CHAPTER FOUR
ARRAY

4.1 Introduction

This chapter introduces the important topic of data structures—collections of related data items. Arrays are data structures consisting of related data items of the same type. Arrays are fixed-length entities—they remain the same length once they're created, although an array variable may be reassigned such that it refers to a new array of a different length.

After discussing how arrays are declared, created and initialized, I present a series of examples that demonstrate several common array manipulations. The chapter demonstrates C#'s last structured control statement—the ***foreach*** *repetition statement—which provides a concise notation for accessing data in arrays.*

4.2 Arrays

An array is a group of variables (called elements) containing values that all have the same type. Recall that types are divided into two categories—value types and reference types. Arrays are reference types. As you'll see, what I typically think of as an array is actually a reference to an array object. The elements of an array can be either value types or reference types, including other arrays. To refer to a particular element in an array, we specify the name of the reference to the array and the position number of the element in the array, which is known as the element's index.

Figure ***4.1*** *shows a logical representation of an integer array called* ***c****. This array contains* ***12*** *elements. An application refers to any one of these elements with an array-access expression that includes the name of the array, followed by* *the index of the particular element in square brackets ([]). The first element in every array has index zero and is some- times called the zeroth element. Thus, the elements of array c are c[0], c[1], c[2] and so on. The highest index in array c is 11, which is one less than the number of elements in the array, because indices begin at 0. Array names follow the same conventions as other variable names. An index must be a nonnegative integer and can be an expression. For example, if we assume that variable a is 5 and variable b is 6, then the statement*

```
                c[ a + b ] += 2;
```

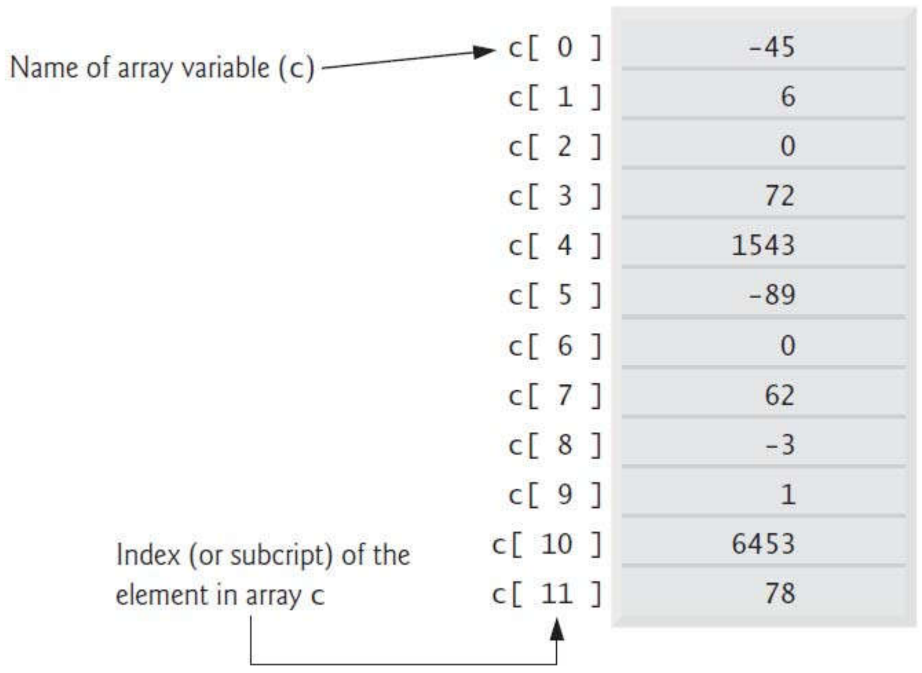

Fig. 4.1 | A 12-element array.

adds 2 to array element c[11]. An indexed array name is an array-access expression. Such expressions can be used on the left side of an assignment to place a new value into an array element. The array index must be a value of type int, uint, long or ulong, or a value of a type that can be implicitly promoted to one of these types.

Let's examine array c in Fig. 4.1 more closely. The name of the variable that references the array is c. Every array instance knows its own length and provides access to this information with the Length property. For example, the expression *c.Length uses array c's Length property to determine the length of the array (that is, 12). The Length property of an array cannot be changed, because it does not provide a set accessor. The array's 12 elements are referred to as c[0], c[1], c[2], ..., c[11]. Referring to elements outside of this range, such as c[-1] or c[12] is a runtime error. The value of c[0] is -45, the value of c[1] is 6, the value of c[2] is 0, the value of c[7] is 62 and the value of c[11] is 78. To calculate the sum of the values contained in the first three elements of array c and store the result in variable sum, we would write*

```
        sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

To divide the value of c[6] by 2 and assign the result to the variable x, we would write

```
        x = c[ 6 ] / 2;
```

4.3 Declaring and Creating Arrays

Arrays occupy space in memory. Since they're objects, they're typically created with key- word new. To create an array object, you specify the type and the number of array elements as part of an array-creation expression that uses keyword new. Such an expression returns a reference that can be stored in an array variable. The following declaration and array- creation expression create an array object containing 12 int elements and store the array's reference in variable c:

```
    int[] c = new  int[ 12 ];
```

This expression can be used to create the array shown in Fig. 4.1 (but not the initial values in the array—we'll show how to initialize the elements of an array momentarily). This task also can be performed as follows:

```
    int[] c; // declare the array variable
    c = new int[ 12 ]; // create the array; assign to array variable
```

In the declaration, the square brackets following the type int indicate that c is a variable that will refer to an array of ints (i.e., c will store a reference to an array object). In the assignment statement, the array variable c receives the *reference to a new array object of 12 int elements. The number of elements can also be specified as an expression that's calculated at execution time. When an array is created, each element of the array receives a default value—0 for the numeric simple-type elements, false for bool elements and null for references. As we'll soon see, we can provide specific, non default initial element values when we create an array. An application can create several arrays in a single declaration. The following statement reserves 100 elements for string array b and 27 elements for string array x:*

```
        string[] b = new string[ 100 ], x = new string[ 27 ];
```

In this statement, string[] applies to each variable. For readability and ease of commenting, we prefer to split the preceding statement into two statements, as in:

```
        string[] b = new string[ 100 ]; // create string array b
        string[] x = new string[ 27 ]; // create string array x
```

An application can declare variables that will refer to arrays of value-type elements or reference-type elements. For example, every element of an int array is an int value, and every element of a string array is a reference to a string object.

Resizing an Array

Though arrays are fixed-length entities, you can use the static Array method Resize, which takes two arguments—the array to be resized and the new length—to create a new array with the specified length. This method copies the contents of the old array into the new array and sets the variable it receives as its first argument to reference the new array. For example, consider the following statements:

```
        int[] newArray = new int[ 5 ];
        Array.Resize( ref newArray, 10 );
```

The variable newArray initially refers to a five-element array. The resize method sets newArray to refer to a new 10-element array. If the new array is smaller than the old array, any content that cannot fit into the new array is truncated without warning.

4.4 Examples Using Arrays

This section presents several examples that demonstrate declaring arrays, creating arrays, initializing arrays and manipulating array elements.

Creating and Initializing an Array

The application of Fig. 4.2 uses keyword new to create an array of five int elements that are initially 0 (the default for int variables).

```
 1  // Fig. 4.2: InitArray.cs
 2  // Creating an array.
 3  using System;
 4
 5  public class InitArray
 6  {
 7     public static void Main( string[] args )
 8     {
 9        int[] array; // declare array named array
10
11        // create the space for array and initialize to default zeros
12        array = new int[ 5 ]; // 5 int elements
13
14        Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings
15
16        // output each array element's value
17        for ( int counter = 0; counter < array.Length; counter++ )
18           Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ]
);
19     } // end Main
20  } // end class InitArray
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

Fig. 4.2 | Creating an array.

Line 9 declares array—a variable capable of referring to an array of int elements. Line 12 creates the five-element array object and assigns its reference to variable array. Line 14 outputs the column headings. The first column contains the index (0–9) of each array element, and the second column contains the default value (0) of each array element and has a field width of 8.

The for statement in lines 17–18 outputs the index number (represented by counter) and the value (represented by array[counter]) of each array element. The loop-control variable counter is initially 0—index values start at 0, so using zero-based counting allows the loop to access every element of the array. The for statement's loop-continuation condition uses the property array.Length (line 17) to obtain the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable counter is less than 10. The highest index value of a 10-element array is 9, so using the less-than operator in the loop-continuation condition guarantees that the loop does not attempt to access an element beyond the end of the array (i.e., during the final iteration of the loop, counter is 9). I'll soon see what happens when such an out- of-range index is encountered at execution time.

Using an Array Initializer

An application can create an array and initialize its elements with an array initializer, which is a comma-separated list of expressions (called an initializer list) enclosed in braces. In this case, the array length is determined by the number of elements in the initializer list. For example, the declaration

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

creates a five-element array with index values 0, 1, 2, 3 and 4. Element n[0] is initialized to 10, n[1] is initialized to 20 and so on. This statement does not require new to create the array object. When the compiler encounters an array initializer list, the compiler counts the number of initializers in the list to determine the size of the array, then sets up the ap- propriate new operation "behind the scenes." The application in Fig. 4.3 initializes an in- teger array with 10 values (line 10) and displays the array in tabular format. The code for displaying the array elements (lines 15–16) is identical to that in Fig. 4.2 (lines 17–18).

```
// Fig. 4.3: InitArray.cs
// Initializing the elements of an array with an array initializer.
using System;

public class InitArray
{
   public static void Main( string[] args )
   {
      // initializer list specifies the value for each element
      int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings

      // output each array element's value
      for ( int counter = 0; counter < array.Length; counter++ )
         Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
   } // end Main
} // end class InitArray
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Fig. 4.3 | Initializing the elements of an array with an array initializer.

Calculating a Value to Store in Each Array Element

Some applications calculate the value to be stored in each array element. The application in Fig. 4.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, ..., 20). Then the application displays the array in tabular format. The for statement at lines 13–14 calculates an array element's value by multiplying the current value of the for loop's control variable counter by 2, then adding 2.

```
1  // Fig. 4.4: InitArray.cs
2  // Calculating values to be placed into the elements of an array.
3  using System;
4
5  public class InitArray
6  {
 7      public static  void Main( string[] args )
 8      {
 9         const int ARRAY_LENGTH = 10; // create a named constant
10         int[] array = new int[ ARRAY_LENGTH ]; // create array
11
12         // calculate value for each array element
13         for ( int counter = 0; counter < array.Length; counter++ )
14            array[ counter ] = 2 + 2 * counter;
15
16         Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings
17
18         // output each array element's value
19         for ( int counter = 0; counter < array.Length; counter++ )
20            Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ]
);
21      } // end Main
22   } // end class InitArray
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Fig. 4.4 | Calculating values to be placed into the elements of an array.

*Line 9 uses the modifier **const** to declare the constant **ARRAY_LENGTH**, whose value is 10. Constants must be initialized when they're declared and cannot be modified thereafter. We declare constants with all capital letters by convention to make them stand out in the code.*

Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, an instructor may wish to total the elements and use that total to calculate the class average for the exam. The application in Fig. 4.5 sums the values contained in a 10-element integer array. The application creates and initializes the array at line 9. The for statement performs the calculations.

```
// Fig. 4.5: SumArray.cs
// Computing the sum of the elements of an array.
using System;

public class SumArray
{
   public static void Main( string[] args )
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int counter = 0; counter < array.Length; counter++ )
         total += array[ counter ];

      Console.WriteLine( "Total of array elements: {0}", total );
   } // end Main
} // end class SumArray
```

```
Total of array elements: 849
```

Fig. 4.5 | Computing the sum of the elements of an array.

Using Bar Charts to Display Array Data Graphically

Many applications present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent proportionally larger numeric values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*). An instructor might graph the number of grades in each of several categories to visualize the grade distribution for the exam. Suppose the grades on an exam were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. There was one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Our next application (Fig. 4.6) stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, array[0] indicates the number of grades in the range 0–9, array[7] the number of grades in the range 70–79 and array[10] the number of 100 grades. For now, we manually create array by examining the set of grades and initializing the elements of array to the number of values in each range (line 9).

The application reads the numbers from the array and graphs the information as a bar chart. Each grade range is followed by a bar of asterisks indicating the number of grades in that range.

To label each bar, lines 17–21 output a grade range (e.g., "70-79: ") based on the current value of counter. When counter is 10, line 18 outputs " 100: " to align the colon with the other bar labels. When counter is not 10, line 20 uses the format items {0:D2} and {1:D2} to output the label of the grade range. The format specifier D indicates that the value should be formatted as an integer, and the number after the D indicates how

```
 1  // Fig. 4.6: BarChart.cs
 2  // Bar chart displaying application.
 3  using System;
 4
 5  public class BarChart
 6  {
 7     public static void Main( string[] args )
 8     {
 9        int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
10
11        Console.WriteLine( "Grade distribution:" );
12
13        // for each array element, output a bar of the chart
14        for ( int counter = 0; counter < array.Length; counter++ )
15        {
16           // output bar labels ( "00-09: ", ..., "90-99: ", "100: " )
17           if ( counter == 10 )
18              Console.Write( "  100: " );
19           else
20              Console.Write( "{0:D2}-{1:D2}: ",
21                 counter * 10, counter * 10 + 9 );
22
23           // display bar of asterisks
24           for ( int stars = 0; stars < array[ counter ]; stars++ )
25              Console.Write( "*" );
26
27           Console.WriteLine(); // start a new line of output
28        } // end outer for
29     } // end Main
30  } // end class BarChart
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 4.6 | Bar chart displaying application.

many digits this formatted integer must contain. The 2 indicates that values with fewer than two digits should begin with a leading 0.The nested for statement (lines 24–25) outputs the bars. Note the loop-continuation condition at line 24 (stars < array[counter]). Each time the application reaches the inner for, the loop counts from 0 up to one less than array[counter], thus using a value in array to determine the number of asterisks to display. In this example, array[0]–array[5] contain 0s because no students received a grade below 60. Thus, the application displays no asterisks next to the first six grade ranges.

4.6 foreach Statement

In previous examples, we demonstrated how to use counter-controlled for statements to iterate through the elements in an array. In this section, we introduce the ***foreach*** *statement, which iterates through the elements of an entire array or collection. This section dis- cusses how to use the* ***foreach*** *statement to loop through an array. The syntax of a* ***foreach*** *statement is:*

```
foreach ( type identifier in arrayName )
   statement
```

where type and identifier are the type and name (e.g., int number) of the iteration variable, and arrayName is the array through which to iterate. The type of the iteration variable must be consistent with the type of the elements in the array. As the next example illustrates, the iteration variable represents successive values in the array on successive iterations of the foreach statement.

Figure 4.12 uses the foreach statement (lines 13–14) to calculate the sum of the integers in an array of student grades. The type specified is int, because array contains int values—therefore, the loop will select one int value from the array during each iteration. The foreach statement iterates through successive values in the array one by one. The foreach header can be read concisely as "for each iteration, assign the next element of array to int variable number, then execute the following statement." Thus, for each iteration, identifier number represents the next int value in the array. Lines 13–14 are equivalent to the following counter-controlled repetition used in lines 13–14 of Fig. 4.5 to total the integers in array:

```
for ( int counter = 0; counter < array.Length; counter++ )
    total += array[ counter ];
```

```
// Fig. 4.12: ForEachTest.cs
// Using the foreach statement to total integers in an array.
using System;

public class ForEachTest
{
   public static void Main( string[] args )
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      foreach ( int number in array )
         total += number;

      Console.WriteLine( "Total of array elements: {0}", total );
   } // end Main
} // end class ForEachTest
```

```
Total of array elements: 849
```

Fig. 4.12 | Using the `foreach` statement to total integers in an array.

The foreach statement can be used in place of the for statement whenever code looping through an array does not require access to the counter indicating the index of the current array element. For example, totaling the integers in an array requires access only to the element values—the index of each element is irrelevant. However, if an application must use a counter for some reason other than simply to loop through an array (e.g., to display an index number next to each array element value, as in the examples earlier in this chapter), use the for statement.

Implicitly Typed Local Variables

In each for statement presented so far and in the foreach statement of Fig. 4.12, we declared the type of the control variable either in the for or foreach statement's header. C# provides a new feature—called implicitly typed local variables—that enables the compiler to infer a local variable's type based on the type of the variable's initializer. To distinguish such an initialization from a simple assignment statement, the **`var`** *keyword is used in place of the variable's type. Recall that a local variable is any variable declared in the body of amethod. In the declaration*

```
var x = 7;
```

the compiler infers that the variable x should be of type int, because the compiler assumes that whole-number values, like 7, are of type int. Similarly, in the declaration

```
var y = -123.45;
```

the compiler infers that the variable y should be of type double, because the compiler assumes that floating-point number values, like -123.45, are of type double. You can also use local type inference with control variables in the header of a for or foreach statement. For example, the for statement header

```
for ( int counter = 1; counter < 10; counter++ )
```

can be written as

```
for ( var counter = 1; counter  < 10; counter++ )
```

In this case, counter is of type int because it's initialized with a whole-number value (1). Similarly, assuming that myArray is an array of ints, the foreach statement header

```
foreach ( int number in myArray )
```

can be written as

```
foreach ( var number in myArray )
```

In this case, number is of type int because it's used to process elements of the int array myArray. For example, the following statement creates an array of int values:

```
var array = new[] { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
```

4.7 Passing Arrays and Array Elements to Methods

*To pass an array argument to a method, specify the name of the array without any brackets. For example, if **hourlyTemperatures** is declared as*

```
double[] hourlyTemperatures  = new double[ 24 ];
```

then the method call

```
ModifyArray( hourlyTemperatures  );
```

passes the reference of array hourlyTemperatures to method ModifyArray. Every array object "knows" its own length (and makes it available via its Length property). Thus, when we pass an array object's reference to a method, we need not pass the array length as an additional argument. For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter. For example, the method header for method ModifyArray might be written as

```
        void ModifyArray( double[] b )
```

indicating that ModifyArray receives the reference of an array of doubles in parameter b. The method call passes array hourlyTemperature's reference, so when the called method uses the array variable b, it refers to the same array object as hourlyTemperatures in the calling method. When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference. However, when an argument to a method is an individual array element of a value type, the called method receives a copy of the element's value. To pass an individual array element to a method, use the indexed name of the array as an argument in the method call. If you want to pass a value- type array element to a method by reference, you must use the ref keyword .

Figure 4.13 demonstrates the difference between passing an entire array and passing a value-type array element to a method. The foreach statement at lines 17–18 outputs the five elements of array (an array of int values). Line 20 invokes method ModifyArray, passing array as an argument. Method ModifyArray (lines 37–41) receives a copy of array's reference and uses the reference to multiply each of array's elements by 2. To prove that array's elements (in Main) were modified, the foreach statement at lines 24–25 outputs the five elements of array again. As the output shows, method ModifyArray doubled the value of each element.

```
// Fig. 4.13: PassArray.cs
// Passing arrays and individual array elements to methods.
using System;

public class PassArray
{
   // Main creates array and calls ModifyArray and ModifyElement
   public static void Main( string[] args )
   {
      int[] array = { 1, 2, 3, 4, 5 };

      Console.WriteLine(
         "Effects of passing reference to entire array:\n" +
         "The values of the original array are:" );

      // output original array elements
      foreach ( int value in array )
         Console.Write( "   {0}", value );

      ModifyArray( array ); // pass array reference
      Console.WriteLine( "\n\nThe values of the modified array are:" );

      // output modified array elements
      foreach ( int value in array )
         Console.Write( "   {0}", value );

      Console.WriteLine(
         "\n\nEffects of passing array element value:\n" +
         "array[3] before ModifyElement: {0}", array[ 3 ] );

      ModifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
      Console.WriteLine(
         "array[3] after ModifyElement: {0}", array[ 3 ] );
   } // end Main

   // multiply each element of an array by 2
   public static void ModifyArray( int[] array2 )
   {
      for ( int counter = 0; counter < array2.Length; counter++ )
         array2[ counter ] *= 2;
   } // end method ModifyArray

   // multiply argument by 2
   public static void ModifyElement( int element )
   {
      element *= 2;
      Console.WriteLine(
         "Value of element in ModifyElement: {0}", element );
   } // end method ModifyElement
} // end class PassArray
```

Fig. 4.13 | Passing arrays and individual array elements to methods

Output of previous Progarm

```
Effects of passing reference to entire array: The
values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before ModifyElement: 8
Value of element in ModifyElement:
16 array[3] after ModifyElement: 8
```

Figure 4.13 next demonstrates that when a copy of an individual value-type array element is passed to a method, modifying the copy in the called method does not affect the original value of that element in the calling method's array. To show the value of array[3] before invoking method ModifyElement, lines 27–29 output the value of array[3], which is 8. Line 31 calls method ModifyElement and passes array[3] as an argument. Remember that array[3] is actually one int value (8) in array. Therefore, the application passes a copy of the value of array[3]. Method ModifyElement (lines 44–49) multiplies the value received as an argument by 2, stores the result in its parameter element, then outputs the value of element (16). Since method parameters, like local variables, cease to exist when the method in which they're declared completes execution, the method parameter ele- ment is destroyed when method ModifyElement terminates. Thus, when the application returns control to Main, lines 32–33 output the unmodified value of array[3] (i.e., 8).

4.8 Passing Arrays by Value and by Reference

In C#, a variable that "stores" an object, such as an array, does not actually store the object itself. Instead, such a variable stores a reference to the object. The distinction between reference-type variables and value-type variables raises some subtle issues that you must understand to create secure, stable programs.

As you know, when an application passes an argument to a method, the called method receives a copy of that argument's value. Changes to the local copy in the called method do not affect the original variable in the caller. If the *argument is of a reference type, the method makes a copy of the reference, not a copy of the actual object that's referenced. The local copy of the reference also refers to the original object, which means that changes to the object in the called method affect the original object.*

In Section 4.8, you learned that C# allows variables to be passed by reference with keyword ref. You can also use keyword ref to pass a reference-type variable by reference, which allows the called method to modify the original variable in the caller and make that variable refer to a different object. This is a subtle capability, which, if misused, can lead to problems.

For instance, when a reference-type object like an array is passed with ref, the called method actually gains control over the reference itself, allowing the called method to replace the original reference in the caller with a reference to a different object, or even with null. Such behavior can lead to unpredictable effects, which can be disastrous in mission-critical applications.

The application in Fig. 4.14 demonstrates the subtle difference between passing a reference by value and passing a reference by reference with keyword ref.

```
// Fig. 4.14: ArrayReferenceTest.cs
// Testing the effects of passing array references
// by value and by reference.
using System;

public class ArrayReferenceTest
{
   public static void Main( string[] args )
   {
      // create and initialize firstArray
      int[] firstArray = { 1, 2, 3 };

      // copy the reference in variable firstArray
      int[] firstArrayCopy = firstArray;

      Console.WriteLine(
         "Test passing firstArray reference by value" );

      Console.Write( "\nContents of firstArray " +
         "before calling FirstDouble:\n\t" );

      // display contents of firstArray
      for ( int i = 0; i < firstArray.Length;  i++ )
         Console.Write( "{0} ", firstArray[ i ] );

      // pass variable firstArray by value to FirstDouble
      FirstDouble( firstArray );

      Console.Write( "\n\nContents of firstArray after " +
         "calling  FirstDouble\n\t");

      // display contents of firstArray
      for ( int i = 0; i < firstArray.Length;  i++ )
         Console.Write( "{0} ", firstArray[ i ] );

      // test whether reference was changed by FirstDouble
      if ( firstArray  == firstArrayCopy )
         Console.WriteLine(
            "\n\nThe references refer  to the  same array"  );
      else
         Console.WriteLine(
            "\n\nThe references refer  to different arrays" );

      // create and initialize secondArray
      int[] secondArray  = { 1, 2, 3 };

      // copy the reference in variable secondArray
      int[] secondArrayCopy  = secondArray;

      Console.WriteLine(  "\nTest passing secondArray " +
         "reference by reference" );

      Console.Write( "\nContents of secondArray " +
         "before calling  SecondDouble:\n\t" );

      // display contents of secondArray before method call
      for ( int i = 0; i < secondArray.Length;  i++ )
         Console.Write( "{0} ", secondArray[ i ] );

      // pass variable secondArray by reference to SecondDouble
      SecondDouble( ref secondArray );

      Console.Write( "\n\nContents of secondArray " +
         "after calling SecondDouble:\n\t" );

      // display contents of secondArray after method call
      for ( int i = 0; i < secondArray.Length;  i++ )
         Console.Write( "{0} ", secondArray[ i ] );

      // test whether reference was changed by SecondDouble
      if ( secondArray  == secondArrayCopy  )
         Console.WriteLine(
            "\n\nThe references refer  to the  same array"  );
      else
```

```
            Console.WriteLine(
               "\n\nThe references refer to different arrays" );
      } // end Main

      // modify elements of array and attempt to modify reference
      public static void FirstDouble( int[] array )
      {
         // double each element's value
         for ( int i = 0; i < array.Length; i++ )
            array[ i ] *= 2;

         // create new object and assign its reference to array
         array = new int[] { 11, 12, 13 };
      } // end method FirstDouble

      // modify elements of array and change reference array
      // to refer to a new array
      public static void SecondDouble(ref int[] array )
      {
         // double each element's value
         for ( int i = 0; i < array.Length; i++ )
            array[ i ] *= 2;

         // create new object and assign its reference to array
         array = new int[] { 11, 12, 13 };
      } // end method SecondDouble
   } // end class ArrayReferenceTest
```

```
Test passing firstArray reference by value

Contents of firstArray before calling FirstDouble:
         1 2 3

Contents of firstArray after calling FirstDouble
         2 4 6

The references refer to the same array

Test passing secondArray reference by reference

Contents of secondArray before calling SecondDouble:
         1 2 3

Contents of secondArray after calling SecondDouble:
         11 12 13

The references refer to different arrays
```

Fig. 4.14 | Passing an array reference by value and by reference.

Lines 11 and 14 declare two integer array variables, firstArray and firstArrayCopy. Line 11 initializes firstArray with the values 1, 2 and 3. The assignment statement at line 14 copies the reference stored in firstArray to variable firstArrayCopy, causing these variables to reference the same array object. We make the copy of the reference so that we can determine later whether reference firstArray gets overwritten. The for statement at lines 23–24 displays the contents of firstArray before it's passed to method FirstDouble (line 27) so that we can verify that the called method indeed changes the array's contents.

The for statement in method FirstDouble (lines 83–84) multiplies the values of all the elements in the array by 2. Line 87 creates a new array containing the values 11, 12 and 13, and assigns the array's reference to parameter array in an attempt to overwrite reference firstArray in the caller—this, of course, does not happen, because the reference was passed by value. After method FirstDouble executes, the for statement at lines 33–34 dis- plays the contents of firstArray, demonstrating that the values of the elements have been changed by the method. The if...else statement at lines 37–42 uses the == operator to compare references firstArray (which we just attempted to overwrite) and firstArray- Copy. The expression in line 37 evaluates to true if the operands of operator == reference the same object. In this case, the object represented by firstArray is the array created in line 11—not the array created in method FirstDouble (line 87)—so the original reference stored in firstArray was not modified.

Lines 45–76 perform similar tests, using array variables secondArray and second- ArrayCopy, and method SecondDouble (lines 92–100). Method SecondDouble performs the same operations as FirstDouble, but receives its array argument using keyword ref. In this case, the reference stored in secondArray after the method call is a reference to the array created in line 99 of SecondDouble, demonstrating that a variable passed with key- word ref can be modified by the called method so that the variable in the caller actually points to a different object—in this case, an array created in SecondDouble. The if...else statement in lines 71–76 confirms that secondArray and secondArrayCopy no longer refer to the same array.