CHAPTER 9

Advanced Array Concepts

In this chapter, you will:

- Sort array elements using the bubble sort algorithm
- Sort array elements using the insertion sort algorithm
- Use two-dimensional and other multidimensional arrays
- Use the `Arrays` class
- Use the `ArrayList` class
- Create enumerations

Sorting Array Elements Using the Bubble Sort Algorithm

Sorting is the process of arranging a series of objects in some logical order. When you place objects in order, beginning with the object that has the lowest value, you are sorting in **ascending order**; conversely, when you start with the object that has the largest value, you are sorting in **descending order**.

The simplest possible sort involves two values that are out of order. To place the values in order, you must swap the two values. Suppose that you have two variables—`valA` and `valB`—and further suppose that `valA = 16` and `valB = 2`. To exchange the values of the two variables, you cannot simply use the following code:

```
valA = valB; // 2 goes to valA
valB = valA; // 2 goes to valB
```

If `valB` is 2, after you execute `valA = valB;`, both variables hold the value 2. The value 16 that was held in `valA` is lost. When you execute the second assignment statement, `valB = valA;`, each variable still holds the value 2.

The solution that allows you to retain both values is to employ a variable to hold `valA`'s value temporarily during the swap:

```
temp = valA; // 16 goes to temp
valA = valB; // 2 goes to valA
valB = temp; // 16 goes to valB
```

Using this technique, `valA`'s value (16) is assigned to the `temp` variable. The value of `valB` (2) is then assigned to `valA`, so `valA` and `valB` are equivalent. Then, the `temp` value (16) is assigned to `valB`, so the values of the two variables finally are swapped.

If you want to sort any two values, `valA` and `valB`, in ascending order so that `valA` is always the lower value, you use the following `if` statement to make the decision whether to swap. If `valA` is more than `valB`, you want to swap the values. If `valA` is not more than `valB`, you do not want to swap the values.

```
if(valA > valB)
{
   temp = valA;
   valA = valB;
   valB = temp;
}
```

Sorting two values is a fairly simple task; sorting more values (`valC`, `valD`, `valE`, and so on) is more complicated. The task becomes manageable when you know how to use an array.

Using the Bubble Sort Algorithm

As an example, you might have a list of five numbers that you want to place in ascending order. Multiple sorting algorithms have been developed; an **algorithm** is a process or set of steps that solve a problem. In the **bubble sort** algorithm, you continue to compare pairs of items, swapping them if they are out of order, so that the smallest items "bubble" to the top of the list, eventually creating a sorted list. The bubble sort is neither the fastest nor most efficient sorting technique, but it is one of the simplest to comprehend and provides deeper understanding of array element manipulation.

To use a bubble sort, you place the original, unsorted values in an array, such as the following:

```
int[] someNums = {88, 33, 99, 22, 54};
```

You compare the first two numbers; if they are not in ascending order, you swap them. You compare the second and third numbers; if they are not in ascending order, you swap them. You continue down the list. Generically, for any `someNums[x]`, if the value of `someNums[x]` is larger than `someNums[x + 1]`, you want to swap the two values.

With the numbers 88, 33, 99, 22, and 54, the process proceeds as follows:

- Compare 88 and 33. They are out of order. Swap them. The list becomes 33, 88, 99, 22, 54.
- Compare the second and third numbers in the list—88 and 99. They are in order. Do nothing.
- Compare the third and fourth numbers in the list—99 and 22. They are out of order. Swap them. The list becomes 33, 88, 22, 99, 54.
- Compare the fourth and fifth numbers—99 and 54. They are out of order. Swap them. The list becomes 33, 88, 22, 54, 99.

When you reach the bottom of the list, the numbers are not in ascending order, but the largest number, 99, has moved to the bottom of the list. This feature gives the bubble sort its name—the "heaviest" value has sunk to the bottom of the list as the "lighter" values have bubbled to the top.

Assuming `b` and `temp` both have been declared as integer variables, the code so far is as follows:

```
for(b = 0; b < someNums.length - 1; ++b)
   if(someNums[b] > someNums[b + 1])
   {
      temp = someNums[b];
      someNums[b] = someNums[b + 1];
      someNums[b + 1] = temp;
   }
```

Instead of comparing b to someNums.length - 1 on every pass through the loop, it would be more efficient to declare a variable to which you assign someNums.length - 1 and use that variable in the comparison. That way, the arithmetic is performed just once. That step is omitted here to reduce the number of steps in the example.

Notice that the for statement tests every value of b from 0 through 3. The array someNums contains five integers, so the subscripts in the array range in value from 0 through 4. Within the for loop, each someNums[b] is compared to someNums[b + 1], so the highest legal value for b is 3. For a sort on any size array, the value of b must remain less than the array's length minus 1.

The list of numbers that began as 88, 33, 99, 22, 54 is currently 33, 88, 22, 54, 99. To continue to sort the list, you must perform the entire comparison-swap procedure again.

- Compare the first two values—33 and 88. They are in order; do nothing.
- Compare the second and third values—88 and 22. They are out of order. Swap them so the list becomes 33, 22, 88, 54, 99.
- Compare the third and fourth values—88 and 54. They are out of order. Swap them so the list becomes 33, 22, 54, 88, 99.
- Compare the fourth and fifth values—88 and 99. They are in order; do nothing.

After this second pass through the list, the numbers are 33, 22, 54, 88, and 99—close to ascending order, but not quite. You can see that with one more pass through the list, the values 22 and 33 will swap, and the list is finally placed in order. To fully sort the worst-case list, one in which the original numbers are descending (as out-of-ascending order as they could possibly be), you need to go through the list four times, making comparisons and swaps. At most, you always need to pass through the list as many times as its length minus one. Figure 9-1 shows the entire procedure.

```
for(a = 0; a < someNums.length - 1; ++a)
   for(b = 0; b < someNums.length - 1; ++b)
      if(someNums[b] > someNums[b + 1])
      {
         temp = someNums[b];
         someNums[b] = someNums[b + 1];
         someNums[b + 1] = temp;
      }
```

Figure 9-1 Ascending bubble sort of the someNums array elements

To place the list in descending order, you need to make only one change in the code in Figure 9-1: You change the greater-than sign (>) in if(someNums[b] > someNums[b + 1]) to a less-than sign (<).

When you use a bubble sort to sort any array into ascending order, the largest value "falls" to the bottom of the array after you have compared each pair of values in the array one time. The second time you go through the array making comparisons, there is no need to check the last pair of values. The largest value is guaranteed to already be at the bottom of the array. You can make the sort process even more efficient by using a new variable for the inner `for` loop and reducing the value by one on each cycle through the array. Figure 9-2 shows how you can use a new variable named `comparisonsToMake` to control how many comparisons are made in the inner loop during each pass through the list of values to be sorted. In the shaded statement, the `comparisonsToMake` value is decremented by 1 on each pass through the list.

```
int comparisonsToMake = someNums.length - 1;
for(a = 0; a < someNums.length - 1; ++a)
{
   for(b = 0; b < comparisonsToMake; ++b)
   {
      if(someNums[b] > someNums[b + 1])
      {
         temp = someNums[b];
         someNums[b] = someNums[b + 1];
         someNums[b + 1] = temp;
      }
   }
   --comparisonsToMake;
}
```

Figure 9-2 More efficient ascending bubble sort of the `someNums` array elements

Watch the video *Sorting*.

Sorting Arrays of Objects

You can sort arrays of objects in much the same way that you sort arrays of primitive types. The major difference occurs when you make the comparison that determines whether you want to swap two array elements. When you construct an array of the primitive element type, you compare the two array elements to determine whether they are out of order. When array elements are objects, you usually want to sort based on a particular object field.

Assume that you have created a simple `Employee` class, as shown in Figure 9-3. The class holds four data fields and get and set methods for the fields.

```
public class Employee
{
   private int empNum;
   private String lastName;
   private String firstName;
   private double salary;
   public int getEmpNum()
   {
      return empNum;
   }
   public void setEmpNum(int emp)
   {
      empNum = emp;
   }
   public String getLastName()
   {
      return lastName;
   }
   public void setLastName(String name)
   {
      lastName = name;
   }
   public String getFirstName()
   {
      return firstName;
   }
   public void setFirstName(String name)
   {
      firstName = name;
   }
   public double getSalary()
   {
      return salary;
   }
   public void setSalary(double sal)
   {
      salary = sal;
   }
}
```

Figure 9-3 The `Employee` class

You can write a program that contains an array of five `Employee` objects using the following statement:

```
Employee[] someEmps = new Employee[5];
```

Assume that after you assign employee numbers and salaries to the `Employee` objects, you want to sort the `Employees` in `salary` order. You can pass the array to a `bubbleSort()` method that is prepared to receive `Employee` objects. Figure 9-4 shows the method.

```
public static void bubbleSort(Employee[] array)
{
   int a, b;
   Employee temp;
   int highSubscript = array.length - 1;
   for(a = 0; a < highSubscript; ++a)
    for(b = 0; b < highSubscript; ++b)
      if(array[b].getSalary() > array[b + 1].getSalary())
      {
         temp = array[b];
         array[b] = array[b + 1];
         array[b + 1] = temp;
      }
}
```

Figure 9-4 The `bubbleSort()` method that sorts `Employee` objects by their salaries

Examine Figure 9-4 carefully, and notice that the `bubbleSort()` method is very similar to the `bubbleSort()` method you use for an array of any primitive type, but there are three major differences:

- The `bubbleSort()` method header shows that it receives an array of type `Employee`.
- The `temp` variable created for swapping is type `Employee`. The `temp` variable will hold an `Employee` object, not just one number or one field. It is important to note that even though only employee salaries are compared, you do not just swap employee salaries. You do not want to substitute one employee's salary for another's. Instead, you swap each `Employee` object's `empNum` and `salary` as a unit.
- The comparison for determining whether a swap should occur uses method calls to the `getSalary()` method to compare the returned salary for each `Employee` object in the array with the salary of the adjacent `Employee` object.

TWO TRUTHS & A LIE

Sorting Array Elements Using the Bubble Sort Algorithm

1. In an ascending bubble sort, you compare pairs of items, swapping them if they are out of order, so that the largest items “bubble” to the top of the list, eventually creating a sorted list.
2. When you sort objects, you usually want to sort based on a particular object field.
3. When you make a swap while sorting an array of objects, you typically swap entire objects and not just the field on which the comparison is made.

The false statement is #1. In an ascending bubble sort, you compare pairs of items, swapping them if they are out of order, so that the smallest items “bubble” to the top of the list, eventually creating a sorted list.

You Do It

Using a Bubble Sort

In this section, you create a program in which you enter values that you sort using the bubble sort algorithm. You display the values during each iteration of the outer sorting loop so that you can track the values as they are repositioned in the array.

1. Open a new file in your text editor, and create the shell for a `BubbleSortDemo` program as follows:

```
import java.util.*;
class BubbleSortDemo
{
   public static void main(String[] args)
   {
   }
}
```

2. Make some declarations between the curly braces of the `main()` method. Declare an array of five integers and a variable to control the number of comparisons to make during the sort. Declare a `Scanner` object, two integers to use as subscripts for handling the array, and a temporary integer value to use during the sort.

(continues)

(continued)

```
int[] someNums = new int[5];
int comparisonsToMake = someNums.length - 1;
Scanner keyboard = new Scanner(System.in);
int a, b, temp;
```

3. Write a `for` loop that prompts the user for a value for each array element and accepts them.

```
for(a = 0; a < someNums.length; ++a)
{
   System.out.print("Enter number " + (a + 1) + " >> ");
   someNums[a] = keyboard.nextInt();
}
```

4. Next, call a method that accepts the array and the number of sort iterations performed so far, which is 0. The purpose of the method is to display the current status of the array as it is being sorted.

```
display(someNums, 0);
```

5. Add the nested loops that perform the sort. The outer loop controls the number of passes through the list, and the inner loop controls the comparisons on each pass through the list. When any two adjacent elements are out of order, they are swapped. At the end of the nested loop, the current list is output and the number of comparisons to be made on the next pass is reduced by one.

```
for(a = 0; a < someNums.length - 1; ++a)
{
   for(b = 0; b < comparisonsToMake; ++b)
   {
      if(someNums[b] > someNums[b + 1])
      {
         temp = someNums[b];
         someNums[b] = someNums[b + 1];
         someNums[b + 1] = temp;
      }
   }
   display(someNums, (a + 1));
   --comparisonsToMake;
}
```

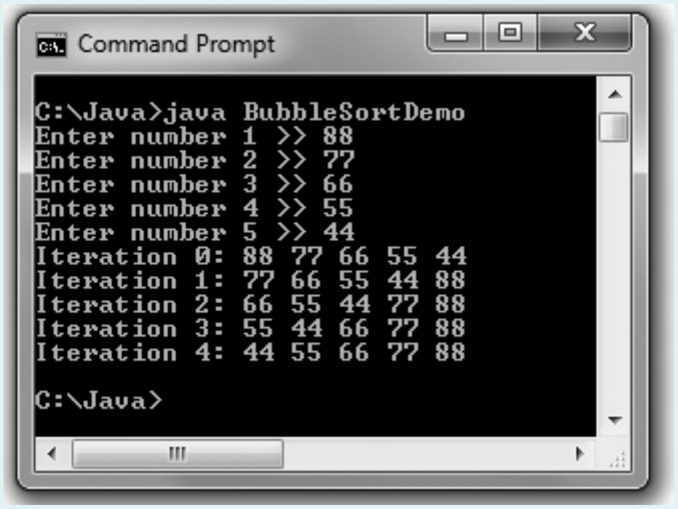
6. After the closing brace for the `main()` method, but before the closing brace for the class, insert the `display()` method. It accepts the array and the current outer loop index, and it displays the array contents.

(continues)

(continued)

```
public static void display(int[] someNums, int a)
{
   System.out.print("Iteration " + a + ": ");
   for(int x = 0; x < someNums.length; ++x)
      System.out.print(someNums[x] + " ");
   System.out.println();
}
```

7. Save the file as **BubbleSortDemo.java**, and then compile and execute it. Figure 9-5 shows a typical execution. Notice that after the first iteration, the largest value has sunk to the bottom of the list. After the second iteration, the two largest values are at the bottom of the list, and so on.



```
C:\Java>java BubbleSortDemo
Enter number 1 >> 88
Enter number 2 >> 77
Enter number 3 >> 66
Enter number 4 >> 55
Enter number 5 >> 44
Iteration 0: 88 77 66 55 44
Iteration 1: 77 66 55 44 88
Iteration 2: 66 55 44 77 88
Iteration 3: 55 44 66 77 88
Iteration 4: 44 55 66 77 88

C:\Java>
```

Figure 9-5 Typical execution of the `BubbleSortDemo` application

8. Modify the `BubbleSortDemo` application to any size array you choose. Confirm that no matter how many array elements you specify, the sorting algorithm works correctly and ends with a completely sorted list, regardless of the order of your entered values.

Sorting Array Elements Using the Insertion Sort Algorithm

The bubble sort works well and is relatively easy to understand and manipulate, but many other sorting algorithms have been developed. For example, when you use an **insertion sort**, you look at each list element one at a time. If an element is out of order relative to any of the items earlier in the list, you move each earlier item down one position and then insert the tested element. The insertion sort is similar to the technique you would most likely use to sort a group of objects manually. For example, if a list contains the values 2, 3, 1, and 4, and you want to place them in ascending order using an insertion sort, you test the values 2 and 3, but you do not move them because they are in order. However, when you test the third value in the list, 1, you move both 2 and 3 to later positions and insert 1 at the first position.

Figure 9-6 shows the logic that performs an ascending insertion sort using a five-element integer array named `someNums`. The logic assumes that `a`, `b`, and `temp` have all been declared as integers.

```
int[] someNums = {90, 85, 65, 95, 75};
a = 1;
while(a < someNums.length)
{
   temp = someNums[a];
   b = a - 1;
   while(b >= 0 && someNums[b] > temp)
   {
      someNums[b + 1] = someNums[b];
      --b;
   }
   someNums[b + 1] = temp;
   ++a;
}
```

Figure 9-6 The insertion sort

The outer loop in Figure 9-6 varies a loop control variable `a` from 1 through one less than the size of the array. The logic proceeds as follows:

First `a` is set to 1, and then the `while` loop begins.

1. The value of `temp` is set to `someNums[1]`, which is 85, and `b` is set to 0.
2. Because `b` is greater than or equal to 0 and `someNums[b]` (90) is greater than `temp`, the inner loop is entered. (If you were performing a descending sort, then you would ask whether `someNums[b]` was less than `temp`.)
3. The value of `someNums[1]` becomes 90 and `b` is decremented, making it –1, so `b` is no longer greater than or equal to 0, and the inner loop ends.
4. Then `someNums[0]` is set to `temp`, which is 85.

After these steps, 90 was moved down one position and 85 was inserted in the first position, so the array values are in slightly better order than they were originally. The values are as follows: 85, 90, 65, 95, 75.

Now, in the outer loop, `a` becomes 2. The logic in Figure 9-6 proceeds as follows:

1. The value of `temp` becomes 65, and `b` is set to 1.
2. The value of `b` is greater than or equal to 0, and `someNums[b]` (90) is greater than `temp`, so the inner loop is entered.
3. The value of `someNums[2]` becomes 90 and `b` is decremented, making it 0, so the loop executes again.
4. The value of `someNums[1]` becomes 85 and `b` is decremented, making it –1, so the loop ends.
5. Then `someNums[0]` becomes 65.

After these steps, the array values are in better order than they were originally, because 65 and 85 now both come before 90. The values are: 65, 85, 90, 95, 75. Now, `a` becomes 3. The logic in Figure 9-6 proceeds to work on the new list as follows:

1. The value of `temp` becomes 95, and `b` is set to 2.
2. For the loop to execute, `b` must be greater than or equal to 0, which it is, and `someNums[b]` (90) must be greater than `temp`, which it is *not*. So, the inner loop does not execute.
3. Therefore, `someNums[2]` is set to 90, which it already was. In other words, no changes are made.

Now, `a` is increased to 4. The logic in Figure 9-6 proceeds as follows:

1. The value of `temp` becomes 75, and `b` is set to 3.
2. The value of `b` is greater than or equal to 0, and `someNums[b]` (95) is greater than `temp`, so the inner loop is entered.
3. The value of `someNums[4]` becomes 95 and `b` is decremented, making it 2, so the loop executes again.
4. The value of `someNums[3]` becomes 90 and `b` is decremented, making it 1, so the loop executes again.
5. The value of `someNums[2]` becomes 85 and `b` is decremented, making it 0; `someNums[b]` (65) is no longer greater than `temp` (75), so the inner loop ends. In other words, the values 85, 90, and 95 are each moved down one position, but 65 is left in place.
6. Then `someNums[1]` becomes 75.

   After these steps, all the array values have been rearranged in ascending order as follows: 65, 75, 85, 90, 95.

Watch the video *The Insertion Sort*.

Many sorting algorithms exist in addition to the bubble sort and insertion sort. You might want to investigate the logic used by the *selection sort*, *cocktail sort*, *gnome sort*, and *quick sort*.

TWO TRUTHS & A LIE

Sorting Array Elements Using the Insertion Sort Algorithm

1. When you use an insertion sort, you look at each list element one at a time and move items down if the tested element should be inserted before them.
2. You can create an ascending list using an insertion sort, but not a descending one.
3. The insertion sort is similar to the technique you would most likely use to sort a group of objects manually.

The false statement is #2. You can create both ascending and descending lists using an insertion sort.

You Do It

Using an Insertion Sort

In this section, you modify the `BubbleSortDemo` program so it performs an insertion sort.

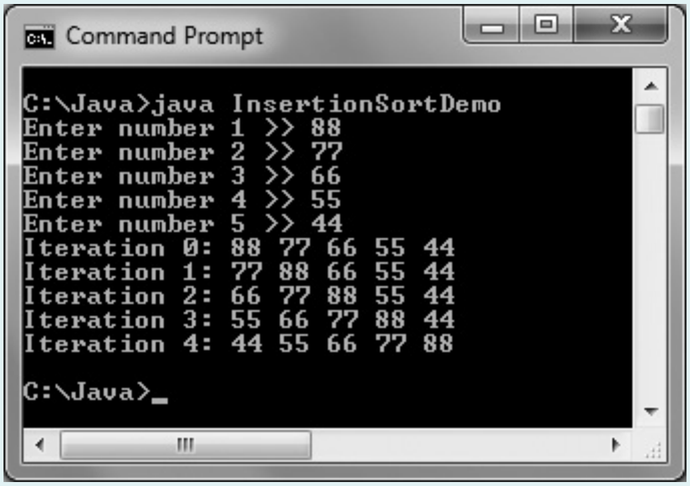
1. Open the **BubbleSortDemo.java** file. Change the class name to **`InsertionSortDemo`**, and immediately save the file as **InsertionSortDemo.java**.
2. Remove the declaration for `comparisonsToMake`.
3. Remove the 14 lines of code that constitute the nested loops that perform the bubble sort. In other words, remove all the lines from the start of the second `for` loop through the closing curly brace following the statement that decrements `comparisonsToMake`.
4. Replace the removed lines with the statements that perform the insertion sort. These are the same statements you saw in Figure 9-6 with the addition of a call to the `display()` method so that you can track the progress of the sort:

(continues)

(continued)

```
a = 1;
while(a < someNums.length)
{
   temp = someNums[a];
   b = a - 1;
   while(b >= 0 && someNums[b] > temp)
   {
      someNums[b + 1] = someNums[b];
      --b;
   }
   someNums[b + 1] = temp;
   display(someNums, a);
   ++a;
}
```

5. Save the file as **InsertionSortDemo.java**, and then compile and execute it. Figure 9-7 shows a typical execution. During the first loop, 77 is compared with 88 and inserted at the beginning of the array. In the second loop, 66 is compared with both 77 and 88 and inserted at the beginning of the array. Then the same thing happens with 55 and 44 until all the values are sorted.



```
C:\Java>java InsertionSortDemo
Enter number 1 >> 88
Enter number 2 >> 77
Enter number 3 >> 66
Enter number 4 >> 55
Enter number 5 >> 44
Iteration 0: 88 77 66 55 44
Iteration 1: 77 88 66 55 44
Iteration 2: 66 77 88 55 44
Iteration 3: 55 66 77 88 44
Iteration 4: 44 55 66 77 88

C:\Java>_
```

Figure 9-7 Typical execution of the `InsertionSortDemo` program

6. Try the program with other input values, and examine the output so that you understand how the insertion sort algorithm works.

Using Two-Dimensional and Other Multidimensional Arrays

When you declare an array such as `int[] someNumbers = new int[3];`, you can envision the three declared integers as a column of numbers in memory, as shown in Figure 9-8. In other words, you can picture the three 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 size of the array as its height.

`someNumbers[0]`
`someNumbers[1]`
`someNumbers[2]`

Figure 9-8 View of a single-dimensional array in memory

Java also supports two-dimensional arrays. **Two-dimensional arrays** have two or more columns of values, as shown in Figure 9-9. The two dimensions represent the height and width of the array. Another way to picture a two-dimensional array is as an array of arrays. It is easiest to picture two-dimensional arrays as having both rows and 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.

`someNumbers[0][0]`	`someNumbers[0][1]`	`someNumbers[0][2]`	`someNumbers[0][3]`
`someNumbers[1][0]`	`someNumbers[1][1]`	`someNumbers[1][2]`	`someNumbers[1][3]`
`someNumbers[2][0]`	`someNumbers[2][1]`	`someNumbers[2][2]`	`someNumbers[2][3]`

Figure 9-9 View of a two-dimensional array in memory

When you declare a one-dimensional array, you type a set of square brackets after the array's data type. To declare a two-dimensional array in Java, you type two sets of brackets after the array type. For example, the array in Figure 9-9 can be declared as follows, creating an array named `someNumbers` that holds three rows and four columns:

```
int[][] someNumbers = new int[3][4];
```

Just as with a one-dimensional array, if you do not provide values for the elements in a two-dimensional numeric array, the values default to zero. You can assign other values to the array elements later. For example, `someNumbers[0][0] = 14;` assigns the value 14 to the element of the `someNumbers` array that is in the first column of the first row.

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

```
int[][] someNumbers = {{8, 9, 10, 11},
                       {1, 3, 12, 15},
                       {5, 9, 44, 99} };
```

The `someNumbers` array contains three rows and four columns. You do not *need* to place each row of values for a two-dimensional array on its own line. However, doing so makes the positions of values easier to understand. You contain the entire set of values within an outer pair of curly braces. The first row of the array holds the four integers 8, 9, 10, and 11. Notice that these four integers are placed within their own inner set of curly braces to indicate that they constitute one row, or the first row, which is row 0. Similarly, 1, 3, 12, and 15 make up the second row (row 1), which you reference with the subscript 1. Next, 5, 9, 44, and 99 are the values in the third row (row 2), which you reference with the subscript 2. The value of `someNumbers[0][0]` is 8. The value of `someNumbers[0][1]` is 9. The value of `someNumbers[2][3]` is 99. The value within the first set of brackets following the array name always refers to the row; the value within the second brackets refers to the column.

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) and one- and two-bedroom apartments. The monthly rent for each type of apartment is different—the higher the floor, the higher the rent (the view is better), and the rent is higher for apartments with more bedrooms. Table 9-1 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

Table 9-1 Rents charged (in dollars)

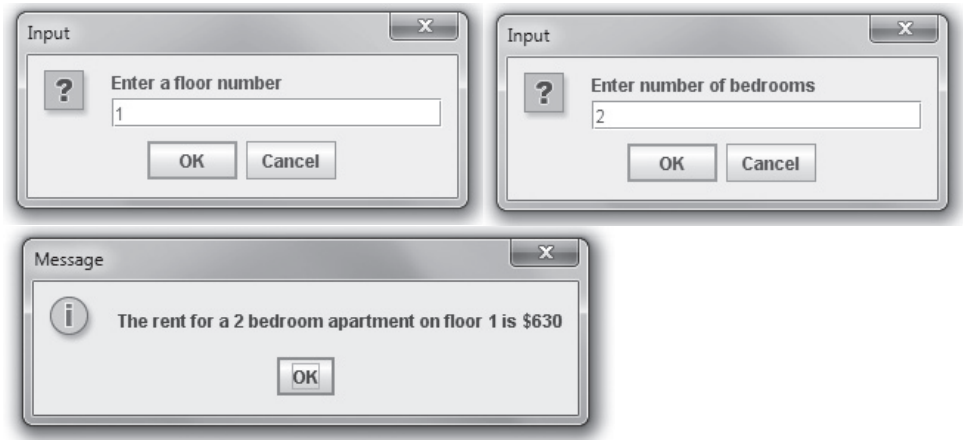
To determine a tenant's rent, you need to know two pieces of information: the floor on which the tenant rents an apartment and the number of bedrooms in the apartment. Within a Java program, you can declare an array of rents using the following code:

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

If you declare two integers named `floor` and `bedrooms`, then any tenant's rent can be referred to as `rents[floor][bedrooms]`. Figure 9-10 shows an application that prompts a user for a floor number and number of bedrooms. Figure 9-11 shows a typical execution.

```
import javax.swing.*;
class FindRent
{
   public static void main(String[] args)
   {
      int[][] rents = { {400, 450, 510},
                        {500, 560, 630},
                        {625, 676, 740},
                        {1000, 1250, 1600} };
      String entry;
      int floor;
      int bedrooms;
      entry = JOptionPane.showInputDialog(null,
         "Enter a floor number ");
      floor = Integer.parseInt(entry);
      entry = JOptionPane.showInputDialog(null,
         "Enter number of bedrooms ");
      bedrooms = Integer.parseInt(entry);
      JOptionPane.showMessageDialog(null,
        "The rent for a " + bedrooms +
        " bedroom apartment on floor " + floor +
        " is $" + rents[floor][bedrooms]);
   }
}
```

Figure 9-10 The `FindRent` class



Enter a floor number

1

Enter number of bedrooms

2

The rent for a 2 bedroom apartment on floor 1 is $630

Figure 9-11 Typical execution of the `FindRent` program

Passing a Two-Dimensional Array to a Method

When you pass a two-dimensional array to a method, you pass the array name just as you do with a one-dimensional array. A method that receives a two-dimensional array uses two bracket pairs following the data type in the parameter list of the method header. For example, the following method headers accept two-dimensional arrays of `int`s, `double`s, and `Employee`s, respectively:

```
public static void displayScores(int[][]scoresArray)
public static boolean areAllPricesHigh(double[][] prices)
public static double computePayrollForAllEmployees(Employee[][] staff)
```

In each case, notice that the brackets indicating the array in the method header are empty. There is no need to insert numbers into the brackets because each passed array name is a starting memory address. The way you manipulate subscripts within the method determines how rows and columns are accessed.

Using the `length` Field with a Two-Dimensional Array

In Chapter 8, you learned that a one-dimensional array has a `length` field that holds the number of elements in the array. With a two-dimensional array, the `length` field holds the number of rows in the array. Each row, in turn, has a `length` field that holds the number of columns in the row. For example, suppose you declare a `rents` array as follows:

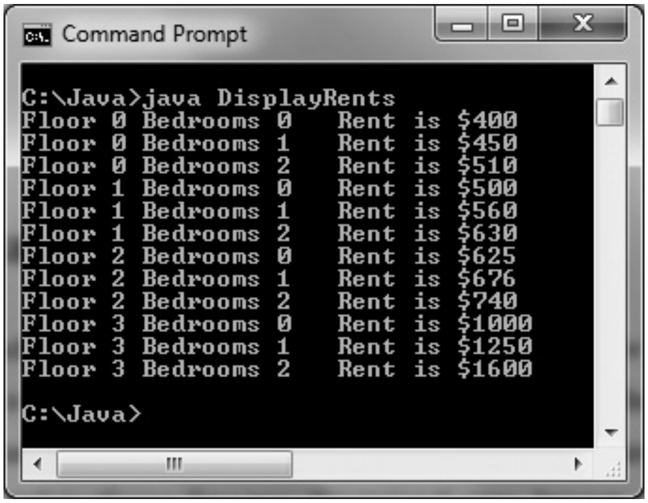
```
int[][] rents = { {400, 450, 510},
                  {500, 560, 630},
                  {625, 676, 740},
                  {1000, 1250, 1600} };
```

The value of `rents.length` is 4 because there are four rows in the array. The value of `rents[0].length` is 3 because there are three columns in the first row of the `rents` array. Similarly, the value of `rents[1].length` also is 3 because there are three columns in the second row.

Figure 9-12 shows an application that uses the `length` fields associated with the `rents` array to display all the rents. The `floor` variable varies from 0 through one less than 4 in the outer loop, and the `bdrms` variable varies from 0 through one less than 3 in the inner loop. Figure 9-13 shows the output.

```
class DisplayRents
{
   public static void main(String[] args)
   {
      int[][] rents = { {400, 450, 510},
                        {500, 560, 630},
                        {625, 676, 740},
                        {1000, 1250, 1600} };
      int floor;
      int bdrms;
      for(floor = 0; floor < rents.length; ++floor)
         for(bdrms = 0; bdrms < rents[floor].length; ++bdrms)
            System.out.println("Floor " + floor +
               " Bedrooms " + bdrms + "   Rent is $" +
               rents[floor][bdrms]);
   }
}
```

Figure 9-12 The `DisplayRents` class



```
C:\Java>java DisplayRents
Floor 0 Bedrooms 0   Rent is $400
Floor 0 Bedrooms 1   Rent is $450
Floor 0 Bedrooms 2   Rent is $510
Floor 1 Bedrooms 0   Rent is $500
Floor 1 Bedrooms 1   Rent is $560
Floor 1 Bedrooms 2   Rent is $630
Floor 2 Bedrooms 0   Rent is $625
Floor 2 Bedrooms 1   Rent is $676
Floor 2 Bedrooms 2   Rent is $740
Floor 3 Bedrooms 0   Rent is $1000
Floor 3 Bedrooms 1   Rent is $1250
Floor 3 Bedrooms 2   Rent is $1600

C:\Java>
```

Figure 9-13 Output of the `DisplayRents` program

Watch the video *Two-Dimensional Arrays*.

Understanding Ragged Arrays

In a two-dimensional array, each row also is an array. In Java, you can declare each row to have a different length. When a two-dimensional array has rows of different lengths, it is a **ragged array** because you can picture the ends of each row as uneven. You create a ragged array by defining the number of rows for a two-dimensional array, but not defining the number of columns in the rows. For example, suppose that you have four sales representatives, each of whom covers a different number of states as their sales territory. Further suppose that you want an array to store total sales for each state for each sales representative. You would define the array as follows:

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

This statement declares an array with four rows, but the rows are not yet created. Then, you can declare the individual rows, based on the number of states covered by each salesperson as follows:

```
sales[0] = new double[12];
sales[1] = new double[18];
sales[2] = new double[9];
sales[3] = new double[11];
```

Using Other Multidimensional Arrays

Besides one- and two-dimensional arrays, Java also supports arrays with three, four, and more dimensions. The general term for arrays with more than one dimension is **multidimensional arrays**. For example, if you own an apartment building with a number of floors and different numbers of bedrooms available in apartments on each floor, you can use a two-dimensional array to store the rental fees. If you own several apartment buildings, you might want to employ a third dimension to store the building number. An expression such as `rents[building][floor][bedrooms]` refers to a specific rent figure for a building whose building number is stored in the `building` variable and whose floor and bedroom numbers are stored in the `floor` and `bedrooms` variables. Specifically, `rents[5][1][2]` refers to a two-bedroom apartment on the first floor of building 5. When you are programming in Java, 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, Java lets you create arrays of any size.

TWO TRUTHS & A LIE

Using Two-Dimensional and Other Multidimensional Arrays

1. Two-dimensional arrays have both rows and columns, so you must use two subscripts when you access an element in a two-dimensional array.
2. The following array contains two columns and three rows:

```
int[][] myArray = {{12, 14, 19},
                   {33, 45, 88}};
```

3. With a two-dimensional array, the `length` field holds the number of rows in the array; each row has a `length` field that holds the number of columns in the row.

The false statement is #2. The array shown has two rows and three columns.

You Do It

Using a Two-Dimensional Array

In this section, you create an application that demonstrates using a two-dimensional array.

1. Open a new file in your text editor, and start a class that will demonstrate a working two-dimensional array:

```
import java.util.Scanner;
class TwoDimensionalArrayDemo
{
   public static void main(String[] args)
   {
```

2. Declare a three-by-three array of integers. By default, the elements will all be initialized to 0.

```
int[][] count = new int[3][3];
```

3. Declare a `Scanner` object for input, variables to hold a row and column, and a constant that can be used to indicate when the user wants to quit the application.

```
Scanner input = new Scanner(System.in);
int row, column;
final int QUIT = 99;
```

(continues)

(continued)

4. Prompt the user to enter a row or the QUIT value to quit, then accept the user's input.

```
System.out.print("Enter a row or " + QUIT +
   " to quit > ");
row = input.nextInt();
```

5. In a loop that continues if the user has not entered the `QUIT` value, prompt the user for a column. If the row and column are both within appropriate ranges, add 1 to the element in the selected position.

```
while(row != QUIT)
{
   System.out.print("Enter a column > ");
   column = input.nextInt();
   if(row < count.length && column < count[row].length)
   {
      count[row][column]++;
```

6. Still within the `if` statement that checks for a valid row and column, add a nested loop that displays each row and column of the newly incremented array. The elements in each row are displayed on the same line, and a new line is started at the end of each row. Add a closing curly brace for the `if` statement.

```
   for(int r = 0; r < count.length; ++r)
   {
      for(int c = 0; c < count[r].length; ++c)
         System.out.print(count[r][c] + "  ");
      System.out.println();
   }
}
```

7. Add an `else` clause to the `if` statement to display an error message when the row or column value is too high.

```
else
   System.out.println("Invalid position selected");
```

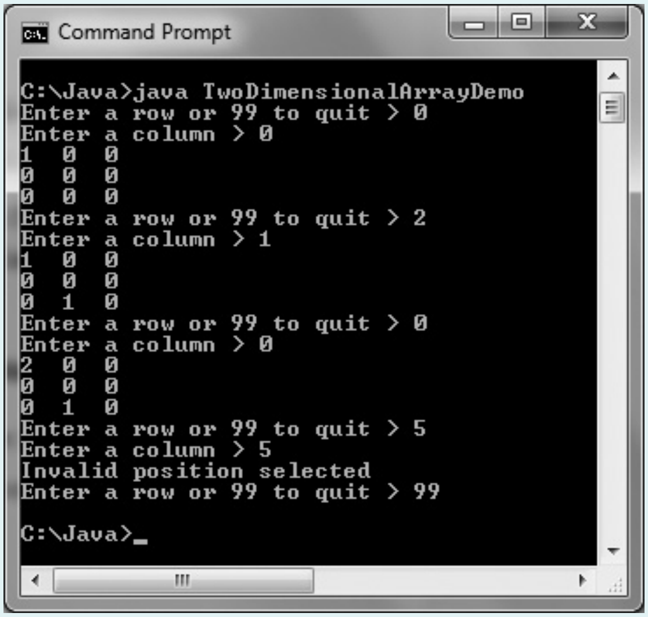
8. At the end of the loop, prompt the user for and accept the next row number. Add closing curly braces for the loop, the `main()` method, and the class.

```
         System.out.print("Enter a row or " + QUIT +
            " to quit > ");
         row = input.nextInt();
      }
   }
}
```

(continues)

(continued)

9. Save the file as **TwoDimensionalArrayDemo.java**. Compile and execute the program. Figure 9-14 shows a typical execution. As the user continues to enter row and column values, the appropriate elements in the array are incremented.



```
C:\Java>java TwoDimensionalArrayDemo
Enter a row or 99 to quit > 0
Enter a column > 0
1  0  0
0  0  0
0  0  0
Enter a row or 99 to quit > 2
Enter a column > 1
1  0  0
0  0  0
0  1  0
Enter a row or 99 to quit > 0
Enter a column > 0
2  0  0
0  0  0
0  1  0
Enter a row or 99 to quit > 5
Enter a column > 5
Invalid position selected
Enter a row or 99 to quit > 99

C:\Java>_
```

Figure 9-14 Typical execution of the `TwoDimensionalArrayDemo` program

Using the `Arrays` Class

When you fully understand the power of arrays, you will want to use them to store all kinds of objects. Frequently, you will want to perform similar tasks with different arrays—for example, filling them with values and sorting their elements. Java provides an **`Arrays` class**, which contains many useful methods for manipulating arrays. Table 9-2 shows some of the useful methods of the `Arrays` class. For each method listed in the left column of the table, `type` stands for a data type; an overloaded version of each method exists for each appropriate data type. For example, there is a version of the `sort()` method to sort `int`, `double`, `char`, `byte`, `float`, `long`, `short`, and `Object` arrays.

You will learn about the `Object` class in the chapter *Advanced Inheritance Concepts*.

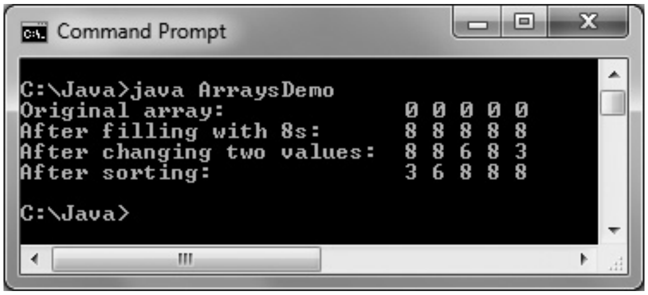
Method	Purpose
`static int binarySearch(type [] a, type key)`	Searches the specified array for the specified key value using the binary search algorithm
`static boolean equals(type[] a, type[] a2)`	Returns `true` if the two specified arrays of the same type are equal to one another
`static void fill(type[] a, type val)`	Assigns the specified value to each element of the specified array
`static void sort(type[] a)`	Sorts the specified array into ascending order
`static void sort(type[] a, int fromIndex, int toIndex)`	Sorts the specified range of the specified array into ascending order

Table 9-2 Useful methods of the `Arrays` class

The methods in the `Arrays` class are `static` methods, which means you use them with the class name without instantiating an `Arrays` object. The `Arrays` class is located in the `java.util` package, so you can use the statement `import java.util.*;` to access it. The `ArraysDemo` application in Figure 9-15 demonstrates how you can use some of the methods in the `Arrays` class. In the `ArraysDemo` class, the `myScores` array is created to hold five integers. Then, a message and the array reference are passed to a `display()` method. The first line of the output in Figure 9-16 shows that the original array is filled with *0*s at creation. After the first display, the `Arrays.fill()` method is called in the first shaded statement in Figure 9-15. Because the arguments are the name of the array and the number 8, when the array is displayed a second time the output is all *8*s. In the application, two of the array elements are changed to 6 and 3, and the array is displayed again. Finally, in the second shaded statement, the `Arrays.sort()` method is called. The output in Figure 9-16 shows that when the `display()` method executes the fourth time, the array elements have been sorted in ascending order.

```
import java.util.*;
public class ArraysDemo
{
   public static void main(String[] args)
   {
      int[] myScores = new int [5];
      display("Original array:             ", myScores);
      Arrays.fill(myScores, 8);
      display("After filling with 8s:      ", myScores);
      myScores[2] = 6;
      myScores[4] = 3;
      display("After changing two values:  ", myScores);
      Arrays.sort(myScores);
      display("After sorting:              ", myScores);
   }

   public static void display(String message, int array[])
   {
      int sz = array.length;
      System.out.print(message);
      for(int x = 0; x < sz; ++x)
         System.out.print(array[x] + " ");
      System.out.println();
   }
}
```

Figure 9-15 The `ArraysDemo` application



```
C:\Java>java ArraysDemo
Original array:              0 0 0 0 0
After filling with 8s:       8 8 8 8 8
After changing two values:   8 8 6 8 3
After sorting:               3 6 8 8 8

C:\Java>
```

Figure 9-16 Output of the `ArraysDemo` application

The `Arrays` class `binarySearch()` methods provide convenient ways to search through sorted lists of values of various data types. It is important that the list be in order before you use it in a call to `binarySearch()`; otherwise, the results are unpredictable. You do not have to understand how a binary search works to use the `binarySearch()` method, but basically the operation takes place as follows:

- You have a sorted array and an item for which you are searching within the array. Based on the array size, you determine the middle position. (In an array with an even number of elements, this can be either of the two middle positions.)
- You compare the item you are looking for with the element in the middle position of the array and decide whether your item is above that point in the array—that is, whether your item's value is less than the middle-point value.
- If it is above that point in the array, you next find the middle position of the top half of the array; if it is not above that point, you find the middle position of the bottom half. Either way, you compare your item with that of the new middle position and divide the search area in half again.
- Ultimately, you find the element or determine that it is not in the array.

Programmers often refer to a binary search as a "divide and conquer" procedure. If you have ever played a game in which you tried to guess what number someone was thinking, you might have used a similar technique.

Suppose your organization uses six single-character product codes. Figure 9-17 contains a `VerifyCode` application that verifies a product code entered by the user. The array `codes` holds six values in ascending order. The user enters a code that is extracted from the first `String` position using the `String` class `charAt()` method. Next, the array of valid characters and the user-entered character are passed to the `Arrays.binarySearch()` method. If the character is found in the array, its position is returned. If the character is not found in the array, a negative integer is returned and the application displays an error message. Figure 9-18 shows the program's execution when the user enters *K*; the character is found in position 2 (the third position) in the array.

The negative integer returned by the `binarySearch()` method when the value is not found is the negative equivalent of the array size. In most applications, you do not care about the exact value returned when there is no match; you care only whether it is negative.

```
import java.util.*;
import javax.swing.*;
public class VerifyCode
{
   public static void main(String[] args)
   {
      char[] codes = {'B', 'E', 'K', 'M', 'P', 'T'};
      String entry;
      char usersCode;
      int position;
      entry = JOptionPane.showInputDialog(null,
         "Enter a product code");
      usersCode = entry.charAt(0);
      position = Arrays.binarySearch(codes, usersCode);
      if(position >= 0)
         JOptionPane.showMessageDialog(null, "Position of " +
            usersCode + " is " + position);
      else
         JOptionPane.showMessageDialog(null, usersCode +
            " is an invalid code");
   }
}
```

Figure 9-17 The `VerifyCode` application

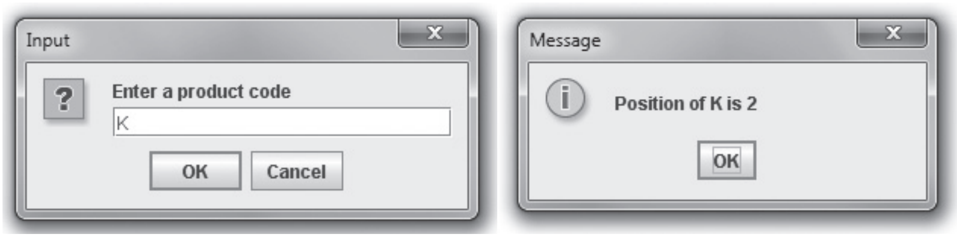


Figure 9-18 Typical execution of the `VerifyCode` application

The `sort()` and `binarySearch()` methods in the `Arrays` class are very useful and allow you to achieve results by writing many fewer instructions than if you had to write the methods yourself. This does not mean you wasted your time reading about sorting and searching methods earlier in this chapter. The more completely you understand how arrays can be manipulated, the more useful, efficient, and creative your future applications will be.

TWO TRUTHS & A LIE

Using the `Arrays` Class

1. The `Arrays` class contains methods for manipulating arrays, such as `binarySearch()`, `fill()`, and `sort()`.
2. You can use the `Arrays` class `binarySearch()` method successfully on any array as soon as you have assigned values to the array elements.
3. The `binarySearch()` method works by continuously deciding whether the element sought is above or below the halfway point in sublists of the original list.

The false statement is #2. Before you can use the `Arrays` class `binarySearch()` method successfully, the array elements must be in order.

You Do It

Using `Arrays` Class Methods

In this section, you create an application that demonstrates several `Arrays` class methods. The application will allow the user to enter a menu of entrees that are available for the day at a restaurant. Then, the application will present the menu to the user, allow a request, and indicate whether the requested item is on the menu.

1. Open a new file in your text editor, and type the `import` statements you need to create an application that will use the `JOptionPane` and the `Arrays` classes:

```
import java.util.*;
import javax.swing.*;
```

2. Add the first few lines of the `MenuSearch` application class:

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

(continues)

(continued)

3. Declare an array to hold the day's menu choices; the user is allowed to enter up to 10 entrees. Also declare two `String`s—one to hold the user's current entry and the other to accumulate the entire menu list as it is entered. The two `String` variables are initialized to empty `String`s using quotation marks; if you do not initialize these `String`s, you receive a compiler error because you might attempt to display them without having entered a legitimate value. Also, declare an integer to use as a subscript for the array, another to hold the number of menu items entered, and a third to hold the highest allowable subscript, which is 1 less than the array size:

```
String[] menuChoices = new String[10];
String entry= "", menuString = "";
int x = 0;
int numEntered;
int highestSub = menuChoices.length - 1;
```

4. Use the `Arrays.fill()` method to fill the menu array with *z* characters, as shown in the following line of code. You use this method so that when you perform a search later, actual values will be stored in any unused menu positions. If you ignore this step and fill less than half the array, your search method might generate an error.

```
Arrays.fill(menuChoices, "zzzzzzz");
```

Lowercase *z*s were purposely chosen as the array fill characters because they have a higher value than any other letter. Therefore, when the user's entries are sorted, the *zzzzzzz* entries will be at the bottom of the list.

5. Display an input dialog box into which the user can enter a menu item. Allow the user to quit before entering 10 items by typing "zzz". (Using a value such as "zzz" is a common programming technique to check for the user's desire to stop entering data. If the data items are numeric instead of text, you might use a value such as 999. Values the user enters that are not "real" data, but just signals to stop, are often called **dummy values**.) After the user enters the first menu item, the application enters a loop that continues to add the entered item to the menu list, increase the subscript, and prompt for a new menu item. The loop continues while the user has not entered "zzz" and the subscript has not exceeded the allowable limit. When the loop ends, save the number of menu items entered:

(continues)

(continued)

```
menuChoices[x] = JOptionPane.showInputDialog(null,
   "Enter an item for today's menu, or zzz to quit:");
while(!menuChoices[x].equals("zzz") && x < highestSub)
{
   menuString = menuString + menuChoices[x] + "\n";
   ++x;
   if(x < highestSub)
      menuChoices[x] = JOptionPane.showInputDialog(null,
        "Enter an item for today's menu, or zzz to quit");
}
numEntered = x;
```

6. When the menu is complete, display it for the user and allow the user to make a request:

```
entry = JOptionPane.showInputDialog(null,
   "Today's menu is:\n" + menuString +
   "Please make a selection:");
```

7. Sort the array from index position 0 to `numEntered` so that it is in ascending order prior to using the `binarySearch()` method. If you do not sort the array, the result of the `binarySearch()` method is unpredictable. You could sort the entire array, but it is more efficient to sort only the elements that hold actual menu items:

```
Arrays.sort(menuChoices, 0, numEntered);
```

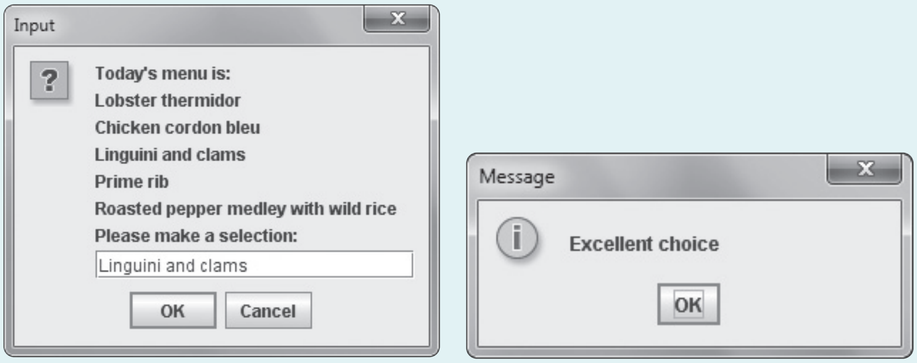
8. Use the `Arrays.binarySearch()` method to search for the requested entry in the previously sorted array. If the method returns a nonnegative value that is less than the `numEntered` value, display the message "Excellent choice"; otherwise, display an error message:

```
x = Arrays.binarySearch(menuChoices, entry);
if(x >= 0 && x < numEntered)
   JOptionPane.showMessageDialog(null, "Excellent choice");
else
   JOptionPane.showMessageDialog(null,
      "Sorry - that item is not on tonight's menu");
```

9. Add the closing curly braces for the `main()` method and the class, and save the file as **MenuSearch.java**. Compile and execute the application. When prompted, enter as many menu choices as you want, and enter "zzz" when you want to quit data entry. When prompted again, enter a menu choice and observe the results. (A choice you enter must match the spelling in the menu exactly.) Figure 9-19 shows a typical menu as it is presented to the user, and the results after the user makes a valid choice.

(continues)

(continued)



Today's menu is:
Lobster thermidor
Chicken cordon bleu
Linguini and clams
Prime rib
Roasted pepper medley with wild rice
Please make a selection:
Linguini and clams

Excellent choice

Figure 9-19 Typical execution of the `MenuSearch` application

Using the `ArrayList` class

In addition to the `Arrays` class, Java provides an **`ArrayList` class** that can be used to create containers that store lists of objects. The `ArrayList` class provides some advantages over the `Arrays` class. Specifically, an `ArrayList` is **dynamically resizable**, meaning that its size can change during program execution. This means that:

- You can add an item at any point in an `ArrayList` container, and the array size expands automatically to accommodate the new item.
- You can remove an item at any point in an `ArrayList` container, and the array size contracts automatically.

To use the `ArrayList` class, you must use one of the following import statements:

```
import java.util.ArrayList;
import java.util.*;
```

Then, to declare an `ArrayList`, you can use the default constructor, as in the following example:

```
ArrayList names = new ArrayList();
```

The default constructor creates an `ArrayList` with a capacity of 10 items. An `ArrayList`'s **capacity** is the number of items it can hold without having to increase its size. By definition, an `ArrayList`'s capacity is greater than or equal to its size. You can also specify a capacity if you like. For example, the following statement declares an `ArrayList` that can hold 20 names:

```
ArrayList names = new ArrayList(20);
```

If you know you will need more than 10 items at the outset, it is more efficient to create an `ArrayList` with a larger capacity.

Table 9-3 summarizes some useful `ArrayList` methods.

Method	Purpose

`public void add(Object)`
`public void add(int, Object)`
	Adds an item to an `ArrayList`; the default version adds an item at the next available location; an overloaded version allows you to specify a position at which to add the item

`public void remove(int)`	Removes an item from an `ArrayList` at a specified location
`public void set(int, Object)`	Alters an item at a specified `ArrayList` location
`Object get(int)`	Retrieves an item from a specified location in an `ArrayList`
`public int size()`	Returns the current `ArrayList` size

Table 9-3 Useful methods of the `ArrayList` class

In the chapter *Advanced Inheritance Concepts*, you will learn that the `Object` class is the most generic Java class.

To add an item to the end of an `ArrayList`, you can use the `add()` method. For example, to add the name *Abigail* to an `ArrayList` named `names`, you can make the following statement:

```
names.add("Abigail");
```

You can insert an item into a specific position in an `ArrayList` by using an overloaded version of the `add()` method that includes the position. For example, to insert the name *Bob* in the first position of the `names ArrayList`, you use the following statement:

```
names.add(0, "Bob");
```

With each of the methods described in this section, you receive an error message if the position number is invalid for the `ArrayList`.

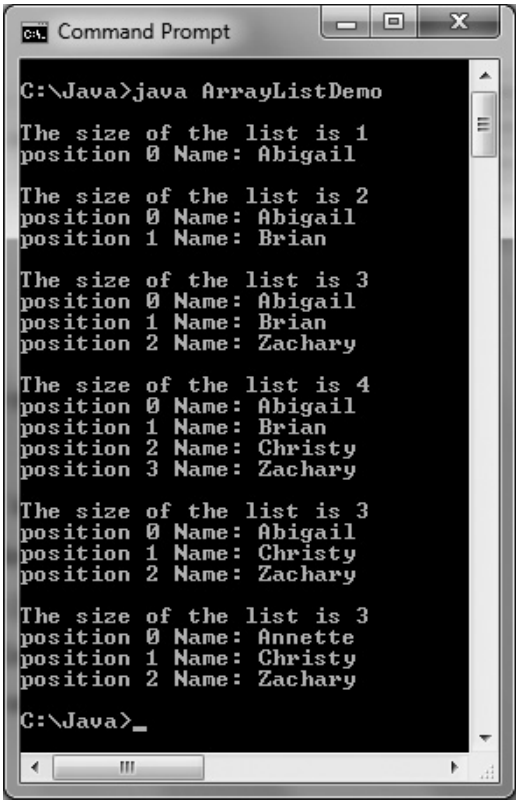
As you can see from Table 9-3, you also can alter and remove items from an `ArrayList`. The `ArrayList` class contains a `size()` method that returns the current size of the `ArrayList`. Figure 9-20 contains a program that demonstrates each of these methods.

```
import java.util.ArrayList;
public class ArrayListDemo
{
   public static void main(String[] args)
   {
      ArrayList names = new ArrayList();
      names.add("Abigail");
      display(names);
      names.add("Brian");
      display(names);
      names.add("Zachary");
      display(names);
      names.add(2, "Christy");
      display(names);
      names.remove(1);
      display(names);
      names.set(0, "Annette");
      display(names);
   }
   public static void display(ArrayList names)
   {
      System.out.println("\nThe size of the list is " + names.size());
      for(int x = 0; x < names.size(); ++x)
         System.out.println("position " + x + " Name: " +
            names.get(x));
   }
}
```

Figure 9-20 The ArrayListDemo program

When you compile the `ArrayListDemo` program, you receive a compiler warning indicating that the program uses unchecked or unsafe operations. You will learn how to eliminate this message in the next section.

In the application in Figure 9-20, an `ArrayList` is created and *Abigail* is added to the list. The `ArrayList` is passed to a `display()` method that displays the current list size and all the names in the list. You can see from the output in Figure 9-21 that at this point, the `ArrayList` size is 1, and the array contains just one name. Examine the program in Figure 9-20 along with the output in Figure 9-21 so that you understand how the `ArrayList` is altered as names are added, removed, and replaced.



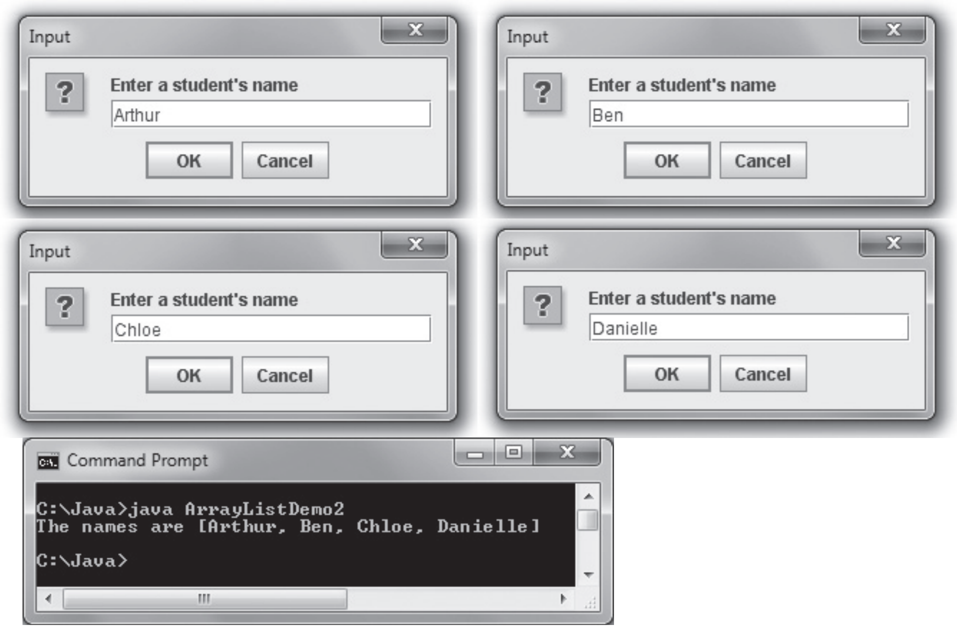
```
C:\Java>java ArrayListDemo

The size of the list is 1
position 0 Name: Abigail

The size of the list is 2
position 0 Name: Abigail
position 1 Name: Brian

The size of the list is 3
position 0 Name: Abigail
position 1 Name: Brian
position 2 Name: Zachary

The size of the list is 4
position 0 Name: Abigail
position 1 Name: Brian
position 2 Name: Christy
position 3 Name: Zachary

The size of the list is 3
position 0 Name: Abigail
position 1 Name: Christy
position 2 Name: Zachary

The size of the list is 3
position 0 Name: Annette
position 1 Name: Christy
position 2 Name: Zachary

C:\Java>_
```

Figure 9-21 Output of the `ArrayListDemo` program

You can display the contents of an `ArrayList` of `Strings` without looping through the values. For example, Figure 9-22 shows an `ArrayList` named `students` that the user populates interactively. Displaying the array name as shown in the shaded statement produces a comma-separated list between square brackets. Figure 9-23 shows a typical execution.

```
import javax.swing.*;
import java.util.ArrayList;
public class ArrayListDemo2
{
   public static void main(String[] args)
   {
      ArrayList students = new ArrayList();
      String name;
      final int LIMIT = 4;
      for(int x = 0; x < LIMIT; ++x)
      {
         name = JOptionPane.showInputDialog(null,
           "Enter a student's name");
         students.add(name);
      }
      System.out.println("The names are " + students);
   }
}
```

Figure 9-22 The ArrayListDemo2 class



```
C:\Java>java ArrayListDemo2
The names are [Arthur, Ben, Chloe, Danielle]

C:\Java>
```

Figure 9-23 Typical execution of the ArrayListDemo2 application

You can achieve similar results using the name for ArrayLists of any class type. In Chapter 7, you learned that every class contains a toString() method that converts its objects to Strings; this method is used when you display an ArrayList's name. However, unless you have overridden the toString() method within a class, the String that is returned by toString() is not very useful. You will learn more about writing this method in the chapter *Advanced Inheritance Concepts*.

You can sort an ArrayList using the Collections.sort() method and providing the ArrayList as the argument—for example:

```
Collections.sort(students);
```

To use this method, you must import the java.util.Collections package at the top of the file.

Understanding the Limitations of the ArrayList Class

An ArrayList can be used to store any type of object reference. In fact, one ArrayList can store multiple types. However, this creates two drawbacks:

- You cannot use an ArrayList to store primitive types such as int, double, or char because those types are not references. If you want to work with primitive types, you can create an array or use the Arrays class, but you cannot use the ArrayList class.
- When you want to store ArrayList elements, you must cast them to the appropriate reference type before you can do so, or you must declare a reference type in the ArrayList declaration.

For example, if you want to declare a String to hold the first name in the names ArrayList, you must make statements such as the following:

```
String firstName;
firstName = (String)names.get(0);
```

The cast operator (String) converts the generic returned object from the get() method to a String. If you do not perform this cast, you receive an error message indicating that you are using incompatible types. (You first learned about the cast operator in Chapter 2.)

You can eliminate the need to perform a cast with ArrayList objects by specifying the type that an ArrayList can hold. For example, you can declare an ArrayList of names as follows:

```
ArrayList<String> names = new ArrayList<String>();
```

Creating an ArrayList declaration with a specified type provides several advantages:

- You no longer have to use the cast operator when retrieving an item from the ArrayList.
- Java checks to make sure that only items of the appropriate type are added to the list.
- The compiler warning that indicates your program uses an unchecked or unsafe operation is eliminated.

TWO TRUTHS & A LIE

Using the `ArrayList` Class

1. An advantage of the `ArrayList` class over the `Arrays` class is that an `ArrayList` is dynamically resizable.
2. An advantage of the `ArrayList` class over the `Arrays` class is that it can hold multiple object types.
3. An advantage of the `ArrayList` class over the `Arrays` class is that it can hold primitive data types such as `int` and `double`.

The false statement is #3. A disadvantage of the `ArrayList` class is that it cannot hold primitive types.

Creating Enumerations

Data types have a specific set of values. For example, in Chapter 2 you learned that a `byte` cannot hold a value larger than 127 and an `int` cannot hold a value larger than 2,147,483,647. You can also create your own data types that have a finite set of legal values. A programmer-created data type with a fixed set of values is an **enumerated data type**.

In Java, you create an enumerated data type in a statement that uses the keyword `enum`, an identifier for the type, and a pair of curly braces that contain a list of the **`enum` constants**, which are the allowed values for the type. For example, the following code creates an enumerated type named `Month` that contains 12 values:

```
enum Month {JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC};
```

By convention, the identifier for an enumerated type begins with an uppercase letter. This convention makes sense because an enumerated type is a class. Also, by convention, the `enum` constants, like other constants, appear in all uppercase letters. The constants are not strings and they are not enclosed in quotes; they are Java identifiers.

After you create an enumerated data type, you can declare variables of that type. For example, you might declare the following:

```
Month birthMonth;
```

You can assign any of the `enum` constants to the variable. Therefore, you can code a statement such as the following:

```
birthMonth = Month.MAY;
```

An enumeration type like `Month` is a class, and its `enum` constants act like objects instantiated from the class, including having access to the methods of the class. These built-in methods include the ones shown in Table 9-4. Each of these methods is nonstatic; that is, each is used with an `enum` object.

Method	Description	Example if `birthMonth = Month.MAY`
`toString()`	The `toString()` method returns the name of the calling constant object.	`birthMonth.toString()` has the value "MAY" You can pass `birthMonth` to `print()` or `println()`, and it is automatically converted to its string equivalent.
`ordinal()`	The `ordinal()` method returns an integer that represents the constant's position in the list of constants. As with arrays, the first position is 0.	`birthMonth.ordinal()` is 4

`equals()`	The `equals()` method returns `true` if its argument is equal to the calling object's value.	`birthMonth.equals(Month.MAY)` is `true`
		`birthMonth.equals(Month.NOV)` is `false`

`compareTo()`	The `compareTo()` method returns a negative integer if the calling object's ordinal value is less than that of the argument, 0 if they are the same, and a positive integer if the calling object's ordinal value is greater than that of the argument.	`birthMonth.compareTo(Month.JUL)` is negative `birthMonth.compareTo(Month.FEB)` is positive `birthMonth.compareTo(Month.MAY)` is 0

Table 9-4 Some useful nonstatic `enum` methods

Several static methods are also available to use with enumerations. These are used with the type and not with the individual constants. Table 9-5 describes two useful static methods.

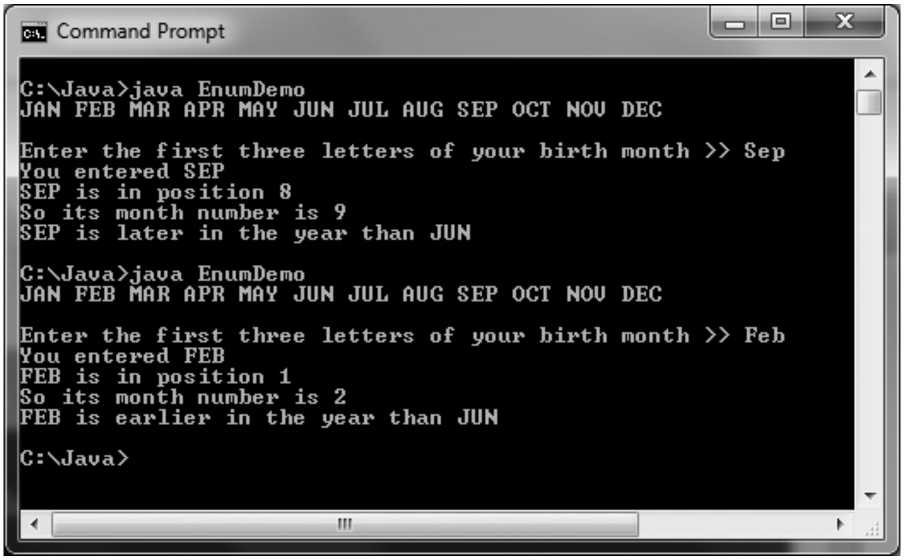
Method	Description	Example with `Month` Enumeration
`valueOf()`	The `valueOf()` method accepts a string parameter and returns an enumeration constant.	`Month.valueOf("DEC")` returns the `DEC` `enum` constant.
`values()`	The `values()` method returns an array of the enumerated constants.	`Month.values()` returns an array with 12 elements that contain the `enum` constants.

Table 9-5 Some static `enum` methods

You can declare an enumerated type in its own file, in which case the filename matches the type name and has a .java extension. You will use this approach in a “You Do It” exercise later in this chapter. Alternatively, you can declare an enumerated type within a class but not within a method. Figure 9-24 is an application that declares a `Month` enumeration and demonstrates its use. Figure 9-25 shows two typical executions.

```
import java.util.Scanner;
public class EnumDemo
{
   enum Month {JAN, FEB, MAR, APR, MAY, JUN,
      JUL, AUG, SEP, OCT, NOV, DEC};

   public static void main(String[] args)
   {
      Month birthMonth;
      String userEntry;
      int position;
      int comparison;
      Scanner input = new Scanner(System.in);
      System.out.println("The months are:");
      for(Month mon : Month.values())
         System.out.print(mon + " ");
      System.out.print("\n\nEnter the first three letters of " +
         "your birth month >> ");
      userEntry = input.nextLine().toUpperCase();
      birthMonth = Month.valueOf(userEntry);
      System.out.println("You entered " + birthMonth);
      position =  birthMonth.ordinal();
      System.out.println(birthMonth + " is in position " + position);
      System.out.println("So its month number is " + (position + 1));
      comparison = birthMonth.compareTo(Month.JUN);
      if(comparison < 0)
          System.out.println(birthMonth +
            " is earlier in the year than " + Month.JUN);
      else
         if(comparison > 0)
            System.out.println(birthMonth +
               " is later in the year than " + Month.JUN);
         else
            System.out.println(birthMonth + " is " + Month.JUN);
   }
}
```

Figure 9-24 The EnumDemo class



```
C:\Java>java EnumDemo
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

Enter the first three letters of your birth month >> Sep
You entered SEP
SEP is in position 8
So its month number is 9
SEP is later in the year than JUN

C:\Java>java EnumDemo
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

Enter the first three letters of your birth month >> Feb
You entered FEB
FEB is in position 1
So its month number is 2
FEB is earlier in the year than JUN

C:\Java>
```

Figure 9-25 Two typical executions of the `EnumDemo` application

In the application in Figure 9-24, a `Month` enumeration is declared; in the `main()` method, a `Month` variable is declared in the first shaded statement. The second shaded statement uses the enhanced `for` loop, which you first learned to use with arrays in Chapter 8. The enhanced `for` loop declares a local `Month` variable named `mon` that takes on the value of each element in the `Month.value()` array in turn so it can be displayed.

In the program in Figure 9-24, the user then is prompted to enter the first three letters for a month, which are converted to their uppercase equivalents. The third shaded statement in the figure uses the `valueOf()` method to convert the user's string to an enumeration value. The fourth shaded statement gets the position of the month in the enumeration list. The last shaded statement compares the entered month to the `JUN` constant. This is followed by an `if` statement that displays whether the user's entered month comes before or after `JUN` in the list or is equivalent to it.

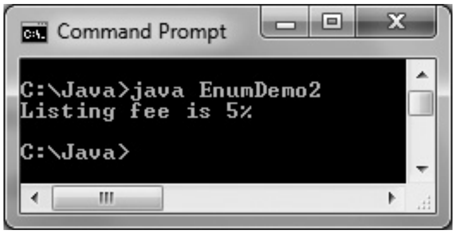
Starting with Java 7, you can use comparison operators with enumeration constants instead of using the `compareTo()` method to return a number. For example, you can write the following:

```
if(birthMonth < Month.JUN)
   System.out.println(birthMonth +
   " is earlier in the year than " + Month.JUN);
```

You can use enumerations to control a switch structure. Figure 9-26 contains a class that declares a `Property` enumeration for a real estate company. The program assigns one of the values to a `Property` variable and then uses a switch structure to display an appropriate message. Figure 9-27 shows the result.

```
import java.util.Scanner;
public class EnumDemo2
{
   enum Property {SINGLE_FAMILY, MULTIPLE_FAMILY,
      CONDOMINIUM, LAND, BUSINESS};
   public static void main(String[] args)
   {
      Property propForSale = Property.MULTIPLE_FAMILY;
      switch(propForSale)
      {
         case SINGLE_FAMILY:
         case MULTIPLE_FAMILY:
            System.out.println("Listing fee is 5%");
            break;
         case CONDOMINIUM:
            System.out.println("Listing fee is 6%");
            break;
         case LAND:
         case BUSINESS:
            System.out.println
               ("We do not handle this type of property");
      }
   }
}
```

Figure 9-26 The EnumDemo2 class



```
C:\Java>java EnumDemo2
Listing fee is 5%

C:\Java>
```

Figure 9-27 Output of the EnumDemo2 application

Creating an enumeration type provides you with several advantages. For example, the `Month` enumeration improves your programs in the following ways:

- If you did not create an enumerated type for month values, you could use another type—for example, `int`s or `String`s. The problem is that any value could be assigned to an `int` or `String` variable, but only the 12 allowed values can be assigned to a `Month`.
- If you did not create an enumerated type for month values, you could create another type to represent months, but invalid behavior could be applied to the values. For example, if you used integers to represent months, you could add, subtract, multiply, or divide two months, which is not logical. Programmers say using `enum`s makes the values type-safe. **Type-safe** describes a data type for which only appropriate behaviors are allowed.

- The `enum` constants provide a form of self-documentation. Someone reading your program might misinterpret what 9 means as a month value, but there is less confusion when you use the identifier `OCT`.
- As with other classes, you can also add methods and other fields to an `enum` type.

Watch the video *Enumerations.*

TWO TRUTHS & A LIE

Creating Enumerations

Assume that you have coded the following:

```
enum Color {RED, WHITE, BLUE};
Color myColor = Color.RED;
```

1. The value of `myColor.ordinal()` is 1.
2. The value of `myColor.compareTo(Color.RED)` is 0.
3. The value of `myColor < Color.WHITE` is `true`.

The false statement is #1. As the first enum constant, the value of `myColor.ordinal()` is 0.

You Do It

Creating Enumerations

In this section, you create two enumerations that hold colors and car model types. You will use them as field types in a `Car` class and write a demonstration program that shows how the enumerations are used.

1. Open a new file in your text editor, and type the following `Color` enumeration:

   ```
   enum Color {BLACK, BLUE, GREEN, RED, WHITE, YELLOW};
   ```

2. Save the file as **Color.java**.

(continues)

(continued)

3. Open a new file in your text editor, and create the following `Model` enumeration:

```
enum Model {SEDAN, CONVERTIBLE, MINIVAN};
```

4. Save the file as **Model.java**. Next, open a new file in your text editor, and start to define a `Car` class that holds three fields: a year, a model, and a color.

```
public class Car
{
   private int year;
   private Model model;
   private Color color;
```

5. Add a constructor for the `Car` class that accepts parameters that hold the values for year, model, and color as follows:

```
public Car(int yr, Model m, Color c)
{
   year = yr;
   model = m;
   color = c;
}
```

6. Add a `display()` method that displays a `Car` object's data, then add a closing curly brace for the class.

```
   public void display()
   {
      System.out.println("Car is a " + year +
         " " + color + " " + model);
   }
}
```

7. Save the file as **Car.java**.

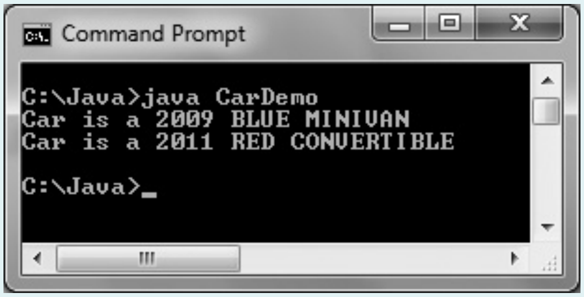
8. Open a new file in your text editor, and write a short demonstration program that instantiates two `Car` objects and assigns values to them using enumeration values for the models and colors.

```
public class CarDemo
{
   public static void main(String[] args)
   {
      Car firstCar = new Car(2009, Model.MINIVAN, Color.BLUE);
      Car secondcar = new Car(2011, Model.CONVERTIBLE,
         Color.RED);
      firstCar.display();
      secondcar.display();
   }
}
```

(continues)

(continued)

9. Save the file as **CarDemo.java**, and then compile and execute it. Figure 9-28 shows that the values are assigned correctly.



```
C:\Java>java CarDemo
Car is a 2009 BLUE MINIVAN
Car is a 2011 RED CONVERTIBLE

C:\Java>_
```

Figure 9-28 Output of the `CarDemo` program

Don’t Do It

- Don’t forget that the first subscript used with a two-dimensional array represents the row and that the second subscript represents the column.
- Don’t try to store primitive data types in an `ArrayList` structure.
- Don’t think `enum` constants are strings; they are not enclosed in quotes.

Key Terms

Sorting is the process of arranging a series of objects in some logical order.

Ascending order describes the order of objects arranged from lowest to highest value.

Descending order describes the order of objects arranged from highest to lowest value.

An **algorithm** is a process or set of steps that solve a problem.

A **bubble sort** is a type of sort in which you continue to compare pairs of items, swapping them if they are out of order, so that the smallest items “bubble” to the top of the list, eventually creating a sorted list.

With an **insertion sort**, you look at each list element one at a time, and if an element is out of order relative to any of the items earlier in the list, you move each earlier item down one position and then insert the tested element.

A **one-dimensional array** or **single-dimensional array** contains one column of values; you access its elements using a single subscript.

Two-dimensional arrays have two or more columns of values, and you must use two subscripts to access an element.

Matrix and **table** are names used for two-dimensional arrays.

A **ragged array** is a two-dimensional array that has rows of different lengths.

Multidimensional arrays contain two or more dimensions.

The Java **`Arrays` class** is a built-in class that contains many useful methods for manipulating arrays, such as methods to search, fill, compare, and sort arrays.

Dummy values are values the user enters that are not "real" data; they are just signals to stop data entry.

The **`ArrayList` class** provides a dynamically resizable container that stores lists of objects.

Dynamically resizable describes an object whose size can change during program execution.

An `ArrayList`'s **capacity** is the number of items it can hold without having to increase its size.

An **enumerated data type** is a programmer-created data type with a fixed set of values.

The **`enum` constants** are the allowed values for an enumerated data type.

Type-safe describes a data type for which only appropriate behaviors are allowed.

Chapter Summary

- Sorting is the process of arranging a series of objects in ascending or descending order. With a bubble sort, you continue to compare pairs of items, swapping them if they are out of order, so that the smallest items "bubble" to the top of the list, eventually creating a sorted list.
- With an insertion sort, you look at each list element one at a time, and if an element is out of order relative to any of the items earlier in the list, you move each earlier item down one position and then insert the tested element.
- You can sort arrays of objects in much the same way that you sort arrays of primitive types. The major difference occurs when you make the comparison that determines whether you want to swap two array elements. When array elements are objects, you usually want to sort based on a particular object field.

- 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. Two-dimensional arrays have both rows and columns. You must use two subscripts when you access an element in a two-dimensional array. To declare a two-dimensional array, you type two sets of brackets after the array type.
- The Java `Arrays` class contains many useful methods for manipulating arrays. These methods provide ways to easily search, compare, fill, and sort arrays.
- The Java `ArrayList` class contains useful methods for manipulating dynamically sized arrays. You can add objects to, remove objects from, and replace objects in `ArrayList` containers.
- A programmer-created data type with a fixed set of values is an enumerated data type. In Java, you create an enumerated data type in a statement that uses the keyword `enum`, an identifier for the type, and a pair of curly braces that contain a list of the `enum` constants, which are the allowed values for the type.

Review Questions

1. When you place objects in order beginning with the object with the highest value, you are sorting in ____________ order.
   a. acquiescing
   b. ascending
   c. demeaning
   d. descending
2. Using a bubble sort involves ____________.
   a. comparing parallel arrays
   b. comparing each array element to the average
   c. comparing each array element to the adjacent array element
   d. swapping every array element with its adjacent element
3. When you use a bubble sort to perform an ascending sort, after the first pass through an array the largest value is ____________.
   a. at the beginning of the list
   b. in the middle of the list
   c. at the end of the list
   d. It is impossible to determine the answer without more information.

4. When you use a bubble sort to perform an ascending sort, after the first pass through an array the smallest value is ____________.

   a. at the beginning of the list
   b. in the middle of the list
   c. at the end of the list
   d. It is impossible to determine the answer without more information.

5. When array elements are objects, you usually want to sort based on a particular ____________ of the object.

   a. field
   b. method
   c. name
   d. type

6. The following defines a ____________ array:

```
int[][]nums={{1, 2}, {3, 4}, {5, 6}};
```

   a. one-dimensional
   b. two-dimensional
   c. three-dimensional
   d. six-dimensional

7. How many rows are contained in the following array?

```
double[][] prices = {{2.56, 3.57, 4.58, 5.59},
                     {12.35, 13.35, 14.35, 15.00}};
```

   a. 1
   b. 2
   c. 4
   d. 8

8. How many columns are contained in the following array?

```
double[][] prices = {{2.56, 3.57, 4.58, 5.59},
                     {12.35, 13.35, 14.35, 15.00}};
```

   a. 1
   b. 2
   c. 4
   d. 8

9. In the following array, what is the value of `code[2][1]`?

```
char[][] code = {{ 'A ', 'D ', 'M '},
                 { 'P ', 'R ', 'S '},
                 { 'U ', 'V ', 'Z '}};
```

   a. `'P'`
   b. `'R'`
   c. `'U'`
   d. `'V'`

10. In the following array, what is the value of `address[1][1]`?

```
String address = {{ "123 Oak ", "345 Elm "},
                  { "87 Maple ", "901 Linden "}};
```

   a. `"123 Oak "`
   b. `"345 Elm "`
   c. `"87 Maple "`
   d. `"901 Linden "`

11. In the following array, what is the value of `fees.length`?

```
double[][] fees = {{3.00, 3.50, 4.00, 5.00},
                   {6.35, 7.35, 8.35, 9.00}};
```

   a. 2
   b. 4
   c. 8
   d. none of the above

12. In the following array, what is the value of `fees[1].length`?

```
double[][] fees = {{3.00, 3.50, 4.00, 5.00},
                   {6.35, 7.35, 8.35, 9.00}};
```

   a. 2
   b. 4
   c. 8
   d. none of the above

13. You place ___________ after the data type in the parameter list of a method that receives a two-dimensional array.

   a. a pair of empty brackets
   b. two pairs of empty brackets
   c. a pair of brackets that contain the number of rows followed by a pair of empty brackets
   d. a pair of empty brackets followed by brackets that contain the number of columns

14. A ___________ array has rows of different lengths.

   a. ragged
   b. jagged
   c. haggard
   d. tattered

15. If the value of `credits[0].length` is not equal to `credits[1].length`, you know `credits` is ___________.

   a. a three-dimensional array
   b. an uninitialized array
   c. a partially populated array
   d. a jagged array

16. Which of the following is true if a successfully running program contains the following statement:

    `Arrays.fill(tax, 10);`

    a. `tax` is a two-dimensional array.
    b. `fill()` is a nonstatic method.
    c. `tax` is an array with 10 elements.
    d. none of the above

17. Which of the following is a requirement when you use a binary search method with an array?

    a. The array must be numeric.
    b. The array must have been sorted in ascending order.
    c. The array must have at least three elements.
    d. none of the above

18. The chief advantage to using the `ArrayList` class instead of the `Arrays` class is that an `ArrayList` ___________.

    a. can be much larger
    b. is easier to search
    c. is dynamically resizable
    d. can be used as an argument to a `static` method

19. The chief disadvantage to using the `ArrayList` class instead of the `Arrays` class is that an `ArrayList` ___________.

    a. cannot be sorted
    b. cannot store primitive data types
    c. cannot be accessed using subscripts
    d. All of the above are disadvantages to using an `ArrayList`.

20. An advantage to using an enumerated data type is ___________.

    a. errors are reduced because only a limited set of values can be used with the type
    b. time is saved because programs with enumerated types compile faster
    c. coding time is reduced because enumerated types are created automatically by the compiler
    d. All of the above are true.

Exercises

Programming Exercises

1. Write an application containing an array of 10 `String` values, and display them in ascending order. Save the file as **SortStrings.java**.

2. a. The mean of a list of numbers is its arithmetic average. The median of a list is its middle value when the values are placed in order. For example, if a list contains 1, 4, 7, 8, and 10, then the mean is 6 and the median is 7. Write an application that allows you to enter five integers and displays the values, their mean, and their median. Save the file as **MeanMedian.java**.

   b. Revise the `MeanMedian` class so that the user can enter any number of values up to 20. If the list has an even number of values, the median is the numeric average of the values in the two middle positions. Save the file as **MeanMedian2.java**.

3. a. Radio station JAVA wants a class to keep track of recordings it plays. Create a class named `Recording` that contains fields to hold methods for setting and getting a `Recording`'s title, artist, and playing time in seconds. Save the file as **Recording.java**.

   b. Write an application that instantiates five `Recording` objects and prompts the user for values for the data fields. Then prompt the user to enter which field the `Recordings` should be sorted by—song title, artist, or playing time. Perform the requested sort procedure, and display the `Recording` objects. Save the file as **RecordingSort.java**.

4. In Chapter 8, you created a `Salesperson` class with fields for an ID number and sales values. Now, create an application that allows a user to enter values for an array of seven `Salesperson` objects. Offer the user the choice of displaying the objects in order by either ID number or sales value. Save the application as **SalespersonSort.java**.

5. In Chapter 8, you created a `Salesperson` class with fields for an ID number and sales values. Now, create an application that allows you to store an array that acts as a database of any number of `Salesperson` objects up to 20. While the user decides to continue, offer three options: to add a record to the database, to delete a record from the database, or to change a record in the database. Then proceed as follows:

   - If the user selects the add option, issue an error message if the database is full. Otherwise, prompt the user for an ID number. If the ID number already exists in the database, issue an error message. Otherwise, prompt the user for a sales value, and add the new record to the database.

- If the user selects the delete option, issue an error message if the database is empty. Otherwise, prompt the user for an ID number. If the ID number does not exist, issue an error message. Otherwise, do not access the record for any future processing.
- If the user selects the change option, issue an error message if the database is empty. Otherwise, prompt the user for an ID number. If the requested record does not exist, issue an error message. Otherwise, prompt the user for a new sales value, and change the sales value for the record.

After each option executes, display the updated database in ascending order by `Salesperson` ID number, and prompt the user to select the next action. Save the application as **SalespersonDatabase.java**.

6. Write an application that stores at least four different course names and meeting days and times in a two-dimensional array. Allow the user to enter a course name (such as "CIS 110"), and display the day of the week and time that the course is held (such as "Th 3:30"). If the course does not exist, display an error message. Save the file as **Schedule.java**.

7. a. Table 9-6 shows the various services offered by a hair salon, including its prices and times required:

Service Description	Price ($)	Time (Minutes)
Cut	8.00	15
Shampoo	4.00	10
Manicure	18.00	30
Style	48.00	55
Permanent	18.00	35
Trim	6.00	5

Table 9-6 Salon services, prices, and times

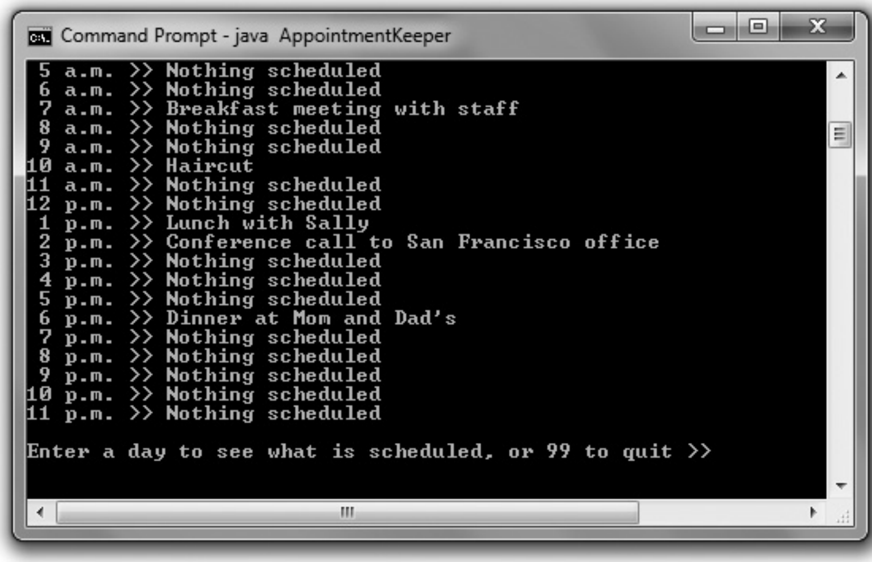
Create a class that holds the service description, price, and number of minutes it takes to perform the service. Include a constructor that requires arguments for all three data fields and three get methods that each return one of the data field's values. Save the class as **Service.java**.

b. Write an application named `SalonReport` that contains an array to hold six `Service` objects, and fill it with the data from Table 9-6. Include methods to sort the array in ascending order by each of the data fields. Prompt the user for the preferred sort order, and display the list of services in the requested order. Save the program as **SalonReport.java**.

8. Create an application that contains an enumeration that represents the days of the week. Display a list of the days, then prompt the user for a day. Display business hours for the chosen day. Assume that the business is open from 11 to 5 on Sunday, 9 to 9 on weekdays, and 10 to 6 on Saturday. Save the file as **DayOfWeek.java**.

9. Create a class named `Majors` that includes an enumeration for the six majors offered by a college as follows: ACC, CHEM, CIS, ENG, HIS, PHYS. Display the enumeration values for the user, then prompt the user to enter a major. Display the college division in which the major falls. ACC and CIS are in the Business Division, CHEM and PHYS are in the Science Division, and ENG and HIS are in the Humanities Division. Save the file as **Majors.java**.

10. Create a class named `AppointmentKeeper` that stores a two-dimensional `String` array with 31 rows and 24 columns. The program allows a user to choose a day and an hour continuously and then to enter a `String` value that describes an appointment. After the user finishes entering data, prompt the user continuously to enter a day, and show the 24 appointment slots for the day. Save the file as **AppointmentKeeper.java**. Figure 9-29 shows part of a typical execution.



```
 5 a.m. >> Nothing scheduled
 6 a.m. >> Nothing scheduled
 7 a.m. >> Breakfast meeting with staff
 8 a.m. >> Nothing scheduled
 9 a.m. >> Nothing scheduled
10 a.m. >> Haircut
11 a.m. >> Nothing scheduled
12 p.m. >> Nothing scheduled
 1 p.m. >> Lunch with Sally
 2 p.m. >> Conference call to San Francisco office
 3 p.m. >> Nothing scheduled
 4 p.m. >> Nothing scheduled
 5 p.m. >> Nothing scheduled
 6 p.m. >> Dinner at Mom and Dad's
 7 p.m. >> Nothing scheduled
 8 p.m. >> Nothing scheduled
 9 p.m. >> Nothing scheduled
10 p.m. >> Nothing scheduled
11 p.m. >> Nothing scheduled

Enter a day to see what is scheduled, or 99 to quit >>
```

Figure 9-29 Part of a typical execution of the `AppointmentKeeper` program

Debugging Exercises

1. Each of the following files in the Chapter09 folder in 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, DebugNine1.java will become FixDebugNine1.java.
   a. DebugNine1.java
   b. DebugNine2.java
   c. DebugNine3.java
   d. DebugNine4.java
   e. Movie.java, which is used by DebugNine2.java.

Game Zone

1. Create a guessing game called "Does It Have Legs?", in which you think of an animal and the computer tries to guess what it is. The game continually becomes "smarter" as you play because you teach it how to become more specific with its questions. For example, a typical game session might proceed as follows:

   Computer: Does the animal you are thinking of have legs?

   Player: Yes.

   Computer: Is it a dog?

   Player: Yes.

   Computer: Yay! I win. Do you want to play again?

   Player: Yes.

   Computer: Does the animal you are thinking of have legs?

   Player: Yes.

   Computer: Is it a dog?

   Player: No.

   Computer: I give up. What was your animal?

   Player: Horse.

   Computer: Type a question for which the answer is Yes for Dog but No for Horse.

   Player: Does it live in the house?

   Computer: Do you want to play again?

   Player: Yes.

   Computer: Does the animal you are thinking of have legs?

   Player: Yes.

Computer: Does it live in the house?

Player: No.

Computer: Is it a Horse?

Player: No.

Computer: I give up. What was your animal?

Player: Cow.

Computer: Type a question for which the answer is Yes for Horse but No for Cow.

Player: Do people ride it?

Computer: Do you want to play again?

Player: Yes.

Computer: Does the animal you are thinking of have legs?

Player: Yes.

Computer: Does it live in the house?

Player: No.

Computer: Do people ride it?

Player: No.

Computer: Is it a Cow?

Player: Yes.

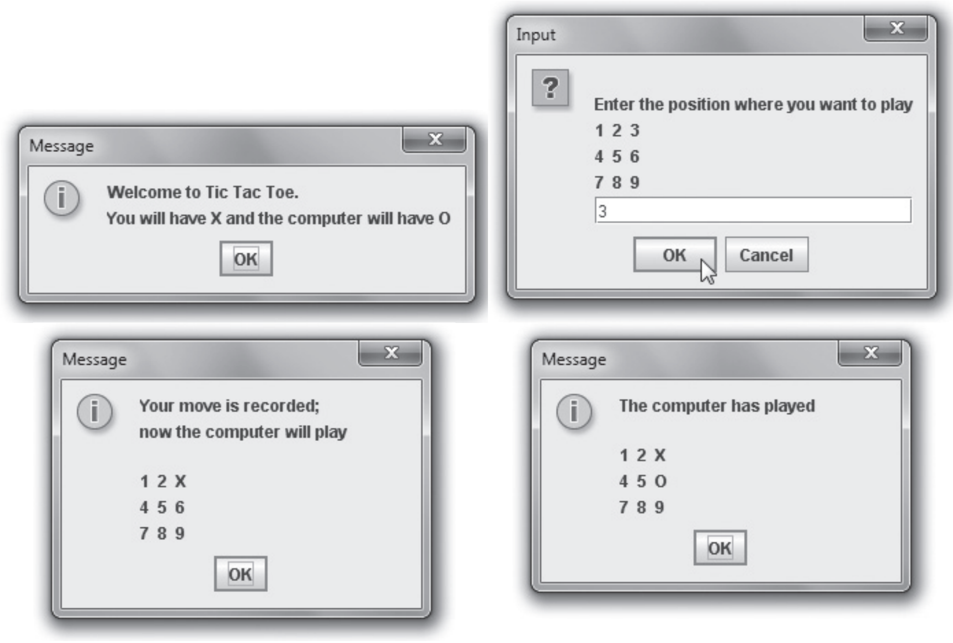
Computer: Yay! I win. Do you want to play again?

Player: No.

(*Hint*: You might want to look up "binary trees" on the Internet as a method to organize this application.)

Save the application as **DoesItHaveLegs.java**.

2. a. Create a Tic-Tac-Toe game. In this game, two players alternate placing *X*s and *O*s into a grid until one player has three matching symbols in a row, either horizontally, vertically, or diagonally. Create a game in which the user is presented with a three-by-three grid containing the digits 1 through 9. When the user chooses a position by typing a number, place an *X* in the appropriate spot. Generate a random number for the position where the computer will place an *O*. Do not allow the player or the computer to place a symbol where one has already been placed. Figure 9-30 shows the first four windows in a typical game. When either the player or computer has three symbols in a row, declare a winner; if all positions have been exhausted and no one has three symbols in a row, declare a tie. Save the game as **TicTacToe.java**.



```
Message
Welcome to Tic Tac Toe.
You will have X and the computer will have O
```

```
Input
Enter the position where you want to play
1 2 3
4 5 6
7 8 9
3
```

```
Message
Your move is recorded;
now the computer will play

1 2 X
4 5 6
7 8 9
```

```
Message
The computer has played

1 2 X
4 5 O
7 8 9
```

Figure 9-30 Typical game of `TicTacToe` in progress

b. In the `TicTacToe` application, the computer's selection is chosen randomly. Improve the `TicTacToe` game so that when the computer has two *O*s in any row, column, or diagonal, it selects the winning position for its next move rather than selecting a position randomly. Save the improved game as **TicTacToe2.java**.

3. In Chapter 8, you created an application class named `FullDeck` that implemented a 52-element array that represented each card in a standard deck of playing cards. Now, create an enumeration that holds the four suits: `SPADES`, `HEARTS`, `DIAMONDS`, and `CLUBS`. Save the enumeration in a file named **Suit.java**. Modify the `Card` class from Chapter 8 to use the enumeration, and save the class as **Card2.java**. Modify the `FullDeck` application to use the new `Card` class, and save the application as **FullDeck2.java**.

4. In Chapter 7, you improved a Rock Paper Scissors game played between a user and the computer. Add an enumeration that holds three values that represent `ROCK`, `PAPER`, and `SCISSORS`, and use it for all comparisons in the program. Save the file as **RockPaperScissors3.java**.

Case Problems

1. In the last chapter, you modified the `EventDemo` program for Carly's Catering to accept and display data for an array of three `Event` objects. Now, modify the program to use an array of eight `Event` objects. Prompt the user to choose an option to sort `Events` in ascending order by event number, number of guests, or event type. Display the sorted list and continue to prompt the user for sorting options until the user enters a sentinel value. Save the file as **EventDemo.java**.
2. In the last chapter, you modified the `RentalDemo` program for Sammy's Seashore Supplies to accept and display data for an array of three `Rental` objects. Now, modify the program to use an array of eight `Rental` objects. Prompt the user to choose an option to sort `Rentals` in ascending order by contract number, price, or equipment type. Display the sorted list and continue to prompt the user for sorting options until the user enters a sentinel value. Save the file as **RentalDemo.java**.