CHAPTER 6

Using Arrays

In this chapter you will:

- Declare an array and assign values to array elements
- Access array elements
- Search an array using a loop
- Use the `BinarySearch()`, `Sort()`, and `Reverse()` methods
- Use multidimensional arrays
- Learn about array issues in GUI programs

Storing values in variables provides programs with flexibility—a program that uses variables to replace constants can manipulate different values each time the program executes. When you add loops to your programs, the same variable can hold different values during successive cycles through the loop within the same program execution. This ability makes the program even more flexible. Learning to use the data structure known as an array offers further flexibility—you can store multiple values in adjacent memory locations and access them by varying a value that indicates which of the stored values to use. In this chapter, you will learn to create and manage C# arrays.

Declaring an Array and Assigning Values to Array Elements

Sometimes, storing just one value in memory at a time is not adequate. For example, a sales manager who supervises 20 employees might want to determine whether each employee has produced sales above or below the average amount. When you enter the first employee's sales figure into a program, you cannot determine whether it is above or below average because you will not know the average until you have entered all 20 figures. You might plan to assign 20 sales figures to 20 separate variables, each with a unique name, then sum and average them. That process is awkward and unwieldy, however—you need 20 prompts, 20 input statements using 20 separate storage locations (in other words, 20 separate variable names), and 20 addition statements. This method might work for 20 salespeople, but what if you have 30, 40, or 10,000 salespeople?

A superior approach is to assign the sales value to the same variable in 20 successive iterations through a loop that contains one prompt, one input statement, and one addition statement. Unfortunately, when you enter the sales value for the second employee, that data item replaces the figure for the first employee, and the first employee's value is no longer available to compare to the average of all 20 values. With this approach, when the data-entry loop finishes, the only sales value left in memory is the last one entered.

The best solution to this problem is to create an array. An **array** is a list of data items that all have the same data type and the same name. Each object in an array is an **array element**. You can distinguish each element from the others in an array with a subscript. A **subscript** (also called an **index**) is an integer contained within square brackets that indicates the position of one of an array's elements.

In some programming languages, such as C++ and Java, you also can declare an array variable by placing the square brackets after the array name, as in `double sales[];`. This format is illegal in C#.

You declare an array variable with a data type, a pair of square brackets, and an identifier. For example, to declare an array of `double` values to hold sales figures for salespeople, you write the following:

```
double[] sales;
```

After you create an array variable, you still need to create the actual array. Declaring an array and actually reserving memory space for it are two distinct processes. To reserve memory locations for 20 `sales` objects, you declare the array variable with the following two statements:

```
double[] sales;
sales = new double[20];
```

The keyword **`new`** is also known as the **`new` operator**; it is used to create objects. In this case, it creates 20 separate `sales`. You also can declare and create an array in one statement, such as the following:

```
double[] sales = new double[20];
```

The statement `double[] sales = new double[20];` reserves 20 memory locations. In C#, an array's elements are numbered beginning with 0, so if an array has 20 elements, you can use any subscript from 0 through 19. In other words, the first `sales` array element is `sales[0]` and the last `sales` element is `sales[19]`. Figure 6-1 shows how the array of 20 sales figures appears in computer memory. The figure assumes that the array begins at memory address 20000. Because a `double` takes eight bytes of storage, each element of the array is stored in succession at an address that is eight bytes higher than the previous one.

You can change the size of an array associated with an identifier, if necessary. For example, if you declare `int[] array;`, you can assign five elements later with `array = new int[5];`; later in the program, you might alter the array size to 100 with `array = new int[100];`. Still later, you could alter it again to be either larger or smaller. Most other programming languages do not provide this capability. If you resize an array in C#, the same identifier refers to a new array in memory and all the values are set to 0.

In C#, an array subscript must be an integer. For example, no array contains an element with a subscript of 1.5. A subscript can be an integer constant or variable or an expression that evaluates to an integer.

The first element in an array is sometimes called the "zeroth element."

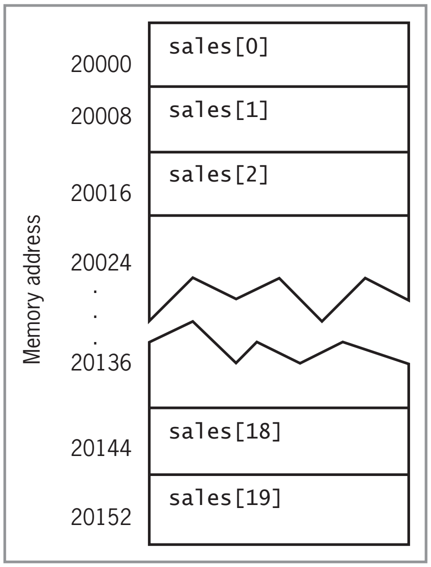


Figure 6-1 An array of 20 `sales` items in memory

When you instantiate an array, you cannot choose its location in memory any more than you can choose the location of any other variable. However, you do know that after the first array element, the subsequent elements will follow immediately.

Some other languages, such as COBOL, BASIC, and Visual Basic, use parentheses rather than square brackets to refer to individual array elements. By using brackets, the creators of C# made it easier for you to distinguish arrays from methods. Like C#, C++ and Java also use brackets surrounding array subscripts.

To remember that array elements begin with element 0, it might be helpful to think of the first array element as being "zero elements away from" the beginning of the array, the second element as being "one element away from" the beginning of the array, and so on.

A common mistake is to forget that the first element in an array is element 0, especially if you know another programming language in which the first array element is element 1. Making this mistake means you will be "off by one" in your use of any array. If you are "off by one" but still using a valid subscript when accessing an array element, your program will produce incorrect output. If you are "off by one" so that your subscript becomes larger than the highest value allowed, you will cause a program error.

When you work with any individual array element, you treat it no differently than you treat a single variable of the same type. For example, to assign a value to the first `sales` in an array, you use a simple assignment statement, such as the following:

```
sales[0] = 2100.00;
```

To output the value of the last `sales` in a 20-element array, you write:

```
Console.WriteLine(sales[19]);
```

An array subscript can be an expression, as long as the expression evaluates to an integer. For example, if `x` and `y` are integers and their sum is at least 0 but less than the size of an array named `array`, then it is legal to refer to `array[x + y]`.

Initializing an Array

In C#, arrays are objects. When you instantiate an array, you are creating a specific instance of a class that derives from, or builds upon, the built-in class named `System.Array`. When you declare arrays or any other objects, the following are default values:

- Numeric fields are set to 0.
- Character fields are set to '\u0000' or `null`.
- `bool` fields are set to `false`.

Watch the video *Declaring Arrays.*

For example, when you declare an array of five `int`s, each of the elements is initialized to 0.

You learned about the notation '\u0000' in Chapter 2.

You can assign nondefault values to array elements upon creation by declaring a comma-separated list of values enclosed within curly braces. For example, if you want to create an array named `myScores` and store five test scores within the array, you can use any of the following declarations:

```
int[] myScores = new int[5] {100, 76, 88, 100, 90};
int[] myScores = new int[] {100, 76, 88, 100, 90};
int[] myScores = {100, 76, 88, 100, 90};
```

In the chapter *Introduction to Inheritance* you will learn more about deriving classes.

You first learned the term *instance* to refer to an object in Chapter 1. You will understand instances more thoroughly after you complete the chapter *Using Classes and Objects*.

When you use curly braces at the end of a block of code, you do not follow the closing curly brace with a semicolon. However, when you use curly braces to enclose a list of array values, you must complete the statement with a semicolon.

The list of values provided for an array is an **initializer list**. When you initialize an array by providing a size and an initializer list, as in the first example, the stated size and number of list elements must match. However, when you initialize an array with values, you are not required to give the array a size, as shown in the second example; in that case, the size is assigned based on the number of values in the initializing list. The third example shows that when you initialize an array, you do not need to use the keyword `new` and repeat the type; instead, memory is assigned based on the stated array type and the length of the list of provided values. Of these three examples, the first is most explicit, but it requires two changes if the number of elements is altered. The third example requires the least typing, but might not clarify that a new object is being created. Use the form of array initialization that is clearest to you or that is conventional in your organization.

Programmers who have used other languages such as C++ and Java might expect that when an initialization list is shorter than the number of declared array elements, the "extra" elements will be set to default values. This is not the case in C#; if you declare a size, then you must list a value for each element.

An array of characters can be assigned to a string. For example, you can write the following:

```
char[] arrayOfLetters = {'h', 'e', 'l', 'l', 'o'};
string word = new string(arrayOfLetters);
```

TWO TRUTHS & A LIE

Declaring an Array and Assigning Values to Array Elements

1. To reserve memory locations for ten `testScore` objects, you can use the following statement:

   ```
   int[] testScore = new int[9];
   ```

2. To assign 60 to the last element in a 10-element array named `testScore`, you can use the following statement:

   ```
   testScore[9] = 60;
   ```

3. The following statement creates an array named `purchases` and stores four values within the array:

   ```
   double[] purchases = new double[] {23.55, 99.20, 4.67, 9.99};
   ```

The false statement is #1. To reserve memory locations for ten `testScore` objects, you must use ten within the second set of square braces. The ten elements will use the subscripts 0 through 9.

Accessing Array Elements

If you treat each array element as an individual entity, declaring an array does not offer much of an advantage over declaring individual variables. The power of arrays becomes apparent when you use subscripts that are variables rather than constant values.

For example, when you declare an array of five integers, such as the following, you often want to perform the same operation on each array element:

```
int[] myScores = {100, 76, 88, 100, 90};
```

To increase each array element by 3, for example, you can write the following five statements:

```
myScores[0] += 3;
myScores[1] += 3;
myScores[2] += 3;
myScores[3] += 3;
myScores[4] += 3;
```

You can shorten the task by using a variable as a subscript. Then you can use a loop to perform arithmetic on each element in the array. For example, you can use a `while` loop, as follows:

```
int sub = 0;
while(sub < 5)
{
   myScores[sub] += 3;
   ++sub;
}
```

You also can use a `for` loop, as follows:

```
for(int sub = 0; sub < 5; ++sub)
   myScores[sub] += 3;
```

In both examples, the variable `sub` is declared and initialized to 0, then compared to 5. Because it is less than 5, the loop executes and `myScores[0]` increases by 3. The variable `sub` is incremented and becomes 1, which is still less than 5, so when the loop executes again, `myScores[1]` increases by 3, and so on. If the array had 100 elements, individually increasing the array values by 3 would require 95 additional statements, but the only change required using either loop would be to change the limiting value for `sub` from 5 to 100.

New array users sometimes think there is a permanent connection between a variable used as a subscript and the array with which it is used, but that is not the case. For example, if you vary `sub` from 0 to 10 to fill an array, you do not need to use `sub` later when displaying the array elements—either the same variable or a different variable can be used as a subscript elsewhere in the program.

Using the `Length` Property

When you work with array elements, you must ensure that the subscript you use remains in the range of 0 through one less than the array's length. If you declare an array with five elements and use a subscript that is negative or more than 4, you will receive the error message "IndexOutOfRangeException" when you run the program. This message means the index, or subscript, does not hold a value that legally can access an array element. For example, if you declare an array of five integers, you can display them as follows:

```
int[] myScores = {100, 75, 88, 100, 90};
for(int sub = 0; sub < 5; ++sub)
   Console.WriteLine("{0} ", myScores[sub]);
```

You will learn about the `IndexOutOfRangeException` in the chapter *Exception Handling*.

An array's `Length` is a read-only property—you cannot assign it a new value. It is capitalized, like all property identifiers. You will create property identifiers for your own classes in the chapter *Using Classes and Objects*.

In C#, every string also has a built-in `Length` property. For example, if you declare `string firstName = "Chloe"`, then `firstName.Length` is 5.

If you modify your program to change the size of the array, you must remember to change the comparison in the `for` loop as well as every other reference to the array size within the program. Many text editors have a "find and replace" feature that lets you change (for example) all of the 5s in a file, either simultaneously or one by one. However, you must be careful not to change 5s that have nothing to do with the array; for example, do not change the 5 in the score 75 inadvertently—it is the second listed value in the `myScores` array and has nothing to do with the array size.

A better approach is to use a value that is automatically altered when you declare an array. Because every array automatically derives from the class **`System.Array`**, you can use the fields and methods that are part of the `System.Array` class with any array you create. The **`Length` property** is a member of the `System.Array` class that automatically holds an array's length and is always updated to reflect any changes you make to an array's size. The following segment of code displays "Array size is 5" and subsequently displays the array's contents:

```
int[] myScores = {100, 76, 88, 100, 90};
Console.WriteLine("Array size is {0}", myScores.Length);
for(int x = 0; x < myScores.Length; ++x)
   Console.WriteLine(myScores[x]);
```

Using `foreach`

You can easily navigate through arrays using a `for` or `while` loop that varies a subscript from 0 through `Array.Length - 1`. C# also supports a **`foreach` statement** that you can use to cycle through every array element without using a subscript. With the `foreach` statement, you provide a temporary **iteration variable** that automatically holds each array value in turn.

For example, the following code displays each element in the `payRate` array in sequence:

```
double[] payRate = {6.00, 7.35, 8.12, 12.45, 22.22};
foreach(double money in payRate)
   Console.WriteLine("{0}", money.ToString("C"));
```

The variable `money` is declared as a `double` within the `foreach` statement. During the execution of the loop, `money` holds each `payRate` value in turn—first, `payRate[0]`, then `payRate[1]`, and so on. As a simple variable, `money` does not require a subscript, making it easier to work with.

The `foreach` statement is used only under certain circumstances:

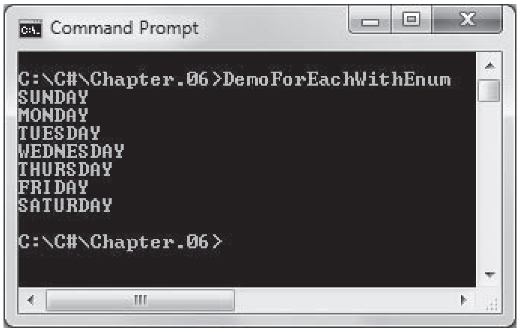
- You typically use `foreach` only when you want to access every array element; to access only selected array elements, you must manipulate subscripts using some other technique—for example, using a `for` loop or `while` loop.
- The `foreach` iteration variable is read-only—that is, you cannot assign a value to it. If you want to assign a value to array elements, you must use a different type of loop.

Using `foreach` with Enumerations

The `foreach` statement can provide a convenient way to display enumeration values. For example, Figure 6-2 shows a program that contains an enumeration for days of the week, and Figure 6-3 shows the output. The `foreach` block in the `Main()` method gets and displays each string in the enumeration using the `GetNames()` method that is built into the `Enum` class. You will learn more about built-in classes and methods in the coming chapters, but for now you can use the format shown in Figure 6-2 to conveniently work with enumeration values.

```
using System;
public class DemoForEachWithEnum
{
   enum Day
   {
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
         THURSDAY, FRIDAY, SATURDAY
   }
   public static void Main()
   {
      foreach(string day in Enum.GetNames(typeof(Day)))
         Console.WriteLine(day);
   }
}
```

Figure 6-2 `DemoForEachWithEnum` program



```
C:\C#\Chapter.06>DemoForEachWithEnum
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY

C:\C#\Chapter.06>
```

Figure 6-3 Output of `DemoForEachWithEnum` program

TWO TRUTHS & A LIE

Accessing Array Elements

1. Assume you have declared an array of six `doubles` named `balances`. The following statement displays all the elements:

   ```
   for(int index = 0; index < 6; ++index)
      Console.WriteLine(balances[index]);
   ```

2. Assume you have declared an array of eight `doubles` named `prices`. The following statement subtracts 2 from each element:

   ```
   for(double pr = 0; pr < 8; ++pr)
      prices[pr] -= 2;
   ```

3. The following code displays 3:

   ```
   int[] array = {1, 2, 3};
   Console.WriteLine(array.Length);
   ```

The false statement is #2. You can only use an `int` as the subscript to an array, and this example attempts to use a `double`.

Searching an Array Using a Loop

When you want to determine whether a variable holds one of many possible valid values, one option is to use `if` statements to compare the variable to valid values. For example, suppose that a company manufactures ten items. When a customer places an order for an item, you need to determine whether the item number is valid. If valid item numbers are sequential, say 101 through 110, then the following simple `if` statement that uses a logical AND operator can verify the order number and set a Boolean field to `true`:

```
if(itemOrdered >= 101 && itemOrdered <= 110)
   isValidItem = true;
```

If the valid item numbers are nonsequential, however—for example, 101, 108, 201, 213, 266, 304, and so on—you must code the following deeply nested `if` statement or a lengthy OR comparison to determine the validity of an item number:

```
if(itemOrdered == 101)
   isValidItem = true;
else if(itemOrdered == 108)
   isValidItem = true;
else if(itemOrdered == 201)
   isValidItem = true;
// and so on
```

Using a `for` Loop to Search an Array

Instead of creating a long series of `if` statements, a more elegant solution is to compare the `itemOrdered` variable to a list of values in an array. You can initialize the array with the valid values by using the following statement:

```
int[] validValues = {101, 108, 201, 213, 266, 304, 311,
   409, 411, 412};
```

Next, you can use a `for` statement to loop through the array and set a Boolean variable to `true` when a match is found:

```
for(int x = 0; x < validValues.Length; ++x)
   if(itemOrdered == validValues[x])
      isValidItem = true;
```

This simple `for` loop replaces the long series of `if` statements. What's more, if a company carries 1000 items instead of ten, then the list of valid items in the array must be altered, but the `for` statement does not change at all. As an added bonus, if you set up another array as a **parallel array** with the same number of elements and corresponding data, you can use the same subscript to access additional information. For example, if the ten items your company carries have ten different prices, then you can set up an array to hold those prices as follows:

```
double[] prices = {0.89, 1.23, 3.50, 0.69...}; // and so on
```

The prices must appear in the same order as their corresponding item numbers in the `validValues` array. Now the same `for` loop that finds the valid item number also finds the price, as shown in the program in Figure 6-4. In other words, if the item number is found in the second position in the `validValues` array, then you can find the correct price in the second position in the `prices` array. In the program in Figure 6-4, the variable used as a subscript, `x`, is set to 0 and

You might prefer to declare the `validValues` array as a constant because, presumably, the valid item numbers should not change during program execution. In C#, you must use the keywords `static` and `readonly` prior to the constant declaration. To keep these examples simple, all arrays in this chapter are declared as variable arrays.

In place of the `for` loop, you could use a `foreach` loop. You also could use a `while` loop, but `for` and `foreach` loops are used more commonly with arrays.

This type of search is called a **sequential search** because each array element is examined in sequence.

If you initialize parallel arrays, it is convenient to use spacing so that the corresponding values visually align on the screen or printed page.

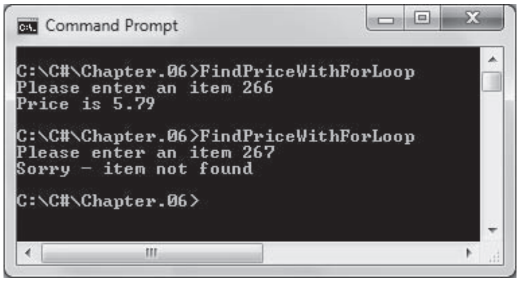
the Boolean variable `isValidItem` is `false`. In the shaded portion of the figure, while the subscript remains smaller than the length of the array of valid item numbers, the subscript is continuously increased so that subsequent array values can be tested. When a match is found between the user's item and an item in the array, `isValidItem` is set to `true` and the price of the item is stored in `itemPrice`. Figure 6-5 shows two typical program executions.

In the fourth statement of the `Main()` method in Figure 6-4, `itemPrice` is set to 0. Setting this variable is required because its value is later altered only if an item number match is found in the `validValues` array. When C# determines that a variable's value is only set depending on an `if` statement, C# will not allow you to display the variable, because the compiler assumes the variable might not have been set to a valid value.

```
using System;
public class FindPriceWithForLoop
{
   public static void Main()
   {
      int[] validValues = {101, 108, 201, 213, 266,
         304, 311, 409, 411, 412};
      double[] prices = {0.89, 1.23, 3.50, 0.69, 5.79,
         3.19, 0.99, 0.89, 1.26, 8.00};
      int itemOrdered;
      double itemPrice = 0;
      bool isValidItem = false;
      Console.Write("Please enter an item ");
      itemOrdered = Convert.ToInt32(Console.ReadLine());
      for(int x = 0; x < validValues.Length; ++x)
      {
         if(itemOrdered == validValues[x])
         {
           isValidItem = true;
           itemPrice = prices[x];
         }
      }
      if(isValidItem)
         Console.WriteLine("Price is {0}", itemPrice);
      else
         Console.WriteLine("Sorry - item not found");
   }
}
```

Figure 6-4 The `FindPriceWithForLoop` program

In an array with many possible matches to a search value, the most efficient strategy is to place the most common items first so they are matched right away. For example, if item 311 is ordered most often, place 311 first in the `validValues` array and its price ($0.99) first in the `prices` array.



```
C:\C#\Chapter.06>FindPriceWithForLoop
Please enter an item 266
Price is 5.79

C:\C#\Chapter.06>FindPriceWithForLoop
Please enter an item 267
Sorry - item not found

C:\C#\Chapter.06>
```

Figure 6-5 Two typical executions of the `FindPriceWithForLoop` program

Within the code shown in Figure 6-4, you compare every `itemOrdered` with each of the ten `validValues`. Even when an `itemOrdered` is equivalent to the first value in the `validValues` array (101), you always make nine additional cycles through the array. On each of these nine additional iterations, the comparison between `itemOrdered` and `validValues[x]` is always `false`. As soon as a match for an `itemOrdered` is found, the most efficient action is to break out of the `for` loop early. An easy way to accomplish this task is to set `x` to a high value within the block of statements executed when a match is found. Then, after a match, the `for` loop will not execute again because the limiting comparison (`x < validValues.Length`) will have been surpassed. The following code shows this approach.

```
for(int x = 0; x < validValues.Length; ++x)
{
   if(itemOrdered == validValues[x])
   {
      isValidItem = true;
      itemPrice = prices[x];
      x = validValues.Length;
        // break out of loop when you find a match
   }
}
```

Although parallel arrays can be very useful, they also can increase the likelihood of mistakes. Any time you make a change to one array, you must remember to make the corresponding change in its parallel array. As you continue to study C#, you will learn superior ways to correlate data items. For example, in the chapter *Using Classes and Objects* you will learn that you can encapsulate corresponding data items in objects and create arrays of objects.

Instead of the statement that sets `x` to `validValues.Length` when a match is found, you could remove that statement and change the comparison in the middle section of the `for` statement to a compound statement, as follows:

```
for(int x = 0; x < validValues.Length && !isValidItem; ++x)...
```

As another alternative, you could remove the statement that sets `x` to `validValues.Length` and place a `break` statement within the loop in its place. Some programmers disapprove of exiting a `for` loop early, whether by setting a variable's value or by using a `break` statement. They argue that programs are easier to debug and maintain if each program segment has only one entry and one exit point. If you (or your instructor) agree with this philosophy, then you can select an approach that uses a `while` statement, as described next.

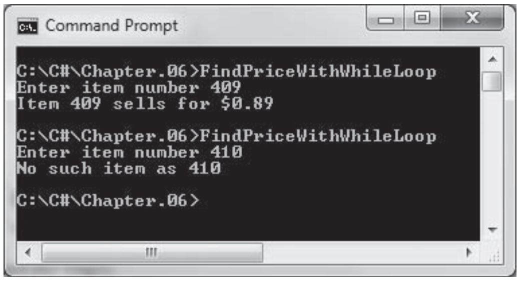
Using a `while` Loop to Search an Array

As an alternative to using a `for` or `foreach` loop to search an array, you can use a `while` loop to search for a match. Using this approach, you set a subscript to 0 and, while the `itemOrdered` is not equal to a value in the array, increase the subscript and keep looking. You search only while the subscript remains lower than the number of elements in the array. If the subscript increases to match `validValues.Length`, then you never found a match in the ten-element array. If the loop ends before the subscript reaches `validValues.Length`, then you found a match and the correct price can be assigned to the `itemPrice` variable. Figure 6-6 shows a program that uses this approach.

```
using System;
public class FindPriceWithWhileLoop
{
   public static void Main()
   {
      int x;
      string inputString;
      int itemOrdered;
      double itemPrice = 0;
      bool isValidItem = false;
      int[] validValues = {101, 108, 201, 213, 266,
        304, 311, 409, 411, 412};
      double[] prices = {0.89, 1.23, 3.50, 0.69, 5.79,
        3.19, 0.99, 0.89, 1.26, 8.00};
      Console.Write("Enter item number ");
      inputString = Console.ReadLine();
      itemOrdered = Convert.ToInt32(inputString);
      x = 0;
      while(x < validValues.Length &&
        itemOrdered != validValues[x])
          ++x;
      if(x != validValues.Length)
      {
        isValidItem = true;
        itemPrice = prices[x];
      }
      if(isValidItem)
        Console.WriteLine("Item {0} sells for {1}",
          itemOrdered, itemPrice.ToString("C"));
      else
        Console.WriteLine("No such item as {0}",
          itemOrdered);
   }
}
```

Figure 6-6 The `FindPriceWithWhileLoop` program that searches with a `while` loop

In the application in Figure 6-6, the variable used as a subscript, `x`, is set to 0 and the Boolean variable `isValidItem` is `false`. In the shaded portion of the figure, while the subscript remains smaller than the length of the array of valid item numbers, and while the user's requested item does not match a valid item, the subscript is increased so that subsequent array values can be tested. The `while` loop ends when a match is found or the array tests have been exhausted, whichever comes first. When the loop ends, if `x` is not equal to the size of the array, then a valid item has been found and its price can be retrieved from the `prices` array. Figure 6-7 shows two executions of the program. In the first execution, a match is found; in the second, an invalid item number is entered, so no match is found.



```
C:\C#\Chapter.06>FindPriceWithWhileLoop
Enter item number 409
Item 409 sells for $0.89

C:\C#\Chapter.06>FindPriceWithWhileLoop
Enter item number 410
No such item as 410

C:\C#\Chapter.06>
```

Figure 6-7 Two executions of the `FindPriceWithWhileLoop` application

Watch the video *Searching an Array.*

Searching an Array for a Range Match

Searching an array for an exact match is not always practical. For example, suppose your mail-order company gives customer discounts based on the quantity of items ordered. Perhaps no discount is given for any order of up to a dozen items, but increasing discounts are available for orders of increasing quantities, as shown in Figure 6-8.

Total Quantity Ordered	Discount (%)
1 to 12	None
13 to 49	10
50 to 99	14
100 to 199	18
200 or more	20

Figure 6-8 Discount table for a mail-order company

One awkward, impractical option is to create a single array to store the discount rates. You could use a variable named `numOfItems` as a subscript to the array, but the array would need hundreds of entries, such as the following:

```
double[] discount = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0.10, 0.10, 0.10 ...}; // and so on
```

When `numOfItems` is 3, for example, then `discount[numOfItems]` or `discount[3]` is 0. When `numOfItems` is 14, then `discount[numOfItems]` or `discount[14]` is 0.10. Because a customer might order thousands of items, the array would need to be ridiculously large.

Notice that 13 zeroes are listed in the discount array in this example. The first array element has a 0 subscript (and a 0 discount for 0 items). The next 12 discounts (1 through 12 items) also have 0 discounts.

A better option is to create parallel arrays. One array will hold the five discount rates, and the other array will hold five discount range limits. Then you can perform a **range match** by determining the pair of limiting values between which a customer's order falls. The Total Quantity Ordered column in Figure 6-8 shows five ranges. If you use only the first figure in each range, then you can create an array that holds five low limits:

```
int[] discountRangeLowLimit = {1, 13, 50, 100, 200};
```

A parallel array will hold the five discount rates:

```
double[] discount = {0, 0.10, 0.14, 0.18, 0.20};
```

Then, starting at the last `discountRangeLowLimit` array element, for any `numOfItems` greater than or equal to `discountRangeLowLimit[4]`, the appropriate discount is `discount[4]`. In other words, for any `numOfItems` less than `discountRangeLowLimit[4]`, you should decrement the subscript and look in a lower range. Figure 6-9 shows the code.

```
// assume numOfItems is a declared integer for which a user
// has input a value
int[] discountRangeLowLimit = {1, 13, 50, 100, 200};
double[] discount =          {0, 0.10, 0.14, 0.18, 0.20};
double customerDiscount;
int sub = discountRangeLowLimit.Length - 1;
while(sub >= 0 && numOfItems < discountRangeLowLimit[sub])
   --sub;
customerDiscount = discount[sub];
```

Figure 6-9 Searching an array of range limits

In the search in Figure 6-9, one less than the `Length` property of either array could have been used to initialize `sub`.

As an alternate approach to the range-checking logic in Figure 6-9, you can choose to create an array that contains the upper limit of each range, such as the following:

```
int[] discountRangeUpperLimit = {12, 49, 99, 199, 9999999};
```

Then the logic can be written to compare `numOfItems` to each range limit until the correct range is located, as follows:

```
int sub = 0;
while(sub < discountRangeUpperLimit.Length && numOfItems >
   discountRangeUpperLimit[sub])
   ++sub;
customerDiscount = discount[sub];
```

In this example, `sub` is initialized to 0. While it remains within array bounds, and while `numOfItems` is more than each upper-range limit, `sub` is increased. In other words, if `numOfItems` is 3, the `while` expression is false on the first loop iteration, the loop ends, `sub` remains 0, and the customer discount is the first discount. However, if `numOfItems` is 30, then the `while` expression is true on the first loop iteration, `sub` becomes 1, the `while` expression is false on the second iteration, and the second discount is used. In this example, the last `discountRangeUpperLimit` array value is 9999999. This very high value was used with the assumption that no `numOfItems` would ever exceed it. As with many issues in programming, multiple correct approaches frequently exist for the same problem.

TWO TRUTHS & A LIE

Searching an Array Using a Loop

1. A parallel array has the same number of elements as another array, and corresponding data.
2. When you search an array for an exact match in a parallel array, you must perform a loop as many times as there are elements in the arrays.
3. One practical solution to creating an array with which to perform a range check is to design the array to hold the lowest value in each range.

The false statement is #2. When you search an array for an exact match in a parallel array, you can perform a loop as many times as there are elements in the arrays, but once a match is found, the additional loop iterations are unnecessary. It is most efficient to terminate the loop cycles as soon as a match is found.

Using the `BinarySearch()`, `Sort()`, and `Reverse()` Methods

You have already learned that because every array in C# is derived from the `System.Array` class, you can use the `Length` property. Additionally, the `System.Array` class contains a variety of useful, built-in methods that can search, sort, and manipulate array elements.

You already have used many built-in C# methods such as `WriteLine()` and `ReadLine()`. You will learn to write your own methods in the next chapter.

Using the `BinarySearch()` Method

The **`BinarySearch()` method** finds a requested value in a sorted array. Instead of employing the logic you used to find a match in the last section, you can take advantage of this built-in method to locate a value within an array, as long as the array items are organized in ascending order.

A binary search is one in which a sorted list of objects is split in half repeatedly as the search gets closer and closer to a match. Perhaps you have played a guessing game, trying to guess a number from 1 to 100. If you asked, "Is it less than 50?," then continued to narrow your guesses upon hearing each subsequent answer, you have performed a binary search.

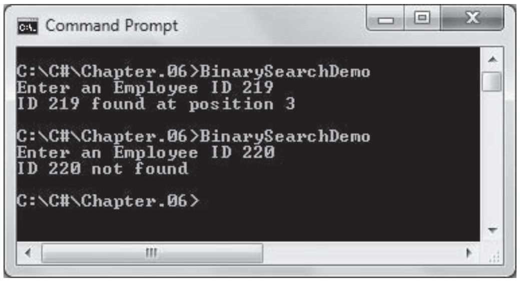
Figure 6-10 shows a program that declares an array of integer `idNumbers` arranged in ascending order. The program prompts a user for a value, converts it to an integer, and, rather than using a loop to examine each array element and compare it to the entered value, simply passes the array and the entered value to the `BinarySearch()` method in the shaded statement. The method returns –1 if the value is not found in the array; otherwise, it returns the array position of the sought value. Figure 6-11 shows two executions of this program.

The `BinarySearch()` method takes two arguments—the array name and the value for which to search. In Chapter 1, you learned that arguments represent information that a method needs to perform its task. When methods require multiple arguments, they are separated by commas. For example, when you have used the `Console.WriteLine()` method, you have passed a format string and values to be displayed, all separated by commas.

```
using System;
public class BinarySearchDemo
{
   public static void Main()
   {
      int[] idNumbers = {122, 167, 204, 219, 345};
      int x;
      string entryString;
      int entryId;
      Console.Write("Enter an Employee ID ");
      entryString = Console.ReadLine();
      entryId = Convert.ToInt32(entryString);
      x = Array.BinarySearch(idNumbers, entryId);
      if(x < 0)
         Console.WriteLine("ID {0} not found", entryId);
      else
         Console.WriteLine("ID {0} found at position {1} ",
            entryId, x);
   }
}
```

These values must be sorted in ascending order for the `BinarySearch()` method to work correctly.

Figure 6-10 `BinarySearchDemo` program



```
C:\C#\Chapter.06>BinarySearchDemo
Enter an Employee ID 219
ID 219 found at position 3

C:\C#\Chapter.06>BinarySearchDemo
Enter an Employee ID 220
ID 220 not found

C:\C#\Chapter.06>
```

Figure 6-11 Two executions of the `BinarySearchDemo` program

When you use the following statement, you send a string to the `Write()` method:

```
Console.Write("Enter an Employee ID ");
```

When you use the following statement, you get a value back from the `ReadLine()` method:

```
entryString = Console.ReadLine();
```

In Figure 6-10, the following single statement both sends a value to a method and gets a value back:

```
x = Array.BinarySearch(idNumbers, entryId);
```

The statement calls the method that performs the search, returning a –1 or the position where `entryId` was found; that value is then stored in `x`. This single line of code is easier to write, less prone to error, and easier to understand than writing a loop to cycle through the `idNumbers` array looking for a match. Still, it is worthwhile to understand how to perform the search without the `BinarySearch()` method, as you learned while studying parallel arrays. You will need to use that technique under the following conditions, when the `BinarySearch()` method proves inadequate:

- If your array items are not arranged in ascending order, the `BinarySearch()` method does not work correctly.
- If your array holds duplicate values and you want to find all of them, the `BinarySearch()` method does not work—it can return only one value, so it returns the position of the first matching value it finds (which is not necessarily the first instance of the value in the array).
- If you want to find a range match rather than an exact match, the `BinarySearch()` method does not work.

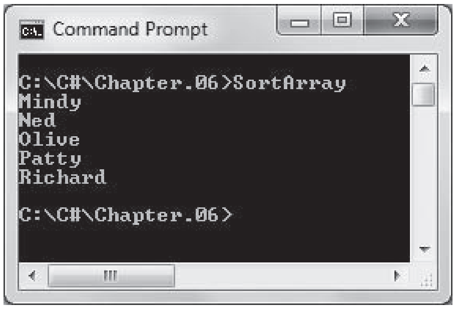
Using the Sort() Method

The **`Sort()` method** arranges array items in ascending order. Ascending order is lowest to highest; it works numerically for number types and alphabetically for characters and strings. To use the method, you pass the array name to `Array.Sort()`, and the element positions within the array are rearranged appropriately. Figure 6-12 shows a program that sorts an array of strings; Figure 6-13 shows its execution.

```
using System;
public class SortArray
{
   public static void Main()
   {
      string[] names = {"Olive", "Patty",
         "Richard", "Ned", "Mindy"};
      int x;
      Array.Sort(names);
      for(x = 0; x < names.Length; ++x)
         Console.WriteLine(names[x]);
   }
}
```

Figure 6-12 `SortArray` program

The `Array.Sort()` method provides a good example of encapsulation—you can use the method without understanding how it works internally. The method actually uses an algorithm named Quicksort. You will learn how to implement this algorithm yourself as you continue to study programming.



```
C:\C#\Chapter.06>SortArray
Mindy
Ned
Olive
Patty
Richard

C:\C#\Chapter.06>
```

Figure 6-13 Execution of `SortArray` program

Because the `BinarySearch()` method requires that array elements be sorted in order, the `Sort()` method is often used in conjunction with it.

Using the `Reverse()` Method

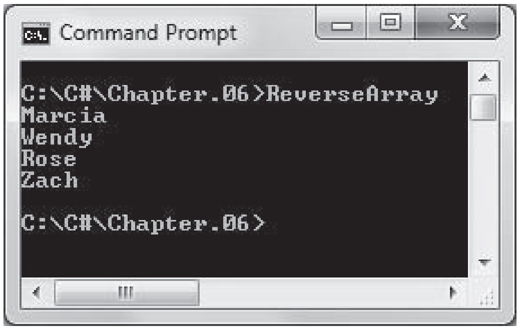
The **`Reverse()` method** reverses the order of items in an array. In other words, for any array, the element that starts in position 0 is relocated to position `Length - 1`, the element that starts in position 1 is relocated to position `Length - 2`, and so on until the element that starts in position `Length - 1` is relocated to position 0. You call the `Reverse()` method the same way you call the `Sort()` method—you simply pass the array name to the method. Figure 6-14 shows a program that uses `Reverse()` with an array of strings, and Figure 6-15 shows its execution.

When you `Reverse()` an array that contains an odd number of elements, the middle element will remain in its original location.

```
using System;
public class ReverseArray
{
   public static void Main()
   {
      string[] names = {"Zach", "Rose", "Wendy", "Marcia"};
      int x;
      Array.Reverse(names);
      for(x = 0; x < names.Length; ++x)
         Console.WriteLine(names[x]);
   }
}
```

Figure 6-14 ReverseArray program

The Reverse() method does not sort array elements; it only rearranges their positions to the opposite order.



```
C:\C#\Chapter.06>ReverseArray
Marcia
Wendy
Rose
Zach

C:\C#\Chapter.06>
```

Figure 6-15 Execution of ReverseArray program

TWO TRUTHS & A LIE

Using the BinarySearch(), Sort(), and Reverse() Methods

1. When you use the BinarySearch() method with an array, the array items must first be organized in ascending order.
2. The Array.Sort() and Array.Reverse() methods are similar in that both require a single argument.
3. The Array.Sort() and Array.Reverse() methods are different in that one places items in ascending order and the other places them in descending order.

The false statement is #3. The Array.Sort() method places items in ascending order, but the Array.Reverse() method simply reverses the existing order of any array whether it was presorted or not.

Using Multidimensional Arrays

You can think of the single dimension of a single-dimensional array as the height of the array.

When you declare an array such as `double[] sales = new double[20];`, you can envision the declared integers as a column of numbers in memory, as shown at the beginning of this chapter in Figure 6-1. In other words, you can picture the 20 declared numbers stacked one on top of the next. An array that you can picture as a column of values, and whose elements you can access using a single subscript, is a **one-dimensional** or **single-dimensional array**.

C# also supports **multidimensional arrays**—those that require multiple subscripts to access the array elements. The most commonly used multidimensional arrays are two-dimensional arrays that are rectangular. **Two-dimensional arrays** have two or more columns of values for each row, as shown in Figure 6-16. In a **rectangular array**, each row has the same number of columns. You must use two subscripts when you access an element in a two-dimensional array. When mathematicians use a two-dimensional array, they often call it a **matrix** or a **table**; you might have used a two-dimensional array called a spreadsheet.

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

Figure 6-16 View of a rectangular, two-dimensional array in memory

You can think of the two dimensions of a two-dimensional array as height and width.

A `sales` array with two dimensions, as shown in Figure 6-16, could have several uses. For example, each row could represent a category of items sold and each column could represent a salesperson who sold them.

When you declare a one-dimensional array, you type a single, empty set of square brackets after the array type, and you use a single subscript in a set of square brackets when reserving memory. To declare a two-dimensional array, you type a comma in the square brackets after the array type, and you use two subscripts, separated by a comma in brackets, when reserving memory. For example, the array in Figure 6-16 can be declared as the following, creating an array named `sales` that holds three rows and four columns:

When you declare a two-dimensional array, spaces surrounding the comma within the square brackets are optional.

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

Just as with a one-dimensional array, if you do not provide values for the elements in a two-dimensional numerical array, the values are set to the default value for the data type (zero for numeric data). You can assign other values to the array elements later. For example, the following statement assigns the value 14.00 to the element of the `sales` array that is in the first column of the first row:

```
sales[0, 0] = 14.00;
```

Alternatively, you can initialize a two-dimensional array with values when it is created. For example, the following code assigns values to `sales` when it is created:

```
double[ , ] sales = {{14.00, 15.00, 16.00, 17.00},
                     {21.99, 34.55, 67.88, 31.99},
                     {12.03, 55.55, 32.89, 1.17}};
```

The `sales` array contains three rows and four columns. You contain the entire set of values within a pair of curly braces. The first row of the array holds the four `doubles` 14.00, 15.00, 16.00, and 17.00. Notice that these four values are placed within their own inner set of curly braces to indicate that they constitute one row, or the first row, which is row 0. Similarly, the next four values make up the second row (row 1), which you reference with the subscript 1. The value of `sales[0, 0]` is 14.00. The value of `sales[0, 1]` is 15.00. The value of `sales[2, 3]` is 1.17. The first value within the brackets following the array name always refers to the row; the second value, after the comma, refers to the column.

You do not need to place each row of values that initializes a two-dimensional array on its own line. However, doing so makes the positions of values easier to understand.

As an example of how useful two-dimensional arrays can be, assume you own an apartment building with four floors—a basement, which you refer to as floor zero, and three other floors numbered one, two, and three. In addition, each of the floors has studio (with no bedroom), one-, and two-bedroom apartments. The monthly rent for each type of apartment is different, and the rent is higher for apartments with more bedrooms. Figure 6-17 shows the rental amounts.

Floor	Zero Bedrooms	One Bedroom	Two Bedrooms
0	400	450	510
1	500	560	630
2	625	676	740
3	1000	1250	1600

Figure 6-17 Rents charged (in dollars)

To determine a tenant's rent, you need to know two pieces of information: the floor on which the tenant rents an apartment and the number of bedrooms in the apartment. Within a C# program, you can declare an array of rents using the following code:

```
int[ , ] rents = { {400, 450, 510},
                   {500, 560, 630},
                   {625, 676, 740},
                   {1000, 1250, 1600} };
```

Assume you declare two integers to hold the floor number and bedroom count, as in the following statement:

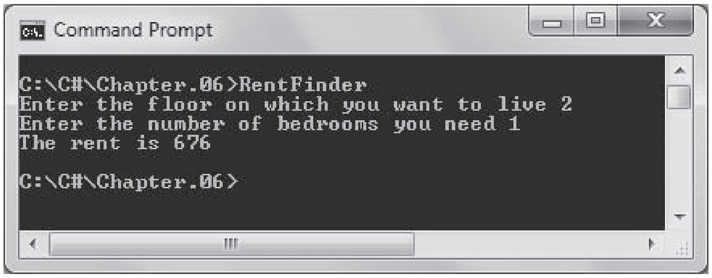
```
int floor, bedrooms;
```

Then any tenant's rent can be referred to as `rents[floor, bedrooms]`.

Figure 6-18 shows a complete program that uses a rectangular, two-dimensional array to hold rent values. Figure 6-19 shows a typical execution.

```
using System;
public class RentFinder
{
   public static void Main()
   {
      int[ , ] rents = { {400, 450, 510},
                         {500, 560, 630},
                         {625, 676, 740},
                         {1000, 1250, 1600} };
      int floor;
      int bedrooms;
      string inputString;
      Console.Write("Enter the floor on which you want to live ");
      inputString = Console.ReadLine();
      floor = Convert.ToInt32(inputString);
      Console.Write("Enter the number of bedrooms you need ");
      inputString = Console.ReadLine();
      bedrooms = Convert.ToInt32(inputString);
      Console.WriteLine("The rent is {0}",
         rents[floor, bedrooms]);
   }
}
```

Figure 6-18 The `RentFinder` program



```
C:\C#\Chapter.06>RentFinder
Enter the floor on which you want to live 2
Enter the number of bedrooms you need 1
The rent is 676

C:\C#\Chapter.06>
```

Watch the video *Using a Two-Dimensional Array.*

Figure 6-19 Typical execution of the `RentFinder` program

C# supports arrays with more than two dimensions. For example, if you own a multistory apartment building with different numbers of bedrooms available in apartments on each floor, you can use a two-dimensional array to store the rental fees. If you own several apartment buildings, you might want to employ a third dimension to store the building number. Suppose you want to store rents for four buildings that have three floors each and that each hold two types of apartments. Figure 6-20 shows how you might define such an array.

```
int[ , , ] rents = { { {400, 500}, {450, 550}, {500, 550}},
                     { {510, 610}, {710, 810}, {910, 1010}},
                     { {525, 625}, {725, 825}, {925, 1025}},
                     { {850, 950}, {1050, 1150}, {1250, 1350}}};
```

Figure 6-20 A three-dimensional array definition

The empty brackets that follow the data type contain two commas, showing that the array supports three dimensions. A set of curly braces surrounds all the data. Four inner sets of braces surround the data for each floor. In this example, each row of values represents a building (0 through 3). Then 12 sets of innermost brackets surround the values for each floor—first a zero-bedroom apartment and then a one-bedroom apartment.

Using the three-dimensional array in Figure 6-20, an expression such as `rents[building, floor, bedrooms]` refers to a specific rent figure for a building whose number is stored in the `building` variable and whose floor and bedroom numbers are stored in the `floor` and `bedrooms` variables. Specifically, `rents[3, 1, 0]` refers to a studio (zero-bedroom) apartment on the first floor of building 3 ($1050 in Figure 6-20). When you are programming in C#, you can use four, five, or more dimensions in an array. As long as you can keep track of the order of the variables needed as subscripts, and as long as you do not exhaust your computer's memory, C# lets you create arrays of any size.

C# also supports jagged arrays. A **jagged array** is a one-dimensional array in which each element is another array. The major difference between jagged and rectangular arrays is that in jagged arrays, each row can be a different length.

For example, consider an application in which you want to store train ticket prices for each stop along five different routes. Suppose some of the routes have as many as ten stops and others have as few as two. Each of the five routes could be represented by a row in a multidimensional array. Then you would have two logical choices for the columns:

- You could create a rectangular, two-dimensional array, allowing ten columns for each row. In some of the rows, as many as eight of the columns would be empty, because some routes have only two stops.
- You could create a jagged two-dimensional array, allowing a different number of columns for each row. Figure 6-21 shows how you could implement this option.

```
double[][] tickets = {
   new double[] {5.50, 6.75, 7.95, 9.00, 12.00,
      13.00, 14.50, 17.00, 19.00, 20.25},
   new double[] {5.00, 6.00},
   new double[] {7.50, 9.00, 9.95, 12.00, 13.00, 14.00},
   new double[] {3.50, 6.45, 9.95, 10.00, 12.75},
   new double[] {15.00, 16.00} };
```

Figure 6-21 A jagged, two-dimensional array

Two square brackets are used following the data type of the array in Figure 6-21. This notation declares a jagged array, which is composed of five separate one-dimensional arrays. Within the jagged array, each row needs its own `new` operator and data type. To refer to a jagged array element, you use two sets of brackets after the array name—for example, `tickets[route][stop].` In Figure 6-21, the value of `tickets[0][0]` is 5.50, the value of `tickets[0][1]` is 6.75, and the value of `tickets[0][2]` is 7.95. The value of `tickets[1][0]` is 5.00, and the value of `tickets[1][1]` is 6.00. Referring to `tickets[1][2]` is invalid because there is no column 2 in the second row (that is, there are only two stops, not three, on the second train route).

TWO TRUTHS & A LIE

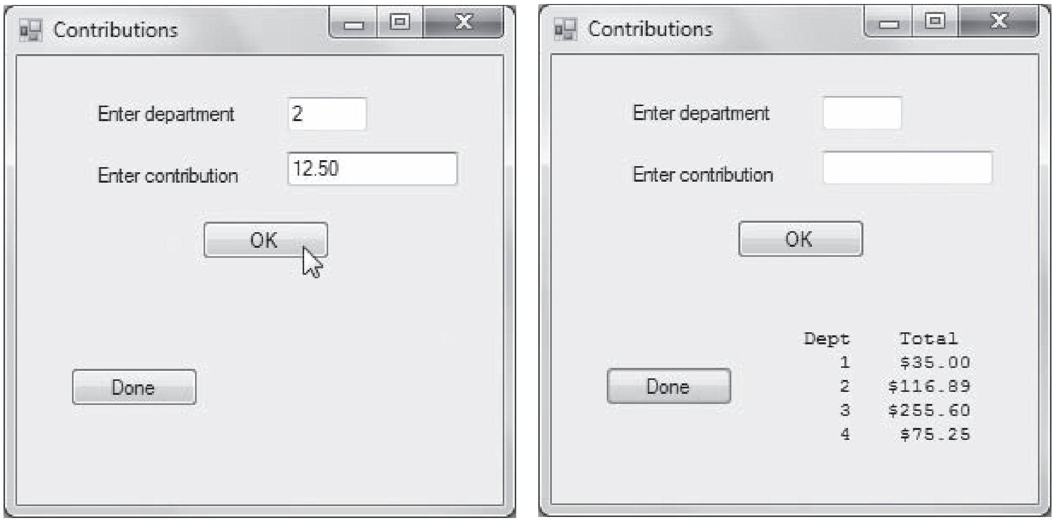
Using Multidimensional Arrays

1. A rectangular array has the same number of columns as rows.
2. The following array contains two rows and three columns:
   ```
   int[ , ] departments = {{12, 54, 16},
                           {22, 44, 47}};
   ```
3. A jagged array is a one-dimensional array in which each element is another array.

The false statement is #1. In a rectangular array, each row has the same number of columns, but the numbers of rows and columns are not required to be the same.

Array Issues In GUI Programs

The major unusual consideration when using an array in a GUI program is that if the array values change based on user input, the array must be stored outside any method that reacts to the user's event. For example, consider an application that accumulates contribution totals for a fund-raising drive competition between four departments in a company. The left side of Figure 6-22 shows a `Form` into which a user types a department number and a contribution amount. The user clicks OK, then enters the next contribution amount. When the user clicks the Done button, a summary of contributions appears, as in the right half of Figure 6-22.



Enter department: 2

Enter contribution: 12.50

```
Dept    Total
   1   $35.00
   2  $116.89
   3  $255.60
   4   $75.25
```

Figure 6-22 The `Form` for the `CountContributions` program as the user enters values and after the user clicks Done

Figure 6-23 shows the code needed to implement the application. An array named `total` is declared outside of any methods. The `okButton_Click()` method accepts a department number and contribution amount from the user. It then adds the contribution into the array element that corresponds to the department and clears the text boxes so they are empty prior to the next entry. The `total` array must be declared outside of the `okButton_Click()` method; if it was inside the method, it would be redeclared and all its elements would be reset to 0 with each button click. The `doneButton_Click()` method displays the array's contents.

```
double[] total = { 0, 0, 0, 0 };
private void okButton_Click(object sender, EventArgs e)
{
   int dept;
   double contribution;
   dept = Convert.ToInt32(deptTextbox.Text);
   contribution = Convert.ToDouble(contributionTextbox.Text);
   --dept;
   total[dept] += contribution;
   deptTextbox.Text = "";
   contributionTextbox.Text = "";
}

private void doneButton_Click(object sender, EventArgs e)
{
   outputLabel.Text = "Dept    Total";
   for (int x = 0; x < total.Length; ++x)
      outputLabel.Text +=
         String.Format("\n {0}{1, 10}", x + 1, total[x].ToString("C"));
}
```

Figure 6-23 Code that declares array and two methods needed for the `CountContributions` application

TWO TRUTHS & A LIE

Array Issues in GUI Programs

1. A GUI program can contain declarations for any number of arrays of any data type.
2. If a method reacts to a user-initiated event, it cannot contain an array declaration.
3. If a method reacts to a user-initiated event and the method contains an array declaration, the array will be redeclared with each event occurrence.

The false statement is #2. If a method reacts to a user-initiated event, it can contain an array. The only problem is that the array will be redeclared with each new event, so it cannot store data that must persist over a number of events.

You Do It

Creating and Using an Array

In the next steps, you will create a small array to see how they are used. The array will hold salaries for four categories of employees.

To create a program that uses an array:

1. Open a new text file in your text editor.
2. Begin the class that will demonstrate array use by typing the following:

```
using System;
public class ArrayDemo1
{
   public static void Main()
   {
```

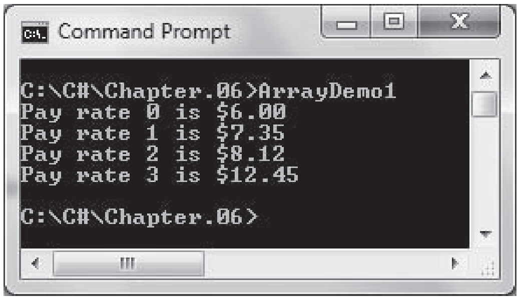
3. Declare and create an array that holds four `double` values by typing:

```
double[] payRate = {6.00, 7.35, 8.12, 12.45};
```

4. To confirm that the four values have been assigned, display them using the following code:

```
for(int x = 0; x < payRate.Length; ++x)
   Console.WriteLine("Pay rate {0} is {1}",
      x, payRate[x].ToString("C"));
```

5. Add the two closing curly brackets that end the `Main()` method and the `ArrayDemo1` class.
6. Save the program as **ArrayDemo1.cs**.
7. Compile and run the program. The program's output appears in Figure 6-24.



```
C:\C#\Chapter.06>ArrayDemo1
Pay rate 0 is $6.00
Pay rate 1 is $7.35
Pay rate 2 is $8.12
Pay rate 3 is $12.45

C:\C#\Chapter.06>
```

Figure 6-24 Output of `ArrayDemo1` program

Using the `Sort()` and `Reverse()` Methods

In the next steps, you will create an array of integers and use the `Sort()` and `Reverse()` methods to manipulate it.

To use the `Sort()` and `Reverse()` methods:

1. Open a new file in your text editor.
2. Type the beginning of a class named `ArrayDemo2` that includes an array of eight integer test scores, an integer you will use as a subscript, and a string that will hold user-entered data.

```
using System;
public class ArrayDemo2
{
   public static void Main()
   {
      int[] scores = new int[8];
      int x;
      string inputString;
```

3. Add a loop that prompts the user, accepts a test score, converts the score to an integer, and stores it as the appropriate element of the `scores` array.

```
for(x = 0; x < scores.Length; ++x)
{
   Console.Write("Enter your score on test {0} ", x + 1);
   inputString = Console.ReadLine();
   scores[x] = Convert.ToInt32(inputString);
}
```

The program displays `x + 1` with each `score[x]` because, although array elements are numbered starting with 0, people usually count items starting with 1.

4. Add a statement that creates a dashed line to visually separate the input from the output. Display "Scores in original order:", then use a loop to display each score in a field that is six characters wide.

```
Console.WriteLine("\n--------------------------------");
Console.WriteLine("Scores in original order:");
for(x = 0; x < scores.Length; ++x)
   Console.Write("{0, 6}", scores[x]);
```

You learned to set display field sizes when you learned about format strings in Chapter 2.

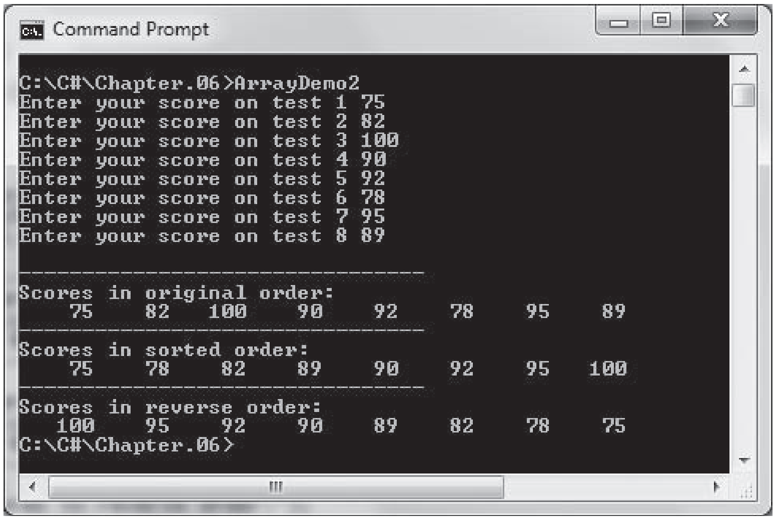
5. Add another dashed line for visual separation, then pass the `scores` array to the `Array.Sort()` method. Display "Scores in sorted order:", then use a loop to display each of the newly sorted scores.

```
Console.WriteLine("\n--------------------------------");
Array.Sort(scores);
Console.WriteLine("Scores in sorted order:");
for(x = 0; x < scores.Length; ++x)
   Console.Write("{0, 6}", scores[x]);
```

6. Add one more dashed line, reverse the array elements by passing `scores` to the `Array.Reverse()` method, display "Scores in reverse order:", and show the rearranged scores.

```
Console.WriteLine("\n-------------------------------");
Array.Reverse(scores);
Console.WriteLine("Scores in reverse order:");
for(x = 0; x < scores.Length; ++x)
   Console.Write("{0, 6}", scores[x]);
```

7. Add two closing curly braces—one for the `Main()` method and one for the class. Save the file as **ArrayDemo2.cs**. Compile and execute the program. Figure 6-25 shows a typical execution of the program. The user-entered scores are not in order, but after the call to the `Sort()` method, they appear in ascending order. After the call to the `Reverse()` method, they appear in descending order.



```
C:\C#\Chapter.06>ArrayDemo2
Enter your score on test 1 75
Enter your score on test 2 82
Enter your score on test 3 100
Enter your score on test 4 90
Enter your score on test 5 92
Enter your score on test 6 78
Enter your score on test 7 95
Enter your score on test 8 89

-------------------------------
Scores in original order:
    75    82   100    90    92    78    95    89
-------------------------------
Scores in sorted order:
    75    78    82    89    90    92    95   100
-------------------------------
Scores in reverse order:
   100    95    92    90    89    82    78    75
C:\C#\Chapter.06>
```

Figure 6-25 Typical execution of `ArrayDemo2` program

Chapter Summary

- An array is a list of data items, all of which have the same type and the same name, but are distinguished from each other using a subscript or index. You declare an array variable by inserting a pair of square brackets after the type, and reserve memory for an array by using the keyword `new`. Any array's elements are numbered 0 through one less than the array's length. In C#, arrays are objects that derive from a class named `System.Array`. An array's fields are initialized to default values. To initialize an array to nondefault values, you use a list of values that are separated by commas and enclosed within curly braces.

- The power of arrays becomes apparent when you begin to use subscripts that are variables rather than constant values, and when you use loops to process array elements. When you work with array elements, you must ensure that the subscript you use remains in the range of 0 through `length - 1`. You can use the `Length` property, which is a member of the `System.Array` class, to automatically hold an array's length. You can use the `foreach` statement to cycle through every array element without using subscripts. With the `foreach` statement, you provide a temporary variable that automatically holds each array value in turn.

- When you want to determine whether a variable holds one of many possible valid values, you can compare the variable to a list of values in an array. If you set up a parallel array with the same number of elements and corresponding data, you can use the same subscript to access additional information. You can create parallel arrays to more easily perform a range match.

- The `BinarySearch()` method finds a requested value in a sorted array. The method returns –1 if the value is not found in the array; otherwise, it returns the array position of the sought value. You cannot use the `BinarySearch()` method if your array items are not arranged in ascending order, if the array holds duplicate values and you want to find all of them, or if you want to find a range match rather than an exact match. The `Sort()` method arranges array items in ascending order. The `Reverse()` method reverses the order of items in an array.

- C# supports multidimensional arrays—those that require multiple subscripts to access the array elements. The most commonly used multidimensional arrays are two-dimensional arrays that are rect-angular. Two-dimensional arrays have two or more columns of values for each row. In a rectangular array, each row has the same number of columns. C# also supports jagged arrays, which are arrays of arrays.

- The major unusual consideration when using an array in a GUI program is that if the array values change based on user input, the array must be stored outside any method that reacts to the user's event.

Key Terms

An **array** is a list of data items that all have the same data type and the same name, but are distinguished from each other by a subscript or index.

Each object in an array is an **array element**.

A **subscript** (also called an **index**) is an integer contained within square brackets that indicates the position of one of an array's elements.

The keyword **`new`** is also known as the **`new` operator**; it is used to create objects.

An **initializer list** is the list of values provided for an array.

The class **`System.Array`** defines fields and methods that belong to every array.

The **`Length` property** is a member of the `System.Array` class that automatically holds an array's length.

The **`foreach` statement** is used to cycle through every array element without using a subscript.

A temporary **iteration variable** holds each array value in turn in a `foreach` statement.

A **sequential search** is conducted by examining a list in sequence.

A **parallel array** has the same number of elements as another array and corresponding data.

A **range match** determines the pair of limiting values between which a value falls.

The **`BinarySearch()` method** finds a requested value in a sorted array.

The **`Sort()` method** arranges array items in ascending order.

The **`Reverse()` method** reverses the order of items in an array.

A **one-dimensional** or **single-dimensional array** is an array whose elements you can access using a single subscript.

Multidimensional arrays require multiple subscripts to access the array elements.

Two-dimensional arrays have two or more columns of values for each row.

In a **rectangular array,** each row has the same number of columns.