CHAPTER 8

Arrays

In this chapter, you will:

- Declare arrays
- Initialize an array
- Use variable subscripts with an array
- Declare and use arrays of objects
- Search an array and use parallel arrays
- Pass arrays to and return arrays from methods

Declaring Arrays

While completing the first five chapters in this book, you stored values in variables. In those early chapters, you simply stored a value and used it, usually only once, but never more than a few times. In Chapter 6, you created loops that allow you to "recycle" variables and use them many times; that is, after creating a variable, you can assign a value, use the value, and then, in successive cycles through the loop, reuse the variable as it holds different values.

At times, however, you might encounter situations in which storing just one value at a time in memory does not meet your needs. 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 an application, you can't determine whether it is above or below average because you don't know the average until you have all 20 values. Unfortunately, if you attempt to assign 20 sales values to the same variable, when you assign the value for the second employee, it replaces the value for the first employee.

A possible solution is to create 20 separate employee sales variables, each with a unique name, so you can store all the sales until you can determine an average. A drawback to this method is that if you have 20 different variable names to be assigned values, you need 20 separate assignment statements. For 20 different variable names, the statement that calculates total sales will be unwieldy, such as:

```
total = firstAmt + secondAmt + thirdAmt + …
```

This method might work for 20 salespeople, but what if you have 10,000 salespeople?

The best solution is to create an array. An **array** is a named list of data items that all have the same type. Each data item is an **element** of the array. You declare an array variable in the same way you declare any simple variable, but you insert a pair of square brackets after the type. For example, to declare an array of `double` values to hold sales figures for salespeople, you can write the following:

```
double[] salesFigures;
```

Similarly, to create an array of integers to hold student ID numbers, you can write the following:

```
int[] idNums;
```

In Java, you can also declare an array variable by placing the square brackets after the array name, as in `double salesFigures[];`. This format is familiar to C and C++ programmers, but the preferred format among Java programmers is to place the brackets following the variable type and before the variable name.

You can provide any legal identifier you want for an array, but Java programmers conventionally name arrays by following the same rules they use for variables—array names start with a lowercase letter and use 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 `salesFigures`.
- Arrays are often named by adding a final word that implies a group, such as `salesList`, `salesTable`, or `salesArray`.

After you create an array variable, you still need to reserve memory space. You use the same procedure to create an array that you use to create an object. Recall that when you create a class named `Employee`, you can declare an `Employee` object with a declaration such as:

```
Employee oneWorker;
```

However, that declaration does not actually create the `oneWorker` object. You create the `oneWorker` object when you use the keyword `new` and the constructor method, as in:

```
oneWorker = new Employee();
```

Similarly, declaring an array and reserving memory space for it are two distinct processes. To reserve memory locations for 20 `salesFigures` values, you can declare the array variable and create the array with two separate statements as follows:

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

Alternatively, just as with objects, you can declare and create an array in one statement with the following:

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

In Java, the size of an array follows the data type and is never declared immediately following the array name, as it is in some other languages such as C++. Other languages, such as Visual Basic, BASIC, and COBOL, use parentheses rather than brackets to refer to individual array elements. By using brackets, the creators of Java made it easier for you to distinguish array names from methods.

The statement `double[] salesFigures = new double[20];` reserves 20 memory locations for 20 `double` values. You can distinguish each `salesFigures` item from the others with a subscript. A **subscript** is an integer contained within square brackets that specifies one of an array's elements. In Java, any array's elements are numbered beginning with 0, so you can legally use any subscript from 0 through 19 when working with an array that has 20 elements. In other words, the first `salesFigures` array element is `salesFigures[0]` and the last `salesFigures` element is `salesFigures[19]`. Figure 8-1 shows how the first few and last few elements of an array of 20 `salesFigures` values appears in computer memory.

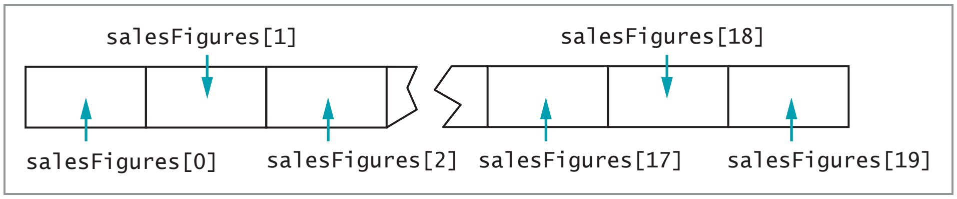


Figure 8-1 The first few and last few elements of an array of 20 `salesFigures` items in memory

A subscript is also called an **index**. In particular, you will see the term *index* in some error messages issued by the compiler.

It is a common mistake 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. It is also common to forget that the last element's subscript is one less than the array's size and not the array's size. For example, the highest allowed subscript for a 100-element array is 99. To remember that array elements begin with element 0, it might help if you 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. If you use a subscript that is too small (that is, negative) or too large for an array, the subscript is **out of bounds** and an error message is generated.

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

```
salesFigures[0] = 2100.00;
```

To display the last `salesFigures` element in an array of 20, you can write:

```
System.out.println(salesFigures[19]);
```

When programmers talk about these statements, they typically say things like, "`salesFigures` sub zero is assigned 2100.00," and "`salesFigures` sub 19 is output." In other words, they use *sub* as shorthand for "with the subscript."

When you declare or access an array, you can use any expression to represent the size, as long as the expression is an integer. For example, to declare a `double` array named `moneyValues`, you might use any of the following:

- A literal integer constant; for example:

  ```
  double[] moneyValues = new double[10];
  ```

- A named integer constant; for example:

  ```
  double[] moneyValues = new double[NUMBER_ELS];
  ```

  In this example, the constant `NUMBER_ELS` must have been declared previously and assigned a value.

- An integer variable; for example:

  ```
  double[] moneyValues = new double[numberOfEls];
  ```

  In this example, the variable `numberOfEls` must have been declared previously and assigned a value.

- A calculation; for example:

  ```
  double[] moneyValues = new double[x + y * z];
  ```

  In this example, the variables `x`, `y`, and `z` must have been declared previously and assigned values, and the result of the mathematical expression must be an integer.
- A method's `return` value; for example:

  ```
  double[] moneyValues = new double[getElements()];
  ```

  In this example, the method `getElements()` must return an integer.

Some other programming languages, such as C++, allow only named or unnamed constants to be used for the size when an array is declared. Java allows variables, arithmetic expressions, and method return values, which makes array declaration more flexible.

TWO TRUTHS & A LIE

Declaring Arrays

1. The statement `int[] idNums = new int[35];` reserves enough memory for exactly 34 integers.
2. The first element in any array has a subscript of 0, no matter what data type is stored.
3. In Java, you can use a variable as well as a constant to declare an array's size.

The false statement is #1. The statement `int[] idNums = new int[35];` reserves enough memory for exactly 35 integers numbered 0 through 34.

You Do It

Declaring an Array

In this section, you create a small array to see how arrays are used. The array holds salaries for four categories of employees.

1. Open a new text file in your text editor.
2. Begin the class that demonstrates how arrays are used by typing the following class and `main()` headers and their corresponding opening curly braces:

(continues)

(continued)

```
public class DemoArray
{
   public static void main(String[] args)
   {
```

3. On a new line, declare and create an array that can hold four `double` values by typing the following:

```
double[] salaries = new double[4];
```

4. One by one, assign four values to the four array elements by typing the following:

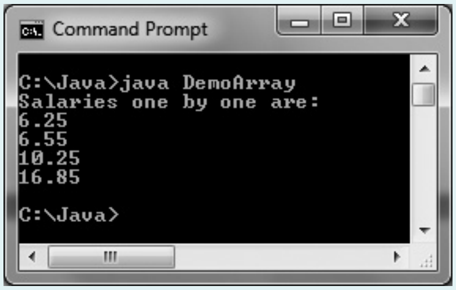
```
salaries[0] = 6.25;
salaries[1] = 6.55;
salaries[2] = 10.25;
salaries[3] = 16.85;
```

5. To confirm that the four values have been assigned, display the salaries one by one using the following code:

```
System.out.println("Salaries one by one are:");
System.out.println(salaries[0]);
System.out.println(salaries[1]);
System.out.println(salaries[2]);
System.out.println(salaries[3]);
```

6. Add the two closing curly braces that end the `main()` method and the `DemoArray` class.

7. Save the program as **DemoArray.java**. Compile and run the program. The program's output appears in Figure 8-2.



```
C:\Java>java DemoArray
Salaries one by one are:
6.25
6.55
10.25
16.85

C:\Java>
```

Figure 8-2 Output of the `DemoArray` application

(continues)

(continued)

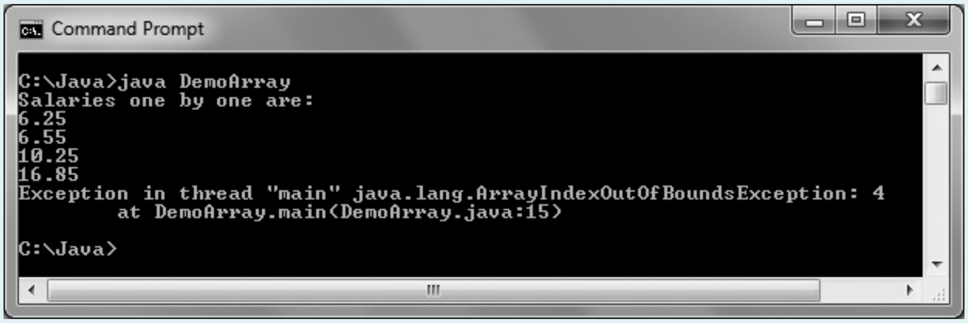
Using a Subscript that is Out of Bounds

In this section, you purposely generate an out-of-bounds error so you can familiarize yourself with the error message generated.

1. As the last executable line in the DemoArray.java file, add a new output statement that attempts to display a `salaries` value using a subscript that is beyond the range of the array:

   ```
   System.out.println(salaries[4]);
   ```

2. Save the file, and then compile and execute it. The output looks like Figure 8-3. The program runs successfully when the subscript used with the array is 0, 1, 2, or 3. However, when the subscript reaches 4, the error in Figure 8-3 is generated. The message indicates that an `ArrayIndexOutOfBoundsException` has occurred and that the offending index is 4.



```
C:\Java>java DemoArray
Salaries one by one are:
6.25
6.55
10.25
16.85
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at DemoArray.main(DemoArray.java:15)

C:\Java>
```

Figure 8-3 Output of the `DemoArray` application when a subscript is out of bounds

In Chapter 12, you will learn more about the term *exception* and learn new ways to deal with exceptions.

3. Remove the offending statement from the `DemoArray` class. Save the program, and then compile and run it again to confirm that it again executes correctly.

Initializing an Array

A variable that has a primitive type, such as `int`, holds a value. A variable with a reference type, such as an array, holds a memory address where a value is stored. In other words, array names contain references, as do all Java objects.

No memory address is assigned when you declare an array using only a data type, brackets, and a name. Instead, the array variable name has the special value `null`, which means the identifier is not associated with a memory address. For example, when you define `someNums` in the following statement, the value of `someNums` is `null`:

```
int[] someNums;
```

When you use the keyword `new` to define an array, the array name acquires an actual memory address value. For example, when you define `someNums` in the following statement, a memory address is assigned:

```
int[] someNums = new int[10];
```

When you declare `int[] someNums = new int[10];`, `someNums` holds an address, but each element of `someNums` has a value of 0 because `someNums` is an integer array. Each element in a `double` or `float` array is assigned 0.0. By default, `char` array elements are assigned '\u0000', which is the Unicode value for a `null` character, and `boolean` array elements automatically are assigned the value `false`. In arrays of objects, including `String`s, each element is assigned `null` by default.

You already know how to assign a different value to a single element of an array, as in:

```
someNums[0] = 46;
```

You can also assign nondefault values to array elements upon creation. To initialize an array, you use an **initialization list** of values separated by commas and enclosed within curly braces. Providing values for all the elements in an array also is called **populating an array**.

For example, if you want to create an array named `tenMults` and store the first six multiples of 10 within the array, you can declare `tenMults` as follows:

```
int[] tenMults = {10, 20, 30, 40, 50, 60};
```

Notice the semicolon at the end of the statement. You don't use a semicolon following a method's closing curly brace, but you do use one following the closing brace of an array initialization list.

When you populate an array upon creation by providing an initialization list, you do not give the array a size—the size is assigned based on the number of values you place in the initializing list. For example, the `tenMults` array just defined has a size of 6. Also, when you initialize an array, you do not need to use the keyword `new`; instead, new memory is assigned based on the length of the list of provided values.

In Java, you cannot directly initialize part of an array. For example, you cannot create an array of 10 elements and initialize only five; you either must initialize every element or none of them.

Watch the video *Arrays*.

TWO TRUTHS & A LIE

Initializing an Array

1. When you declare `int[] idNums = new int[35];`, each element of the array has a value of 0.
2. When you declare `double[] salaries = new double[10];`, each element of the array has a value of 0.0.
3. When you declare `int[] scores = {100, 90, 80};`, the first three elements of the array are assigned the values listed, but all the remaining elements are assigned 0.

The false statement is #3. When you provide an initialization list for an array, the array contains exactly that many elements.

You Do It

Initializing an Array

Next, you alter your `DemoArray` program to initialize the array of `doubles`, rather than declaring the array and assigning values later.

1. Open the **DemoArray.java** file in your text editor. Immediately save the file as **DemoArray2.java**. Change the class name to **`DemoArray2`**. Delete the statement that declares the array of four `doubles` named `salaries`, and then replace it with the following initialization statement:

   ```
   double[] salaries = {6.25, 6.55, 10.25, 16.85};
   ```

2. Delete the following four statements that individually assign the values to the array:

   ```
   salaries[0] = 6.25; salaries[1] = 6.55; salaries[2] = 10.25;
      salaries[3] = 16.85;
   ```

3. Save the file (as **DemoArray2.java**), compile, and test the application. The values that are output are the same as those shown for the `DemoArray` application in Figure 8-2.

Using Variable Subscripts with an Array

If you treat each array element as an individual entity, there isn't much of an advantage to declaring an array over declaring individual primitive type variables, such as those with the type `int`, `double`, or `char`. The power of arrays becomes apparent when you begin to use subscripts that are variables, rather than subscripts that are constant values.

For example, suppose you declare an array of five integers that holds quiz scores, such as the following:

```
int[] scoreArray = {2, 14, 35, 67, 85};
```

You might want to perform the same operation on each array element, such as increasing each score by a constant amount. To increase each `scoreArray` element by three points, for example, you can write the following:

```
final int INCREASE = 3;
scoreArray[0] += INCREASE;
scoreArray[1] += INCREASE;
scoreArray[2] += INCREASE;
scoreArray[3] += INCREASE;
scoreArray[4] += INCREASE;
```

With five `scoreArray` elements, this task is manageable, requiring only five statements. However, you can reduce the amount of program code needed by using a variable as the subscript. Then, you can use a loop to perform arithmetic on each array element, as in the following example:

```
final int INCREASE = 3;
for(sub = 0; sub < 5; ++sub)
   scoreArray[sub] += INCREASE;
```

The variable `sub` is set to 0, and then it is compared to 5. Because the value of `sub` is less than 5, the loop executes and 3 is added to `scoreArray[0]`. Then, the variable `sub` is incremented and it becomes 1, which is still less than 5, so when the loop executes again, `scoreArray[1]` is increased by 3, and so on. A process that took five statements now takes only one. In addition, if the array had 100 elements, the first method of increasing the array values by 3 in separate statements would result in 95 additional statements. The only changes required using the second method would be to change the array size to 100 by inserting additional initial values for the scores, and to change the middle portion of the `for` statement to compare `sub` to 100 instead of to 5. The loop to increase 100 separate scores by 3 each is:

```
for(sub = 0; sub < 100; ++sub)
   scoreArray[sub] += INCREASE;
```

When an application contains an array and you want to use every element of the array in some task, it is common to perform loops that vary the loop control variable from 0 to one less than the size of the array. For example, if you get input values for the elements in the array, alter every value in the array, sum all the values in the array, or display every element in the array, you need to perform a loop that executes the same number of times as there are elements. When there are 10 array elements, the subscript varies from 0 through 9; when there are 800 elements, the subscript varies from 0 through 799. Therefore, in an application that includes an array, it is convenient to declare a symbolic constant equal to the size of the array and use the symbolic constant as a limiting value in every loop that processes the array. That way, if the array size changes in the future, you need to modify only the value stored in the symbolic constant, and you do not need to search for and modify the limiting value in every loop that processes the array.

For example, suppose you declare an array and a symbolic constant as follows:

```
int[] scoreArray = {2, 14, 35, 67, 85};
final int NUMBER_OF_SCORES = 5;
```

Then, the following two loops are identical:

```
for(sub = 0; sub < 5; ++sub)
   scoreArray[sub] += INCREASE;
for(sub = 0; sub < NUMBER_OF_SCORES; ++sub)
   scoreArray[sub] += INCREASE;
```

The second format has two advantages. First, by using the symbolic constant, NUMBER_OF_SCORES, the reader understands that you are processing every array element for the size of the entire array. If you use the number 5, the reader must look back to the array declaration to confirm that 5 represents the full size of the array. Second, if the array size changes because you remove or add scores, you change the symbolic constant value only once, and all loops that use the constant are automatically altered to perform the correct number of repetitions.

As another option, you can use a field (instance variable) that is automatically assigned a value for every array you create; the **length field** contains the number of elements in the array. For example, when you declare an array using either of the following statements, the field scoreArray.length is assigned the value 5:

```
int[] scoreArray = {2, 14, 35, 67, 85};
int[] scoreArray = new int[5];
```

Therefore, you can use the following loop to add 3 to every array element:

```
for(sub = 0; sub < scoreArray.length; ++sub)
   scoreArray[sub] += INCREASE;
```

Later, if you modify the size of the array and recompile the program, the value in the length field of the array changes appropriately. When you work with array elements, it is always better to use a symbolic constant or the length field when writing a loop that manipulates an array.

A frequent programmer error is to attempt to use length as an array method, referring to scoreArray.length(). However, length is not an array method; it is a field. An instance variable or object field such as length is also called a **property** of the object.

In Chapter 6, you learned to use the for loop. Java also supports an **enhanced for loop**. This loop allows you to cycle through an array without specifying the starting and ending points for the loop control variable. For example, you can use either of the following statements to display every element in an array named scoreArray:

```
for(int sub = 0; sub < scoreArray.length; ++sub)
   System.out.println(scoreArray[sub]);
for(int val : scoreArray)
   System.out.println(val);
```

In the second example, `val` is defined to be the same type as the array named following the colon. Within the loop, `val` takes on, in turn, each value in the array. You can read the second example as, "For each `val` in `scoreArray`, display `val`." As a matter of fact, you will see the enhanced `for` loop referred to as a **`foreach` loop**.

You also can use the enhanced `for` loop with more complicated Java objects, as you will see in the next section.

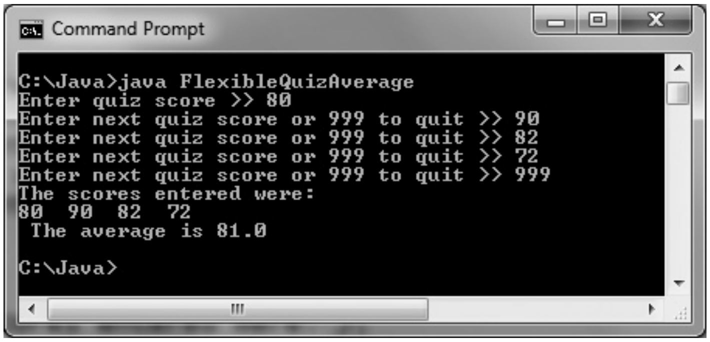
Using Part of an Array

Sometimes, you do not want to use every value in an array. For example, suppose that you write a program that allows a student to enter up to 10 quiz scores and then computes and displays the average. To allow for 10 quiz scores, you create an array that can hold 10 values, but because the student might enter fewer than 10 values, you might use only part of the array. Figure 8-4 shows such a program.

```
import java.util.*;
public class FlexibleQuizAverage
{
   public static void main(String[] args)
   {
      int[] scores = new int[10];
      int score = 0;
      int count = 0;
      int total = 0;
      final int QUIT = 999;
      final int MAX = 10;
      Scanner input = new Scanner(System.in);
      System.out.print("Enter quiz score >> ");
      score = input.nextInt();
      while(count < MAX && score != QUIT)
      {
         if(score != QUIT)
         {
            scores[count] = score;
            total += scores[count];
            System.out.print("Enter next quiz score or " +
               QUIT + " to quit >> ");
            score = input.nextInt();
         }
         count++;
      }
      System.out.println("The scores entered were:");
      for(int x = 0; x < count; ++x)
         System.out.print(scores[x] + "  ");
      System.out.println("\n The average is " + (total * 1.0 / count));
   }
}
```

Figure 8-4 The `FlexibleQuizAverage` application

The `FlexibleQuizAverage` program declares an array that can hold 10 quiz scores. The user is prompted for a first quiz score; then, in the `while` loop that starts with the first shaded line in Figure 8-4, the score is placed in the `scores` array. Then the score is added to a running total, and the user is prompted to enter another quiz score or a `QUIT` value of 999 to stop. The `while` loop continuously checks to ensure that no more than 10 scores have been entered and that the user has not entered 999 to quit. After each score entry, the variable `count` is incremented, which serves two purposes: It indicates the element where the next score should be stored, and when the loop eventually ends, `count` holds the number of scores entered. The variable `count` then can be used to control the output loop (in the shaded `for` statement) and to help calculate the average score. Figure 8-5 shows a typical execution of the program.



```
C:\Java>java FlexibleQuizAverage
Enter quiz score >> 80
Enter next quiz score or 999 to quit >> 90
Enter next quiz score or 999 to quit >> 82
Enter next quiz score or 999 to quit >> 72
Enter next quiz score or 999 to quit >> 999
The scores entered were:
80  90  82  72
 The average is 81.0

C:\Java>
```

Figure 8-5 Typical execution of the `FlexibleQuizAverage` application

TWO TRUTHS & A LIE

Using Variable Subscripts with an Array

1. When an application contains an array, it is common to perform loops that vary the loop control variable from 0 to one less than the size of the array.
2. An array's `length` field contains the highest value that can be used as the array's subscript.
3. The enhanced `for` loop allows you to cycle through an array without specifying the starting and ending points for the loop control variable.

The false statement is #2. An array's `length` field contains the number of elements in the array.

You Do It

Using a `for` *Loop to Access Array Elements*

Next, you modify the `DemoArray2` program to use a `for` loop with the array.

1. Open the **DemoArray2.java** file in your text editor. Immediately save the file as **DemoArray3.java**. Change the class name to **`DemoArray3`**. Delete the four `println()` statements that display the four array values, and then replace them with the following `for` loop:

   ```
   for(int x = 0; x < salaries.length; ++x)
      System.out.println(salaries[x]);
   ```

2. Save the program (as **DemoArray3.java**), compile, and run the program. Again, the output is the same as that shown in Figure 8-2.

Declaring and Using Arrays of Objects

Just as you can declare arrays of `int`s or `double`s, you can declare arrays that hold elements of any type, including objects. For example, assume you create the `Employee` class shown in Figure 8-6. This class has two data fields (`empNum` and `empSal`), a constructor, and a get method for each field.

```
public class Employee
{
   private int empNum;
   private double empSal;
   Employee(int e, double s)
   {
      empNum = e;
      empSal = s;
   }
   public int getEmpNum()
   {
      return empNum;
   }
   public double getSalary()
   {
      return empSal;
   }
}
```

Figure 8-6 The `Employee` class

You can create separate Employee objects with unique names, such as either of the following:

```
Employee painter, electrician, plumber;
Employee firstEmployee, secondEmployee, thirdEmployee;
```

However, in many programs it is far more convenient to create an array of Employee objects. An array named emps that holds seven Employee objects can be defined as:

```
Employee[] emps = new Employee[7];
```

This statement reserves enough computer memory for seven Employee objects named emps[0] through emps[6]. However, the statement does not actually construct those Employee objects; instead, you must call the seven individual constructors. According to the class definition shown in Figure 8-6, the Employee constructor requires two arguments: an employee number and a salary. If you want to number your Employees 101, 102, 103, and so on, and start each Employee at a salary of $6.35, the loop that constructs seven Employee objects is as follows:

```
final int START_NUM = 101;
final double PAYRATE = 6.35;
for(int x = 0; x < emps.length; ++x)
   emps[x] = new Employee(START_NUM + x, PAYRATE);
```

As x varies from 0 through 6, each of the seven emps objects is constructed with an employee number that is 101 more than x, and each of the seven emps objects holds the same salary of $6.35, as assigned in the constant PAYRATE.

Unlike the Employee class in Figure 8-6, which contains a constructor that requires arguments, some classes contain only a default constructor, which might be supplied automatically when no other constructors are created or might be written explicitly. To construct an array of objects using a default constructor, you must still call the constructor using the keyword new for each declared array element. For example, suppose you have created a class named InventoryItem but have not written a constructor. To create an array of 1,000 InventoryItem objects, you would write the following:

```
final int NUM_ITEMS = 1000;
InventoryItem[] items = new InventoryItem[NUM_ITEMS];
for(int x = 0; x < NUM_ITEMS; ++x)
   items[x] = new InventoryItem();
```

To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot that precedes the method name. For example, to display data for seven Employees stored in the emps array, you can write the following:

```
for(int x = 0; x < emps.length; ++x)
   System.out.println (emps[x].getEmpNum() + " " +
      emps[x].getSalary());
```

Pay attention to the syntax of the Employee objects' method calls, such as emps[x].getEmpNum(). Although you might be tempted to place the subscript at the end of the expression after the method name—as in emps.getEmpNum[x] or emps.getEmpNum()[x]—you cannot; the values in x (0 through 6) refer to a particular emps object, each of which has access to a single getEmpNum() method. Placement of the bracketed subscript so it follows emps means the method "belongs" to a particular element of emps.

Using the Enhanced `for` Loop with Objects

You can use the enhanced `for` loop to cycle through an array of objects. For example, to display data for seven `Employees` stored in the `emps` array, you can write the following:

```
for(Employee worker : emps)
   System.out.println(worker.getEmpNum() + " " + worker.getSalary());
```

In this loop, `worker` is a local variable that represents each element of `emps` in turn. Using the enhanced `for` loop eliminates the need to use a limiting value for the loop and eliminates the need for a subscript following each element.

Manipulating Arrays of `Strings`

As with any other object, you can create an array of `String` objects. For example, you can store three company department names as follows:

```
String[] deptNames = {"Accounting", "Human Resources", "Sales"};
```

You can access these department names like any other array object. For example, you can use the following code to display the list of `Strings` stored in the `deptNames` array:

```
for(int a = 0; a < deptNames.length; ++a)
   System.out.println(deptNames[a]);
```

Notice that `deptNames.length;` refers to the length of the array `deptNames` (three elements) and not to the length of any `String` objects stored in the `deptNames` array. Arrays use a `length` field (no parentheses follow). Each `String` object has access to a `length()` method that returns the length of a `String`. For example, if `deptNames[0]` is "Accounting", then `deptNames[0].length()` is 10 because "Accounting" contains 10 characters.

TWO TRUTHS & A LIE

Declaring and Using Arrays of Objects

1. The following statement declares an array named `students` that holds 10 `Student` objects:

   ```
   Student[] students = new Student[10];
   ```

2. When a class has a default constructor and you create an array of objects from the class, you do not need to call the constructor explicitly.
3. To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot that precedes the method name.

The false statement is #2. Whether a class has a default constructor or not, when you create an array of objects from the class, you must call the constructor using the keyword `new` for each declared array element.

You Do It

Creating a Class That Contains an Array of `Strings`

In this section, you create a class named `BowlingTeam` that contains the name of a bowling team and an array that holds the names of the four team members.

1. Open a new file in your text editor, and type the header and curly braces for the `BowlingTeam` class:

```
public class BowlingTeam
{
}
```

2. Create a field for the team name and an array that holds the team members' names.

```
private String teamName;
private String[] members = new String[4];
```

3. Create get and set methods for the `teamName` field as follows:

```
public void setTeamName(String team)
{
   teamName = team;
}
public String getTeamName()
{
   return teamName;
}
```

4. Add a method that sets a team member's name. The method requires a position and a name, and it uses the position as a subscript to the `members` array.

```
public void setMember(int number, String name)
{
   members[number] = name;
}
```

5. Add a method that returns a team member's name. The method requires a value used as the subscript that determines which member's name to return.

```
public String getMember(int number)
{
   return members[number];
}
```

6. Save the file as **BowlingTeam.java**. Compile it and correct any errors.

(continues)

(continued)

Creating a Program to Demonstrate an Instance of the `BowlingTeam` *Class*

In this section, you write a program in which you create an instance of the `BowlingTeam` class and provide values for it.

1. Open a new file in your text editor, and enter the following code to begin the class.

```
public class BowlingTeamDemo
{
   public static void main(String[] args)
   {
```

2. Add five declarations. These include a `String` that holds user input, a `BowlingTeam` object, an integer to use as a subscript, a constant that represents the number of members on a bowling team, and a `Scanner` object for input.

```
String name;
BowlingTeam bowlTeam = new BowlingTeam();
int x;
final int NUM_TEAM_MEMBERS = 4;
Scanner input = new Scanner(System.in);
```

3. Prompt the user for a bowling team name. Accept it, and then assign it to the `BowlingTeam` object:

```
System.out.print("Enter team name >> ");
name = input.nextLine();
bowlTeam.setTeamName(name);
```

4. In a loop that executes four times, prompt the user for a team member's name. Accept the name and assign it to the `BowlingTeam` object using the subscript to indicate the team member's position in the array in the `BowlingTeam` class.

```
for(x = 0; x < NUM_TEAM_MEMBERS; ++x)
{
   System.out.print("Enter team member's name >> ");
   name = input.nextLine();
   bowlTeam.setMember(x, name);
}
```

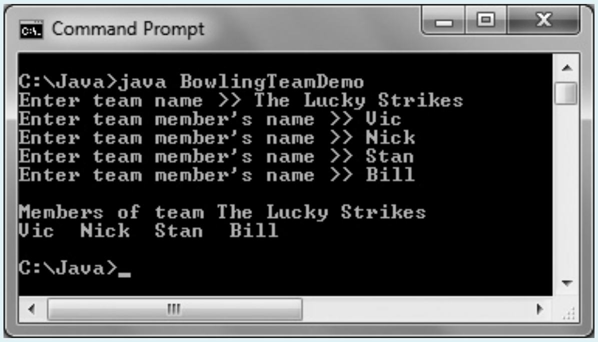
5. Display the details of the `BowlingTeam` object using the following code:

```
System.out.println("\nMembers of team " +
   bowlTeam.getTeamName());
for(x = 0; x < NUM_TEAM_MEMBERS; ++x)
   System.out.print(bowlTeam.getMember(x) + "  ");
System.out.println();
```

(continues)

(continued)

6. Add a closing curly brace for the `main()` method and another for the class.
7. Save the file as **BowlingTeamDemo.java**, and then compile and execute it. Figure 8-7 shows a typical execution.



```
C:\Java>java BowlingTeamDemo
Enter team name >> The Lucky Strikes
Enter team member's name >> Vic
Enter team member's name >> Nick
Enter team member's name >> Stan
Enter team member's name >> Bill

Members of team The Lucky Strikes
Vic   Nick   Stan   Bill

C:\Java>_
```

Figure 8-7 Typical execution of the `BowlingTeamDemo` class

Creating a Program That Declares an Array of `BowlingTeam` Objects

Next, you create and use an array of `BowlingTeam` objects.

1. Open the **BowlingTeamDemo.java** file. Rename the class **`BowlingTeamDemo2`**, and immediately save the file as **BowlingTeamDemo2.java**.
2. Above the declaration of the `BowlingTeam` object, add a new named constant that holds a number of `BowlingTeams`, and then replace the statement that declares the single `BowlingTeam` object with an array declaration of four `BowlingTeam` objects.

   ```
   final int NUM_TEAMS = 4;
   BowlingTeam[] teams = new BowlingTeam[NUM_TEAMS];
   ```

3. The current program declares `x`, which is used as a subscript to display team member names. Now, following the declaration of `x`, add a variable that is used as a subscript to display the teams:

   ```
   int y;
   ```

4. Following the declaration of the `Scanner` object, and before the team name prompt, insert a `for` loop that executes as many times as there

(continues)

(continued)

are `BowlingTeams`. Add the opening curly brace, and within the loop, allocate memory for each array element:

```
for(y = 0; y < NUM_TEAMS; ++y)
{
   teams[y] = new BowlingTeam();
```

5. Delete the statement that uses the `setTeamName()` method with the single `bowlTeam` object. In its place, insert a statement that uses the method with one of the array elements:

```
    teams[y].setTeamName(name);
```

6. Within the first `for` loop controlled by `x`, delete the statement that uses the `setMember()` method with the single `bowlTeam` object. In its place, insert a statement that uses the method with one of the array elements:

```
teams[y].setMember(x, name);
```

7. Add a closing curly brace for the `for` loop controlled by the variable `y`.
8. Adjust the indentation of the program statements so that the program logic is easy to follow with the new nested loops. The nested loops that you just modified should look like the following 13 lines of code:

```
for(y = 0; y < NUM_TEAMS; ++y)
{
   teams[y] = new BowlingTeam();
   System.out.print("Enter team name >> ");
   name = input.nextLine();
   teams[y].setTeamName(name);
   for(x = 0; x < NUM_TEAM_MEMBERS; ++x)
   {
      System.out.print("Enter team member's name >> ");
      name = input.nextLine();
      teams[y].setMember(x, name);
   }
}
```

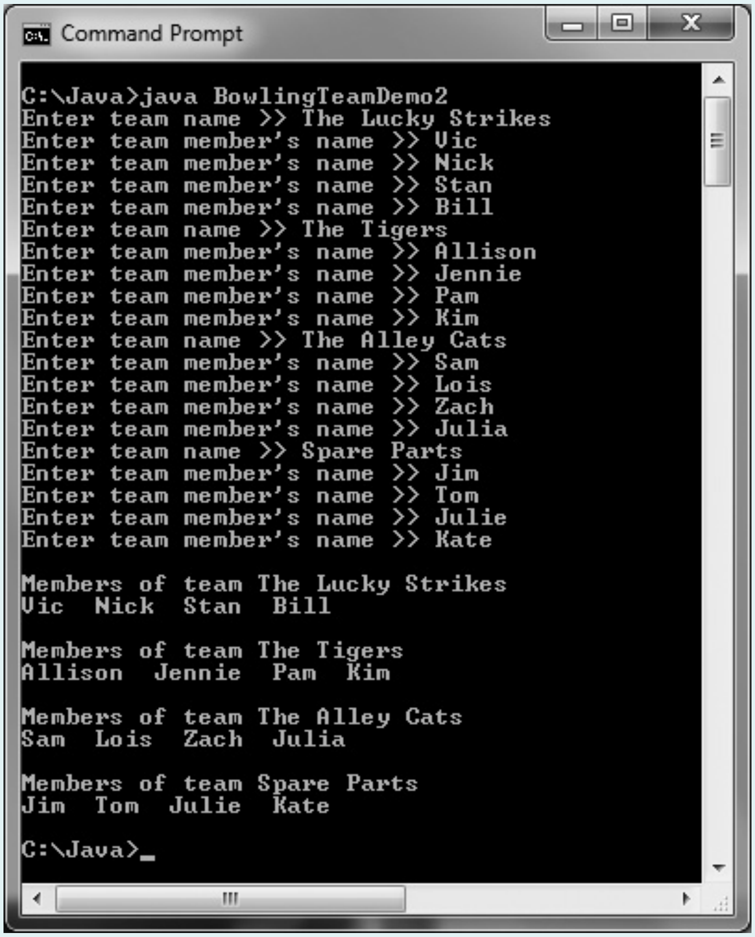
9. The `for` loop at the end of the current program lists four team members' names. Replace this loop with the following nested version that lists four members' names for each of four teams:

```
for(y = 0; y < NUM_TEAMS; ++y)
{
   System.out.println("\nMembers of team " +
      teams[y].getTeamName());
   for(x = 0; x < NUM_TEAM_MEMBERS; ++x)
      System.out.print(teams[y].getMember(x) + "  ");
   System.out.println();
}
```

(continues)

(continued)

10. Save the file, and then compile and execute the program. Figure 8-8 shows a typical execution. The user can enter data into the array of `BowlingTeam` objects, including the array of `Strings` within each object, and then see all the entered data successfully displayed.



```
C:\Java>java BowlingTeamDemo2
Enter team name >> The Lucky Strikes
Enter team member's name >> Vic
Enter team member's name >> Nick
Enter team member's name >> Stan
Enter team member's name >> Bill
Enter team name >> The Tigers
Enter team member's name >> Allison
Enter team member's name >> Jennie
Enter team member's name >> Pam
Enter team member's name >> Kim
Enter team name >> The Alley Cats
Enter team member's name >> Sam
Enter team member's name >> Lois
Enter team member's name >> Zach
Enter team member's name >> Julia
Enter team name >> Spare Parts
Enter team member's name >> Jim
Enter team member's name >> Tom
Enter team member's name >> Julie
Enter team member's name >> Kate

Members of team The Lucky Strikes
Vic  Nick  Stan  Bill

Members of team The Tigers
Allison  Jennie  Pam  Kim

Members of team The Alley Cats
Sam  Lois  Zach  Julia

Members of team Spare Parts
Jim  Tom  Julie  Kate

C:\Java>_
```

Figure 8-8 Typical execution of the `BowlingTeamDemo2` application

Searching an Array and Using Parallel Arrays

Suppose that a company manufactures 10 items. When a customer places an order for an item, you need to determine whether the item number on the order form is valid. When you want to determine whether a variable holds one of many valid values, one option is to use a series of `if` statements to compare the variable to a series of valid values. If valid item numbers are sequential, such as 101 through 110, the following simple `if` statement that uses a logical AND can verify the order number and set a Boolean field to `true`:

```
final int LOW = 101;
final int HIGH = 110;
boolean validItem = false;
if(itemOrdered >= LOW && itemOrdered <= HIGH)
   validItem = true;
```

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

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

Instead of a long series of `if` statements, a more elegant solution is to compare the `itemOrdered` variable to a list of values in an array, a process called **searching an array**. You can initialize the array with the valid values using the following statement, which creates exactly 10 array elements with subscripts 0 through 9:

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

After the list of valid values is initialized, 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])
      validItem = true;
}
```

This simple `for` loop replaces the long series of `if` statements; it checks the `itemOrdered` value against each of the 10 array values in turn. Also, if a company carries 1,000 items instead of 10, nothing changes in the `for` statement—the value of `validValues.length` is updated automatically.

Using Parallel Arrays

As an added bonus, if you set up another array with the same number of elements and corresponding data, you can use the same subscript to access additional information.

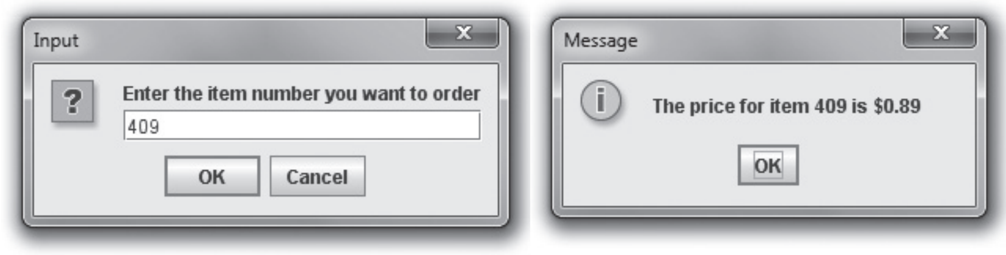
A **parallel array** is one with the same number of elements as another and for which the values in corresponding elements are related. For example, if the 10 items your company carries have 10 different prices, you can set up an array to hold those prices as follows:

```
double[] prices = {0.29, 1.23, 3.50, 0.69…};
```

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 application in Figure 8-9. In the shaded portion of the code, notice that when the ordered item's number is found in the `validValues` array, the `itemPrice` value is "pulled" from the `prices` array. In other words, if the item number is found in the second position in the `validValues` array, you can find the correct price in the second position in the `prices` array. Figure 8-10 shows a typical execution of the program. A user requests item 409, which is the eighth element in the `validValues` array, so the price displayed is the eighth element in the `prices` array.

```
import javax.swing.*;
public class FindPrice
{
   public static void main(String[] args)
   {
      final int NUMBER_OF_ITEMS = 10;
      int[] validValues = {101,  108,  201,  213,  266,
         304,  311,  409,  411,  412};
      double[] prices = {0.29,  1.23,  3.50,  0.69,  6.79,
         3.19,  0.99,  0.89,  1.26,  8.00};
      String strItem;
      int itemOrdered;
      double itemPrice = 0.0;
      boolean validItem = false;
      strItem = JOptionPane.showInputDialog(null,
         "Enter the item number you want to order");
      itemOrdered = Integer.parseInt(strItem);
      for(int x = 0; x < NUMBER_OF_ITEMS; ++x)
      {
         if(itemOrdered == validValues[x])
         {
            validItem = true;
            itemPrice = prices[x];
         }
      }
      if(validItem)
         JOptionPane.showMessageDialog(null, "The price for item " +
            itemOrdered + " is $" + itemPrice);
      else
         JOptionPane.showMessageDialog(null,
            "Sorry - invalid item entered");
    }
}
```

Figure 8-9 The `FindPrice` application that accesses information in parallel arrays


Input
Enter the item number you want to order
409

Message
The price for item 409 is $0.89

Figure 8-10 Typical execution of the `FindPrice` application

When you initialize parallel arrays, it is convenient to use spacing so that the values that correspond to each other visually align on the screen or printed page.

Instead of parallel arrays containing item numbers and prices, you might prefer to create a class named `Item` in which each instance contains two fields—`itemOrdered` and `itemPrice`. Then you could create a single array of objects that encapsulate item numbers and prices. There are almost always multiple ways to approach programming problems.

Within the code shown in Figure 8-9, you compare 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. On each of these nine additional cycles, the comparison between `itemOrdered` and `validValues[x]` is always `false`. As soon as a match for an `itemOrdered` is found, it is most efficient to break out of the `for` loop early. An easy way to accomplish this is to set `x` to a high value within the block of statements executed when there is a match. Then, after a match, the `for` loop does not execute again because the limiting comparison (`x < NUMBER_OF_ITEMS`) is surpassed. Figure 8-11 shows this loop. In an array with many possible matches, it is most efficient to place the more 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 place its price ($0.99) first in the `prices` array.

```
for(int x = 0; x < NUMBER_OF_ITEMS; ++x)
{
   if(itemOrdered == validValues[x])
   {
      validItem = true;
      itemPrice = prices[x];
      x = NUMBER_OF_ITEMS;
   }
}
```

Figure 8-11 A `for` loop with an early exit

In the code in Figure 8-11, the loop control variable is altered within the loop body. Some programmers object to altering a loop control variable within the body of a `for` loop; they feel that the loop control variable should be altered only in the third section of the `for` clause (where `x` is incremented). These programmers would prefer the loop in Figure 8-12, in which two Boolean expressions appear in the shaded section in the middle portion of the `for` clause. In this example, the loop control variable is not altered within the loop body. Instead, `x` must be within range before each iteration and `validItem` must not yet have been set to `true`.

```
for(int x = 0; x < NUMBER_OF_ITEMS && !validItem; ++x)
{
   if(itemOrdered == validValues[x])
   {
      validItem = true;
      itemPrice = prices[x];
   }
}
```

Figure 8-12 A `for` loop that uses a compound test for termination

Searching an Array for a Range Match

Searching an array for an exact match is not always practical. Suppose your company gives customer discounts based on the quantity of items ordered. Perhaps no discount is given for any order of fewer than a dozen items, but there are increasing discounts available for orders of increasing quantities, as shown in Table 8-1.

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

Table 8-1 Discount table

One awkward programming 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, as in the following example:

```
double[] discounts = {0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0.10, 0.10, 0.10 …};
```

Thirteen zeroes are listed in the `discounts` array. The first array element has a 0 subscript and represents a zero discount for zero items. The next 12 discounts (for items 1 through 12) are also zero. When `numOfItems` is 13, `discounts[numOfItems]`, or `discounts[13]`, is 0.10. The array would store 37 copies of 0.10 for elements 13 through 49. The `discounts` array would need to be ridiculously large to hold an exact value for each possible quantity ordered.

A better option is to create two corresponding arrays and perform a **range match**, in which you compare a value to the endpoints of numerical ranges to find the category in which a value belongs. For example, one array can hold the five discount rates, and the other array can hold five discount range limits. The Total Quantity Ordered column in Table 8-1 shows five ranges. If you use only the first figure in each range, you can create an array that holds five low limits:

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

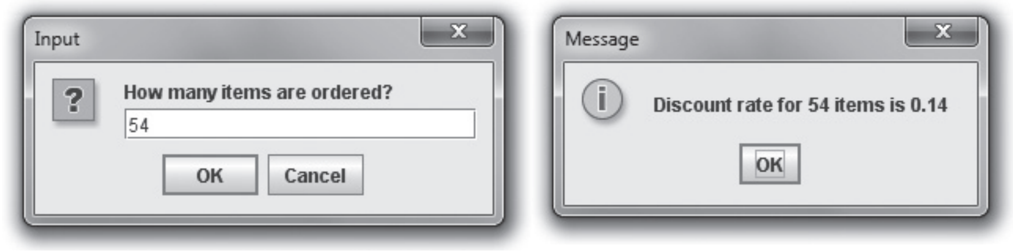
A parallel array can hold the five discount rates:

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

Then, starting at the last `discountRangeLimits` array element, for any `numOfItems` greater than or equal to `discountRangeLimits[4]`, the appropriate discount is `discounts[4]`. In other words, for any `numOrdered` less than `discountRangeLimits[4]`, you should decrement the subscript and look in a lower range. Figure 8-13 shows an application that uses the parallel arrays, and Figure 8-14 shows a typical execution of the program.

```
import javax.swing.*;
public class FindDiscount
{
   public static void main(String[] args)
   {
      final int NUM_RANGES = 5;
      int[] discountRangeLimits = {   1,   13,   50,  100,  200};
      double[] discountRates =    {0.00, 0.10, 0.14, 0.18, 0.20};
      double customerDiscount;
      String strNumOrdered;
      int numOrdered;
      int sub = NUM_RANGES - 1;
      strNumOrdered = JOptionPane.showInputDialog(null,
         "How many items are ordered?");
      numOrdered = Integer.parseInt(strNumOrdered);
      while(sub >= 0 && numOrdered < discountRangeLimits[sub])
         --sub;
      customerDiscount = discountRates[sub];
      JOptionPane.showMessageDialog(null, "Discount rate for " +
          numOrdered + " items is " + customerDiscount);
   }
}
```

Figure 8-13 The `FindDiscount` class



Input

How many items are ordered?

54

OK Cancel

Message

Discount rate for 54 items is 0.14

OK

Figure 8-14 Typical execution of the `FindDiscount` class

In the `while` loop in the application in Figure 8-13, `sub` is required to be greater than or equal to 0 before the second half of the statement that compares `numOrdered` to `discountRangeLimits[sub]` executes. It is a good programming practice to ensure that a subscript to an array does not fall below zero, causing a runtime error.

Watch the video *Searching an Array*.

TWO TRUTHS & A LIE

Searching an Array and Using Parallel Arrays

1. A parallel array is one with the same number of elements as another and for which the values in corresponding elements are related.
2. When searching an array, it is usually most efficient to abandon the search as soon as the sought-after element is found.
3. In a range match, you commonly compare a value to the midpoint of each of a series of numerical ranges.

The false statement is #3. In a range match, you commonly compare a value to the low or high endpoint of each of a series of numerical ranges but not to the midpoint.

You Do It

Searching an Array

In this section, you modify the `BowlingTeamDemo2` program so that after the bowling team data has been entered, a user can request the roster for a specific team.

1. Open the **BowlingTeamDemo2.java** file, and change the class name to **`BowlingTeamDemo3`**. Immediately save the file as **BowlingTeamDemo3.java**.
2. At the end of the existing application, just before the two final closing curly braces, insert a prompt asking the user to enter a team name. Then accept the entered value.

```
System.out.print("\n\nEnter a team name to see its roster >> ");
name = input.nextLine();
```

3. Next, insert a nested `for` loop. The outer loop varies `y` from 0 through the highest subscript allowed in the `teams` array. Within this loop, the team name requested by the user is compared to each stored team name; when they are equal, another `for` loop displays the four team member names.

```
for(y = 0; y < teams.length; ++y)
   if(name.equals(teams[y].getTeamName()))
      for(x = 0; x < NUM_TEAM_MEMBERS; ++x)
         System.out.print(teams[y].getMember(x) + " ");
```

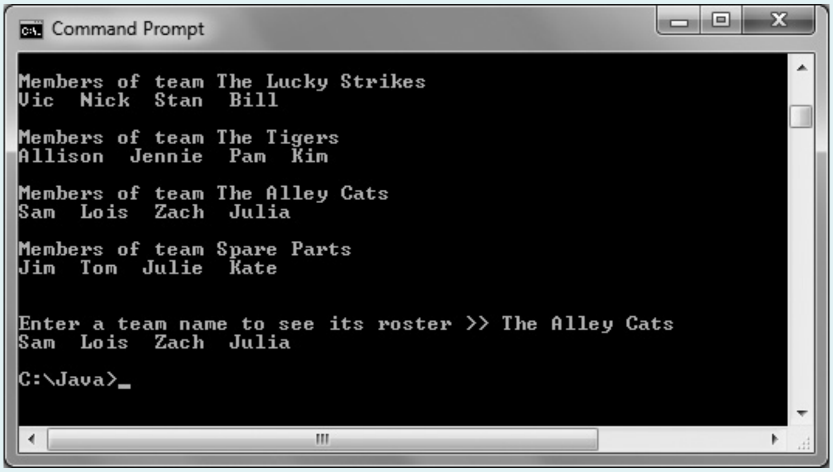
4. Insert an additional empty `println()` method call.

```
System.out.println();
```

5. Save the file, and then compile and execute the program. Figure 8-15 shows the last part of a typical execution.

(continues)

(continued)



```
Members of team The Lucky Strikes
Vic  Nick  Stan  Bill

Members of team The Tigers
Allison  Jennie  Pam  Kim

Members of team The Alley Cats
Sam  Lois  Zach  Julia

Members of team Spare Parts
Jim  Tom  Julie  Kate


Enter a team name to see its roster >> The Alley Cats
Sam  Lois  Zach  Julia

C:\Java>_
```

Figure 8-15 Typical execution of the `BowlingTeamDemo3` application

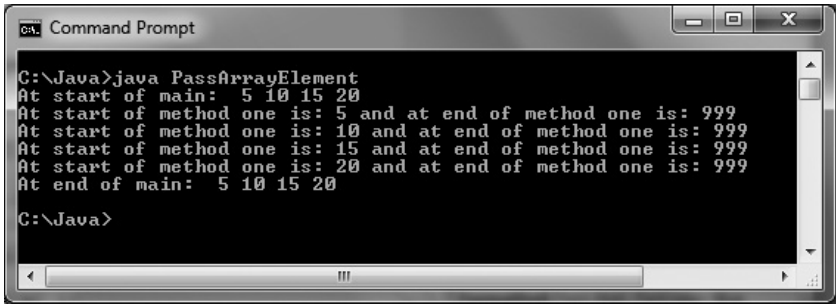
Passing Arrays to and Returning Arrays from Methods

You have already seen that you can use any individual array element in the same manner as you use any single variable of the same type. That is, if you declare an integer array as `int[] someNums = new int[12];`, you can subsequently display `someNums[0]`, or increment `someNums[1]`, or work with any element just as you do for any integer. Similarly, you can pass a single array element to a method in exactly the same manner as you pass a variable.

Examine the `PassArrayElement` application shown in Figure 8-16 and the output shown in Figure 8-17. The application creates an array of four integers and displays them. Then, the application calls the `methodGetsOneInt()` method four times, passing each element in turn. The method displays the number, changes the number to 999, and then displays the number again. Finally, back in the `main()` method, the four numbers are displayed again.

```
public class PassArrayElement
{
   public static void main(String[] args)
   {
      final int NUM_ELEMENTS = 4;
      int[] someNums = {5, 10, 15, 20};
      int x;
      System.out.print("At start of main: ");
      for(x = 0; x < NUM_ELEMENTS; ++x)
         System.out.print(" " + someNums[x] );
      System.out.println();
      for(x = 0; x < NUM_ELEMENTS; ++x)
         methodGetsOneInt(someNums[x]);
      System.out.print("At end of main: ");
      for(x = 0; x < NUM_ELEMENTS; ++x)
         System.out.print(" " + someNums[x]);
      System.out.println();
   }
   public static void methodGetsOneInt(int one)
   {
      System.out.print("At start of method one is: " + one);
      one = 999;
      System.out.println(" and at end of method one is: " + one);
   }
}
```

Figure 8-16 The `PassArrayElement` class



```
C:\Java>java PassArrayElement
At start of main:  5 10 15 20
At start of method one is: 5 and at end of method one is: 999
At start of method one is: 10 and at end of method one is: 999
At start of method one is: 15 and at end of method one is: 999
At start of method one is: 20 and at end of method one is: 999
At end of main:  5 10 15 20

C:\Java>
```

Figure 8-17 Output of the `PassArrayElement` application

As you can see in Figure 8-17, the four numbers that were changed in the `methodGetsOneInt()` method remain unchanged back in `main()` after the method executes. The variable named `one` is local to the `methodGetsOneInt()` method, and any changes to variables passed into the method are not permanent and are not reflected in the array in the `main()` program. Each variable named `one` in the `methodGetsOneInt()` method holds only a copy of the array element passed into the method. The individual array elements are **passed by value**;

that is, a copy of the value is made and used within the receiving method. When any primitive type (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, or `double`) is passed to a method, the value is passed.

Arrays, like all nonprimitive objects, are **reference types**; this means that the object actually holds a memory address where the values are stored. Because an array name is a reference, you cannot assign another array to it using the = operator, nor can you compare two arrays using the == operator. Additionally, when you pass an array (that is, pass its name) to a method, the receiving method gets a copy of the array's actual memory address. This means that the receiving method has access to, and the ability to alter, the original values in the array elements in the calling method.

The class shown in Figure 8-18 creates an array of four integers. After the integers are displayed, the array name (its address) is passed to a method named `methodGetsArray()`. Within the method, the numbers are displayed, which shows that they retain their values from `main()`, but then the value 888 is assigned to each number. Even though `methodGetsArray()` is a `void` method—meaning nothing is returned to the `main()` method—when the `main()` method displays the array for the second time, all of the values have been changed to 888, as you can see in the output in Figure 8-19. Because the method receives a reference to the array, the `methodGetsArray()` method "knows" the address of the array declared in `main()` and makes its changes directly to the original array.

In some languages, arrays are **passed by reference**, meaning that a receiving method gets the memory address. It is a subtle distinction, but in Java, the receiving method gets a copy of the original address. In other words, in Java, an array is not passed by reference, but a reference to an array is passed by value.

```java
public class PassArray
{
   public static void main(String[] args)
   {
      final int NUM_ELEMENTS = 4;
      int[] someNums = {5, 10, 15, 20};
      int x;
      System.out.print("At start of main: ");
      for(x = 0; x < NUM_ELEMENTS; ++x)
         System.out.print(" " + someNums[x] );
      System.out.println();
      methodGetsArray(someNums);
      System.out.print("At end of main: ");
      for(x = 0; x < NUM_ELEMENTS; ++x)
         System.out.print(" " + someNums[x]);
      System.out.println();
   }
```

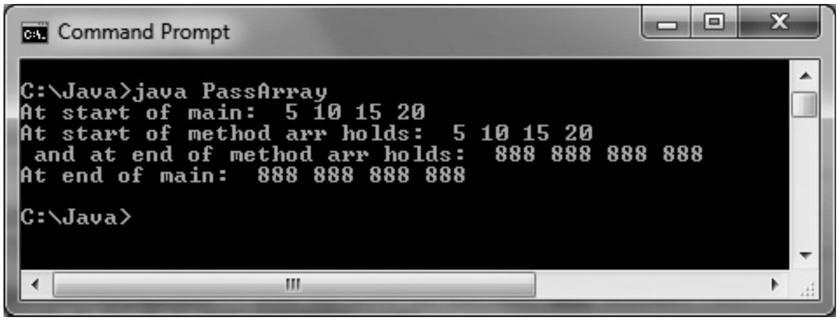
Figure 8-18 The `PassArray` class *(continues)*

(continued)

```
   public static void methodGetsArray(int[] arr)
   {
      int x;
      System.out.print("At start of method arr holds: ");
      for(x = 0; x < arr.length; ++x)
         System.out.print(" " + arr[x] );
      System.out.println();
      for(x = 0; x < arr.length; ++x)
         arr[x] = 888;
      System.out.print(" and at end of method arr holds: ");
      for(x = 0; x < arr.length; ++x)
         System.out.print(" " + arr[x] );
      System.out.println();
   }
}
```

Figure 8-18 The `PassArray` class

Notice that in the first shaded statement in Figure 8-18, the array name is passed to the method and no brackets are used. In the method header, brackets are used to show that the parameter is an array of integers (a reference) and not a simple `int`.



```
C:\Java>java PassArray
At start of main:  5 10 15 20
At start of method arr holds:  5 10 15 20
 and at end of method arr holds:  888 888 888 888
At end of main:  888 888 888 888

C:\Java>
```

Figure 8-19 Output of the `PassArray` application

In some other languages, notably C, C++, and C#, you can choose to pass variables to methods by value or reference. In Java, you cannot make this choice. Primitive type variables are always passed by value. When you pass an object, a copy of the reference to the object is always passed.

Returning an Array from a Method

A method can return an array reference. When a method returns an array reference, you include square brackets with the return type in the method header. For example, Figure 8-20 shows a `getArray()` method that returns a locally declared array of `int`s. Square brackets are used as part of the return type; the `return` statement returns the array name without any brackets.

```
public static int[] getArray()
{
   int[] scores = {90, 80, 70, 60};
   return scores;
}
```

Figure 8-20 The `getArray()` method

When you call the `getArray()` method in Figure 8-20, you can store its returned value in any integer array reference. For example, you might declare an array and make the method call in the following statement:

```
int[] scoresFromMethod = getArray();
```

Watch the video *Arrays and Methods*.

TWO TRUTHS & A LIE

Passing Arrays to and Returning Arrays from Methods

1. You pass a single array element to a method using its name, and the method must be prepared to receive the appropriate data type.
2. You pass an array to a method using its name followed by a pair of brackets; arrays are passed by value.
3. When a method returns an array reference, you include square brackets with the return type in the method header.

The false statement is #2. You pass an array to a method using its name; a copy of the array's address is passed to the method.

You Do It

Passing an Array to a Method

Next, you add a method to the `BowlingTeamDemo3` application. The improvement allows you to remove the data entry process from the main program and encapsulate the process in its own method.

1. In your text editor, open the **BowlingTeamDemo3.java** file if it is not already open. Immediately save the file as **BowlingTeamDemo4.java**. Change the class name to match the filename.

2. Just before the closing curly brace for the class, add the following shell for a method that accepts a `BowlingTeam` array argument.

```
public static void getTeamData(BowlingTeam[] teams)
{
}
```

3. Within the `getTeamData()` method, add the following six declarations (or copy them from the `main()` method):

```
String name;
final int NUM_TEAMS = 4;
int x;
int y;
final int NUM_TEAM_MEMBERS = 4;
Scanner input = new Scanner(System.in);
```

4. Cut the 13 lines of code that assign memory to the `BowlingTeam` array and obtain all the data values. Place these 13 lines within the `getTeamData()` method following the declarations.

```
for(y = 0; y < NUM_TEAMS; ++y)
{
   teams[y] = new BowlingTeam();
   System.out.print("Enter team name >> ");
   name = input.nextLine();
   teams[y].setTeamName(name);
   for(x = 0; x < NUM_TEAM_MEMBERS; ++x)
   {
      System.out.print("Enter team member's name >> ");
      name = input.nextLine();
      teams[y].setMember(x, name);
   }
}
```

(continues)

(continued)

5. In place of the 13 cut lines, insert a method call. This call passes a copy of the array reference to the method. Notice that this call does not assign a return value. The method is a `void` method and returns nothing. Nevertheless, the array in the `main()` method will be updated because the method is receiving access to the array's memory address.

```
getTeamData(teams);
```

6. Save the file (as **BowlingTeamDemo4.java**), and then compile and execute the program. Confirm that the program works exactly as it did before the new method was added.

Don't Do It

- Don't forget that the lowest array subscript is 0.
- Don't forget that the highest array subscript is one less than the length of the array.
- Don't forget the semicolon following the closing curly brace in an array initialization list.
- Don't forget that `length` is an array property and not a method. Conversely, `length()` is a `String` method and not a property.
- Don't place a subscript after an object's field or method name when accessing an array of objects. Instead, the subscript for an object follows the object and comes before the dot and the field or method name.
- Don't assume that an array of characters is a string. Although an array of characters can be treated like a string in languages like C++, you can't do this in Java. For example, if you display the name of a character array, you will see its address, not its contents.
- Don't forget that array names are references. Therefore, you cannot assign one array to another using the = operator, nor can you compare array contents using the == operator.
- Don't use brackets with an array name when you pass it to a method. Do use brackets in the method header that accepts the array.

Key Terms

An **array** is a named list of data items that all have the same type.

An **element** is one variable or object in an array.

A **subscript** is an integer contained within square brackets that indicates one of an array's elements.

An **index** is a subscript.

Out of bounds describes a subscript that is not within the allowed range for an array.

An **initialization list** is a series of values provided for an array when it is declared.

Populating an array is the act of providing values for all the elements.

The **`length` field** contains the number of elements in an array.

An object's instance variable or field is also called a **property** of the object.

The **enhanced `for` loop** allows you to cycle through an array without specifying the starting and ending points for the loop control variable.

A **`foreach` loop** is an enhanced `for` loop.

Searching an array is the process of comparing a value to a list of values in an array, looking for a match.

A **parallel array** is one with the same number of elements as another and for which the values in corresponding elements are related.

A **range match** is the process of comparing a value to the endpoints of numerical ranges to find a category in which the value belongs.

When a variable is **passed by value** to a method, a copy is made in the receiving method.

Arrays are **reference types**, meaning that the object actually holds a memory address where the values are stored.

When a value is **passed by reference** to a method, the address is passed to the method.

Chapter Summary

- An array is a named list of data items that all have the same type. You declare an array variable by inserting a pair of square brackets after the type. To reserve memory space for an array, you use the keyword `new`. You use a subscript contained within square brackets to refer to one of an array's variables, or elements. In Java, any array's elements are numbered beginning with zero.
- Array names represent computer memory addresses. When you declare an array name, no computer memory address is assigned to it. Instead, the array variable name has the special value `null`. When you use the keyword `new` or supply an initialization list, then an array acquires an actual memory address. When an initialization list is not provided, each data type has a default value for its array elements.

- You can shorten many array-based tasks by using a variable as a subscript. When an application contains an array, it is common to perform loops that execute from 0 to one less than the size of the array. The `length` field is an automatically created field that is assigned to every array; it contains the number of elements in the array.
- Just as you can declare arrays of integers or `double`s, you can declare arrays that hold elements of any type, including `String`s and other objects. To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot that precedes the method name.
- By looping through an array and making comparisons, you can search an array to find a match to a value. You can use a parallel array with the same number of elements to hold related elements. You perform a range match by placing end values of a numeric range in an array and making greater-than or less-than comparisons.
- You can pass a single array element to a method in exactly the same manner as you would pass a simple variable, and the array receives a copy of the passed value. However, arrays, like all objects, are reference types; this means that when an array name is passed to a method, the method receives a copy of the array's memory address and has access to the values in the original array.

Review Questions

1. An array is a list of data items that ____________.
   a. all have the same type
   b. all have different names
   c. all are integers
   d. all are `null`

2. When you declare an array, ____________.
   a. you always reserve memory for it in the same statement
   b. you might reserve memory for it in the same statement
   c. you cannot reserve memory for it in the same statement
   d. the ability to reserve memory for it in the same statement depends on the type of the array

3. You reserve memory locations for an array when you ____________.
   a. declare the array name
   b. use the keyword `new`
   c. use the keyword `mem`
   d. explicitly store values within the array elements

4. For how many integers does the following statement reserve room?

```
int[] value = new int[34];
```

a. 0
b. 33
c. 34
d. 35

5. Which of the following can be used as an array subscript?

a. character
b. `double`
c. `int`
d. `String`

6. If you declare an array as follows, how do you indicate the final element of the array?

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

a. `num[0]`
b. `num[5]`
c. `num[6]`
d. impossible to tell

7. If you declare an integer array as follows, what is the value of `num[2]`?

```
int[] num = {101, 202, 303, 404, 505, 606};
```

a. 101
b. 202
c. 303
d. impossible to tell

8. Array names represent ____________.

a. values
b. functions
c. references
d. allusions

9. Unicode value '\u0000' is also known as ____________.

a. `nil`
b. `void`
c. `nada`
d. `null`

10. When you initialize an array by giving it values upon creation, you ____________.

a. do not explicitly give the array a size
b. also must give the array a size explicitly
c. must make all the values zero, blank, or `false`
d. must make certain each value is different from the others

11. In Java, you can declare an array of 12 elements and initialize ____________.

a. only the first one
b. all of them
c. Both of these are true.
d. Neither of these is true.

12. Assume an array is declared as follows. Which of the following statements correctly assigns the value 100 to each of the array elements?

    `int[] num = new int[4];`

    a. `for(x = 0; x < 3; ++x) num[x] = 100;`
    b. `for(x = 0; x < 4; ++x) num[x] = 100;`
    c. `for(x = 1; x < 4; ++x) num[x] = 100;`
    d. `for(x = 1; x < 5; ++x) num[x] = 100;`

13. Suppose you have declared an array as follows:

    `int[] creditScores = {670, 720, 815};`

    What is the value of `creditScores.length`?

    a. `0`
    b. `1`
    c. `2`
    d. `3`

14. If a class named `Student` contains a method `setID()` that takes an `int` argument, and you write an application in which you create an array of 20 `Student` objects named `scholar`, which of the following statements correctly assigns an ID number to the first `Student scholar`?

    a. `Student[0].setID(1234);`
    b. `scholar[0].setID(1234);`
    c. `Student.setID[0](1234);`
    d. `scholar.setID[0](1234);`

15. A parallel array is one that ____________.

    a. holds values that correspond to those in another array
    b. holds an even number of values
    c. is placed adjacent to another array in code
    d. is placed adjacent to another array in memory

16. In which of the following situations would setting up parallel arrays be most useful?

    a. You need to look up an employee's ID number to find the employee's last name.
    b. You need to calculate interest earned on a savings account balance.
    c. You need to store a list of 20 commonly misspelled words.
    d. You need to determine the shortest distance between two points on a map.

17. When you pass an array element to a method, the method receives ____________.

    a. a copy of the array
    b. the address of the array
    c. a copy of the value in the element
    d. the address of the element

18. A single array element of a primitive type is passed to a method by ____________.

    a. value
    b. reference
    c. address
    d. osmosis

19. When you pass an array to a method, the method receives ____________.

    a. a copy of the array
    b. a copy of the first element in the array
    c. the address of the array
    d. nothing

20. If a method should return an array to its calling method, ____________.

    a. the method's return type must match its parameter type
    b. the return type in the method header is preceded by an ampersand
    c. the return type in the method header is followed by square brackets
    d. A Java method cannot return an array.

Exercises

Programming Exercises

1. Write an application that can hold eight integers in an array. Display the integers from first to last, and then display the integers from last to first. Save the file as **EightInts.java**.

2. Allow a user to enter any number of `double` values up to 10. The user should enter 99999 to quit entering numbers. Display an error message if the user quits without entering any numbers; otherwise, display each entered value and its distance from the average. Save the file as **DistanceFromAverage.java**.

3. a. Write an application for Carl's Carpentry that shows a user a list of available items: table, desk, dresser, or entertainment center. Allow the user to enter a string that corresponds to one of the options, and display the price as $250, $325, $420, or $600, accordingly. Display an error message if the user enters an invalid item. Save the file as **CarpentryChoice.java**.

   b. It might not be reasonable to expect users to type long entries such as "entertainment center" accurately. Modify the `CarpentryChoice` class so that as long as the user enters the first three characters of a furniture item, the choice is considered valid. Save the file as **CarpentryChoice2.java**.

4. Create an application containing an array that stores eight integers. The application should (1) display all the integers, (2) display all the integers in reverse order, (3) display the sum of the eight integers, (4) display all values less than 5, (5) display the lowest value, (6) display the highest value, (7) calculate and display the average, and (8) display all values that are higher than the calculated average value. Save the file as **NumberListDemo.java**.

5. Write an application that accepts up to 20 `Strings`. Divide them into two lists—one for short `Strings` that are five characters or fewer, and the other for long `Strings`. After data entry is complete, prompt the user to enter which type of `String` to display, and then output the correct list. If there are no `Strings` in a requested list, then output an appropriate message. Save the file as **DivideStrings.java**.

6. a. Create a class named `Salesperson`. Data fields for `Salesperson` include an integer ID number and a `double` annual sales amount. Methods include a constructor that requires values for both data fields, as well as get and set methods for each of the data fields. Write an application named `DemoSalesperson` that declares an array of 10 `Salesperson` objects. Set each ID number to 9999 and each sales value to zero. Display the 10 `Salesperson` objects. Save the files as **Salesperson.java** and **DemoSalesperson.java**.

   b. Modify the `DemoSalesperson` application so each `Salesperson` has a successive ID number from 111 through 120 and a sales value that ranges from $25,000 to $70,000, increasing by $5,000 for each successive `Salesperson`. Save the file as **DemoSalesperson2.java**.

7. a. Create a `CollegeCourse` class. The class contains fields for the course ID (for example, "CIS 210"), credit hours (for example, 3), and a letter grade (for example, 'A'). Include get and set methods for each field. Create a `Student` class containing an ID number and an array of five `CollegeCourse` objects. Create a `get()` and `set()` method for the `Student` ID number. Also create a `get()` method that returns one of the `Student`'s `CollegeCourses`; the method takes an integer argument and returns the `CollegeCourse` in that position (0 through 4). Next, create a `set()` method that sets the value of one of the `Student`'s `CollegeCourses`; the method takes two arguments—a `CollegeCourse` and an integer representing the `CollegeCourse`'s position (0 through 4). Save the files as **CollegeCourse.java** and **Student.java**.

   b. Write an application that prompts a professor to enter grades for five different courses each for 10 students. Prompt the professor to enter data for one student at a time, including student ID and course data for five courses. Use prompts containing the number of the student whose data is being entered and the course number—for example, "Enter ID for student #s", where *s* is an integer from 1 through 10, indicating the student, and "Enter course ID #n", where *n* is an integer from 1 through 5, indicating the course number. Verify that the professor enters only A, B, C, D, or F for the grade value for each course. Save the file as **InputGrades.java**.

8. Write an application in which the user can enter a date using digits and slashes (for example, "6/24/2014"), and receive output that displays the date with the month shown as a word (such as "June 24, 2014"). Allow for the fact that the user might or might not precede a month or day number with a zero (for example, the user might type "06/24/2014" or "6/24/2014"). Do not allow the user to enter an invalid date, defined as one for which the month is less than 1 or more than 12, or one for which the day number is less than 1 or greater than the number of days in the specified month. Also display the date's ordinal position in the year; for example, 6/24/14 is the 175th day of the year. In this application, use your knowledge of arrays to store the month names, as well as values for the number of days in each month so that you can calculate the number of days that have passed. Save the application as **ConvertDate.java**.

When determining whether a date is valid and when calculating the number of days that have passed, remember that some years are leap years. In a leap year, February 29th is added to the calendar. A leap year is any year that is evenly divisible by 4, unless the year is also evenly divisible by 100. So 1908 and 2008 were both leap years, but 1900 was not a leap year. Another exception occurs when a year is evenly divisible by 400—the year is a leap year. Therefore, 2000 was a leap year, but 2100 will not be one.

9. Write an application that allows a user to enter the names and phone numbers of up to 20 friends. Continue to prompt the user for names and phone numbers until the user enters "zzz" or has entered 20 names, whichever comes first. When the user is finished entering names, produce a count of how many names were entered, but make certain not to count the application-ending dummy "zzz" entry. Then display the names. Ask the user to type one of the names and display the corresponding phone number. Save the application as **PhoneBook.java**.

10. Write an application containing three parallel arrays that hold 10 elements each. The first array holds four-digit student ID numbers, the second holds first names, and the third holds the students' grade point averages. Use dialog boxes to accept a student ID number and display the student's first name and grade point average. If a match is not found, display an error message that includes the invalid ID number and allow the user to search for a new ID number. Save the file as **StudentIDArray.java**.

11. A personal phone directory contains room for first names and phone numbers for 30 people. Assign names and phone numbers for the first 10 people. Prompt the user for a name, and if the name is found in the list, display the corresponding phone number. If the name is not found in the list, prompt the user for a phone number, and add the new name and phone number to the list. Continue to prompt the user for names until the user enters "quit". After the arrays are full (containing 30 names), do not allow the user to add new entries. Save the file as **PhoneNumbers.java**.

12. In the Exercises in Chapter 4, you created a `CertificateOfDeposit` class and a `TestCertificateOfDeposit` application that instantiated two `CertificateOfDeposit` objects. Now, modify the application to accept data for an array of five `CertificateOfDeposit` objects, and then display the data. Save the application as **TestCertificateOfDepositArray.java**.

13. In the Exercises in Chapter 5, you created a `DigitalCamera` class and a `TestDigitalCamera` application that accepted and displayed data for four `DigitalCamera` objects. Now, modify the `TestDigitalCamera` application to use an array of four `DigitalCamera` objects instead of four individual ones. Save the application as **TestDigitalCameraArray.java**.

Debugging Exercises

1. Each of the following files in the Chapter08 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugEight1.java will become FixDebugEight1.java.

   a. DebugEight1.java
   b. DebugEight2.java
   c. DebugEight3.java
   d. DebugEight4.java

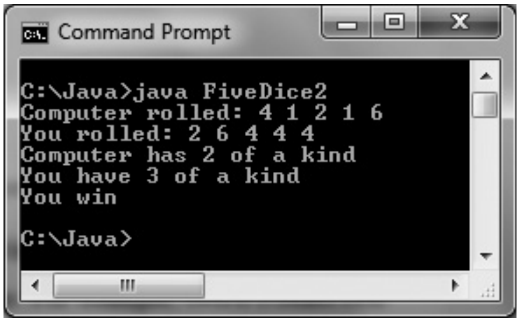
Game Zone

1. Write an application that contains an array of 10 multiple-choice quiz questions related to your favorite hobby. Each question contains three answer choices. Also create an array that holds the correct answer to each question—A, B, or C. Display each question and verify that the user enters only A, B, or C as the answer—if not, keep prompting the user until a valid response is entered. If the user responds to a question correctly, display "Correct!"; otherwise, display "The correct answer is" and the letter of the correct answer. After the user answers all the questions, display the number of correct and incorrect answers. Save the file as **Quiz.java**.

2. a. In Chapter 4, you created a `Die` application that randomly "throws" five dice for the computer and five dice for the player. The application displays the values. Modify the application to decide the winner based on the following hierarchy of `Die` values. Any higher combination beats a lower one; for example, five of a kind beats four of a kind.

   - Five of a kind
   - Four of a kind
   - Three of a kind
   - A pair

   For this game, the dice values do not count; for example, if both players have three of a kind, it's a tie, no matter what the values of the three dice are. Additionally, the game does not recognize a full house (three of a kind plus two of a kind). Figure 8-21 shows a sample execution. Save the application as **FiveDice2.java**.

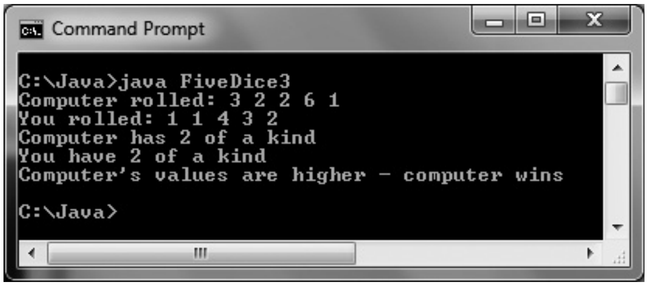


```
C:\Java>java FiveDice2
Computer rolled: 4 1 2 1 6
You rolled: 2 6 4 4 4
Computer has 2 of a kind
You have 3 of a kind
You win

C:\Java>
```

Figure 8-21 Typical execution of the `FiveDice2` application

b. Improve the `FiveDice2` game so that when both players have the same combination of dice, the higher value wins. For example, two 6s beats two 5s. Figure 8-22 shows an example execution. Save the application as **FiveDice3.java**.



```
C:\Java>java FiveDice3
Computer rolled: 3 2 2 6 1
You rolled: 1 1 4 3 2
Computer has 2 of a kind
You have 2 of a kind
Computer's values are higher - computer wins

C:\Java>
```

Figure 8-22 Typical execution of the `FiveDice3` application

3. a. In Chapter 7, you modified a previously created `Card` class so that each `Card` would hold the name of a suit ("Spades", "Hearts", "Diamonds", or "Clubs") as well as a value ("Ace", "King", "Queen", "Jack", or a number value). Now, create an array of 52 `Card` objects, assigning a different value to each `Card`, and display each `Card`. Save the application as **FullDeck.java**.

b. In Chapter 7, you created a `War2` card game that randomly selects two `Card` objects (one for the player and one for the computer) and declares a winner or a tie based on the card values. Now create a game that plays 26 rounds of War, dealing a full deck with no repeated cards. Some hints:

- Start by creating an array of all 52 playing cards, as in Part a of this exercise.
- Select a random number for the deck position of the player's first card, and assign the card at that array position to the player.

- Move every higher-positioned card in the deck "down" one to fill in the gap. In other words, if the player's first random number is 49, select the card at position 49, move the card that was in position 50 to position 49, and move the card that was in position 51 to position 50. Only 51 cards remain in the deck after the player's first card is dealt, so the available-card array is smaller by one.
- In the same way, randomly select a card for the computer and "remove" the card from the deck.
- Display the values of the player's and computer's cards, compare their values, and determine the winner.
- When all the cards in the deck are exhausted, display a count of the number of times the player wins, the number of times the computer wins, and the number of ties.

Save the game as **War3.java**.

4. In Chapter 7, you created a Secret Phrase game similar to Hangman, in which the user guesses letters in a partially hidden phrase in an attempt to determine the complete phrase. Modify the program so that:

   - The phrase to be guessed is selected randomly from a list of at least 10 phrases.
   - The clue is presented to the user with asterisks replacing letters to be guessed but with spaces in the appropriate locations. For example, if the phrase to be guessed is "No man is an island," then the user sees the following as a first clue:

     ```
     ** *** ** ** ******
     ```

     The spaces provide valuable clues as to where individual words start and end.
   - Make sure that when a user makes a correct guess, all the matching letters are filled in, regardless of case.

   Save the game as **SecretPhrase2.java**.

Case Problems

1. In previous chapters, you developed classes that work with catering event information for Carly's Catering. Now modify the `Event` and `EventDemo` classes as follows:

   - Modify the `Event` class to include an integer field that holds an event type. Add a `final String` array that holds names of the types of events that Carly's caters—wedding, baptism, birthday, corporate, and other. Include get and set methods for the integer event type field. If the argument passed to the method that sets the event type is larger than the size of the array of `String` event types, then set the integer to the element number occupied by "other". Include a get method that returns an event's `String` event type based on the numeric event type.

- To keep the `EventDemo` class simple, remove all the statements that compare event sizes and that display the invitation `String`s.
- Modify the `EventDemo` class so that instead of creating three single `Event` objects, it uses an array of three `Event` objects. Get data for each of the objects, and then display all the details for each object.

Save the files as **Event.java** and **EventDemo.java**.

2. In previous chapters, you developed classes that hold rental contract information for Sammy's Seashore Supplies. Now modify the `Rental` and `RentalDemo` classes as follows:

- Modify the `Rental` class to include an integer field that holds an equipment type. Add a `final String` array that holds names of the types of equipment that Sammy's rents—jet ski, pontoon boat, rowboat, canoe, kayak, beach chair, umbrella, and other. Include get and set methods for the integer equipment type field. If the argument passed to the method that sets the equipment type is larger than the size of the array of `String` equipment types, then set the integer to the element number occupied by "other". Include a get method that returns a rental's `String` equipment type based on the numeric equipment type.
- To keep the `RentalDemo` class simple, remove all the statements that compare rental times and that display the coupon `String`s.
- Modify the `RentalDemo` class so that instead of creating three single `Rental` objects, it uses an array of three `Rental` objects. Get data for each of the objects, and then display all the details for each object.

Save the files as **Rental.java** and **RentalDemo.java**.