

Storing values in variables provides programs with flexibility; a program that uses variables to replace constants can manipulate different values each time it 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. Learning to use the data structure known as an array offers further flexibility. Arrays allow you to 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

Sometimes, storing just one value in memory at a time isn't 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 value into a program, you can't determine whether it is above or below average because you won't know the average until you have entered all 20 values. You might plan to assign 20 sales values to 20 separate variables, each with a unique name, then sum and average them. However, that process is awkward and unwieldy: 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?

You could enter data for 20 salespeople using just one 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 value 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 individual 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 that indicates the position of a particular array element. In C#, a subscript is written between square brackets that follow an array name.

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 values for salespeople, you write the following:

```
double[] sales;
```



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 can provide any legal identifier you want for an array, but programmers conventionally name arrays like they name variables—starting with a lowercase letter and using uppercase letters to begin subsequent words. Additionally, many programmers observe one of the following conventions to make it more obvious that the name represents a group of items:

- Arrays are often named using a plural noun such as `sales`.
- Arrays are often named by adding a final word that implies a group, such as `salesList`, `salesTable`, or `salesArray`.

After you declare an array variable, you still need to create the actual array because declaring an array and reserving memory space for it are two distinct processes. You can declare an array variable and reserve memory locations for 20 `sales` objects using 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` elements. You also can declare and create an array in one statement, such as the following:

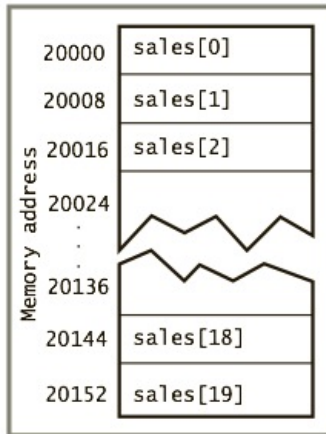
```
double[] sales = new double[20];
```



You can change the size of an array associated with an identifier, if necessary. For example, if you declare `int[] myArray`, you can assign five elements later with `myArray = new int[5]`; later in the program, you might alter the array size to 100 with `myArray = new int[100]`. 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 the default value for the data type.

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` values appears in computer memory. The figure assumes that the array begins at memory address 20000. 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. Because a `double` takes eight bytes of storage, each element of a `double` array is stored in succession at an address that is eight bytes higher than the previous one.



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

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. 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 the element `array[x + y]`.



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.

A common mistake is to forget that the first element in an array is element 0 (sometimes called the *zeroth element*), 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 most likely 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.

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.

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 element in the `sales` 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:

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



Watch the video *Declaring an Array*.

## 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`. (In the chapter “Introduction to Inheritance,” you will learn more about deriving classes.)

When you declare arrays or any other objects, the following default values are assigned to the elements:

- Numeric fields are set to 0.
- Character fields are set to `'\u0000'` or `null`. (You learned about escape sequences that start with `'\u'` in Chapter 2.)
- `bool` fields are set to `false`.

You can assign nondefault values to array elements at declaration by including 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};
```

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. Microsoft's documentation prefers the third example because it is most concise, but you should use the form of array initialization that is clearest to you or that is conventional in your organization.

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.



Programmers who have used other languages such as C++ 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 and list any values, 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);
```

You also can access a single character in a string using a subscript. For example, if you have defined `string greeting = "Hello"`, then `greeting[0]` is `'H'`. However, a string is not an array of characters, and you cannot assign a character to a portion of a string such as in the invalid assignment `word[0] = 'A'`.

**TWO TRUTHS & A LIE****Declaring an Array and Assigning Values**

1. To reserve memory locations for 10 `testScores` objects, you can use the following statement:  

```
int[] testScores = new int[9];
```
2. To assign 60 to the last element in a 10-element array named `testScores`, you can use the following statement:  

```
testScores[9] = 60;
```
3. The following statement creates an array named `testScores` and stores four values within the array:  

```
int[] testScores = new int[] {90, 85, 76, 92};
```

The false statement is #1. To reserve memory locations for 10 `testScores` objects, you must use 10 within the second set of square braces. The 10 elements will use the subscripts 0 through 9.

**Accessing Array Elements**

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;
```

If you treat each array element as an individual entity, declaring an array doesn't 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. 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. (You will learn about the *IndexOutOfRangeException* in the chapter "Exception Handling.") 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)
    WriteLine("{0} ", myScores[sub]);
```

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. In Chapter 2, you learned that every `string` has a `Length` property. Similarly, every array has a `Length` property that is a member of the `System.Array` class and that automatically holds an array’s length. The `Length` property 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};
WriteLine("Array size is {0}", myScores.Length);
for(int x = 0; x < myScores.Length; ++x)
    WriteLine(myScores[x]);
```



An array’s `Length` is a **read-only** property—a property you can access, but to which you cannot assign a new value. It is capitalized, as is the convention with all C# property identifiers. You will create property identifiers for your own classes in the chapter “Using Classes and Objects.”



When you write a program in which a large array is required, you might want to test it first with a much smaller array. For example, if a program requires interactive data entry for 50 items, you might want to test it first with only three or four items so you can work out the bugs without wasting too much time on data entry. After the program works correctly with just a few items, if you have used the `Length` property to access the array, you need only make one change to the array size at the top of the program when you are ready to test the full version.

## 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 `payRates` array in sequence:

```
double[] payRates = {12.00, 17.35, 21.12, 27.45, 32.22};
foreach(double money in payRates)
    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 `payRates` element value in turn—first, `payRates[0]`, then `payRates[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 can access it, but 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.

## TWO TRUTHS & A LIE

### Accessing Array Elements

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

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

2. Assume you have declared an array of eight `double`s 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};
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`.



 You Do It*Creating and Using an Array*

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

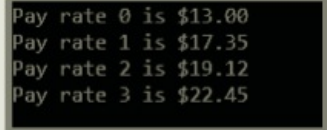
1. Open a new file, and begin a console-based program named **ArrayDemo1** that demonstrates array use:

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

2. Declare and create an array that holds four double values by typing:  
`double[] payRate = {13.00, 17.35, 19.12, 22.45};`
3. To confirm that the four values have been assigned, display them using the following code:

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

4. Add the two closing curly braces that end the `Main()` method and the `ArrayDemo1` class.
5. Save the program, and then compile and run it. The output appears in Figure 6-2.



```
Pay rate 0 is $13.00
Pay rate 1 is $17.35
Pay rate 2 is $19.12
Pay rate 3 is $22.45
```

**Figure 6-2** Output of the `ArrayDemo1` program

## 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 10 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 an 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
```

Instead of creating a long series of `if` statements, a more elegant solution to determining whether a value is valid is to compare it to a list of values in an array. For example, you can initialize an array with the valid values by using the following statement:

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



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.

After the `validValues` array is declared, you can use either a `for` loop or a `while` loop to search whether the `itemOrdered` variable value matches any of the array entries.

## Using a for Loop to Search an Array

One way to determine whether an `itemOrdered` value equals a value in the `validValues` array is to use a `for` statement to loop through the array and set a Boolean variable to `true` when a match is found:

234

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

This type of search is called a **sequential search** because each array element is examined in sequence. This simple `for` loop replaces the long series of `if` statements. What's more, if a company carries 1000 items instead of 10, 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 10 items your company carries have 10 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 6-3. 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 6-3, 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, 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 6-4 shows two typical program executions.

If you initialize parallel arrays, it is convenient to use spacing (as shown in Figure 6-3) so that the corresponding values visually align on the screen or printed page.



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.

```
using System;
using static System.Console;
class FindPriceWithForLoop
{
    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;
        Write("Please enter an item ");
        itemOrdered = Convert.ToInt32(ReadLine());
        for(int x = 0; x < validValues.Length; ++x)
        {
            if(itemOrdered == validValues[x])
            {
                isValidItem = true;
                itemPrice = prices[x];
            }
        }
        if(isValidItem)
            WriteLine("Price is {0}", itemPrice);
        else
            WriteLine("Sorry - item not found");
    }
}
```

Figure 6-3 The FindPriceWithForLoop program

```
Please enter an item 266
Price is 5.79
```

```
Please enter an item 267
Sorry - item not found
```

Figure 6-4 Two typical executions of the FindPriceWithForLoop program

In the fourth statement of the `Main()` method in Figure 6-3, `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 that the `if` statement's Boolean expression could have been `false` and the variable might not have been set to a valid value.

## Improving a Loop's Efficiency

The code shown in Figure 6-3 can be made more efficient. Currently, the program compares every `itemOrdered` with each of the 10 `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 comparing all the values. 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. Figure 6-5 shows this approach.

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

**Figure 6-5** Loop with forced early exit

In the code segment in Figure 6-5, 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. A `break` statement exits the current code block immediately.

If you decide to leave a loop as soon as a match is found, the most efficient strategy is to place the most common items first so they are matched sooner. 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. However, it might be more convenient for people to view the item numbers in ascending numerical order. In many business applications, your first consideration is how easily users can read, understand, and modify your programs. However, in other applications, such as programming for mobile devices, speed and memory considerations are more important. You should follow the recommendations of your instructors or supervisors.

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` 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 until it matches `validValues.Length`, then you never found a match in the 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;
using static System.Console;
class FindPriceWithWhileLoop
{
    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};
        Write("Enter item number ");
        inputString = 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)
            WriteLine("Item {0} sells for {1}",
                itemOrdered, itemPrice.ToString("C"));
        else
            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.

```

Enter item number 409
Item 409 sells for $0.89

Enter item number 410
No such item as 410

```

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

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 value in each range, then you can create an array that holds five low limits:

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

A parallel array will hold the five discount rates:

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

Then, starting at the last `discountRangeLowLimits` array element, for any `numOfItems` greater than or equal to `discountRangeLowLimits[4]`, the appropriate discount is `discounts[4]`. In other words, for any `numOfItems` less than `discountRangeLowLimits[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[] discountRangeLowLimits = {1, 13, 50, 100, 200};
double[] discounts = {0, 0.10, 0.14, 0.18, 0.20};
double customerDiscount;
int sub = discountRangeLowLimits.Length - 1;
while(sub >= 0 && numOfItems < discountRangeLowLimits[sub])
    --sub;
customerDiscount = discounts[sub];
```

**Figure 6-9** Searching an array of range limits

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[] discountRangeUpperLimits = {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 < discountRangeUpperLimits.Length &&
    numOfItems > discountRangeUpperLimits[sub])
    ++sub;
customerDiscount = discounts[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 `discountRangeUpperLimits` array value is 9999999. This very high value was used with the assumption that no `numOfItems` would ever exceed it, but, because this assumption could

possibly be wrong, many programmers prefer to use a range-checking method that uses lower range limits. 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. Terminating the loop cycles as soon as a match is found is the most efficient approach.

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

You have already learned that every array in C# can use the Length property it gets from the System.Array class. Additionally, the System.Array class contains a variety of useful, built-in methods that you can use with arrays. This section shows you how to use the methods Array.BinarySearch() method to find an element in an array, the Array.Sort() method to sort an array's elements, and the Array.Reverse() method to reverse the order of elements. If you include the statement using static System.Array();, you can use each of these methods without using the Array class name and the dot.

### Using the BinarySearch() Method

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?" and continued to narrow your guesses upon hearing each subsequent answer, then you have performed a binary search. In C#, the BinarySearch() method finds a requested value in a sorted array.

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 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 `WriteLine()` method, you have passed a format string and values to be displayed, all separated by commas.

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

**Figure 6-10** The `BinarySearchDemo` program

```
Enter an Employee ID 219
ID 219 found at position 3
```

```
Enter an Employee ID 220
ID 220 not found
```

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

You have sent arguments to methods, as in the following statement:

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

You also have accepted methods' returned values, as in the following statement:

```
entryString = ReadLine();
```

When you use the `BinarySearch()` method, you both send arguments and receive returned values:

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

The statement calls the method that performs the search, returning `-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 earlier in this chapter. 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 doesn't 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, you can't use the `BinarySearch()` method.

## Using the Sort() Method

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

**Figure 6-12** The SortArray program



```
Mindy
Ned
Olive
Patty
Richard
```

**Figure 6-13** Execution of the SortArray program

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

244



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.

## 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. When you `Reverse()` an array that contains an odd number of elements, the middle element will remain in its original location. The `Reverse()` method does not sort array elements; it only rearranges their values to the opposite order.

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.

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

**Figure 6-14** The `ReverseArray` program

```
Marcia
Wendy
Rose
Zach
```

**Figure 6-15** Execution of the `ReverseArray` program

## TWO TRUTHS &amp; A LIE

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

1. When you use the BinarySearch() method, the searched 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.



## You Do It

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

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

1. Open a new file and type the beginning of a program 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;
using static System.Console;
class ArrayDemo2
{
    static void Main()
    {
        int[] scores = new int[8];
        int x;
        string inputString;
```

2. Add two more items that will be used to improve the appearance of the output: a counter to count dashes to be displayed in a line and a constant for the number of dashes:

```
int count;
const int DASHES = 50;
```

(continues)

(continued)

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. When the loop is complete, execute a `WriteLine()` statement so that the next output appears on a new line.

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

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 loop that creates a dashed line to visually separate the input from the output.

```
for(count = 0; count < DASHES; ++count)
    Write("-");
WriteLine();
```

5. Display *Scores in original order.*, then use a loop to display each score in a field that is six characters wide.

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

6. 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.

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

7. 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.

(continues)

(continued)

```

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

```

8. Add two closing curly braces—one for the `Main()` method and one for the class.
9. Save the file, and then compile and execute the program. Figure 6-16 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.

```

Enter your score on test 1 75
Enter your score on test 2 88
Enter your score on test 3 62
Enter your score on test 4 100
Enter your score on test 5 95
Enter your score on test 6 90
Enter your score on test 7 89
Enter your score on test 8 91

-----
Scores in original order:
  75  88  62 100  95  90  89  91
-----
Scores in sorted order:
  62  75  88  89  90  91  95 100
-----
Scores in reverse order:
 100  95  91  90  89  88  75  62

```

**Figure 6-16** Typical execution of the `ArrayDemo2` program

## Using Multidimensional Arrays

When you declare an array such as `double[] sales = new double[20];`, you can envision the declared integers as a list or 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**. You can think of the single dimension of a single-dimensional array as the height of the 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 6-17. You can think of the two dimensions of a two-dimensional array as height and width.

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 6-17** View of a rectangular, two-dimensional array in memory

The array in Figure 6-17 is a rectangular array. 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. You might want to create a sales array with two dimensions, as shown in Figure 6-17, if, for example, each row represented a category of items sold, and each column represented 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 6-17 can be declared as the following, creating an array named sales that holds three rows and four columns:

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

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

Just as with a one-dimensional array, every element in a two-dimensional array is the same data type. Also, just as with a one-dimensional array if you do not provide values for the elements in a two-dimensional array, the values are set to the default value for the data type (for example, 0 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 by assigning values when it is created. For example, the following code assigns values to sales upon declaration:

```
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. The row and its curly braces are separated from the next row with a comma. The next four values in their own set of braces make up the second row (row 1), which you reference with the subscript 1, and the last four values constitute the third row (row 2).

When you refer to an element in a two-dimensional array, 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. As examples, the value of `sales[0, 0]` is 14.00, the value of `sales[0, 1]` is 15.00, the value of `sales[1, 0]` is 21.99, and the value of `sales[2, 3]` is 1.17. 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 that 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 6-18 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 6-18** Rents charged (in dollars)

To determine a tenant's rent, you need to know two pieces of information about the apartment: the floor and the number of bedrooms. 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} };
```

If `floor` and `bedrooms` are integers with in-range values, then any tenant's rent can be referred to as `rents[floor, bedrooms]`.

Figure 6-19 shows a complete program that uses a rectangular, two-dimensional array to hold rent values. Figure 6-20 shows a typical execution.

```

using System;
using static System.Console;
class RentFinder
{
    static void Main()
    {
        int[ , ] rents = { {400, 450, 510},
                          {500, 560, 630},
                          {625, 676, 740},
                          {1000, 1250, 1600} };

        int floor;
        int bedrooms;
        string inputString;
        Write("Enter the floor on which you want to live ");
        inputString = ReadLine();
        floor = Convert.ToInt32(inputString);
        Write("Enter the number of bedrooms you need ");
        inputString = ReadLine();
        bedrooms = Convert.ToInt32(inputString);
        WriteLine("The rent is {0}", rents[floor, bedrooms]);
    }
}

```

Figure 6-19 The RentFinder program

```

Enter the floor on which you want to live 2
Enter the number of bedrooms you need 1
The rent is 676

```

Figure 6-20 Typical execution of the RentFinder program



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

C# supports arrays with more than two dimensions. For example, as in the program in Figure 6-19, 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. However, 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 6-21 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 6-21** 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; the inner curly braces represent the following:

- Four inner sets of braces surround the data for each building—each row of values represents a building (0 through 3).
- Within each row, the three sets of inner braces represent each floor—first a basement, then floor one, and floor two. For example, in building 0, {400, 500} are rents for floor 0, and {450, 550} are rents for floor 1.
- Within each floor, the two braced values represent the bedrooms—first a zero-bedroom apartment and then a one-bedroom apartment. For example, in building 0, floor 0, 400 is the rent for a zero-bedroom apartment, and 500 is the rent for a one-bedroom apartment.

Using the three-dimensional array in Figure 6-21, an expression such as `rents[building, floor, bedrooms]` refers to a specific rent value 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 (which is the fourth building). The value of `rents[3, 1, 0]` is \$1050 in Figure 6-21. 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 don't exhaust your computer's memory, C# lets you create arrays of any size.

## Using Jagged Arrays

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 10 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 10 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 array, allowing a different number of elements for each row. Figure 6-22 shows how you could implement this option.

252

```
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 6-22** A jagged array

Two square brackets are used following the data type of the array in Figure 6-22. This notation declares a jagged array that 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 6-22, 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

- A rectangular array has the same number of columns as rows.
- The following array contains two rows and three columns:
 

```
int[ , ] departments = {{12, 54, 16},
                        {22, 44, 47}};
```
- 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.