row 0	scores[0,0]	scores[0,1]	scores[0,2]	scores[0,3]
Row 1	scores[1,0]	scores[1,1]	scores[1,2]	scores[1,3]
Row 2	scores[2,0]	scores[2,1]	scores[2,2]	scores[2,3]

## Accessing the Elements in a Two-Dimensional Array

To access one of the elements in a two-dimensional array, you must use two subscripts. For example, suppose we have the following declarations in a program:

```
const int ROWS = 5;
const int COLS = 10;
int[,] values = new int[ROWS, COLS];
```

The following statement assigns the number 95 to `values[2,1]`:

```
values[2,1] = 95;
```

Programs often use nested loops to process two-dimensional arrays. For example, the following code assigns a random number to each element of the `values` array:

```
1 // Create a Random object.
2 Random rand = new Random();
3
4 // Create a two-dimensional int array.
5 const int ROWS = 5;
6 const int COLS = 10;
7 int[,] values = new int[ROWS, COLS];
8
9 // Fill the array with random numbers.
10 for (int row = 0; row < ROWS; row++)
11 {
12     for (int col = 0; col < COLS; col++)
13     {
14         values[row, col] = rand.Next(100);
15     }
16 }
```

And the following set of nested loops displays all the elements of the `values` array in a `ListBox` control named `outputListBox`:

```
1 // Display the array contents.
2 for (int row = 0; row < ROWS; row++)
3 {
4     for (int col = 0; col < COLS; col++)
5     {
6         outputListBox.Items.Add(values[row, col].ToString());
7     }
8 }
```

## Implicit Sizing and Initialization of Two-Dimensional Arrays

As with a one-dimensional array, you may provide an initialization list for a two-dimensional array. Recall that when you provide an initialization list for an array, you cannot provide the upper subscript numbers. When initializing a two-dimensional array, you must provide the comma to indicate the number of dimensions. The following is an example of a two-dimensional array declaration with an initialization list:

```
int[,] values = { {1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9} };
```

Initialization values for each row are enclosed in their own set of braces. In this example, the initialization values for row 0 are {1, 2, 3}, the initialization values for row 1 are {4, 5, 6}, and the initialization values for row 2 are {7, 8, 9}. So, this statement declares an array with three rows and three columns. The values are assigned to the `values` array in the following manner:

```
values[0, 0] is set to 1.
values[0, 1] is set to 2.
values[0, 2] is set to 3.
values[1, 0] is set to 4.
values[1, 1] is set to 5.
values[1, 2] is set to 6.
values[2, 0] is set to 7.
values[2, 1] is set to 8.
values[2, 2] is set to 9.
```

Tutorial 7-3 gives you hands-on practice working with a two-dimensional array.



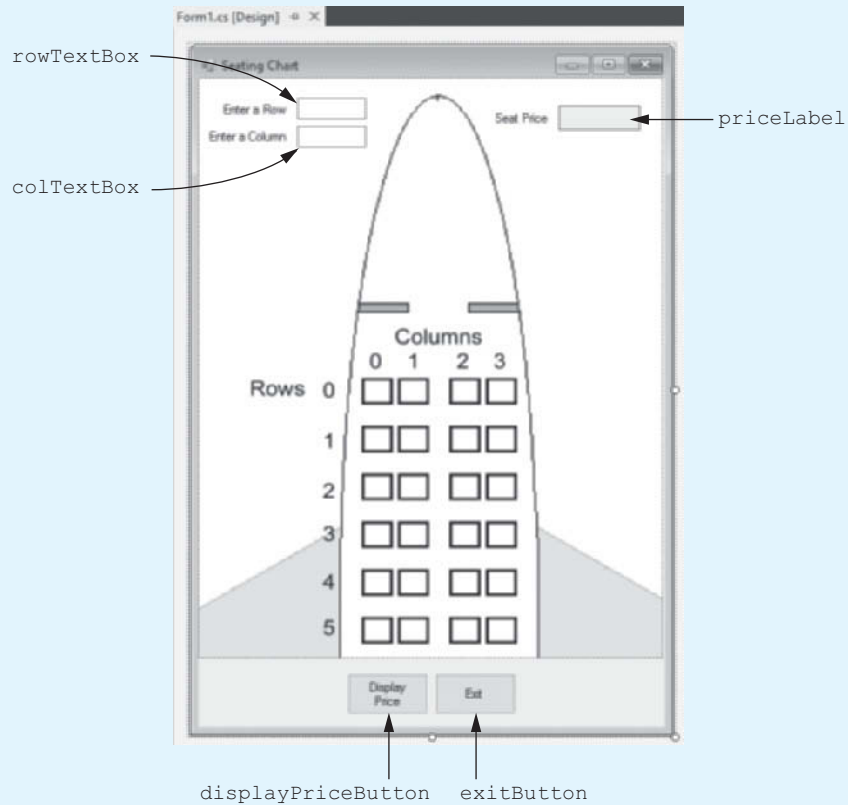
### Tutorial 7-3: Completing the *Seating Chart* Application



VideoNote

**Tutorial 7-3:**  
Completing  
the *Seating  
Chart*  
application

In this tutorial, you complete the *Seating Chart* application. The application's form, which is shown in Figure 7-40, uses a `PictureBox` control to display an airplane seating chart that is arranged in rows and columns. When completed, the application allows the user to enter valid row and column numbers in the `rowTextBox` and `colTextBox` text boxes and then click the *Display Price* button. The price of the selected seat will be displayed in the `priceLabel` control. The following table shows the seat prices:

**Figure 7-40** The *Seating Chart* application's form

Columns	0	1	2	3
Row 0	\$450	\$450	\$450	\$450
Row 1	\$425	\$425	\$425	\$425
Row 2	\$400	\$400	\$400	\$400
Row 3	\$375	\$375	\$375	\$375
Row 4	\$375	\$375	\$375	\$375
Row 5	\$350	\$350	\$350	\$350

When you write the code for the application, you will create a two-dimensional array to hold these values.

**Step 1:** Start Visual Studio. Open the project named *Seating Chart* in the *Chap07* folder of this book's Student Sample Programs.

**Step 2:** Open the Form1 form in the *Designer*. Double-click the `displayPriceButton` control. This opens the code editor, and you will see an empty event handler named `displayPriceButton_Click`. Complete the event handler by typing the code shown in lines 22–78 in Program 7-4. Let's take a closer look at the code:

**Line 23:** This statement declares two `int` variables, `row` and `col`, to hold the row and column selected by the user.

**Lines 26–27:** These statements declare `int` constants named `MAX_ROW` and `MAX_COL`, set to the values 5 and 3, respectively. These are used as array size declarators.

**Lines 30–36:** This statement creates a two-dimensional `decimal` array named `prices`, initialized with the seat prices previously shown.


**Line 39:** This `if` statement converts the value entered into the `rowTextBox` control to an `int` and stores the result in the `row` variable. If the conversion is successful, the program continues. If the conversion fails, the program jumps to the `else` clause in line 74, and then line 77 displays an error message.

**Line 42:** This `if` statement converts the value entered into the `colTextBox` control to an `int` and stores the result in the `col` variable. If the conversion is successful, the program continues. If the conversion fails, the program jumps to the `else` clause in line 68, and then line 71 displays an error message.

**Line 45:** This `if` statement determines whether `row` is in the range of 0 through `MAX_ROW`. If so, the program continues. Otherwise, the program jumps to the `else` clause in line 61, and then lines 64–65 display an error message.

**Line 48:** This `if` statement determines whether `col` is in the range of 0 through `MAX_COL`. If so, the program continues. Otherwise, the program jumps to the `else` clause in line 54, and then lines 57–58 display an error message.

**Lines 51–52:** This statement uses `row` and `col` as subscripts to retrieve the selected seat's price from the `prices` array and then displays that value in the `priceLabel` control.

- Step 3:** Switch your view back to the *Designer* and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the event handler by typing the code shown in lines 83–84 in Program 7-4.
- Step 4:** Save the project. Then, press `F5` on the keyboard or click the *Start Debugging* button (  ) on the toolbar to compile and run the application. When the application runs, experiment by entering row and column numbers for different seats and comparing the displayed price with the table previously shown. When you are finished, click the *Exit* button to end the application.

#### Program 7-4 Completed code for Form1 in the *Seating Chart* application

```

1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10
11 namespace Seating_Chart
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void displayPriceButton_Click(object sender, EventArgs e)
21         {
22             // Variables for the selected row and column

```

```

23         int row, col;
24
25         // Constants for the maximum row and column subscripts
26         const int MAX_ROW = 5;
27         const int MAX_COL = 3;
28
29         // Create an array with the seat prices.
30         decimal[,] prices = { {450m, 450m, 450m, 450m},
31                               {425m, 425m, 425m, 425m},
32                               {400m, 400m, 400m, 400m},
33                               {375m, 375m, 375m, 375m},
34                               {375m, 375m, 375m, 375m},
35                               {350m, 350m, 350m, 350m}
36                               };
37
38         // Get the selected row number.
39         if (int.TryParse(rowTextBox.Text, out row))
40         {
41             // Get the selected column number.
42             if (int.TryParse(colTextBox.Text, out col))
43             {
44                 // Make sure the row is within range.
45                 if (row >= 0 && row <= MAX_ROW)
46                 {
47                     // Make sure the column is within range.
48                     if (col >= 0 && col <= MAX_COL)
49                     {
50                         // Display the selected seat's price.
51                         priceLabel.Text =
52                             prices[row, col].ToString("c");
53                     }
54                     else
55                     {
56                         // Error message for invalid column.
57                         MessageBox.Show("Column must be 0 through " +
58                             MAX_COL);
59                     }
60                 }
61                 else
62                 {
63                     // Error message for invalid row.
64                     MessageBox.Show("Row must be 0 through " +
65                         MAX_ROW);
66                 }
67             }
68             else
69             {
70                 // Display an error message for noninteger column.
71                 MessageBox.Show("Enter an integer for the column.");
72             }
73         }
74         else
75         {
76             // Display an error message for noninteger row.
77             MessageBox.Show("Enter an integer for the row.");
78         }
79     }
80
81     private void exitButton_Click(object sender, EventArgs e)

```

```
82     {
83         // Close the form.
84         this.Close();
85     }
86 }
87 }
```

## Summing All the Elements of a Two-Dimensional Array

To sum all the elements of a two-dimensional array, you can use a pair of nested loops to add the contents of each element to an accumulator. The following code shows an example:

```
1 const int ROWS = 3;
2 const int COLS = 3;
3 int[,] numbers = { {1, 2, 3, 4},
4                   {5, 6, 7, 8},
5                   {9, 10, 11, 12}
6                   };
7
8 int total = 0; // Accumulator, set to 0
9
10 // Sum the array elements.
11 for (int row = 0; row < ROWS; row++)
12 {
13     for (int col = 0; col < COLS; col++)
14     {
15         total += numbers[row, col];
16     }
17 }
18 // Display the sum.
19 MessageBox.Show("The total is " + total);
```

## Summing the Rows of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each row in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students. Each row in the array is a set of test scores for one student. To get the sum of a student's test scores (perhaps so an average may be calculated), you use a loop to add all the elements in one row. The following code shows an example:

```
1 const int ROWS = 3;
2 const int COLS = 3;
3 int[,] numbers = { {1, 2, 3, 4},
4                   {5, 6, 7, 8},
5                   {9, 10, 11, 12}
6                   };
7
8 int total; // Accumulator
9
10 // Sum each row in the array.
11 for (int row = 0; row < ROWS; row++)
12 {
13     // Set the accumulator to 0.
14     total = 0;
15
16     // Total the row.
17     for (int col = 0; col < COLS; col++)
18     {
```

```

19         total += numbers[row, col];
20     }
21
22     // Display the row's total.
23     MessageBox.Show("The total of row " + row +
24                     " is " + total);
25 }

```

## Summing the Columns of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each column in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students and you wish to calculate the class average for each of the test scores. To do this, you calculate the average of each column in the array. This is accomplished with a set of nested loops. The outer loop controls the column subscript, and the inner loop controls the row subscript. The inner loop calculates the sum of a column, which is stored in an accumulator. The following code demonstrates:

```

1  const int ROWS = 3;
2  const int COLS = 4;
3  int[,] numbers = { {1, 2, 3, 4},
4                     {5, 6, 7, 8},
5                     {9, 10, 11, 12}
6                     };
7
8  int total;    // Accumulator
9
10 // Sum each column in the array.
11 for (int col = 0; col < COLS; col++)
12 {
13     // Set the accumulator to 0.
14     total = 0;
15
16     // Total the column.
17     for (int row = 0; row < ROWS; row++)
18     {
19         total += numbers[row, col];
20     }
21
22     // Display the column's total.
23     MessageBox.Show("The total of column " + col +
24                     " is " + total);
25 }

```



### Checkpoint

- 7.17 How many rows and how many columns are in the following array?
- ```
int[,] values = new decimal[200, 100];
```
- 7.18 Write a statement that assigns the value 50 to the very last element in the `values` array declared in Checkpoint 7.17.
- 7.19 Write a declaration for a two-dimensional `int` array initialized with the following table of data:

|    |    |    |    |    |
|----|----|----|----|----|
| 12 | 24 | 32 | 21 | 42 |
| 99 | 8  | 68 | 32 | 92 |
| 95 | 34 | 21 | 11 | 7  |



## 7.8 Jagged Arrays

**CONCEPT:** A jagged array is similar to a two-dimensional array, but the rows in a jagged array can have different lengths.

In a traditional two-dimensional array, each row has the same number of columns. Mentally, we visualize a two-dimensional array as a rectangular structure. Figure 7-37, previously shown, is an example. For this reason, two-dimensional arrays are sometimes referred to as **rectangular arrays**.

A **jagged array** is similar to a two-dimensional array, but the rows in a jagged array can have different numbers of columns. This is possible because a jagged array is actually an array of arrays. To be more specific, a jagged array is a one-dimensional array, and each element of the array is also a one-dimensional array. Figure 7-41 shows an example. In the figure, row 0 has four columns, row 1 has three columns, and row 2 has five columns.

**Figure 7-41** A jagged array

|       |   |   |    |    |    |
|-------|---|---|----|----|----|
| Row 0 | 1 | 2 | 3  | 4  |    |
| Row 1 | 5 | 6 | 7  |    |    |
| Row 2 | 8 | 9 | 10 | 11 | 12 |

Because a jagged array is an array of arrays, you set it up differently than a two-dimensional array. First you create an array, and then you create each of the arrays that are the elements of the first array. The following code shows an example of how the jagged array in Figure 7-41 might be created and initialized.

```

1 // Create an array of 3 int arrays.
2 int[][] jaggedArray = new int[3][];
3
4 // Create each array that is an element
5 // of the jagged array.
6 jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
7 jaggedArray[1] = new int[3] { 5, 6, 7 };
8 jaggedArray[2] = new int[5] { 8, 9, 10, 11, 12 };

```

Let's take a closer look at the code:

- Line 2 declares an array named `jaggedArray`. Notice that the data type is `int[][]`, with two sets of brackets. This indicates that we are declaring an array of `int` arrays. Also notice that the expression `new int[3][]` uses only one size declarator, specifying the number of rows. The column sizes must be set individually.
- Line 6 creates element 0, which is an `int` array with four columns. The columns are initialized with the values 1, 2, 3, and 4.
- Line 7 creates element 1, which is an `int` array with three columns. The columns are initialized with the values 5, 6, and 7.
- Line 8 creates element 2, which is an `int` array with five columns. The columns are initialized with the values 8, 9, 10, 11, and 12.

To access an item that is stored at a particular row and column in a jagged array, you enclose the row and column subscripts in their own sets of brackets. For example, the

following statement displays the value stored at row 1, column 2, of the `jaggedArray` that was previously declared:

```
MessageBox.Show(jaggedArray[1][2].ToString());
```

The following statement shows another example. It assigns the value 99 to row 0, column 3, of `jaggedArray`:

```
jaggedArray[0][3] = 99;
```

A jagged array has a `Length` property that holds the number of rows, and then each row has its own `Length` property. You can use a row's `Length` property to determine the number of columns in that row. For example, the following set of nested loops displays all the values stored in the `jaggedArray` that was previously declared:

```
1 for (int row = 0; row < jaggedArray.Length; row++)
2 {
3     for (int col = 0; col < jaggedArray[row].Length; col++)
4     {
5         MessageBox.Show(jaggedArray[row][col].ToString());
6     }
7 }
```



## Checkpoint

- 7.20 Why are two-dimensional arrays sometimes referred to as rectangular arrays?
- 7.21 Write a statement that declares a jagged array of `int` values and initialize the columns of each row with the values in the following table of data:

|   |   |    |    |    |
|---|---|----|----|----|
| 2 | 4 | 6  |    |    |
| 3 | 5 | 7  | 9  |    |
| 5 | 9 | 11 | 17 | 21 |

## 7.9 The List Collection

**CONCEPT:** `List` is a class in the .NET Framework that is similar to an array. Unlike an array, a `List` object's size is automatically adjusted to accommodate the number of items being stored in it.

The .NET Framework provides a class named `List`, which can be used for storing and retrieving items. Once you create a `List` object, you can think of it as a container for holding other objects. A `List` object is similar to an array but offers many advantages over an array. Here are a few:

- When you create a `List` object, you do not have to know the number of items that you intend to store in it.
- A `List` object automatically expands as items are added to it.
- In addition to adding items to a `List`, you can remove items as well.
- A `List` object automatically shrinks as items are removed from it.

### Creating a List

Here is an example of how you create a `List` object that can be used to hold strings:

```
List<string> nameList = new List<string>();
```

This statement creates a `List` object, referenced by the `nameList` variable. Notice that in this example the word `string` is written inside angled brackets `<>` immediately after the word `List`. This specifies that the `List` can hold objects of the `string` data type. If you try to store any other type of object in this `List`, an error occurs.

Here is an example of how you create a `List` object that can be used to hold integers:

```
List<int> numberList = new List<int>();
```

This statement creates a `List` object, referenced by the `numberList` variable. Notice that in this example the word `int` is written inside angled brackets `<>` immediately after the word `List`.

## Initializing a List

You can optionally initialize a `List` object when you declare it. Here is an example:

```
List<int> numberList = new List<int>() { 1, 2, 3 };
```

This statement creates a `List` object that can hold integers and initializes it with the values 1, 2, and 3. Here is an example that creates a `List` object to hold strings and initializes it with three strings:

```
List<string> nameList = new List<string>() { "Chris",  
    "Kathryn", "Bill" };
```

## Adding Items to a List

To add items to an existing `List` object, you use the **Add method**. For example, the following statements create a `List` object and add a series of strings to it:

```
List<string> nameList = new List<string>();  
nameList.Add("Chris");  
nameList.Add("Kathryn");  
nameList.Add("Bill");
```

After these statements execute, the `nameList` object will hold the three strings "Chris", "Kathryn", and "Bill".

The items that are stored in a `List` have a corresponding index. The index specifies the item's location in the `List`, so it is much like an array subscript. The first item that is added to a `List` is stored at index 0. The next item that is added to the `List` is stored at index 1, and so forth. After the previously shown statements execute, "Chris" is stored at index 0, "Kathryn" is stored at index 1, and "Bill" is stored at index 2.

## The Count Property

A `List` object has a **Count property** that holds the number of items stored in the `List`. For example, the following statement uses the `Count` property to display the number of items stored in `nameList`:

```
MessageBox.Show("The List has " + nameList.Count +  
    " objects stored in it.");
```

Assuming that `nameList` holds the strings "Chris", "Kathryn", and "Bill", the following statement will be displayed in a message box:

```
The List has 3 objects stored in it.
```

## Accessing Items in a List

You can use subscript notation to access the items in a `List`, just as you can with an array. For example, the following `for` loop displays the items in the `nameList` object:

```
for (int index = 0; index < nameList.Count; index++)
{
    MessageBox.Show(nameList[index]);
}
```

Notice that the loop uses the `List` object's `Count` property in the test expression to control the number of iterations. Here is an example that reads values from a text file and adds them to a `List`:

```
1 // Open the Names.txt file.
2 StreamReader inputFile = File.OpenText("Names.txt");
3
4 // Create a List object to hold strings.
5 List<string> nameList = new List<string>();
6
7 // Read the file's contents.
8 while (!inputFile.EndOfStream)
9 {
10     // Read a line and add it to the List.
11     nameList.Add(inputFile.ReadLine());
12 }
```

Let's take a closer look at this code:

- Line 2 opens a file named `Names.txt` and associates it with a `StreamReader` object that is referenced by the `inputFile` variable.
- Line 5 creates a `List` object, referenced by the `nameList` variable. The object can hold strings.
- The `while` loop that starts in line 8 iterates until the end of the file is reached.
- The statement in line 11 reads a line from the file and adds it to the `nameList` object.

After this code executes, the `nameList` object contains all the lines that were read from the `Names.txt` file.

You can also use the `foreach` loop to iterate over the items in a `List`, just as you can with an array. Here is an example:

```
foreach (string str in nameList)
{
    MessageBox.Show(str);
}
```

## Passing a List to a Method

Sometimes you will want to write a method that accepts a `List` as an argument and performs an operation on the `List`. For example, the following code shows a method named `DisplayList`. The method accepts a `List` of strings as an argument and displays each item in `List`.

```
1 private void DisplayList(List<string> sList)
2 {
3     foreach (string str in sList)
4     {
5         MessageBox.Show(str);
6     }
7 }
```

Notice in line 1 that the method has a parameter variable named `sList` and that the parameter's data type is `List<string>`. The parameter variable is a reference to a `List<string>` object. When you call this method, you must pass a `List<string>` object as an argument.

When you call a method and pass a `List<string>` object as an argument, you simply pass the variable that references the `List`. The following code shows an example of how the `DisplayList` method (previously shown) might be called:

```
1 // Create a List of strings.
2 List<string> nameList = new List<string>() { "Chris",
3     "Kathryn", "Bill" };
4
5 // Pass the List to the DisplayList method.
6 DisplayList(nameList);
```

The statement in lines 2 and 3 creates a `List` containing the strings "Chris", "Kathryn", and "Bill". Line 6 calls the `DisplayList` method, passing the `nameList` object as an argument.



**NOTE:** `List` objects, like arrays, are always passed by reference.

## Removing Items from a List

You can use the **RemoveAt** method to remove an item at a specific index in a `List`. The following code shows an example:

```
1 // Create a List of strings.
2 List<string> nameList = new List<string>() { "Chris",
3     "Kathryn", "Bill" };
4
5 // Remove the item at index 0.
6 nameList.RemoveAt(0);
```

The statement in lines 2 and 3 creates a `List` containing the strings "Chris", "Kathryn", and "Bill". Then, the statement in line 6 removes the string at index 0. After this statement executes, the `List` contains the strings "Kathryn" and "Bill".

If you know the value of the item that you want to remove from a `List`, but you do not know the item's index, you can use the **Remove** method. You pass the item that you want to remove as an argument, and the `Remove` method searches for that item in the `List`. If the item is found, it is removed. Here is an example:

```
1 // Create a List of strings.
2 List<string> nameList = new List<string>() { "Chris",
3     "Kathryn", "Bill" };
4
5 // Remove "Bill" from the List.
6 nameList.Remove("Bill");
```

The statement in lines 2 and 3 creates a `List` containing the strings "Chris", "Kathryn", and "Bill". Then, the statement in line 6 removes "Bill" from the `List`. After this statement executes, the `List` contains the strings "Chris" and "Kathryn".

The `Remove` method returns a Boolean value indicating whether the item was actually removed from the `List`. If the specified item was found in the `List` and removed, the `Remove` method returns `true`. If the item was not found in the `List`, the `Remove` method

returns `false`. The following code demonstrates how you can use the value returned from the method:

```
1 // Create a List of strings.
2 List<string> nameList = new List<string>() { "Chris",
3     "Kathryn", "Bill" };
4
5 // Remove "Susan".
6 if (!nameList.Remove("Susan"))
7 {
8     MessageBox.Show("Susan was not found.");
9 }
```

The statement in lines 2 and 3 creates a `List` containing the strings "Chris", "Kathryn", and "Bill". Then, the statement in line 6 attempts to remove "Susan" from the `List`. The `List` does not contain the string "Susan", so the `Remove` method returns `false`. The message "Susan was not found" is displayed. After this code executes, the `List`

An easy way to search for item in a `List`, however, is to use the `IndexOf` method. The `IndexOf` method accepts a value as an argument, and it searches for that value in the `List`. If the value is found, the method returns its index. If the value is not found, the method returns `-1`. The following code shows an example:

```
1 // Create a List of strings.
2 List<string> nameList = new List<string>() { "Chris",
3     "Kathryn", "Bill" };
4
5 // Search for "Kathryn".
6 int position = nameList.IndexOf("Kathryn");
7
8 // Was Kathryn found in the List?
9 if (position != -1)
10 {
11     MessageBox.Show("Kathryn was found at index " +
12         position);
13 }
14 else
15 {
16     MessageBox.Show("Kathryn was not found.");
17 }
```

The statement in lines 2 and 3 creates a `List` containing the strings "Chris", "Kathryn", and "Bill". The statement in line 6 calls the `IndexOf` method to search for "Kathryn" in the `List`. The value that is returned from the method is assigned to the `position` variable. After this statement executes, the `position` variable contains the index of "Kathryn" or `-1` if "Kathryn" was not found in the `List`. The `if` statement in lines 9–17 displays one of two possible messages, depending on whether "Kathryn" was found. (If this code were executed, it would display the message "Kathryn was found at index 1".)

There are two additional versions of the `IndexOf` method that allow you to specify the area of the `List` that should be searched. The following statement shows an example of one of these:

```
position = nameList.IndexOf("Diane", 2);
```

Notice that two arguments are passed to the `IndexOf` method. The first argument, "Diane", is the item to search for. The second argument, 2 is the starting index of the search. This specifies that the search should begin at index 2 and end at the last item in the `List`. (The beginning index is included in the search. If you pass an invalid index as an argument, an exception occurs.)

Here is an example of another version of the `IndexOf` method:

```
position = nameList.IndexOf("Diane", 2, 5);
```

In this example, three arguments are passed to the `IndexOf` method. The first argument, "Diane", is the item to search for. The second argument, 2 is the starting index of the search. The third argument, 5, is the ending index of the search. This specifies that the search should begin at index 2, and end at index 5. (The beginning and ending indices are included in the search. If either index is invalid, an exception occurs.)



**NOTE:** The `IndexOf` method performs a sequential search to locate the specified item. If the `List` contains a large number of items, its performance will be slow.

In Tutorial 7-4, you will complete an application that reads the contents of a file into a `List`, and then performs various operations on the `List`.



## Tutorial 7-4: Completing the *Test Score List* Application

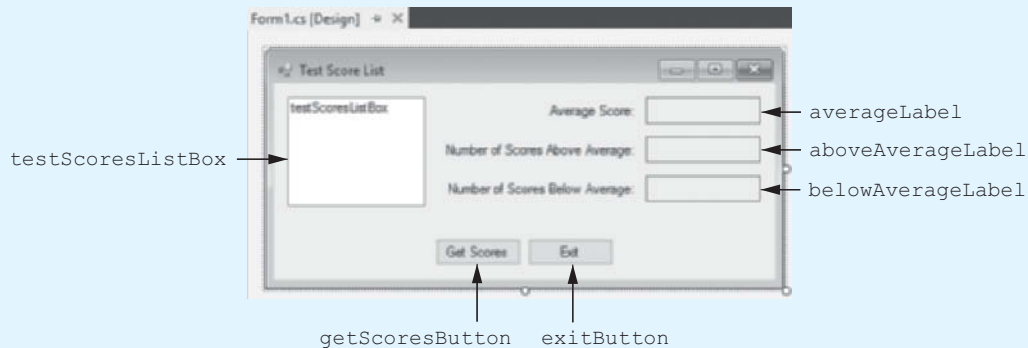


VideoNote

**Tutorial 7-4:**  
Completing  
the *Test  
Score List*  
Application

In this tutorial, you complete the *Test Score List* application. The application's form, which is shown in Figure 7-42, has already been created for you. When you complete the application, it will read a set of test scores from a file into a `List`. (The file has also been created for you.) The test scores are displayed in the `ListBox` control. The average test score is calculated and displayed, as well as the number of above-average test scores and below-average test scores.

**Figure 7-42** The *Test Score List* application's form



- Step 1:** Start Visual Studio. Open the project named *Test Score List* in the *Chap07* folder of the Student Sample Programs.
- Step 2:** Open the `Form1` form's code in the code editor. Insert the `using System.IO;` directive shown in line 10 of Program 7-5 at the end of this tutorial. This statement is necessary because we will be using the `StreamReader` class, and it is part of the `System.IO` namespace in the .NET Framework.
- Step 3:** With the code editor still open, type the comments and code for the `ReadScores` method, shown in lines 21–44 of Program 7-5. The purpose of the `ReadScores` method is to accept a `List<int>` object as an argument and read the contents of the `TestScores.txt` file into the list.
- Step 4:** Type the comments and code for the `DisplayScores` method, shown in lines 46–54 of Program 7-5. The purpose of the `DisplayScores` method is to accept a `List<int>` object as an argument and display its contents in the `testScoresListBox` control.
- Step 5:** Type the comments and code for the `Average` method, shown in lines 56–74 of Program 7-5. The purpose of the `Average` method is to accept a `List<int>` object as an argument and return the average of the values in the `List`.
- Step 6:** Type the comments and code for the `AboveAverage` method, shown in lines 76–96 of Program 7-5. The purpose of the `AboveAverage` method is to accept a `List<int>` object as an argument and return the number of above average scores it contains.
- Step 7:** Type the comments and code for the `BelowAverage` method, shown in lines 98–118 of Program 7-5. The purpose of the `BelowAverage` method is to accept a `List<int>` object as an argument and return the number of below average scores it contains.



**Step 8:** Next, you create the Click event handlers for the Button controls. Switch back to the *Designer* and double-click the `getScoresButton` control. This opens the code editor, and you will see an empty event handler named `getScoresButton_Click`. Complete the `getScoresButton_Click` event handler by typing the code shown in lines 122–145 in Program 7-5. Let's review this code:

**Lines 122–124:** These statements declare the following variables:

- `averageScore`—This variable is used to hold the average test score.
- `numAboveAverage`—This variable is used to hold the number of above-average test scores.
- `numBelowAverage`—This variable is used to hold the number of below-average test scores.

**Line 127:** This statement creates a `List<int>` object, referenced by the `scoresList` variable.

**Line 130:** This statement calls the `ReadScores` method, passing the `scoresList` object as an argument. After this statement executes, the `scoresList` object contains the test scores that are in the `TestScores.txt` file.

**Line 133:** This statement calls the `DisplayScores` method, passing the `scoresList` object as an argument. After this statement executes, the items in the `scoresList` object are displayed in the `testScoresListBox` control.

**Line 136:** This statement calls the `Average` method, passing the `scoresList` object as an argument. The method returns the average of the values in the `scoresList` object, which is assigned to the `averageScore` variable.

**Line 137:** This statement displays the average score in the `averageLabel` control.



**Line 140:** This statement calls the `AboveAverage` method, passing the `scoresList` object as an argument. The method returns the number of above-average scores in the `scoresList` object, which is assigned to the `numAboveAverage` variable.

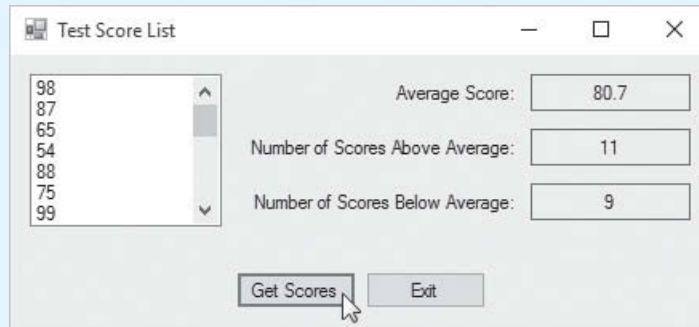
**Line 141:** This statement displays the number of above-average scores in the `aboveAverageLabel` control.

**Line 144:** This statement calls the `BelowAverage` method, passing the `scoresList` object as an argument. The method returns the number of below-average scores in the `scoresList` object, which is assigned to the `numBelowAverage` variable.

**Line 145:** This statement displays the number of below-average scores in the `belowAverageLabel` control.

**Step 9:** Switch your view back to the *Designer* and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the event handler by typing the code shown in lines 150–151 in Program 7-5.

**Step 10:** Save the project. Then, press  on the keyboard or click the *Start Debugging* button () on the toolbar to compile and run the application. When the application runs, click the *Get Scores* button. This should display a set of test scores in the `ListBox`, as well as the average score, the number of above-average scores, and the number of below-average scores, as shown in Figure 7-43. Click the *Exit* button to exit the application.

**Figure 7-43** The *Test Score List* application**Program 7-5** Completed code for Form1 in the *Test Scores List* application

```

1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10 using System.IO;
11
12 namespace Test_Score_List
13 {
14     public partial class Form1 : Form
15     {
16         public Form1()
17         {
18             InitializeComponent();
19         }
20
21         // The ReadScores method reads the scores from the
22         // TestScores.txt file into the scoresList parameter.
23         private void ReadScores(List<int> scoresList)
24         {
25             try
26             {
27                 // Open the TestScores.txt file.
28                 StreamReader inputFile = File.OpenText("TestScores.txt");
29
30                 // Read the scores into the list.
31                 while (!inputFile.EndOfStream)
32                 {
33                     scoresList.Add(int.Parse(inputFile.ReadLine()));
34                 }
35
36                 // Close the file.
37                 inputFile.Close();
38             }
39             catch (Exception ex)
40             {

```

```

41         // Display an error message.
42         MessageBox.Show(ex.Message);
43     }
44 }
45
46 // The DisplayScores method displays the contents of the
47 // scoresList parameter in the ListBox control.
48 private void DisplayScores(List<int> scoresList)
49 {
50     foreach (int score in scoresList)
51     {
52         testScoresListBox.Items.Add(score);
53     }
54 }
55
56 // The Average method returns the average of the values
57 // in the scoresList parameter.
58 private double Average(List<int> scoresList)
59 {
60     int total = 0;           // Accumulator
61     double average;        // To hold the average
62
63     // Calculate the total of the scores.
64     foreach (int score in scoresList)
65     {
66         total += score;
67     }
68
69     // Calculate the average of the scores.
70     average = (double)total / scoresList.Count;
71
72     // Return the average.
73     return average;
74 }
75
76 // The AboveAverage method returns the number of
77 // above average scores in scoresList.
78 private int AboveAverage(List<int> scoresList)
79 {
80     int numAbove = 0;       // Accumulator
81
82     // Get the average score.
83     double avg = Average(scoresList);
84
85     // Count the number of above average scores.
86     foreach (int score in scoresList)
87     {
88         if (score > avg)
89         {
90             numAbove++;
91         }
92     }
93
94     // Return the number of above average scores.
95     return numAbove;
96 }
97

```

```
98 // The BelowAverage method returns the number of
99 // below average scores in scoresList.
100 private int BelowAverage(List<int> scoresList)
101 {
102     int numBelow = 0; // Accumulator
103
104     // Get the average score.
105     double avg = Average(scoresList);
106
107     // Count the number of below average scores.
108     foreach (int score in scoresList)
109     {
110         if (score < avg)
111         {
112             numBelow++;
113         }
114     }
115
116     // Return the number of below average scores.
117     return numBelow;
118 }
119
120 private void getScoresButton_Click(object sender, EventArgs e)
121 {
122     double averageScore; // To hold the average score
123     int numAboveAverage; // Number of above average scores
124     int numBelowAverage; // Number of below average scores
125
126     // Create a List to hold the scores.
127     List<int> scoresList = new List<int>();
128
129     // Read the scores from the file into the List.
130     ReadScores(scoresList);
131
132     // Display the scores.
133     DisplayScores(scoresList);
134
135     // Display the average score.
136     averageScore = Average(scoresList);
137     averageLabel.Text = averageScore.ToString("\n1");
138
139     // Display the number of above average scores.
140     numAboveAverage = AboveAverage(scoresList);
141     aboveAverageLabel.Text = numAboveAverage.ToString();
142
143     // Display the number of below average scores.
144     numBelowAverage = BelowAverage(scoresList);
145     belowAverageLabel.Text = numBelowAverage.ToString();
146 }
147
148 private void exitButton_Click(object sender, EventArgs e)
149 {
150     // Close the form.
151     this.Close();
152 }
153 }
154 }
```



## Checkpoint

- 7.22 Write a statement that initializes a `List` with 4 values of the `double` data type.
- 7.23 Write a statement that adds a new value to the `List` object created in Checkpoint 7.22.
- 7.24 Write a statement that clears the contents of the `List` object created in Checkpoint 7.22.
- 7.25 Is it possible to write code that performs a sequential search, binary search, selection sort, and so forth, on a `List`? Why or why not?

## Key Terms

|                     |                             |
|---------------------|-----------------------------|
| Add method          | off-by-one error            |
| array               | one-dimensional             |
| binary search       | rectangular arrays          |
| Clear method        | reference                   |
| Count property      | reference copy              |
| elements            | reference types             |
| foreach loop        | reference variable          |
| garbage collection  | Remove method               |
| IndexOf method      | RemoveAt method             |
| initialization list | search algorithms           |
| Insert method       | selection sort              |
| iteration variable  | sequential search algorithm |
| jagged array        | size declarator             |
| Length property     | subscript                   |
| List                | two-dimensional             |
| new operator        | value types                 |

## Review Questions

### Multiple Choice

- The memory that is allocated for a \_\_\_\_\_ variable is the actual location that will hold any value that is assigned to that variable.
  - reference type
  - general type
  - value type
  - framework type
- A variable that is used to reference an object is commonly called a(n) \_\_\_\_\_.
  - reference variable
  - resource variable
  - object variable
  - component variable
- When you want to work with an object, you use a variable that holds a special value known as a(n) \_\_\_\_\_ to link the variable to the object.
  - union
  - reference
  - object linker
  - data coupling
- The \_\_\_\_\_ creates an object in memory and returns a reference to that object.
  - = operator
  - object allocator
  - reference variable
  - new operator
- A(n) \_\_\_\_\_ is an object that can hold a group of values that are all of the same data type.
  - array
  - collection

- c. container
  - d. set
6. The \_\_\_\_\_ indicates the number of values that the array should be able to hold.
- a. allocation limit
  - b. size declarator
  - c. data type
  - d. compiler
7. The storage locations in an array are known as \_\_\_\_\_.
- a. elements
  - b. sectors
  - c. pages
  - d. blocks
8. Each element in an array is assigned a unique number known as a(n) \_\_\_\_\_.
- a. element identifier
  - b. subscript
  - c. index
  - d. sequencer
9. When you create an array, you can optionally initialize it with a group of values called a(n) \_\_\_\_\_.
- a. default value group
  - b. initialization list
  - c. defined set
  - d. value list
10. In C#, all arrays have a \_\_\_\_\_ that is set to the number of elements in the array.
- a. Limit property
  - b. Size property
  - c. Length property
  - d. Maximum property
11. A(n) \_\_\_\_\_ occurs when a loop iterates one time too many or one time too few.
- a. general error
  - b. logic error
  - c. loop count error
  - d. off-by-one error
12. C# provides a special loop that, in many circumstances, simplifies array processing. It is known as the \_\_\_\_\_.
- a. for loop
  - b. foreach loop
  - c. while loop
  - d. do-while loop
13. The `foreach` loop is designed to work with a temporary, read-only variable that is known as the \_\_\_\_\_.
- a. element variable
  - b. loop variable
  - c. index variable
  - d. iteration variable

14. \_\_\_\_\_ is a process that periodically runs, removing all unreferenced objects from memory.
  - a. Systematic reallocation
  - b. Memory cleanup
  - c. Garbage collection
  - d. Object maintenance
15. Various techniques known as \_\_\_\_\_ have been developed to locate a specific item in a larger collection of data, such as an array.
  - a. seek functions
  - b. request methods
  - c. traversal procedures
  - d. search algorithms
16. The \_\_\_\_\_ uses a loop to step through an array, starting with the first element, searching for an item.
  - a. sequential search algorithm
  - b. top-down method
  - c. ascending search algorithm
  - d. basic search function
17. A(n) \_\_\_\_\_ is a type of assignment operation that copies a reference to an array and not the contents of the array.
  - a. object copy
  - b. reference copy
  - c. double reference
  - d. parallel copy
18. The \_\_\_\_\_ is a clever algorithm that is much more efficient than the sequential search.
  - a. linear search
  - b. bubble sort
  - c. binary search
  - d. selection sort
19. A \_\_\_\_\_ is similar to a two-dimensional array, but the rows can have different numbers of columns.
  - a. one-dimensional array
  - b. columnar array
  - c. jagged array
  - d. split row array
20. The .NET Framework provides a class named \_\_\_\_\_, which can be used for storing and retrieving items.
  - a. Matrix
  - b. Database
  - c. Container
  - d. List

### True or False

1. When you are working with a value type, you are using a variable that holds a piece of data.
2. Reference variables can be used only to reference objects.



- Individual variables are well suited for storing and processing lists of data.
- Arrays are reference type objects.
- You can store a mixture of data types in an array.
- When you create a numeric array in C#, its elements are set to the value 0 by default.
- The subscript of the last element will always be one less than the array's `Length` property.
- You use the `==` operator to compare two array reference variables and determine whether the arrays are equal.
- A jagged array is similar to a two-dimensional array, but the rows in a jagged array can have different numbers of columns.
- When you create a `List` object, you do not have to know the number of items that you intend to store in it.

### Short Answer

- How much memory is allocated by the compiler when you declare a value type variable?
- What type of variable is needed to work with an object in code?
- What two steps are typically required for creating a reference type object?
- Are variables well suited for processing lists of data? Why or why not?
- What value is returned by the `Length` property of an array?
- What can cause an off-by-one error when working with an array?
- How do you keep track of elements that contain data in a partially filled array?
- Briefly describe the selection sort algorithm.
- How is the binary search more efficient than the sequential search algorithm?
- What advantages does a `List` have over an array?

### Algorithm Workbench

- Assume `names` is a variable that references an array of 20 `string` values. Write a `foreach` loop that displays each of the elements of the array in a `ListBox` control.
- The variables `numberArray1` and `numberArray2` reference arrays that have 100 elements each. Write code that copies the values from `numberArray1` to `numberArray2`.
- Write code for a sequential search that determines whether the value `-1` is stored in an array with a reference variable named `values`. The code should display a message indicating whether the value was found.
- Write a declaration statement that creates a two-dimensional array referenced by a variable named `grades`. The array should store `int` values using 18 rows and 12 columns.
- Write code that sums each column in the array in Question 4.
- Create a `List` object that uses the binary search algorithm to search for the string "A". Display a message box indicating whether the value was found.

## Programming Problems



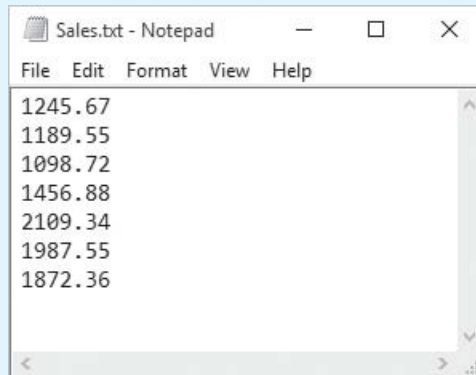
VideoNote

Solving the  
Total Sales  
Problem

### 1. Total Sales

In the *Chap07* folder of the Student Sample Programs, you will find a file named *Sales.txt*. Figure 7-44 shows the file's contents displayed in Notepad. Create an application that reads this file's contents into an array, displays the array's contents in a `ListBox` control, and calculates and displays the total of the array's values.

**Figure 7-44** The *Sales.txt* file



### 2. Sales Analysis

Modify the application that you created in Programming Exercise 1 so it also displays the following:

- The average of the values in the array
- The largest value in the array
- The smallest value in the array

### 3. Charge Account Validation

In the *Chap07* folder of the Student Sample Programs, you will find a file named *ChargeAccounts.txt*. The file contains a list of a company's valid charge account numbers. There are a total of 18 charge account numbers in the file, and each one is a 7-digit number, such as 5658845.

Create an application that reads the contents of the file into an array or a `List`. The application should then let the user enter a charge account number. The program should determine whether the number is valid by searching for it in the array or `List` that contains the valid charge account numbers. If the number is in the array or `List`, the program should display a message indicating the number is valid. If the number is not in the array or `List`, the program should display a message indicating the number is invalid.

### 4. Driver's License Exam

The local driver's license office has asked you to create an application that grades the written portion of the driver's license exam. The exam has 20 multiple-choice questions. Here are the correct answers:

- |       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 1. B  | 2. D  | 3. A  | 4. A  | 5. C  |
| 6. A  | 7. B  | 8. A  | 9. C  | 10. D |
| 11. B | 12. C | 13. D | 14. A | 15. D |
| 16. C | 17. C | 18. B | 19. D | 20. A |

Your program should store these correct answers in an array. The program should read the student's answers for each of the 20 questions from a text file and store the answers in another array. (Create your own text file to test the application.) After the student's answers have been read from the file, the program should display a message indicating whether the student passed or failed the exam. (A student must correctly answer 15 of the 20 questions to pass the exam.) It should then display the total number of correctly answered questions, the total number of incorrectly answered questions, and a list showing the question numbers of the incorrectly answered questions.

### 5. World Series Champions

In the *Chap07* folder of the Student Sample Programs, you will find the following files:

- **Teams.txt**—This file contains a list of several Major League baseball teams in alphabetical order. Each team listed in the file has won the World Series at least once.
- **WorldSeriesWinners.txt**—This file contains a chronological list of the World Series' winning teams from 1903 through 2012. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2012. Note that the World Series was not played in 1904 or 1994.)

Create an application that displays the contents of the **Teams.txt** file in a **ListBox** control. When the user selects a team in the **ListBox**, the application should display the number of times that team has won the World Series in the time period from 1903 through 2012.



**TIP:** Read the contents of the **WorldSeriesWinners.txt** file into a **List** or an array. When the user selects a team, an algorithm should step through the list or array counting the number of times the selected team appears.

### 6. Name Search

In the *Chap07* folder of the Student Sample Programs, you will find the following files:

- **GirlNames.txt**—This file contains a list of the 200 most popular names given to girls born in the United States from 2000 through 2009.
- **BoyNames.txt**—This file contains a list of the 200 most popular names given to boys born in the United States from 2000 through 2009.

Create an application that reads the contents of the two files into two separate arrays or **Lists**. The user should be able to enter a boy's name, a girl's name, or both, and the application should display messages indicating whether the names were among the most popular.

### 7. Population Data

In the *Chap07* folder of the Student Sample Programs, you will find a file named **USPopulation.txt**. The file contains the midyear population of the United States, in thousands, during the years 1950 through 1990. The first line in the file contains the population for 1950, the second line contains the population for 1951, and so forth.

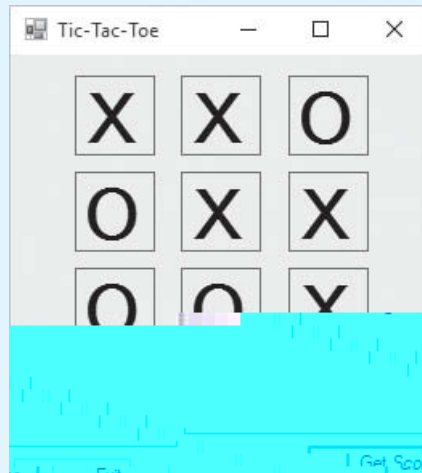
Create an application that reads the file's contents into an array or a **List**. The application should display the following data:

- The average annual change in population during the time period
- The year with the greatest increase in population during the time period
- The year with the least increase in population during the time period

## 8. Tic-Tac-Toe Simulator

Create an application that simulates a game of tic-tac-toe. Figure 7-45 shows an example of the application's form. The form shown in the figure uses eight large Label controls to display the Xs and Os.

Figure 7-45 The Tic-Tac-Toe application



The application should use a two-dimensional `int` array to simulate the game board in memory. When the user clicks the `New Game` button, the application should step through the array, storing a random number in the range of 0 through 1 in each element. The number 0 represents the letter O, and the number 1 represents the letter X. The form should then be updated to display the game board. The application should display a message indicating whether player X won, player Y won, or the game was a tie.

## 9. Jagged Array of Exam Scores

Dr. Hunter teaches three sections of her Intro to Computer Science class. She has 12 students in section 1, 8 students in section 2, and 10 students in section 3. In the `Chap07` folder of the Student Sample Programs, you will find the following files:

a 6HFWLRQ W[W`7KLV ILOH FRQWDLQV WKH ILQDO H[DP VFRU  
(There are 12 integer scores in the file.)

a 6HFWLRQ W[W`7KLV ILOH FRQWDLQV WKH ILQDO H[DP VFRU  
(There are 8 integer scores in the file.)

a 6HFWLRQ W[W`7KLV ILOH FRQWDLQV WKH ILQDO H[DP VFRU  
(There are 10 integer scores in the file.)

Create an application that reads these three files and stores their contents in a jagged array. The array's first row should hold the exam scores for the students in section 1, the second row should hold the exam scores for the students in section 2, and the third row should hold the exam scores for the students in section 3.

The application should display each section's exam scores in a separate `ListBox` control and then use the jagged array to determine the following:

a 7KH DYHUDJH H[DP VFRUH IRU HDFK LQGLYLGXDO VHFWRU

a 7KH DYHUDJH H[DP VFRUH IRU DOO WKH VWXGHQWV LQ WK

a 7KH KLJKHVW H[DP VFRUH DPRQJ DOO WKUHH VHFWRQV D  
that score was found

a 7KH ORZHVW H[DP VFRUH DPRQJ DOO WKUHH VHFWRQV D  
that score was found