ARRAYS, RECURSION, AND COMPLEXITY

Unit 3

Estimated Time for Unit: 14.5 hours

CHAPTER 10

Introduction to Arrays

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Write programs that handle collections of similar items.
- Declare array variables and instantiate array objects.
- Manipulate arrays with loops, including the enhanced `for` loop.
- Write methods to manipulate arrays.
- Create parallel arrays and two-dimensional arrays.

Estimated Time: 5 hours

VOCABULARY

Array
Element
Enhanced `for` loop
Index
Initializer list
Logical size
Parallel arrays
Physical size
Procedural decomposition
Range-bound error
Structure chart
Subscript

There are situations in which programs need to manipulate many similar items, a task that would be extremely awkward using the language features encountered so far. Earlier we developed a `Student` class with a name and three test scores. Each test score required a separate instance variable. Imagine how tedious and lengthy the code would have become if a student had 20 scores. Fortunately, there is a way to handle this dilemma. Most programming languages, including Java, provide a data structure called an ***array***, which consists of an ordered collection of similar items. An array, as a whole, has a single name, and the items in an array are referred to in terms of their position within the array. This chapter explains the mechanics of declaring arrays and several basic algorithms for manipulating them. Using an array, it is as easy to manipulate a million test scores as it is three.

10.1 Conceptual Overview

To demonstrate the need for arrays, let us consider the data for a `Student` class if there are no arrays, but there are 20 rather than 3 test scores. The declarations for the instance variables look like this:

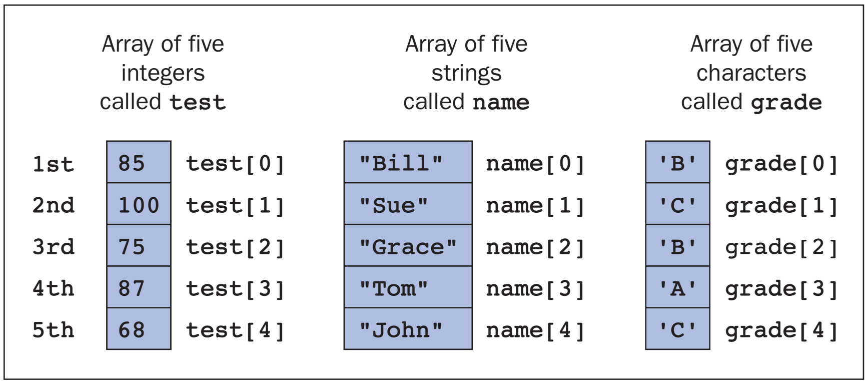
```
private String name;
private int test1,  test2,  test3,  test4,  test5,
            test6,  test7,  test8,  test9,  test10,
            test11, test12, test13, test14, test15,
            test16, test17, test18, test19, test20;
```

and the computation of the average score looks like this:

```
// Compute and return a student's average
public int getAverage(){
    int average;
    average = (test1 + test2  + test3  + test4  + test5 +
               test6  + test7  + test8  + test9  + test10 +
               test11 + test12 + test13 + test14 + test15 +
               test16 + test17 + test18 + test19 + test20) / 20;
    return average;
}
```

Other methods are affected in a similar manner; however, arrays restore sanity to the situation. The items in an array are called ***elements***, and for any particular array, all of the elements must be of the same type. The type can be any primitive or reference type. For instance, we can have an array of test scores, an array of names, or even an array of student objects. Figure 10-1 illustrates these ideas. In the figure, each array contains five elements, or has a *length* of five. The first element in the array `test` is referred to as `test[0]`, the second as `test[1]`, and so on. Here we encounter Java's convention of numbering starting with 0 rather than 1, a convention that is guaranteed to cause us grief whenever we accidentally revert to our lifelong habit of counting from 1. Thus, the elements in an array of length 100 are numbered from 0 to 99. An item's position within an array is called its ***index*** or ***subscript***. In Figure 10-1, the array indexes appear within square brackets (`[]`).

FIGURE 10-1
Three arrays, each containing five elements



EXERCISE 10.1

1. A program needs many variables to store and process data. How does an array solve this problem?
2. How does the programmer access an item in an array?
3. Mary is using an array of doubles to store an employee's wage amounts for each day of the week (Monday through Friday). Draw a picture of this array with sample items and references to each one.

10.2 Simple Array Manipulations

The mechanics of manipulating arrays are fairly straightforward, as illustrated in the following segments of code. First, we declare and instantiate an array of 500 integer values. (Section 10.4 discusses array declarations in greater detail.) By default, all of the values are initialized to 0:

```
int[] abc = new int[500];
```

Next, we declare some other variables:

```
int i = 3;
int temp;
double avFirstFive;
```

The basic syntax for referring to an array element has the form

```
<array name>[<index>]
```

where `<index>` must be between 0 and the array's length less 1. The subscript operator (`[]`) has the same precedence as the method selector (`.`). To illustrate, we assign values to the first five elements:

```
abc[0] = 78;                     // 1st element 78
abc[1] = 66;                     // 2nd element 66
abc[2] = (abc[0] + abc[1]) / 2;  // 3rd element average of first two
abc[i] = 82;                     // 4th element 82 because i is 3
abc[i + 1] = 94;                 // 5th element 94 because i + 1 is 4
```

When assigning a value to the 500th element, we must remember that its index is 499, not 500:

```
abc[499] = 76;                   // 500th element 76
```

Fortunately, the JVM checks the values of subscripts before using them and throws an `ArrayIndexOutOfBoundsException` if they are out of bounds (less than 0 or greater than the array length less 1). The detection of a ***range-bound error*** is similar to the JVM's behavior when a program attempts to divide by 0.

In our present example, subscripts must be between 0 and 499. Later in the chapter, we show how to work with arrays of any size and how to write loops that are not tied to a literal value (in this case, 500).

```
abc[-1] = 74;                 // NO! NO! NO! Out of bounds
abc[500] = 88;                // NO! NO! NO! Out of bounds
```

To compute the average of the first five elements, we could write

```
avFirstFive = (abc[0] + abc[1] + abc[2] + abc[3] + abc[4])/5;
```

It often happens that we need to interchange elements in an array. To demonstrate, following is code that interchanges any two adjacent elements:

```
// Initializations
. . .
abc[3] = 82;
abc[4] = 95;
i = 3;
. . .

// Interchange adjacent elements
temp = abc[i];                    // temp       now equals 82
abc[i] = abc[i + 1];              // abc[i]     now equals 95
abc[i + 1] = temp;                // abc[i + 1] now equals 82
```

We frequently need to know an array's length, but we do not have to remember it. The array itself makes this information available by means of a public instance variable called `length`:

```
System.out.println ("The size of abc is: " + abc.length);
```

EXERCISE 10.2

1. Assume that the array `a` contains the five integers 34, 23, 67, 89, and 12. Write the values of the following expressions:

   a. a[1]

   b. a[a.length – 1]

   c. a[2] + a[3]

2. What happens when a program attempts to access an item at an index that is less than 0 or greater than or equal to the array's length?

10.3 Looping Through Arrays

There are many situations in which it is necessary to write a loop that iterates through an array one element at a time. This process is also called a *traversal*. Following are some examples based on the array abc of 500 integers. Later in this section, we show how to work with arrays of any size and how to write loops that are not tied to a literal value (in this case, 500).

Sum the Elements

The following is code that sums the numbers in the array abc. Each time through the loop we add a different element to the sum. On the first iteration we add abc[0] and on the last abc[499].

```
int sum;
sum = 0;
for (int i = 0; i < 500; i++)
   sum += abc[i];
```

Count the Occurrences

We can determine how many times a number x occurs in the array by comparing x to each element and incrementing count every time there is a match:

```
int x;
int count;
x = ...;                             // Assign some value to x
count = 0;
for (int i = 0; i < 500; i++){
   if (abc[i] == x)
      count++;                       // Found another element equal to x
}
```

Determine Presence or Absence

To determine if a particular number is present in the array, we could count the occurrences; alternatively, we could save time by breaking out of the loop as soon as the first match is found. The following is code based on this idea. The Boolean variable found indicates the outcome of the search:

```
int x;
boolean found;
x = ...;
found = false;               // Initially assume x is not present
for (int i = 0; i < 500; i++){
   if (abc[i] == x){
      found = true;
      break;                 // No point in continuing once x is found
   }                         // so break out of the loop
}
if (found)
```

```
   System.out.println("Found");
else
   System.out.println("Not Found");
```

Determine First Location

As a variation on the preceding example, we now show how to find the first location of `x` in the array. The variable `loc` initially equals `-1`, meaning that we have not found `x` yet. We then iterate through the array, comparing each element to `x`. As soon as we find a match, we set `loc` to the location and break out of the loop. If `x` is not found, `loc` remains equal to `-1`.

```
int x;
int loc;
x = ...;
loc = -1;
for (int i = 0; i < 500; i++){
   if (abc[i] == x){
      loc = i;
      break;
   }
}
if (loc == -1)
   System.out.println("Not Found");
else
   System.out.println("Found at index " + loc);
```

Working with Arrays of Any Size

The examples in this section have assumed that the array contains 500 elements. It is possible and also desirable to write similar code that works with arrays of any size, however. We simply replace the literal 500 with a reference to the array's instance variable `length` in each of the loops. For example, this code would sum the integers in an array of any size:

```
int sum;
sum = 0;
for (int i = 0; i < abc.length; i++)
   sum += abc[i];
```

EXERCISE 10.3

1. Write a loop that prints all of the items in an array `a` to the terminal screen.

2. Repeat Question 1 but print the items in reverse order.

3. Write a loop that locates the first occurrence of a negative integer in an array `a`. When the loop is finished, the variable `index` should contain the index of the negative number or the length of the array if there were no negative numbers in the array.

EXERCISE 10.3 Continued

4. Describe what the following code segments do:

   a.

   ```
   for (int i = 0; i < a.length; i++)
      a[i] = Math.abs(a[i]);
   ```

   b.

   ```
   String str = "";
   for (int i = 0; i < a.length; i++)
      str += a[i];
   ```

5. What is the advantage of using the instance variable `length` in a loop with an array?

10.4 Declaring Arrays

Earlier, we declared an array of 500 integers as follows:

```
int[] abc = new int[500];
```

In doing so, we combined two separate statements:

```
int[] abc;               // Declare abc to be a variable that can
                         // reference an array of integers.
abc = new int[500];      // Instantiate an array of 500 integers for abc to
                         // reference.
```

Arrays are objects and must be instantiated before being used. Several array variables can be declared in a single statement like this:

```
int[] abc, xyz;
abc = new int[500];
xyz = new int[10];
```

or like this:

```
int[] abc = new int[500], xyz = new int[10];
```

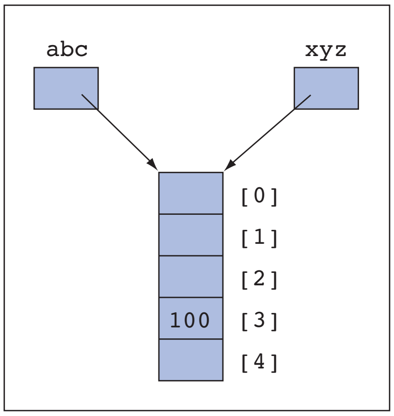
Array variables are `null` before they are assigned array objects. Failure to assign an array object can result in a null pointer exception, as shown in the next code segment:

```
int[] abc;
abc[1] = 10;    // run-time error: null pointer exception
```

Because arrays are objects, two variables can refer to the same array, as indicated in Figure 10-2 and the next segment of code:

```
int[] abc, xyz;
abc = new int[5];                   // Instantiate an array of five
                                    // integers
xyz = abc;                          // xyz and abc refer to the same array
xyz[3] = 100;                       // Changing xyz changes abc as well.
System.out.println (abc[3]);        // 100 is displayed.
```

FIGURE 10-2
Two variables can refer to the same array object



If we want abc and xyz to refer to two separate arrays that happen to contain the same values, we could copy all of the elements from one array to the other, as follows:

```
int[] abc, xyz;             // Declare two array variables
int i;
abc = new int[10];          // Instantiate an array of size 10
for (i = 0; i < 10; i++)    // Initialize the array
   abc[i] = i*i;            // a[0]=0 and a[1]=1 and a[2]=4, etc.

xyz = new int[10];          // Instantiate another array of size 10
for (i = 0; i < 10; i++)    // Initialize the second array
   xyz[i] = abc[i];
```

Also, because arrays are objects, Java's garbage collector sweeps them away when they are no longer referenced:

```
int[] abc, xyz;
abc = new int[10];     // Instantiate an array of 10 integers.
xyz = new int[5];      // Instantiate an array of 5 integers.
xyz = null;            // The array of 5 integers is no longer referenced
                       // so the garbage collector will sweep it away.
```

Arrays can be declared, instantiated, and initialized in one step. The list of numbers between the braces is called an ***initializer list***.

```
int[] abc = {1,2,3,4,5}; // abc now references an array of five integers.
```

As mentioned at the outset, arrays can be formed from any collection of similar items. Following then are arrays of doubles, characters, Booleans, strings, and students:

```
double[]   ddd = new double[10];
char[]     ccc = new char[10];
boolean[]  bbb = new boolean[10];
String[]   ggg = new String[10];
Student[]  sss = new Student[10];
String     str;

ddd[5] = 3.14;
ccc[5] = 'Z';
bbb[5] = true;
ggg[5] = "The cat sat on the mat.";
sss[5] = new Student();

sss[5].setName ("Bill");
str = sss[5].getName() + ggg[5].substring(7);
   // str now equals "Bill sat on the mat."
```

There is one more way to declare array variables, but its use can be confusing. Here it is:

```
int aaa[];                // aaa is an array variable.
```

That does not look confusing, but what about this?

```
int aaa[], bbb, ccc[];    // aaa and ccc are array variables.
                          // bbb is not. This fact might go unnoticed.
```

Instead, it might be better to write:

```
int[] aaa, ccc;           // aaa and ccc are array variables
int bbb;                  // bbb is not. This fact is obvious.
```

Once an array is instantiated, its size cannot be changed, so make sure the array is large enough from the outset.

EXERCISE 10.4

1. Declare and instantiate array variables for the following data:

a. An array of 15 doubles

b. An array of 20 strings

2. What is an initializer list?

EXERCISE 10.4 Continued

3. Use an initializer list to create the following arrays:
   a. five test scores of 100, 90, 75, 60, and 88
   b. three interest rates of 0.12, 0.05, and 0.15
   c. two strings, your first name and last name
4. Why is it better to use the form `<type>[] <variable>` instead of `<type> <variable>[]` when declaring an array variable?

10.5 Working with Arrays That Are Not Full

When an array is instantiated, the computer automatically fills its cells with default values. For example, each cell in an array of `int` initially contains the value 0. The application then replaces these values with new ones as needed. An application might not use all the cells available in an array, however. For example, one might create an array of 20 `int`s but receive only 5 `int`s from interactive input. This array has a ***physical size*** of 20 cells but a ***logical size*** of 5 cells currently used by the application. From the application's perspective, the remaining 15 cells contain garbage. Clearly, the application should only access the first five cells when asked to display the data, so using the array's physical size as an upper bound on a loop will not do. We solve this problem by tracking the array's logical size with a separate integer variable. The following code segment shows the initial state of an array and its logical size:

```
int[] abc = new int[50];
int size = 0;
```

Note that `abc.length` (the physical size) is 50, whereas `size` (the logical size) is 0.

Processing Elements in an Array That Is Not Full

In Section 10.3, we showed how to generalize a loop to process all the data in an array of any size. The loop accesses each cell from position 0 to position `length - 1`, where `length` is the array's instance variable. When the array is not full, one must replace the array's physical length with its logical size in the loop. Following is the code for computing the sum of the integers currently available in the array `abc`:

```
   int[] abc = new int[50];
   int size = 0;

... code that puts values into some initial portion of the array and sets
    the value of size ...

   int sum = 0;
   for (int i = 0; i < size; i++)
      sum += abc[i];
```

Adding Elements to an Array

The simplest way to add a data element to an array is to place it after the last available item. One must first check to see if there is a cell available and then remember to increment the array's logical size. The following code shows how to add an integer to the end of array `abc`:

```
if (size < abc.length){
   abc[size] = anInt;
   size++;
}
```

When `size` equals `abc.length`, the array is full. The `if` statement prevents a range error from occurring. Remember that Java arrays are of fixed size when they are instantiated, so eventually they become full. We examine a way of skirting this limitation of arrays in Chapter 14.

We can also insert an element at an earlier position in the array. This process requires a shifting of other elements and is presented in Chapter 12.

Removing Elements from an Array

Removing a data element from the end of an array requires no change to the array itself. We simply decrement the logical size, thus preventing the application from accessing the garbage elements beyond that point. (Removing a data element from an arbitrary position is discussed in Chapter 12.)

Arrays and Text Files

We conclude this section with two short programs that use arrays in conjunction with text files. The first program prompts the user for integers and inserts them at the logical end of an array. This process stops when the user enters –1 as input or when the array becomes full. The program then outputs the contents of the array to a text file. Here is the code:

```
// Example 10.1: Input numbers from the keyboard into an array and
// output the array's numbers to a text file

import java.io.*;
import java.util.Scanner;

public class ArrayToFile{

   public static void main(String[] args) throws IOException {
      // Set up scanner and array
      Scanner reader = new Scanner(System.in);
      int[] array = new int[10];
      int count = 0;

      // Input numbers until full or user enters -1
      while (count < array.length){
         System.out.print("Enter a number (-1 to quit): ");
         int number = reader.nextInt();
         if (number == -1)
            break;
         array[count] = number;
         count++;
      }
      // Output the numbers to a text file
      PrintWriter writer = new PrintWriter(new File("numbers.txt"));
      for (int i = 0; i < count; i++)
         writer.println(array[i]);
      writer.close();
   }
}
```

The critical variable in this first example is `count`. This variable tracks the number of numbers input as well as the logical size of the array. The output loop also uses `count` to test for the logical end of the array.

Our second example program reads numbers from a text file, inserts them into an array, and then displays them in the terminal window. If there is not enough room in the array, an error message is also displayed.

```
// Example 10.2: Input numbers from a file into an array and
// output the array's numbers to the terminal window

import java.io.*;
import java.util.Scanner;

public class FileToArray{

   public static void main(String[] args) throws IOException {
      // Set up scanner and array
      Scanner reader = new Scanner(new File("numbers.txt"));
      int[] array = new int[10];
      int count = 0;

      // Input numbers until full or end of file is reached
      while (count < array.length  && reader.hasNext()){
         int number = reader.nextInt();
         array[count] = number;
         count++;
      }

      // Output the numbers to the terminal window
      for (int i = 0; i < count; i++)
         System.out.println(array[i]);

      // Display error message if not all data are read from file
      if (reader.hasNext())
         System.out.println("Some data lost during input");
   }
}
```

Once again, the critical variable is `count`, which tracks the number of integers input from the file and the number of integers stored in the array. Arrays that receive data from input are often only partially filled. In extreme cases, an array might not be large enough to hold the number of inputs offered. We examine techniques for dealing with this problem in Chapter 14.

EXERCISE 10.5

1. What happens when the programmer tries to access an array cell whose index is greater than or equal to its logical size?
2. Describe an application that uses an array that might not be full.

10.6 Parallel Arrays

There are situations in which it is convenient to declare what are called ***parallel arrays***. Suppose we want to keep a list of people's names and ages. This can be achieved by using two arrays in which corresponding elements are related. For instance

```
String[] name = {"Bill", "Sue", "Shawn", "Mary", "Ann"};
int[]    age  = {20    , 21   , 19     , 24    , 20};
```

Thus, Bill's age is 20 and Mary's is 24. Note that related items have the same index. There are many other uses for parallel arrays, but continuing with our present example, the following is a segment of code that finds the age of a particular person:

```
String searchName;
int correspondingAge = -1;

searchName = ...;                      // Set this to the desired name
for (int i = 0; i < name.length; i++){ // name.length is the array's size
   if (searchName.equals (name[i])){
      correspondingAge = age[i];
      break;
   }
}

if (correspondingAge == -1)
   System.out.println(searchName + " not found.");
else
   System.out.println("The age is " + correspondingAge);
```

In this example, the parallel arrays are both full and the loops use the instance variable `length`. When the arrays are not full, the code will need an extra variable to track their logical sizes, as discussed earlier.

EXERCISE 10.6

1. What are parallel arrays?
2. Describe an application in which parallel arrays might be used.
3. Declare and instantiate the variables for parallel arrays to track the names, ages, and Social Security numbers of 50 employees.

EXERCISE 10.6 Continued

4. Assume that the array `names` contains the names of people in a phone book and the parallel array `numbers` contains their phone numbers. Write a code segment that displays each name and number in formatted columns (using the method `printf` introduced in Chapter 8). Names should be left-justified in a width of 20 columns. You may assume that each number is the same length.

5. Write a code segment that creates parallel arrays containing the first 10 nonnegative powers of 2. One array should contain the exponent and the other array should contain 2 raised to that power.

10.7 Using the Enhanced `for` Loop

Most of the loops used with arrays follow common patterns, such as visiting every element from the first position to the last one, or visiting elements until a particular element is found. These loops require the use of an index variable, which is initialized to the first position. They must then test the index for less than the last position to continue. Finally, they must increment the index on each pass. Java, version 5.0 or higher, provides an ***enhanced*** **`for`** ***loop*** that frees the programmer from managing these details. This type of loop visits each element in an array from the first position to the last position. On each pass through the loop, the element at the current position is automatically assigned to a temporary variable. No other loop control information is required.

The syntax of the enhanced `for` loop is much simpler than that of the standard `for` loop:

```
for (<temporary variable declaration> : <array object>)
   <statement>
```

The type of the temporary variable must be compatible with the element type of the array.

To see how the enhanced `for` loop can simplify code, let's modify two earlier examples where we computed the sum of an array of integers and the sum of a two-dimensional array of integers. We place the revised code segments in a short tester program, which should display the results 9 and 27 for the two arrays:

```
// Example 10.3: Testing the enhanced for loop

public class TestForLoop{

   public static void main(String[] args){

      // Sum the elements in a one-dimensional array
      int[] abc = {2, 3, 4};
      int sum = 0;
      for (int element : abc)
         sum += element;
      System.out.println("First sum: " + sum);
```

```
         // Sum the elements in a two-dimensional array
         int[][] table = {{2, 3, 4}, {2, 3, 4}, {2, 3, 4}};
         sum = 0;
         for (int[]row : table)
            for (int element : row)
               sum += element;
         System.out.println("Second sum: " + sum);
      }
   }
```

On each pass through the first loop, the integer at the current position in the array `abc` is automatically assigned to the temporary variable `element`. On each pass through the second loop, the two-dimensional array's current row is assigned to the temporary variable `row`. The nested loop then iterates through this array, assigning each integer at the current column position to another temporary variable `element`.

A `break` statement can also be used when we want to terminate an enhanced `for` loop early. For example, we can revise our earlier example of a search loop as follows:

```
int x = ...;
boolean found = false;     // Initially assume x is not present
for (int element : abc){
   if (element == x){
      found = true;
      break;                 // No point in continuing once x is found
   }                         // so break out of the loop
}
if (found)
   System.out.println("Found");
else
   System.out.println("Not Found");
```

An enhanced `for` loop is clearly simpler to write than a standard `for` loop with an index. The enhanced `for` loop is also less error-prone because Java automates the setup and processing of the loop control information. However, this type of loop *cannot* be used to

- Move through an array in reverse, from the last position to the first position
- Assign elements to positions in an array
- Track the index position of the current element in an array
- Access any element other than the current element on each pass

All of these options require a loop with an index. In general, it's also not a good idea to use an enhanced `for` loop on an array that's not filled. Therefore, if you choose an enhanced `for` loop, be sure that the array is filled, that you're going to visit each element from the first to the last, and that you do not need to assign a value to an element at a given position.

EXERCISE 10.7

1. Assume that array `abc` is filled with strings. Convert the following loops to simpler versions using the enhanced `for` loop:

   a.

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

   b.

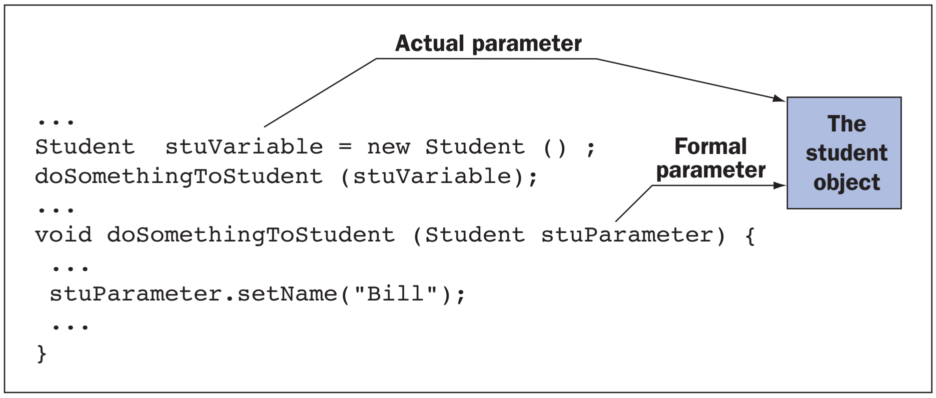
```
String target = …;
    boolean found = false;
    for (int i = 0; i < abc.length; i++)
       if (target.equals(abc[i])){
          found = true;
          break;
       }
```

2. List two different problems for which an enhanced `for` loop would not be appropriate.

10.8 Arrays and Methods

When any object is used as a parameter to a method, what actually gets passed is a reference to the object and not the object itself, as illustrated in Figure 10-3. In other words, the actual and formal parameters refer to the same object, and changes the method makes to the object's state are still in effect after the method terminates. In the figure, the method changes the student's name to Bill, and after the method finishes executing, the name is still Bill.

FIGURE 10-3
Passing a reference to an object as a parameter



```
...
Student  stuVariable = new Student () ;
doSomethingToStudent (stuVariable);
...
void doSomethingToStudent (Student stuParameter) {
 ...
 stuParameter.setName("Bill");
 ...
}
```

Arrays are objects, so the same rules apply. When an array is passed as a parameter to a method, the method manipulates the array itself and not a copy. Changes made to the array in the method are still in effect after the method has completed its execution. Consequently, passing an array to a method leads to trouble if the method accidentally mishandles the array.

A method can also instantiate a new object or a new array and return it using the `return` statement. Following are some illustrations based on examples presented earlier.

Sum the Elements

First, we look at a method that computes the sum of the numbers in an integer array. When the method is written, there is no need to know the array's size. The method works equally well with integer arrays of all sizes, as long as those arrays are full; however, the method cannot be used with arrays of other types, for instance, doubles. Notice that the method makes no changes to the array and therefore is "safe."

```
int sum (int[] a){
   int result = 0;
   for (int element : a)
      result += element;
   return result;
}
```

Using the method is straightforward:

```
int[] array1 = {10, 24, 16, 78, -55, 89, 65};
int[] array2 = {4334, 22928, 33291};
...
if (sum(array1) > sum(array2)) ...
```

Search for a Value

The code to search an array for a value is used so frequently in programs that it is worth placing in a method. Following is a method to search an array of integers. The method returns the location of the first array element equal to the search value or –1 if the value is absent:

```
int search (int[] a, int searchValue){
   for (int i = 0; i < a.length; i++)
      if (a[i] == searchValue)
         return i;
   return -1;
}
```

Sum the Rows

Following is a method that instantiates a new array and returns it. The method computes the sum of each row in a two-dimensional array and returns a one-dimensional array of row sums. The method works even if the rows are not all the same size. We also rely on the fact that Java provides a default value of 0 at each position in the new array.

```
int[] sumRows (int[][] a){
   int[] rowSum = new int[a.length];
   for (int i = 0; i < a.length; i++){
      for (int j = 0; j < a[i].length; j++){
         rowSum[i] += a[i][j];
      }
   }
   return rowSum;
}
```

Following is code that uses the method. Notice that we do not have to instantiate the array `oneD` because that task is done in the method `sumRows`.

```
int[][] twoD = {{1,2,3,4}, {5,6}, {7,8,9}};
int[] oneD;

oneD = sumRows (twoD); // oneD now references the array created and
                       // returned by the method sumRows.
                       // It equals {10, 11, 24}
```

Copy an Array

Earlier, we saw that copying an array must be done with care. Assigning one array variable to another does not do the job. It merely yields two variables referencing the same array. We now examine a method that attempts to solve the problem. The first parameter represents the original array, and the second is the copy. The original is instantiated before the method is called, and the copy is instantiated in the method.

```
void copyOne(int[] original, int[] copy){
   copy = new int[original.length];
   for (int i = 0; i < original.length; i++){
      copy[i] = original[i];
   }
}
```

We now run this method in the following code segment:

```
int[] orig = {1,2,3,4,5};
int[] cp;
...
copyOne (orig, cp);
```

When `copyOne` terminates, we would like the variable `cp` to refer to copy. However, that does not happen. Even though the method creates a copy of the original array and assigns it to the array parameter (`copy = new int[original.length];`), the original variable `cp` is not changed and does not refer to the array created in the method. We can achieve our goal more successfully by writing a method that returns a copy. We then call the method and assign the returned copy to `cp`. Following is the code:

```
// First the method
int[] copyTwo (int[] original){
   int[] copy = new int[original.length];
   for (int i = 0; i < original.length; i++){
      copy[i] = original[i];
   }
   return copy;
}

// And here is how we call it.
int[] orig = {1,2,3,4,5};
int[] cp = copyTwo (orig);
```

EXERCISE 10.8

1. What happens when one uses the assignment operator (=) with two array variables?
2. Discuss the issues involved with copying an array.
3. Write a method that returns the average of the numbers in an array of `double`.
4. Write a method `subArray` that expects an array of `int` and two `int`s as parameters. The integers represent the starting position and the ending position of a subarray within the parameter array. The method should return a new array that contains the elements from the starting position to the ending position.
5. Write a method that searches a two-dimensional array for a given integer. This method should return an object of class `Point`, which contains a row and a column. The constructor for `Point` is `Point(anInteger, anInteger)`.

10.9 Arrays of Objects

We examined the use of an array of strings earlier in this chapter. Arrays can hold objects of any type, or more accurately, references to objects. For example, one can declare, instantiate, and fill an array of students (see Chapter 6) as follows:

```
// Declare and reserve 10 cells for student objects
Student[] studentArray = new Student[10];

// Fill array with students
for (int i = 0; i < studentArray.length; i++)
   studentArray[i] = new Student("Student " + i, 70+i, 80+i, 90+i);
```

When an array of objects is instantiated, each cell is `null` by default until reset to a new object. The next code segment prints the average of all students in the `studentArray`. Pay special attention to the technique used to send a message to each object in the array:

```
// Print the average of all students in the array.
int sum = 0;
for (Student s : studentArray)
   sum += s.getAverage();              // Send message to object in array
System.out.println("The class average is " + sum / studentArray.length);
```

EXERCISE 10.9

1. Write a method `getHighStudent` that expects an array of students as a parameter. The method returns the `Student` object that contains the highest score. You may assume that the `Student` class has a method `getHighScore()`. (*Hint*: The method should declare a local variable of type `Student` to track the student with the highest score. The initial value of this variable should be `null`.)

EXERCISE 10.9 Continued

2. What happens when the following code segment is executed?

```
// Declare and reserve 10 cells for student objects
Student[] studentArray = new Student[10];
// Add 5 students to the array
for (int i = 0; i < 5; i++)
   studentArray[i] = new Student("Student " + i, 70+i, 80+i, 90+i);
// Print the names of the students
for (int i = 0; i < studentArray.length; i++)
   System.out.println(studentArray[i].getName());
```

CASE STUDY: Student Test Scores Again

In Chapter 6 we developed a program for keeping track of student test scores. We now build on that program in two ways:

1. We extend the program so that it allows the user to maintain information on many students.
2. We modify the `Student` class so that the grades are stored in an array rather than in separate instance variables.

   Both changes illustrate the use of arrays to maintain lists of data.

Request

Modify the student test scores program from Chapter 6 so that it allows the user to maintain information on many students.

Analysis

The user interface for this program should allow the user to enter information for a new student, edit existing information, and navigate through the database of students to access each student's information. The student records are arranged in a linear sequence in the database, so the interface allows the user to navigate through this sequence. In addition, the interface should display the overall class average and the student with the highest score on demand. A menu-driven interface will work well. Here are the menu options:

1. Display the current student
2. Display the class average
3. Display the student with the highest grade
4. Display all of the students
5. Edit the current student
6. Add a new student
7. Move to the first student
8. Move to the last student

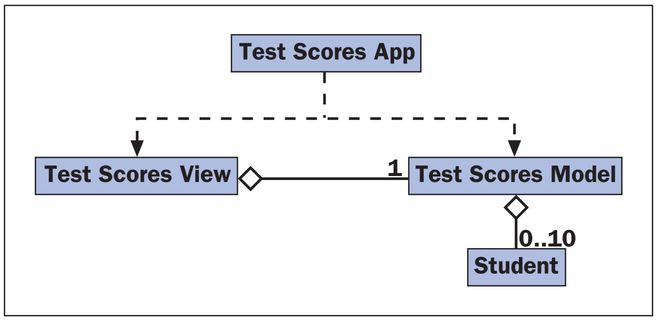
9. Move to the next student
10. Move to the previous student
11. Quit the program

Each option runs a command, displays the results, and waits for the user to press the Enter key before returning to the menu. When the menu is displayed, the program also displays the number of students currently in the database and the index position of the current student.

Because the user interface and the data are complex, we use the model/view pattern introduced in Chapter 6 to structure our code. The view class, called `TestScoresView`, is responsible for displaying the menu and handling interactions with the user. The model classes, called `Student` and `TestScoresModel`, are responsible for managing the students' data. A simple "application" class, called `TestScoresApp`, starts the application in a `main` method by instantiating the view and the model.

Our system is in fact complex enough that it would help to diagram the relationships among its classes. Figure 10-4 shows a *UML diagram* that depicts these relationships. UML, which stands for Unified Modeling Language, is a graphical notation developed by software professionals to design and document object-oriented systems. As you can see, the name of each class appears in a box. Various types of connecting lines designate the relationships between the classes. A dashed line ending in a solid arrow indicates that one class simply depends on another. Thus, `TestScoresApp` depends on both `TestScoresView` and `TestScoresModel`. A solid line ending in a diamond indicates that the class nearest the diamond contains and is in part composed of the class at the other end. The numbers that label these lines show the number of instances of the contained class. These numbers can be simple, indicating a fixed number, or a range from one simple number to another, or a star (*), which means zero or more. Thus, `TestScoresView` contains exactly one `TestScoresModel` object, whereas `TestScoresModel` contains any of 0 through 10 `Student` objects.

FIGURE 10-4
A UML diagram of the classes in the student test scores program



Design of the Data Model

We break the design into two parts, one for each set of classes used in the program.

For this program we make two major changes to the `Student` class described in Chapter 6:

1. The three test scores are stored in an array. This provides more flexibility than did the use of a separate instance variable for each test score and allows clients to use the class to deal with a larger number of scores.
2. The `Student` class provides a `validateData` method. Now any application that needs to validate student data can do so easily. For variety, the approach taken to data validation

is somewhat different than that used in the `Employee` class of Chapter 7. If the validation code is placed in the user interface class, it would need to be repeated in every user interface that works with student objects, an approach that is wasteful, tedious, and difficult to maintain.

The `TestScoresModel` class represents the database of students. Viewed as a black box, it provides an interface or set of public methods listed in Table 10-1.

TABLE 10-1
Public methods of `TestScoresModel`

METHOD	WHAT IT DOES
`int size()`	Returns the number of students in the database
`int currentPosition()`	Returns the index position of the current student
`Student currentStudent()`	If the database is empty, returns `null`, otherwise returns the current student
`String toString()`	Returns a string containing the string representations of all of the students
`int getClassAverage()`	Returns the average of all of the scores in the database
`Student getHighScore()`	If the database is empty, returns `null`; otherwise, returns the first student with the highest score in the database
`String add(Student s)`	If there is not room for the new student, returns an error message; otherwise, adds `s` to the end of the database, makes the last position current, and returns `null`
`String replace(Student s)`	If the database is empty, returns an error message; otherwise, replaces the student at the current position with `s` and returns `null`
`Student first()`	If the database is empty, returns `null`; otherwise, moves to the first student and returns that student
`Student last()`	If the database is empty, returns `null`; otherwise, moves to the last student and returns that student
`Student next()`	If the database is empty, returns `null`; otherwise, if the current student is the last one, returns that student; otherwise, moves to the next student and returns that student
`Student previous()`	If the database is empty, returns `null`; otherwise, if the current student is the first one, returns that student; otherwise, moves to the previous student and returns that student

Note that we have set up the class's interface to make the management of data as easy as possible for its users (easier, at any rate, than it would be to directly manipulate an array). The following short tester program shows how the database of students might be used:

```java
// Case Study 10.1: A tester program for TestScoresModel

public class TestModel{

   public static void main (String[] args){
      // Create and display an empty model
      TestScoresModel model = new TestScoresModel();
      System.out.println(model);

      // Display the size, current position, and current student
      System.out.println(model.size());
      System.out.println(model.currentPosition());
      System.out.println(model.currentStudent());

      // Add and display 3 students
      for (int i = 1; i <= 3; i++){
         Student s = new Student("S" + i);
         model.add(s);
      }
      System.out.println(model);

      // Move to the first student and display it
      System.out.println(model.first());

      // Move to the next and previous and display them
      System.out.println(model.next());
      System.out.println(model.previous());

      // Move to the last and next and display them
      System.out.println(model.last());
      System.out.println(model.next());

      // Display size, current position, and current student
      System.out.println(model.size());
      System.out.println(model.currentPosition());
      System.out.println(model.currentStudent());

      // Replace the current student and display the model
      int[] grades = {99, 88, 77};
      Student newStudent = new Student("Beth", grades);
      model.replace(newStudent);
      System.out.println(model);

      // Add more students and display results
      for (int i = 6; i <= 13; i++){
         Student s = new Student("S" + i);
         System.out.println(model.add(s));
      }
   }
}
```

The `TestScoresModel` class maintains its data in three instance variables:

- An array of `Student` objects
- The selected index (an `int`)
- The current number of students (an `int`).

The use of a separate class to represent the database of students will allow us to choose among several different data structures for holding the students, without disturbing the view classes. We examine some other options for representing collections of objects in later chapters.

Design of the View and Application Classes

At program startup, the `main` method in the `TestScoresApp` class instantiates a `TestScoresModel` and passes it as a parameter to a new instance of `TestScoresView`. The constructor for `TestScoresView` then starts the main command loop. Here is the code for `TestScoresApp`:

```
// Case Study 10.1: The main application class

public class TestScoresApp{
   public static void main(String[] args){
      TestScoresModel model = new TestScoresModel();
      new TestScoresView(model);
   }
}
```

The `TestScoresView` class maintains a single instance variable for the database, of type `TestScoresModel`. The top-level method of `TestScoresModel` is called `run()`. `This` method runs the main command loop. Here is the pseudocode for method `run()`:

```
while (true)
   display the count and current index
   display the menu
   prompt for and input a command
   if the command is to quit then
      break
   run the command
   wait for the user to press Enter
```

Because some of these tasks, such as displaying the menu, getting a valid command number from the user, and running a command, are themselves complex, we can decompose them into separate, private helper methods. A refinement of the pseudocode shows how `run()` calls these methods:

```
while (true)
   display the count and current position
   displayMenu()
   command = getCommand("Enter a number [1-11]: ", 1, 11)
   if command == 11
      break
   runCommand(command)
```

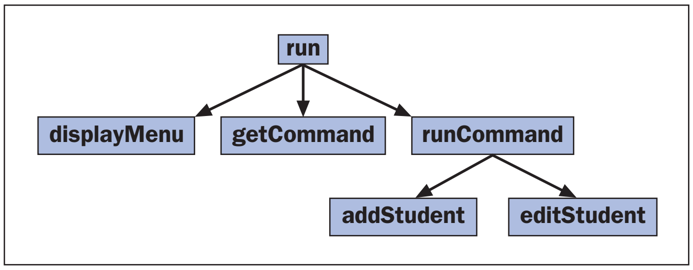
The line `displayMenu()` simply displays the menu options in the terminal window.

The line `getCommand(aString, anInt, anInt)` expects the prompt and the smallest and largest command numbers as parameters. The method displays the prompt for input and attempts to read a number from the keyboard. If the input is a well-formed number and is within the range of valid command numbers, the method returns this number. Otherwise, the method displays an error message and repeats this process.

runCommand(anInt) expects a command number as a parameter. The method performs the appropriate task and waits for the user to press the Enter key. Some of the tasks are simple and require only one line of code, but others, such as adding a new student or editing an existing student, are complex enough to warrant further procedural decomposition.

Figure 10-5 shows a ***structure chart*** that depicts the relationships among the cooperating methods in the class TestScoresView. Needless to say, the single data model object is visible to all of these methods.

FIGURE 10-5
A structure chart for the methods of class TestScoresView



Procedural decomposition is a powerful design tool in situations where a problem calls for one or more complex tasks that operate on the same set of data.

Implementation of the Model

Following is the code for the classes Student and TestScoresModel. To save space we have kept comments to a minimum; however, we have used descriptive names for variables and methods and hope you will find the code fairly self-documenting. Pay special attention to the constructors in the Student class. Note the different options that they give the client for creating a student object and the manner in which chaining them simplifies their code.

```
// Case Study 10.1: Student class

public class Student {

   private String name;
   private int[] tests;

   // Default: name is "" and 3 scores are 0
   public Student(){
      this("");
   }

   // Default: name is nm and 3 scores are 0
   public Student(String nm){
      this(nm, 3);
   }

   // Name is nm and n scores are 0
   public Student(String nm, int n){
      name = nm;
      tests = new int[n];
      for (int i = 0; i < tests.length; i++)
         tests[i] = 0;
   }

   // Name is nm and scores are int
   public Student(String nm, int[] t){
      name = nm;
      tests = new int[t.length];
      for (int i = 0; i < tests.length; i++)
         tests[i] = t[i];
   }

   // Builds a copy of s
   public Student(Student s){
      this(s.name, s.tests);
   }

   public void setName (String nm){
      name = nm;
   }

   public String getName (){
      return name;
   }

   public void setScore (int i, int score){
      tests[i - 1] = score;
   }

   public int getScore (int i){
        return tests[i - 1];
   }

   public int getAverage(){
        int sum = 0;
        for (int score : tests)
           sum += score;
        return sum / tests.length;
   }

   public int getHighScore(){
      int highScore = 0;
      for (int score : tests)
         highScore = Math.max (highScore, score);
      return highScore;
   }

   public String toString(){
      String str = "Name:    " + name  + "\n";
      for (int i = 0; i < tests.length; i++)
         str += "test " + i + ":  " + tests[i] + "\n";
      str += "Average: " + getAverage();
```

```
         return str;
      }

      // Returns null if there are no errors else returns
      // an appropriate error message.
      public String validateData(){
         if (name.equals ("")) return "SORRY: name required";
         for (int score : tests){
            if (score < 0 || score > 100){
               String str = "SORRY: must have "+ 0
                            + " <= test score <= " + 100;
               return str;
            }
         }
         return null;
      }
   }

   // Case Study 10.1: TestScoresModel class

   public class TestScoresModel{

      private Student[] students;          // Array of students
      private int indexSelectedStudent;    // Position of current student
      private int studentCount;            // Current number of students

      public TestScoresModel(){

         // Initialize the data
         indexSelectedStudent = -1;
         studentCount = 0;
         students = new Student[10];
      }

      // Mutator methods for adding and replacing students

      public String add(Student s){
         if (studentCount == students.length)
            return "SORRY: student list is full";
         else{
            students[studentCount] = s;
            indexSelectedStudent = studentCount;
            studentCount++;
            return null;
         }
      }

      public String replace(Student s){
         if (indexSelectedStudent == -1)
            return "Must add a student first";
         else{
            students[indexSelectedStudent] = s;
            return null;
      }
   }

   // Navigation methods

   public Student first(){
      Student s = null;
      if (studentCount == 0)
         indexSelectedStudent = -1;
      else{
         indexSelectedStudent = 0;
         s = students[indexSelectedStudent];
      }
      return s;
   }

    public Student previous(){
      Student s = null;
      if (studentCount == 0)
         indexSelectedStudent = -1;
      else{
         indexSelectedStudent
             = Math.max (0, indexSelectedStudent - 1);
         s = students[indexSelectedStudent];
      }
      return s;
   }

   public Student next(){
      Student s = null;
      if (studentCount == 0)
         indexSelectedStudent = -1;
      else{
         indexSelectedStudent
             = Math.min (studentCount - 1, indexSelectedStudent + 1);
         s = students[indexSelectedStudent];
      }
      return s;
   }

   public Student last(){
      Student s = null;
      if (studentCount == 0)
         indexSelectedStudent = -1;
      else{
         indexSelectedStudent = studentCount - 1;
         s = students[indexSelectedStudent];
      }
      return s;
   }

   // Accessors to observe data
```

```
    public Student currentStudent(){
       if (indexSelectedStudent == -1)
          return null;
       else
          return students[indexSelectedStudent];
    }

    public int size(){
       return studentCount;
    }

    public int currentPosition(){
       return indexSelectedStudent;
    }

    public int getClassAverage(){
       if (studentCount == 0)
          return 0;
       int sum = 0;
       for (int i = 0; i < studentCount; i++)
          sum += students[i].getAverage();
       return sum / studentCount;
    }

    public Student getHighScore(){
       if (studentCount == 0)
          return null;
       else{
          Student s = students[0];
          for (int i = 1; i < studentCount; i++)
             if (s.getHighScore() < students[i].getHighScore())
                s = students[i];
          return s;
       }
    }

    public String toString(){
       String result = "";
       for (int i = 0; i < studentCount; i++)
          result = result + students[i] + "\n";
       return result;
    }
 }
```

Implementation of the View

We include a skeletal listing of the class `TestScoresView` and leave the completion of its methods as an exercise. Even so, the entire program now compiles and allows the user to enter command numbers at run time. Here is the code:

```
// Case Study 10.1: TestScoresView class

import java.util.Scanner;
```

```
public class TestScoresView{

   private TestScoresModel model;

   public TestScoresView(TestScoresModel m){
      model = m;
      run();
   }

   // Menu-driven command loop
   private void run(){
      while (true){
         System.out.println("Number of students: " + model.size());
         System.out.println("Index of current student: " +
                              model.currentPosition());
         displayMenu();
         int command = getCommand("Enter a number [1-11]: ", 1, 11);
         if (command == 11)
            break;
         runCommand(command);
      }
   }

   private void displayMenu(){
   System.out.println("MAIN MENU");// Exercise: List the menu options
   }

   // Prompts the user for a command number and runs until
   // the user enters a valid command number
   // Parameters: prompt is the string to display
   //             low is the smallest command number
   //             high is the largest command number
   // Returns: a valid command number (>= low && <= high)
   private int getCommand(String prompt, int low, int high){
     // Exercise: recover from all input errors
      Scanner reader = new Scanner(System.in);
      System.out.print(prompt);
      return reader.nextInt();
   }

   // Selects a command to run based on a command number,
   // runs the command, and asks the user to continue by
   // pressing the Enter key
   private void runCommand(int command){
      // Exercise
   }
}
```

10.10 Graphics and GUIs: Changing the View of Student Test Scores

In the preceding case study, we organized the code in two sets of classes called the *model* and the *view*. This strategy split the code fairly equally between managing the interface (getting data from the user and displaying results and error messages) and manipulating a database (including worrying about whether or not an array is full and updating the student count and the index of the selected student). In addition to simplifying the code, this separation of concerns allows us to change the style of the user interface without changing the code for managing the database. To illustrate this point, we now show how to attach a GUI to the same data model.

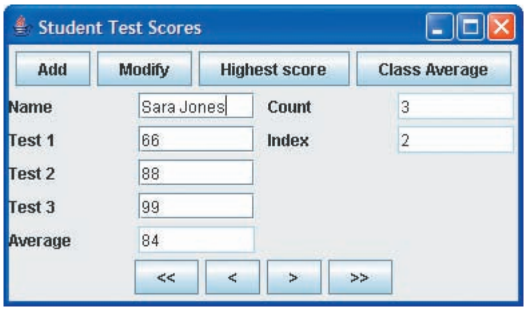
Extra Challenge

This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

Analysis

Figure 10-6 shows a GUI that allows us to view the current student in the database of students. The interface has buttons that support navigation through this database by moving to the first (<<), last (>>), next (>), or previous (<) student. The interface also has buttons that allow the user to add a new student to the end of the database or modify an existing student. The interface displays the index of the current student (**Current Index**) and the current size of the database (Count). Table 10-2 explains each of these features in more detail.

FIGURE 10-6
GUI for the student test scores program



Name: Sara Jones
Test 1: 66
Test 2: 88
Test 3: 99
Average: 84
Count: 3
Index: 2

TABLE 10-2
Description of buttons

BUTTON	WHAT IT DOES
Add	Creates a new student object with the data displayed and inserts it at the end of the array; the new student becomes the current student. Error-checking makes sure that the array of students is not yet full and that the student data is valid.
Modify	Replaces the current student's data with the data displayed, provided it is valid
<<	Moves to the first student in the database and displays its data
<	Moves to the previous student in the database and displays its data
>	Moves to the next student in the database and displays its data
>>	Moves to the last student in the database and displays its data

Design

The structure of the GUI version of the `TestScoresView` class is similar to that of the `GUIWindow` class for the temperature conversion program discussed in Section 7.6. Here are the main changes to `TestScoresView` from the terminal-based version:

- In addition to the instance variable for the data model, the view class must now contain instance variables for the various widgets, such as labels, text fields, and command buttons.
- The constructor now sets up several panels using the appropriate layouts, adds the widgets to the panels, and instantiates and adds listeners to the command buttons. The constructor can also set the title of the window, set its closing action, and show it (no changes are then necessary in the `main` method of the `TestScoresApp` class).
- The code to handle the individual commands now goes in the listener classes. The best way to do this is to define a separate listener class for each command button. Each listener class then has a fairly simple task, which may involve taking data from text fields, sending messages to the model, and updating the text fields with the results. As usual, the listener classes are defined as private inner classes within the view class.
- In addition to the constructor and listener classes, `TestScoresView` defines two private helper methods to perform the tasks of displaying the model's data in the text fields and creating a new `Student` object from the data in the text fields.

Implementation

The listing that follows is a skeletal version but will compile and display a partially functioning GUI. Its completion is left as an exercise.

```java
// Example 10.4: TestScoresView class (GUI version)

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestScoresView extends JFrame{

   // >>>>>>> The model <<<<<<<<

   // Declare the model
   private TestScoresModel model;

   // >>>>>>> The view <<<<<<<<

   // Declare and instantiate the window objects.
   private JButton    addButton       = new JButton("Add");
   private JButton    modifyButton    = new JButton("Modify");
   private JButton    firstButton     = new JButton("<<");
   private JButton    previousButton  = new JButton("<");
   private JButton    nextButton      = new JButton(">");
   private JButton    lastButton      = new JButton(">>");
   private JButton    highScoreButton = new JButton("Highest score");
   private JButton    aveScoreButton  = new JButton("Class Average");
private JLabel     nameLabel       = new JLabel("Name");
private JLabel     test1Label      = new JLabel("Test 1");
private JLabel     test2Label      = new JLabel("Test 2");
private JLabel     test3Label      = new JLabel("Test 3");
private JLabel     averageLabel    = new JLabel("Average");
private JLabel     countLabel      = new JLabel("Count");
private JLabel     indexLabel      = new JLabel("Index");
private JTextField nameField       = new JTextField("");
private JTextField test1Field      = new JTextField("0");
private JTextField test2Field      = new JTextField("0");
private JTextField test3Field      = new JTextField("0");
private JTextField averageField    = new JTextField("0");
private JTextField countField      = new JTextField("0");
private JTextField indexField      = new JTextField("-1");

// Constructor
public TestScoresView(TestScoresModel m){
   model = m;
   // Set attributes of fields
   averageField.setEditable(false);
   countField.setEditable(false);
   indexField.setEditable(false);
   averageField.setBackground(Color.white);
   countField.setBackground(Color.white);
   indexField.setBackground(Color.white);
   // Set up panels to organize widgets and
   // add them to the window
   JPanel northPanel = new JPanel();
   JPanel centerPanel = new JPanel(new GridLayout(5, 4, 10, 5));
   JPanel southPanel = new JPanel();
   Container container = getContentPane();
   container.add(northPanel, BorderLayout.NORTH);
   container.add(centerPanel, BorderLayout.CENTER);
   container.add(southPanel, BorderLayout.SOUTH);
   // Data access buttons
   northPanel.add(addButton);
   northPanel.add(modifyButton);
   northPanel.add(highScoreButton);
   northPanel.add(aveScoreButton);
   // Row 1
   centerPanel.add(nameLabel);
   centerPanel.add(nameField);
   centerPanel.add(countLabel);
   centerPanel.add(countField);
   // Row 2
   centerPanel.add(test1Label);
   centerPanel.add(test1Field);
   centerPanel.add(indexLabel);
   centerPanel.add(indexField);
   // Row 3
   centerPanel.add(test2Label);
   centerPanel.add(test2Field);
   centerPanel.add(new JLabel(""));  // For empty cell in grid
```

```
      centerPanel.add(new JLabel(""));
      // Row 4
      centerPanel.add(test3Label);
      centerPanel.add(test3Field);
      centerPanel.add(new JLabel(""));
      centerPanel.add(new JLabel(""));
      // Row 5
      centerPanel.add(averageLabel);
      centerPanel.add(averageField);
      centerPanel.add(new JLabel(""));
      centerPanel.add(new JLabel(""));
      // Navigation buttons
      southPanel.add(firstButton);
      southPanel.add(previousButton);
      southPanel.add(nextButton);
      southPanel.add(lastButton);
      // Attach listeners to buttons
      addButton.addActionListener(new AddListener());
      previousButton.addActionListener(new PreviousListener());
      // Other attachments will go here (exercise)
      // Set window attributes
      setTitle("Student Test Scores");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      pack();
      setVisible(true);
   }

   // Updates fields with info from the model
   private void displayInfo(){
      Student s = model.currentStudent();
      if (s == null){  // No current student, so clear fields
         nameField.setText("");
         test1Field.setText("0");
         test2Field.setText("0");
         test3Field.setText("0");
         averageField.setText("0");
         countField.setText("0");
         indexField.setText("-1");
      } else{          // Refresh with student's data
         nameField.setText(s.getName());
         test1Field.setText("" + s.getScore(1));
         test2Field.setText("" + s.getScore(2));
         test3Field.setText("" + s.getScore(3));
         averageField.setText("" + s.getAverage());
         countField.setText("" + model.size());
         indexField.setText("" + model.currentPosition());
      }
   }

   // Creates and returns new Student from field info
   private Student getInfoFromScreen(){
      Student s = new Student(nameField.getText());
      s.setScore(1, Integer.parseInt(test1Field.getText()));
```

```
         s.setScore(2, Integer.parseInt(test2Field.getText()));
         s.setScore(3, Integer.parseInt(test3Field.getText()));
         return s;
     }

      // >>>>>>> The controller <<<<<<<<

      // Responds to a click on the Add button
      private class AddListener implements ActionListener{
         public void actionPerformed(ActionEvent e){
            // Get inputs, validate, and display error and quit if invalid
            Student s = getInfoFromScreen();
            String message = s.validateData();
            if (message != null){
               JOptionPane.showMessageDialog(TestScoresView.this, message);
               return;
            }
            // Attempt to add student and display error or update fields
            message = model.add(s);
            if (message != null)
               JOptionPane.showMessageDialog(TestScoresView.this, message);
            else
               displayInfo();
         }
      }

      // Responds to a click on the < button
      private class PreviousListener implements ActionListener{
         public void actionPerformed(ActionEvent e){
            model.previous();
            displayInfo();
         }
      }

      // Other listeners for modify, highest score, class average, and
      // navigation go here (exercise)
   }
```

EXERCISE 10.10

1. Write the code for the listener class that displays the class average in the GUI version of the student test scores program.

2. An alternative way to define listeners for command buttons is to define a single listener class and attach a single instance of this class to all of the buttons. The code for the `actionPerformed` method compares each button to the source of the event and takes the appropriate action. The source of the event is obtained by running the method `getSource()` with the `ActionEvent` parameter. Discuss the advantages and disadvantages of this strategy for implementing listeners.

Design, Testing, and Debugging Hints

- You need to do three things to set up an array:
  1. Declare an array variable.
  2. Instantiate an array object and assign it to the array variable.
  3. Initialize the cells in the array with data, as appropriate.
- When creating a new array object, try to come up with an accurate estimate of the number of cells for the data. If you underestimate, some data will be lost; if you overestimate, some memory will be wasted.
- Remember that array variables are `null` until they are assigned array objects.
- To avoid index out-of-bounds errors, remember that the index of an array cell ranges from 0 (the first position) to the length of the array minus 1.
- To access the last cell in an array, use the expression `<array>.length – 1`.
- As a rule of thumb, it is best to avoid having more than one array variable refer to the same array object. When you want to copy the contents of one array to another, do not use the assignment `A = B`; instead, write a copy method and use the assignment `A = arrayCopy(B)`.
- When an array is not full, take care to track the current number of elements and do not attempt to access a cell that is beyond the last element.

SUMMARY

In this chapter, you learned:

- Arrays are collections of similar items or elements. The items in arrays are ordered by position. Arrays are useful when a program needs to manipulate many similar items, such as a group of students or a number of test scores.
- Arrays are objects. Thus, they must be instantiated and they can be referred to by more than one variable.
- An array can be passed to a method as a parameter and returned as a value.
- Parallel arrays are useful for organizing information with corresponding elements.
- Two-dimensional arrays store values in a row-and-column arrangement similar to a table.
- The enhanced `for` loop is a simplified version of a loop for visiting each element of an array from the first position to the last position.

VOCABULARY *Review*

Define the following terms:

Array	Initializer list	Procedural decomposition
Element	Logical size	Range-bound error
Enhanced `for` loop	Parallel arrays	Structure chart
Index	Physical size	Subscript

REVIEW *Questions*

WRITTEN QUESTIONS

Write a brief answer to the following questions.

1. Assume the following declarations are made and indicate which items below are valid subscripted variables.

```
int a[] = new int[10];
char b[] = new char[6];
int x = 7, y = 2;
double z = 0.0;
```

A. `a[0]`

B. `b[0]`

C. `c[1.0]`

D. `b['a']`

E. `b[a]`

F. `a[x + y]`

G. `a[x % y]`

H. `a[10]`

I. `c[-1]`

J. `a[a[4]]`

2. Assume that the array a defined in Question 1 contains the following values.

```
1   4   6   8   9   3   7   10   2   9
```

Indicate if the following are valid subscripts of `a` and, if so, state the value of the subscript. If invalid, explain why.

A. `a[2]`

B. `a[5]`

C. `a[a[2]]`

D. `a[4 + 7]`

E. `a[a[5] + a[2]]`

F. `a[Math.sqrt(2)]`

3. List the errors in the following array declarations.

   A. `int intArray[] = new double[10];`

   B. `int intArray[] = new int[1.5];`

   C. `double[] doubleArray = new double[-10]`

   D. `int intMatrix[] [] = new int[10];`

4. Write a method `selectRandom` that expects an array of integers as a parameter. The method should return the value of an array element at a randomly selected position.

5. Write code to declare and instantiate a two-dimensional array of integers with five rows and four columns.

6. Write code to initialize the array of Question 5 with randomly generated integers between 1 and 20.

PROJECTS

In some of the following projects, you are asked to write helper methods to process arrays. If you are calling these methods from the `main` method, be sure to begin the helper method's header with the reserved word `static`.

PROJECT 10-1

Write a program that takes 10 integers as input. The program places the even integers into an array called `evenList`, the odd integers into an array called `oddList`, and the negative integers into an array called `negativeList`. The program displays the contents of the three arrays after all of the integers have been entered.

PROJECT 10-2

Write a program that takes 10 floating-point numbers as inputs. The program displays the average of the numbers followed by all of the numbers that are greater than the average. As part of your design, write a method that takes an array of doubles as a parameter and returns the average of the data in the array.

PROJECT 10-3

The mode of a list of numbers is the number listed most often. Write a program that takes 10 numbers as input and displays the mode of these numbers. Your program should use parallel arrays and a method that takes an array of numbers as a parameter and returns the value that appears most often in the array.

PROJECT 10-4

The median of a list of numbers is the value in the middle of the list if the list is arranged in order. Add to the program of Project 10-3 the capability of displaying the median of the list of numbers.

PROJECT 10-5

Modify the program of Project 10-4 so that it displays not only the median and mode of the list of numbers but also a table of the numbers and their associated frequencies.

PROJECT 10-6

Complete the student test scores application from this chapter's case study and test it thoroughly.

PROJECT 10-7

Complete the GUI version of the student test scores program.

CRITICAL *Thinking*

You have been using a method to search for data in arrays like the one described in this chapter, when your friend tells you that it's a poor way to search. She says that you're examining every element in the array to discover that the target element is not there. According to her, a better way is to assume that the elements in the array are in alphabetical order. Start by examining the element at the middle position in the array. If that element matches the target element, you're done. Otherwise, if that element is less than the target element, continue the same kind of search in just the portion of the array to the left of the element just examined. Otherwise, continue the same kind of search in just the portion of the array to the right of the element just examined.

Write an algorithm for this search process, and explain why it is better than the search algorithm discussed in this chapter.