OBJECTIVES

In this chapter you will learn:

■ What arrays are.
■ To use arrays to store data in and retrieve data from lists and tables of values.
■ To declare arrays, initialize arrays and refer to individual elements of arrays.
■ To use the enhanced for statement to iterate through arrays.
■ To pass arrays to methods.
■ To declare and manipulate multidimensional arrays.
■ To write methods that use variable-length argument lists.
■ To read command-line arguments into a program.
Chapter 7 Arrays

7.1 Introduction

This chapter introduces data structures—collections of related data items. Arrays are data structures consisting of related data items of the same type. Arrays are fixed-length entities—they remain the same length once they are created, although an array variable may be reassigned such that it refers to a new array of a different length. We study data structures in depth in Chapters 17–19.

After discussing how arrays are declared, created and initialized, we present a series of practical examples that demonstrate several common array manipulations. We also present a case study that examines how arrays can help simulate the shuffling and dealing of playing cards for use in an application that implements a card game. The chapter then introduces Java’s enhanced for statement, which allows a program to access the data in an array more easily than does the counter-controlled for statement presented in Section 5.3. Two sections of the chapter enhance the case study of class GradeBook in Chapters 3–5. In particular, we use arrays to enable the class to maintain a set of grades in memory and analyze student grades from multiple exams in a semester—two capabilities that were absent from previous versions of the class. These and other chapter examples demonstrate the ways in which arrays allow programmers to organize and manipulate data.

7.2 Arrays

An array is a group of variables (called elements or components) containing values that all have the same type. Recall that types are divided into two categories—primitive types and reference types. Arrays are objects, so they are considered reference types. As you will soon see, what we typically think of as an array is actually a reference to an array object in mem-
ory. The elements of an array can be either primitive types or reference types (including arrays, as we will see in Section 7.9). To refer to a particular element in an array, we specify the name of the reference to the array and the position number of the element in the array. The position number of the element is called the element’s index or subscript.

Figure 7.1 shows a logical representation of an integer array called c. This array contains 12 elements. A program refers to any one of these elements with an array-access expression that includes the name of the array followed by the index of the particular element in square brackets (\([\])\). The first element in every array has index zero and is sometimes called the zeroth element. Thus, the elements of array c are c[0], c[1], c[2] and so on. The highest index in array c is 11, which is 1 less than 12—the number of elements in the array. Array names follow the same conventions as other variable names.

An index must be a nonnegative integer. A program can use an expression as an index. For example, if we assume that variable a is 5 and variable b is 6, then the statement

\[ c[ a + b ] += 2; \]

adds 2 to array element c[11]. Note that an indexed array name is an array-access expression. Such expressions can be used on the left side of an assignment to place a new value into an array element.

**Common Programming Error 7.1**

Using a value of type long as an array index results in a compilation error. An index must be an int value or a value of a type that can be promoted to int—namely, byte, short or char, but not long.

Let us examine array c in Fig. 7.1 more closely. The name of the array is c. Every array object knows its own length and maintains this information in a length field. The expression c.length accesses array c’s length field to determine the length of the array. Note that, even though the length member of an array is public, it cannot be changed because

![Fig. 7.1 | A 12-element array.](image-url)
Chapter 7 Arrays

it is a final variable. This array's 12 elements are referred to as $c[0]$, $c[1]$, $c[2]$, ..., $c[11]$. The value of $c[0]$ is -45, the value of $c[1]$ is 6, the value of $c[2]$ is 0, the value of $c[7]$ is 62 and the value of $c[11]$ is 78. To calculate the sum of the values contained in the first three elements of array $c$ and store the result in variable sum, we would write

```java
sum = c[0] + c[1] + c[2];
```

To divide the value of $c[6]$ by 2 and assign the result to the variable $x$, we would write

```java
x = c[6] / 2;
```

### 7.3 Declaring and Creating Arrays

Array objects occupy space in memory. Like other objects, arrays are created with keyword `new`. To create an array object, the programmer specifies the type of the array elements and the number of elements as part of an array-creation expression that uses keyword `new`. Such an expression returns a reference that can be stored in an array variable. The following declaration and array-creation expression create an array object containing 12 int elements and store the array's reference in variable $c$:

```java
int c[] = new int[12];
```

This expression can be used to create the array shown in Fig. 7.1. This task also can be performed in two steps as follows:

```java
int c[];            // declare the array variable
c = new int[12];   // create the array; assign to array variable
```

In the declaration, the square brackets following the variable name $c$ indicate that $c$ is a variable that will refer to an array (i.e., the variable will store an array reference). In the assignment statement, the array variable $c$ receives the reference to a new array of 12 int elements. When an array is created, each element of the array receives a default value—zero for the numeric primitive-type elements, `false` for `boolean` elements and `null` for references (any nonprimitive type). As we will soon see, we can provide specific, nondefault initial element values when we create an array.

#### Common Programming Error 7.2

In an array declaration, specifying the number of elements in the square brackets of the declaration (e.g., `int c[12];`) is a syntax error.

A program can create several arrays in a single declaration. The following `String` array declaration reserves 100 elements for $b$ and 27 elements for $x$:

```java
String b[] = new String[100], x[] = new String[27];
```

In this case, the class name `String` applies to each variable in the declaration. For readability, we prefer to declare only one variable per declaration, as in:

```java
String b[] = new String[100];  // create array b
String x[] = new String[27];   // create array x
```
Good Programming Practice 7.1

For readability, declare only one variable per declaration. Keep each declaration on a separate line, and include a comment describing the variable being declared.

When an array is declared, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all the identifiers in the declaration are array variables. For example, the declaration

\[ \text{double[]} \text{ array1, array2;} \]

indicates that array1 and array2 are each “array of double” variables. The preceding declaration is equivalent to:

\[ \text{double array1[];} \]
\[ \text{double array2[];} \]

or

\[ \text{double[]} \text{ array1;} \]
\[ \text{double[]} \text{ array2;} \]

The preceding pairs of declarations are equivalent—when only one variable is declared in each declaration, the square brackets can be placed either after the type or after the array variable name.

Common Programming Error 7.3

Declaring multiple array variables in a single declaration can lead to subtle errors. Consider the declaration

\[ \text{int[]} \text{ a, b, c;} \]

If a, b and c should be declared as array variables, then this declaration is correct—placing square brackets directly following the type indicates that all the identifiers in the declaration are array variables. However, if only a is intended to be an array variable, and b and c are intended to be individual int variables, then this declaration is incorrect—the declaration \[ \text{int a[], b, c;} \] would achieve the desired result.

A program can declare arrays of any type. Every element of a primitive-type array contains a value of the array’s declared type. Similarly, in an array of a reference type, every element is a reference to an object of the array’s declared type. For example, every element of an int array is an int value, and every element of a String array is a reference to a String object.

7.4 Examples Using Arrays

This section presents several examples that demonstrate declaring arrays, creating arrays, initializing arrays and manipulating array elements.

Creating and Initializing an Array

The application of Fig. 7.2 uses keyword new to create an array of 10 int elements, which are initially zero (the default for int variables).

Line 8 declares array—a reference capable of referring to an array of int elements. Line 10 creates the array object and assigns its reference to variable array. Line 12 outputs the column headings. The first column contains the index (0–9) of each array element, and the second column contains the default value (0) of each array element.
The `for` statement in lines 15–16 outputs the index number (represented by `counter`) and the value of each array element (represented by `array[counter]`). Note that the loop control variable `counter` is initially 0—index values start at 0, so using zero-based counting allows the loop to access every element of the array. The `for`’s loop-continuation condition uses the expression `array.length` (line 15) to determine the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable `counter` is less than 10. The highest index value of a 10-element array is 9, so using the less-than operator in the loop-continuation condition guarantees that the loop does not attempt to access an element beyond the end of the array (i.e., during the final iteration of the loop, `counter` is 9). We will soon see what Java does when it encounters such an out-of-range index at execution time.

Using an Array Initializer

A program can create an array and initialize its elements with an array initializer, which is a comma-separated list of expressions (called an initializer list) enclosed in braces (`{` and `}`); the array length is determined by the number of elements in the initializer list. For example, the declaration

```java
int n[] = { 10, 20, 30, 40, 50 };```

Fig. 7.2 | Initializing the elements of an array to default values of zero.
creates a five-element array with index values 0, 1, 2, 3 and 4. Element \( a[0] \) is initialized to 10, \( a[1] \) is initialized to 20, and so on. This declaration does not require `new` to create the array object. When the compiler encounters an array declaration that includes an initializer list, it counts the number of initializers in the list to determine the size of the array, then sets up the appropriate `new` operation “behind the scenes.”

The application in Fig. 7.3 initializes an integer array with 10 values (line 9) and displays the array in tabular format. The code for displaying the array elements (lines 14–15) is identical to that in Fig. 7.2 (lines 15–16).

### Calculating the Values to Store in an Array

The application in Fig. 7.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20). Then the application displays the array in tabular format. The `for` statement at lines 12–13 calculates an array element’s value by multiplying the current value of the control variable `counter` by 2, then adding 2.

Line 8 uses the modifier `final` to declare the constant variable `ARRAY_LENGTH` with the value 10. Constant variables (also known as `final` variables) must be initialized before they are used and cannot be modified thereafter. If you attempt to modify a `final` variable after it is initialized in its declaration (as in line 8), the compiler issues the error message

cannot assign a value to final variable `variableName`

---

```java
public class InitArray {
    public static void main( String args[] ) {
        int array[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 }; // initializer list specifies the value for each element
        System.out.printf("%s%8s
", "Index", "Value"); // column headings
        // output each array element's value
        for ( int counter = 0; counter < array.length; counter++ )
            System.out.printf( "%5d%8d
", counter, array[ counter ] );
    }
}
```

---

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>95</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>37</td>
</tr>
</tbody>
</table>

Fig. 7.3 | Initializing the elements of an array with an array initializer.
302 Chapter 7 Arrays

If an attempt is made to access the value of a final variable before it is initialized, the compiler issues the error message

variable variableName might not have been initialized

Good Programming Practice 7.2

Constant variables also are called named constants or read-only variables. Such variables often make programs more readable than programs that use literal values (e.g., 10)—a named constant such as ARRAY_LENGTH clearly indicates its purpose, whereas a literal value could have different meanings based on the context in which it is used.

Common Programming Error 7.4

Assigning a value to a constant after the variable has been initialized is a compilation error.

Common Programming Error 7.5

Attempting to use a constant before it is initialized is a compilation error.
Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the elements of the array and use that sum to calculate the class average for the exam. The examples using class GradeBook later in the chapter, namely Fig. 7.14 and Fig. 7.18, use this technique.

The application in Fig. 7.5 sums the values contained in a 10-element integer array. The program declares, creates and initializes the array at line 8. The for statement performs the calculations. [Note: The values supplied as array initializers are often read into a program rather than specified in an initializer list. For example, an application could input the values from a user or from a file on disk (as discussed in Chapter 14, Files and Streams). Reading the data into a program makes the program more reusable, because it can be used with different sets of data.]

Using Bar Charts to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent proportionally larger numeric values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*).

Professors often like to examine the distribution of grades on an exam. A professor might graph the number of grades in each of several categories to visualize the grade distribution. Suppose the grades on an exam were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. Note that there was one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Our next application (Fig. 7.6) stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, array[0] indicates the number of grades in the 1s category, array[1] indicates the number of grades in the 2s category, and so on. This approach allows the professor to visualize the grade distribution of the exam at a glance.

```java
// Fig. 7.5: SumArray.java
// Computing the sum of the elements of an array.

public class SumArray
{
   public static void main( String args[] )
   {
      int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int counter = 0; counter < array.length; counter++ )
         total += array[ counter ];

      System.out.printf( "Total of array elements: %d\n", total );
   } // end main
} // end class SumArray

Total of array elements: 849
```

Fig. 7.5 Computing the sum of the elements of an array.
range 0–9, array[7] indicates the number of grades in the range 70–79 and array[10] indicates the number of 100 grades. The two versions of class GradeBook later in the chapter (Fig. 7.14 and Fig. 7.18) contain code that calculates these grade frequencies based on a set of grades. For now, we manually create the array by looking at the set of grades.

The application reads the numbers from the array and graphs the information as a bar chart. The program displays each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 16–20 output a grade range (e.g.,

```
// Fig. 7.6: BarChart.java
// Bar chart printing program.

public class BarChart
{
    public static void main( String args[] )
    {
        int array[] = { 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

        System.out.println( "Grade distribution:" );

        // for each array element, output a bar of the chart
        for ( int counter = 0; counter < array.length; counter++ )
        {
            // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
            if ( counter == 10 )
                System.out.printf( "%5d: ", 100 );
            else
                System.out.printf( "%02d-%02d: ",
                                    counter * 10, counter * 10 + 9 );

            // print bar of asterisks
            for ( int stars = 0; stars < array[counter]; stars++ )
                System.out.print( "*" );

            System.out.println(); // start a new line of output
        } // end outer for
    } // end main
} // end class BarChart
```

Fig. 7.6 | Bar chart printing program.
"70–79: ") based on the current value of counter. When counter is 10, line 17 outputs 100 with a field width of 5, followed by a colon and a space, to align the label "100: " with the other bar labels. The nested for statement (lines 23–24) outputs the bars. Note the loop-continuation condition at line 23 (stars < array[counter]). Each time the program reaches the inner for, the loop counts from 0 up to array[counter], thus using a value in array to determine the number of asterisks to display. In this example, array[0]–array[5] contain zeroes because no students received a grade below 60. Thus, the program displays no asterisks next to the first six grade ranges. Note that line 19 uses the format specifier %02d to output the numbers in a grade range. This specifier indicates that an int value should be formatted as a field of two digits. The 0 flag in the format specifier indicates that values with fewer digits than the field width (2) should begin with a leading 0.

Using the Elements of an Array as Counters

Sometimes, programs use counter variables to summarize data, such as the results of a survey. In Fig. 6.8, we used separate counters in our die-rolling program to track the number of occurrences of each side of a die as the program rolled the die 6000 times. An array version of the application in Fig. 6.8 is shown in Fig. 7.7.

```java
// Fig. 7.7: RollDie.java
// Roll a six-sided die 6000 times.
import java.util.Random;

public class RollDie
{
    public static void main( String args[] )
    {
        Random randomNumbers = new Random(); // random number generator
        int frequency[] = new int[ 7 ]; // array of frequency counters

        // roll die 6000 times; use die value as frequency index
        for ( int roll = 1; roll <= 6000; roll++ )
            ++frequency[ 1 + randomNumbers.nextInt( 6 ) ];

        System.out.printf( "%s%10s
", "Face", "Frequency" );

        // output each array element's value
        for ( int face = 1; face < frequency.length; face++ )
            System.out.printf( "%4d%10d
", face, frequency[ face ] );
    }
}
```

**Fig. 7.7** | Die-rolling program using arrays instead of switch.
Fig. 7.7 uses the array frequency (line 10) to count the occurrences of each side of the die. The single statement in line 14 of this program replaces lines 23–46 of Fig. 6.8. Line 14 uses the random value to determine which frequency element to increment during each iteration of the loop. The calculation in line 14 produces random numbers from 1 to 6, so the array frequency must be large enough to store six counters. However, we use a seven-element array in which we ignore frequency[ 0 ]—it is more logical to have the face value 1 increment frequency[ 1 ] than frequency[ 0 ]. Thus, each face value is used as an index for array frequency. We also replaced lines 50–52 from Fig. 6.8 by looping through array frequency to output the results (lines 19–20).

Using Arrays to Analyze Survey Results

Our next example uses arrays to summarize the results of data collected in a survey:

Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (where 1 means awful and 10 means excellent). Place the 40 responses in an integer array, and summarize the results of the poll.

This is a typical array-processing application (see Fig. 7.8). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array responses (lines 9–11) is a 40-element integer array of the students’ responses to the survey. We use an 11-element array frequency (line 12) to count the number of occurrences of each response. Each element of the array is used as a counter for one of the survey responses and is initialized to zero by default. As in Fig. 7.7, we ignore frequency[ 0 ].

```java
// Fig. 7.8: StudentPoll.java
// Poll analysis program.
public class StudentPoll {
    public static void main( String args[] ) {
        // array of survey responses
        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 7, 5, 6, 3, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
        // array of frequency counters
        int frequency[] = new int[ 11 ];
        // for each answer, select responses element and use that value
        for ( int answer = 0; answer < responses.length; answer++ )
            ++frequency[ responses[ answer ] ];
        // as frequency index to determine element to increment
        System.out.printf( "%s%10s", "Rating", "Frequency" );
        // output each array element's value
        for ( int rating = 1; rating < frequency.length; rating++ )
            System.out.printf( "%d%10d", rating, frequency[ rating ] );
    }
} // end main
// end class StudentPoll
```

Fig. 7.8 | Poll analysis program. (Part 1 of 2.)
The for loop at lines 16–17 takes the responses one at a time from array responses and increments one of the 10 counters in the frequency array (frequency[ 1 ] to frequency[ 10 ]). The key statement in the loop is line 17, which increments the appropriate frequency counter, depending on the value of responses[ answer ].

Let’s consider several iterations of the for loop. When control variable answer is 0, the value of responses[ answer ] is the value of responses[ 0 ] (i.e., 1), so the program interprets ++frequency[ responses[ answer ] ] as ++frequency[ 1 ]

which increments the value in array element 1. To evaluate the expression, start with the value in the innermost set of square brackets (answer). Once you know answer’s value (which is the value of the loop control variable in line 16), plug it into the expression and evaluate the next outer set of square brackets (i.e., responses[ answer ], which is a value selected from the responses array in lines 9–11). Then use the resulting value as the index for the frequency array to specify which counter to increment.

When answer is 1, responses[ answer ] is the value of responses[ 1 ] (2), so the program interprets ++frequency[ responses[ answer ] ] as ++frequency[ 2 ]

which increments array element 2.

When answer is 2, responses[ answer ] is the value of responses[ 2 ] (6), so the program interprets ++frequency[ responses[ answer ] ] as ++frequency[ 6 ]

which increments array element 6, and so on. Regardless of the number of responses processed in the survey, the program requires only an 11-element array (ignoring element zero) to summarize the results, because all the response values are between 1 and 10 and the index values for an 11-element array are 0 through 10.

If the data in the responses array had contained invalid values, such as 13, the program would have attempted to add 1 to frequency[ 13 ], which is outside the bounds of the array. Java disallows this. When a Java program executes, the JVM checks array indices to ensure that they are valid (i.e., they must be greater than or equal to 0 and less than the length of the array). If a program uses an invalid index, Java generates a so-called exception to indicate that an error occurred in the program at execution time. A control statement

<table>
<thead>
<tr>
<th>Rating</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 7.8 | Poll analysis program. (Part 2 of 2.)
can be used to prevent such an “out-of-bounds” error from occurring. For example, the condition in a control statement could determine whether an index is valid before allowing it to be used in an array-access expression.

**Error-Prevention Tip 7.1**

An exception indicates that an error has occurred in a program. A programmer often can write code to recover from an exception and continue program execution, rather than abnormally terminating the program. When a program attempts to access an element outside the array bounds, an `ArrayIndexOutOfBoundsException` occurs. Exception handling is discussed in Chapter 13.

**Error-Prevention Tip 7.2**

When writing code to loop through an array, ensure that the array index is always greater than or equal to 0 and less than the length of the array. The loop-continuation condition should prevent the accessing of elements outside this range.

### 7.5 Case Study: Card Shuffling and Dealing Simulation

The examples in the chapter thus far have used arrays containing elements of primitive types. Recall from Section 7.2 that the elements of an array can be either primitive types or reference types. This section uses random number generation and an array of reference-type elements, namely objects representing playing cards, to develop a class that simulates card shuffling and dealing. This class can then be used to implement applications that play specific card games. The exercises at the end of the chapter use the classes developed here to build a simple poker application.

We first develop class `Card` (Fig. 7.9), which represents a playing card that has a face (e.g., "Ace", "Deuce", "Three", ..., "Jack", "Queen", "King") and a suit (e.g., "Hearts", "Diamonds", "Clubs", "Spades"). Next, we develop the `DeckOfCards` class (Fig. 7.10), which creates a deck of 52 playing cards in which each element is a `Card` object. We then build a test application (Fig. 7.11) that demonstrates class `DeckOfCards`'s card shuffling and dealing capabilities.

#### Class Card

Class `Card` (Fig. 7.9) contains two `String` instance variables—face and suit—that are used to store references to the face name and suit name for a specific Card. The constructor for the class (lines 10–14) receives two `String`s that it uses to initialize face and suit. Method `toString` (lines 17–20) creates a `String` consisting of the face of the card, the String " of " and the suit of the card. Recall from Chapter 6 that the + operator can be used to concatenate (i.e., combine) several `String`s to form one larger `String`. Card's `toString` method can be invoked explicitly to obtain a string representation of a Card object (e.g., "Ace of Spades"). The `toString` method of an object is called implicitly when the object is used where a `String` is expected (e.g., when `printf` outputs the object as a `String` using the `%s` format specifier or when the object is concatenated to a `String` using the + operator). For this behavior to occur, `toString` must be declared with the header shown in Fig. 7.9.

#### Class DeckOfCards

Class `DeckOfCards` (Fig. 7.10) declares an instance variable array named `deck` of `Card` objects (line 7). Like primitive-type array declarations, the declaration of an array of objects
7.5 Case Study: Card Shuffling and Dealing Simulation

includes the type of the elements in the array, followed by the name of the array variable and square brackets (e.g., Card deck[]). Class DeckOfCards also declares an integer instance variable currentCard (line 8) representing the next Card to be dealt from the deck array and a named constant NUMBER_OF_CARDS (line 9) indicating the number of Cards in the deck (52).

Fig. 7.9 | Card class represents a playing card.

```java
// Fig. 7.9: Card.java
// Card class represents a playing card.
public class Card
{
    private String face; // face of card ("Ace", "Deuce", ...)
    private String suit; // suit of card ("Hearts", "Diamonds", ...)

    // two-argument constructor initializes card's face and suit
    public Card( String cardFace, String cardSuit )
    {
        face = cardFace; // initialize face of card
        suit = cardSuit; // initialize suit of card
    } // end two-argument Card constructor

    // return String representation of Card
    public String toString()
    {
        return face + " of " + suit;
    } // end method toString
} // end class Card
```

Fig. 7.10 | DeckOfCards class represents a deck of playing cards.

```java
// Fig. 7.10: DeckOfCards.java
// DeckOfCards class represents a deck of playing cards.
import java.util.Random;

public class DeckOfCards
{
    private Card deck[]; // array of Card objects
    private int currentCard; // index of next Card to be dealt
    private final int NUMBER_OF_CARDS = 52; // constant number of Cards
    private Random randomNumbers; // random number generator

    // constructor fills deck of Cards
    public DeckOfCards()
    {
        String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

        deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
        String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

        deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
    } // end constructor

    // return String representation of DeckOfCards
    public String toString()
    {
        return deck.toString();
    } // end method toString
} // end class DeckOfCards
```

Fig. 7.10 | DeckOfCards class represents a deck of playing cards that can be shuffled and dealt one at a time. (Part 1 of 2.)
The class's constructor instantiates the deck array (line 19) to be of size NUMBER_OF_CARDS. When first created, the elements of the deck array are null by default, so the constructor uses a for statement (lines 24–26) to fill the deck array with Cards. The for statement initializes control variable count to 0 and loops while count is less than deck.length, causing count to take on each integer value from 0 to 51 (the indices of the deck array). Each Card is instantiated and initialized with two Strings—one from the faces array (which contains the strings “Ace” through “King”) and one from the suits array (which contains the strings “Hearts”, “Diamonds”, “Clubs” and “Spades”). The calculation count % 13 always results in a value from 0 to 12 (the 13 indices of the faces array in lines 15–16), and the calculation count / 13 always results in a value from 0 to 3.
(the four indices of the suits array in line 17). When the deck array is initialized, it contains the Cards with faces "Ace" through "King" in order for each suit ("Hearts" then "Diamonds" then "Clubs" then "Spades").

Method shuffle (lines 30–46) shuffles the Cards in the deck. The method loops through all 52 Cards (array indices 0 to 51). For each Card, a number between 0 and 51 is picked randomly to select another Card. Next, the current Card object and the randomly selected Card object are swapped in the array. This exchange is performed by the three assignments in lines 42–44. The extra variable temp temporarily stores one of the two Cards being swapped. The swap cannot be performed with only the two statements

```
deed[ first ] = deed[ second ];
deed[ second ] = deed[ first ];
```

If deed[ first ] is the "Ace" of "Spades" and deed[ second ] is the "Queen" of "Hearts", after the first assignment, both array elements contain the "Queen" of "Hearts" and the "Ace" of "Spades" is lost—hence, the extra variable temp is needed. After the for loop terminates, the Card objects are randomly ordered. A total of only 52 swaps are made in a single pass of the entire array, and the array of Card objects is shuffled!

Method dealCard (lines 49–56) deals one Card in the array. Recall that currentCard indicates the index of the next Card to be dealt (i.e., the Card at the top of the deck). Thus, line 52 compares currentCard to the length of the deck array. If the deck is not empty (i.e., currentCard is less than 52), line 53 returns the "top" Card and postincrements currentCard to prepare for the next call to dealCard—otherwise, null is returned. Recall from Chapter 3 that null represents a "reference to nothing."

**Shuffling and Dealing Cards**

The application of Fig. 7.11 demonstrates the card dealing and shuffling capabilities of class DeckOfCards (Fig. 7.10). Line 9 creates a DeckOfCards object named myDeckOfCards. Recall that the DeckOfCards constructor creates the deck with the 52 Card objects in order by suit and face. Line 10 invokes myDeckOfCards's shuffle method to rearrange the Card objects. The for statement in lines 13–19 deals all 52 Cards in the deck and prints them in four columns of 13 Cards each. Lines 16–18 deal and print four Card objects, each obtained by invoking myDeckOfCards's dealCard method. When printf outputs a Card with the %-20s format specifier, the Card's toString method (declared in lines 17–20 of Fig. 7.9) is implicitly invoked, and the result is output left justified in a field of width 20.

```
1 // Fig. 7.11: DeckOfCardsTest.java
2 // Card shuffling and dealing application.
3
4 public class DeckOfCardsTest
5 {
6     // execute application
7     public static void main( String args[] )
8     {
9         DeckOfCards myDeckOfCards = new DeckOfCards();
10         myDeckOfCards.shuffle(); // place Cards in random order
```

**Fig. 7.11** Card shuffling and dealing. (Part 1 of 2.)
In previous examples, we demonstrated how to use counter-controlled for statements to iterate through the elements of an array. In this section, we introduce the enhanced for statement, which iterates through the elements of an array or a collection without using a counter (thus avoiding the possibility of “stepping outside” the array). This section discusses how to use the enhanced for statement to loop through an array. We show how to use the enhanced for statement with collections in Chapter 19, Collections.

The syntax of an enhanced for statement is:

```java
for (parameter : arrayName)
    statement
```

where `parameter` has two parts—a type and an identifier (e.g., `int number`)—and `arrayName` is the array through which to iterate. The type of the parameter must be consistent with the type of the elements in the array. As the next example illustrates, the identifier represents successive values in the array on successive iterations of the enhanced for statement.

Figure 7.12 uses the enhanced for statement (lines 12–13) to sum the integers in an array of student grades. The type specified in the parameter to the enhanced for is `int`, because `array` contains `int` values—the loop selects one `int` value from the array during each iteration. The enhanced for statement iterates through successive values in the array one by one. The enhanced for header can be read as “for each iteration, assign the next element of `array` to `int` variable `number`, then execute the following statement.” Thus, for

```java
for (int i = 0; i < 13; i++)
```

the enhancement avoids the possibility of “stepping outside” the array.
7.7 Passing Arrays to Methods

This section demonstrates how to pass arrays and individual array elements as arguments to methods. At the end of the section, we discuss how all types of arguments are passed to methods. To pass an array argument to a method, specify the name of the array without any brackets. For example, if array `hourlyTemperatures` is declared as

```java
double hourlyTemperatures[] = new double[24];
```

Each iteration, identifier `number` represents an `int` value in array. Lines 12–13 are equivalent to the following counter-controlled repetition used in lines 12–13 of Fig. 7.5 to total the integers in array:

```java
for (int counter = 0; counter < array.length; counter++)
    total += array[counter];
```

The enhanced `for` statement simplifies the code for iterating through an array. Note, however, that the enhanced `for` statement can be used only to obtain array elements—it cannot be used to modify elements. If your program needs to modify elements, use the traditional counter-controlled `for` statement.

The enhanced `for` statement can be used in place of the counter-controlled `for` statement whenever code looping through an array does not require access to the counter indicating the index of the current array element. For example, totaling the integers in an array requires access only to the element values—the index of each element is irrelevant. However, if a program must use a counter for some reason other than simply to loop through an array (e.g., to print an index number next to each array element value, as in the examples earlier in this chapter), use the counter-controlled `for` statement.

```
// Fig. 7.12: EnhancedForTest.java
// Using enhanced for statement to total integers in an array.

public class EnhancedForTest
{
    public static void main(String args[])
    {
        int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
        int total = 0;

        // add each element's value to total
        for (int number : array)
            total += number;

        System.out.printf("Total of array elements: %d\n", total);
    }
}
```

Fig. 7.12 | Using the enhanced for statement to total integers in an array.
then the method call

    modifyArray( hourlyTemperatures );

passes the reference of array hourlyTemperatures to method modifyArray. Every array object "knows" its own length (via its length field). Thus, when we pass an array object's reference into a method, we need not pass the array length as an additional argument.

For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter. For example, the method header for method modifyArray might be written as

    void modifyArray( int b[] )

indicating that modifyArray receives the reference of an integer array in parameter b. The method call passes array hourlyTemperature's reference, so when the called method uses the array variable b, it refers to the same array object as hourlyTemperatures in the caller.

When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference. However, when an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's value. Such primitive values are called scalars or scalar quantities. To pass an individual array element to a method, use the indexed name of the array element as an argument in the method call.

Figure 7.13 demonstrates the difference between passing an entire array and passing a primitive-type array element to a method. The enhanced for statement at lines 16–17 outputs the five elements of array (an array of int values). Line 19 invokes method modifyArray, passing array as an argument. Method modifyArray (lines 36–40) receives a copy of array's reference and uses the reference to multiply each of array's elements by 2. To prove that array's elements were modified, the for statement at lines 23–24 outputs the five elements of array again. As the output shows, method modifyArray doubled the value of each element. Note that we could not use the enhanced for statement in lines 38–39 because we are modifying the array's elements.

```java
// Fig. 7.13: PassArray.java
// Passing arrays and individual array elements to methods.

public class PassArray {
   // main creates array and calls modifyArray and modifyElement
   public static void main( String args[] ) {
      int array[] = { 1, 2, 3, 4, 5 };
      System.out.println( "Effects of passing reference to entire array:\n" +
         "The values of the original array are:" );
      for ( int value : array )
         System.out.printf( "   %d", value );
      System.out.println( "
   // output original array elements
   for ( int value : array )
      System.out.printf( " Xd", value );
```

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 1 of 2.)
Figure 7.13 next demonstrates that when a copy of an individual primitive-type array element is passed to a method, modifying the copy in the called method does not affect the original value of that element in the calling method’s array. Lines 26–28 output the value of array[3] (8) before invoking method modifyElement. Line 30 calls method modifyElement and passes array[3] as an argument. Remember that array[3] is actually one int value (8) in array. Therefore, the program passes a copy of the value of

```java
System.out.printf(
"Effects of passing array element value:
array[3] before modifyElement: %d\n", array[3]);
modifyElement( array[3] ); // attempt to modify array[3]
System.out.printf(
"array[3] after modifyElement: %d\n", array[3]);
}

modifyArray( array ); // pass array reference
System.out.println( "\n\nThe values of the modified array are:" );
// output modified array elements
for ( int value : array )
    System.out.printf( "   %d", value );

System.out.printf(
"\n\nEffects of passing array element value:\narray[3] before modifyElement: %d\n", array[3]);
modifyElement( array[3] ); // attempt to modify array[3]
System.out.printf(
"array[3] after modifyElement: %d\n", array[3]);

modifyArray( array2[] )
{
    for ( int counter = 0; counter < array2.length; counter++ )
        array2[ counter ] *= 2;
}

modifyElement( int element )
{
    element *= 2;
    System.out.printf(
"Value of element in modifyElement: %d\n", element);
}

Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5
The values of the modified array are:
   2   4   6   8  10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 2 of 2.)
```
Chapter 7 Arrays

array[3]. Method modifyElement (lines 43–48) multiplies the value received as an argument by 2, stores the result in its parameter element, then outputs the value of element (16). Since method parameters, like local variables, cease to exist when the method in which they are declared completes execution, the method parameter element is destroyed when method modifyElement terminates. Thus, when the program returns control to main, lines 31–32 output the unmodified value of array[3] (i.e., 8).

Notes on Passing Arguments to Methods

The preceding example demonstrated the different ways that arrays and primitive-type array elements are passed as arguments to methods. We now take a closer look at how arguments in general are passed to methods. Two ways to pass arguments in method calls in many programming languages are pass-by-value and pass-by-reference (also called call-by-value and call-by-reference). When an argument is passed by value, a copy of the argument’s value is passed to the called method. The called method works exclusively with the copy. Changes to the called method’s copy do not affect the original variable’s value in the caller.

When an argument is passed by reference, the called method can access the argument’s value in the caller directly and modify that data, if necessary. Pass-by-reference improves performance by eliminating the need to copy possibly large amounts of data.

Unlike some other languages, Java does not allow programmers to choose pass-by-value or pass-by-reference—all arguments are passed by value. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type int and double) and copies of references to objects (including references to arrays). Objects themselves cannot be passed to methods. When a method modifies a primitive-type parameter, changes to the parameter have no effect on the original argument value in the calling method. For example, when line 30 in main of Fig. 7.13 passes array[3] to method modifyElement, the statement in line 45 that doubles the value of parameter element has no effect on the value of array[3] in main. This is also true for reference-type parameters. If you modify a reference-type parameter by assigning it the reference of another object, the parameter refers to the new object, but the reference stored in the caller’s variable still refers to the original object.

Although an object’s reference is passed by value, a method can still interact with the referenced object by calling its public methods using the copy of the object’s reference. Since the reference stored in the parameter is a copy of the reference that was passed as an argument, the parameter in the called method and the argument in the calling method refer to the same object in memory. For example, in Fig. 7.13, both parameter array2 in method modifyArray and variable array in main refer to the same array object in memory. Any changes made using the parameter array2 are carried out on the same object that is referenced by the variable that was passed as an argument in the calling method. In Fig. 7.13, the changes made in modifyArray using array2 affect the contents of the array object referenced by array in main. Thus, with a reference to an object, the called method can manipulate the caller’s object directly.

Performance Tip 7.1

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would waste time and consume considerable storage for the copies of the arrays.
7.8 Case Study: Class GradeBook Using an Array to Store Grades

This section further evolves class GradeBook, introduced in Chapter 3 and expanded in Chapters 4–5. Recall that this class represents a grade book used by a professor to store and analyze a set of student grades. Previous versions of the class process a set of grades entered by the user, but do not maintain the individual grade values in instance variables of the class. Thus, repeat calculations require the user to reenter the same grades. One way to solve this problem would be to store each grade entered in an individual instance of the class. For example, we could create instance variables grade1, grade2, …, grade10 in class GradeBook to store 10 student grades. However, the code to total the grades and determine the class average would be cumbersome, and the class would not be able to process any more than 10 grades at a time. In this section, we solve this problem by storing grades in an array.

Storing Student Grades in an Array in Class GradeBook

The version of class GradeBook (Fig. 7.14) presented here uses an array of integers to store the grades of several students on a single exam. This eliminates the need to repeatedly input the same set of grades. Array grades is declared as an instance variable in line 7—therefore, each GradeBook object maintains its own set of grades. The class’s constructor (lines 10–14) has two parameters—the name of the course and an array of grades. When an application (e.g., class GradeBookTest in Fig. 7.15) creates a GradeBook object, the application passes an existing int array to the constructor, which assigns the array’s reference to instance variable grades (line 13). The size of the array grades is determined by the class that passes the array to the constructor. Thus, a GradeBook object can process a variable number of grades. The grade values in the passed array could have been input from a user or read from a file on disk (as discussed in Chapter 14). In our test application, we simply initialize an array with a set of grade values (Fig. 7.15, line 10). Once the grades are stored in instance variable grades of class GradeBook, all the class’s methods can access the elements of grades as often as needed to perform various calculations.

```
// Fig. 7.14: GradeBook.java
// Grade book using an array to store test grades.
public class GradeBook {
    private String courseName; // name of course this GradeBook represents
    private int grades[]; // array of student grades

    // two-argument constructor initializes courseName and grades array
    public GradeBook( String name, int gradesArray[] )
    {
        courseName = name; // initialize courseName
        grades = gradesArray; // store grades
    }
}
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 1 of 4.)
Chapter 7  Arrays

// method to set the course name
public void setCourseName( String name )
{
    courseName = name; // store the course name
} // end method setCourseName

// method to retrieve the course name
public String getCourseName()
{
    return courseName;
} // end method getCourseName

// display a welcome message to the GradeBook user
public void displayMessage()
{
    // getCourseName gets the name of the course
    System.out.printf( "Welcome to the grade book for %s!

",
    getCourseName() );
} // end method displayMessage

// perform various operations on the data
public void processGrades()
{
    // output grades array
    outputGrades();

    // call method getAverage to calculate the average grade
    System.out.printf( "Class average is %.2f
", getAverage() ) ;

    // call methods getMinimum and getMaximum
    System.out.printf( "Lowest grade is %d
Highest grade is %d

",
        getMinimum(), getMaximum() ) ;

    // call outputBarChart to print grade distribution chart
    outputBarChart();
} // end method processGrades

// find minimum grade
public int getMinimum()
{
    int lowGrade = grades [ 0 ]; // assume grades[ 0 ] is smallest

    // loop through grades array
    for ( int grade : grades )
    {
        // if grade lower than lowGrade, assign it to lowGrade
        if ( grade < lowGrade )
            lowGrade = grade; // new lowest grade
    } // end for

    return lowGrade; // return lowest grade
} // end method getMinimum

Fig. 7.14  GradeBook class using an array to store test grades. (Part 2 of 4.)
7.8 Case Study: Class GradeBook Using an Array to Store Grades

```java
public int getMaximum()
{
    int highGrade = grades[0]; // assume grades[0] is largest
    // loop through grades array
    for (int grade : grades)
    {
        // if grade greater than highGrade, assign it to highGrade
        if (grade > highGrade)
            highGrade = grade; // new highest grade
    } // end for
    return highGrade; // return highest grade
} // end method getMaximum

public double getAverage()
{
    int total = 0; // initialize total
    // sum grades for one student
    for (int grade : grades)
        total += grade;
    // return average of grades
    return (double) total / grades.length;
} // end method getAverage

public void outputBarChart()
{
    System.out.println("Grade distribution:");
    // stores frequency of grades in each range of 10 grades
    int frequency[] = new int[11];
    // for each grade, increment the appropriate frequency
    for (int grade : grades)
        ++frequency[grade / 10];
    // for each grade frequency, print bar in chart
    for (int count = 0; count < frequency.length; count++)
    {
        // output bar label ("00-09: ", ..., "90-99: ", "100: ")
        if (count == 10)
            System.out.printf("%5d: ", 100);
        else
            System.out.printf("%02d-%02d: ", count * 10, count * 10 + 9);
    }
}
```

Fig. 7.14 GradeBook class using an array to store test grades. (Part 3 of 4.)
320  Chapter 7  Arrays

Method processGrades (lines 37–51) contains a series of method calls that output a report summarizing the grades. Line 40 calls method outputGrades to print the contents of the array of grades. Lines 134–136 in method outputGrades use a for statement to output the students’ grades. A counter-controlled for must be used in this case, because lines 135–136 use counter variable student’s value to output each grade next to a particular student number (see Fig. 7.15). Although array indices start at 0, a professor would typically number students starting at 1. Thus, lines 135–136 output student + 1 as the student number to produce grade labels “Student 1: ”, “Student 2: ”, and so on.

Method processGrades next calls method getAverage (line 43) to obtain the average of the grades in the array. Method getAverage (lines 86–96) uses an enhanced for statement to total the values in array grades before calculating the average. The parameter in the enhanced for’s header (e.g., int grade) indicates that for each iteration, the int variable takes on a value in the array grades. Note that the averaging calculation in line 95 uses grades.length to determine the number of grades being averaged.

Lines 46–47 in method processGrades calls methods getMinimum and getMaximum to determine the lowest and highest grades of any student on the exam, respectively. Each of these methods uses an enhanced for statement to loop through array grades. Lines 59–64 in method getMinimum loop through the array. Lines 62–63 compare each grade to lOWGrade; if a grade is less than lOWGrade, lOWGrade is set to that grade. When line 66 executes, lOWGrade contains the lowest grade in the array. Method getMaximum (lines 70–83) works similarly to method getMinimum.

Finally, line 50 in method processGrades calls method outputBarChart to print a distribution chart of the grade data using a technique similar to that in Fig. 7.6. In that example, we manually calculated the number of grades in each category (i.e., 0–9, 10–19, ..., 90–99 and 100) by simply looking at a set of grades. In this example, lines 107–108 use a technique similar to that in Fig. 7.7 and Fig. 7.8 to calculate the frequency of grades in each category. Line 104 declares and creates array frequency of 11 ints to store the fre-
Case Study: Class GradeBook Using an Array to Store Grades

frequency of grades in each grade category. For each grade in array grades, lines 107–108 increment the appropriate element of the frequency array. To determine which element to increment, line 108 divides the current grade by 10 using integer division. For example, if grade is 85, line 108 increments frequency[8] to update the count of grades in the range 80–89. Lines 111–125 next print the bar chart (see Fig. 7.15) based on the values in array frequency. Like lines 23–24 of Fig. 7.6, lines 121–122 of Fig. 7.14 use a value in array frequency to determine the number of asterisks to display in each bar.

Class GradeBookTest That Demonstrates Class GradeBook

The application of Fig. 7.15 creates an object of class GradeBook (Fig. 7.14) using the int array gradesArray (declared and initialized in line 10). Lines 12–13 pass a course name and gradesArray to the GradeBook constructor. Line 14 displays a welcome message, and

```java
// Fig. 7.15: GradeBookTest.java
// Creates GradeBook object using an array of grades.
public class GradeBookTest {
    // main method begins program execution
    public static void main(String args[])
    {
        // array of student grades
        int gradesArray[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };

        GradeBook myGradeBook = new GradeBook("CS101 Introduction to Java Programming", gradesArray);
        myGradeBook.displayMessage();
        myGradeBook.processGrades();
    } // end main
} // end class GradeBookTest
```

Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

Student 1: 87
Student 2: 68
Student 3: 94
Student 4: 100
Student 5: 83
Student 6: 78
Student 7: 85
Student 8: 91
Student 9: 76
Student 10: 87

(continued…)

Fig. 7.15 | GradeBookTest creates a GradeBook object using an array of grades, then invokes method processGrades to analyze them. (Part 1 of 2.)
Chapter 7  Arrays

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09: *
10-19: **
20-29: ***
30-39: ****
40-49: ****
50-59: *****
60-69: *
70-79: **
80-89: ****
90-99: ****
100: *

Fig. 7.15 | GradeBookTest creates a GradeBook object using an array of grades, then invokes method processGrades to analyze them. (Part 2 of 2.)

line 15 invokes the GradeBook object’s processGrades method. The output summarizes the 10 grades in myGradeBook.

Software Engineering Observation 7.1

A test harness (or test application) is responsible for creating an object of the class being tested and providing it with data. This data could come from any of several sources. Test data can be placed directly into an array with an array initializer, it can come from the user at the keyboard, it can come from a file (as you will see in Chapter 14), or it can come from a network (as you will see in Chapter 24). After passing this data to the class’s constructor to instantiate the object, the test harness should call upon the object to test its methods and manipulate its data. Gathering data in the test harness like this allows the class to manipulate data from several sources.

7.9 Multidimensional Arrays

Multidimensional arrays with two dimensions are often used to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two indices. By convention, the first identifies the element’s row and the second its column. Arrays that require two indices to identify a particular element are called two-dimensional arrays. (Multidimensional arrays can have more than two dimensions.) Java does not support multidimensional arrays directly, but it does allow the programmer to specify one-dimensional arrays whose elements are also one-dimensional arrays, thus achieving the same effect. Figure 7.16 illustrates a two-dimensional array a that contains three rows and four columns (i.e., a three-by-four array). In general, an array with m rows and n columns is called an m-by-n array.

Every element in array a is identified in Fig. 7.16 by an array-access expression of the form a[ row ][ column ]; a is the name of the array, and row and column are the indices that uniquely identify each element in array a by row and column number. Note that the names of the elements in row 0 all have a first index of 0, and the names of the elements in column 3 all have a second index of 3.
7.9 Multidimensional Arrays

Arrays of One-Dimensional Arrays
Like one-dimensional arrays, multidimensional arrays can be initialized with array initializers in declarations. A two-dimensional array \( b \) with two rows and two columns could be declared and initialized with nested array initializers as follows:

```java
int b[][] = {{1, 2}, {3, 4}};
```

The initializer values are grouped by row in braces. So 1 and 2 initialize \( b[0][0] \) and \( b[0][1] \), respectively, and 3 and 4 initialize \( b[1][0] \) and \( b[1][1] \), respectively. The compiler counts the number of nested array initializers (represented by sets of braces within the outer braces) in the array declaration to determine the number of rows in array \( b \). The compiler counts the initializer values in the nested array initializer for a row to determine the number of columns in that row. As we will see momentarily, this means that rows can have different lengths.

Multidimensional arrays are maintained as arrays of one-dimensional arrays. Therefore array \( b \) in the preceding declaration is actually composed of two separate one-dimensional arrays—one containing the values in the first nested initializer list \( \{1, 2\} \) and one containing the values in the second nested initializer list \( \{3, 4\} \). Thus, array \( b \) itself is an array of two elements, each a one-dimensional array of \( int \) values.

Two-Dimensional Arrays with Rows of Different Lengths
The manner in which multidimensional arrays are represented makes them quite flexible. In fact, the lengths of the rows in array \( b \) are not required to be the same. For example,

```java
int b[][] = {{1, 2}, {3, 4, 5}};
```

creates integer array \( b \) with two elements (determined by the number of nested array initializers) that represent the rows of the two-dimensional array. Each element of \( b \) is a reference to a one-dimensional array of \( int \) variables. The \( int \) array for row 0 is a one-dimensional array with two elements (1 and 2), and the \( int \) array for row 1 is a one-dimensional array with three elements (3, 4 and 5).
Creating Two-Dimensional Arrays with Array-Creation Expressions

A multidimensional array with the same number of columns in every row can be created with an array-creation expression. For example, the following lines declare array b and assign it a reference to a three-by-four array:

```java
int b[][] = new int[3][4];
```

In this case, we use the literal values 3 and 4 to specify the number of rows and number of columns, respectively, but this is not required. Programs can also use variables to specify array dimensions, because `new` creates arrays at execution time—not at compile time. As with one-dimensional arrays, the elements of a multidimensional array are initialized when the array object is created.

A multidimensional array in which each row has a different number of columns can be created as follows:

```java
int b[][] = new int[2][ ]; // create 2 rows
b[0] = new int[5]; // create 5 columns for row 0
b[1] = new int[3]; // create 3 columns for row 1
```

The preceding statements create a two-dimensional array with two rows. Row 0 has five columns, and row 1 has three columns.

Two-Dimensional Array Example: Displaying Element Values

Figure 7.17 demonstrates initializing two-dimensional arrays with array initializers and using nested for loops to traverse the arrays (i.e., manipulate every element of each array).

Class `InitArray`'s `main` declares two arrays. The declaration of `array1` (line 9) uses nested array initializers to initialize the first row of the array to the values 1, 2 and 3, and the second row to the values 4, 5 and 6. The declaration of `array2` (line 10) uses nested initializers of different lengths. In this case, the first row is initialized to have two elements with the values 1 and 2, respectively. The second row is initialized to have one element with the value 3. The third row is initialized to have three elements with the values 4, 5 and 6, respectively.

```java
public class InitArray
{
    // create and output two-dimensional arrays
    public static void main( String args[] )
    {
        int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
        int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
        System.out.println( "Values in array1 by row are" );
        outputArray( array1 ); // displays array1 by row
        System.out.println( "\nValues in array2 by row are" );
        outputArray( array2 ); // displays array2 by row
    }
}
```

Fig. 7.17 | Initializing two-dimensional arrays. (Part I of 2.)
7.9 Multidimensional Arrays

Lines 13 and 16 call method `outputArray` (lines 20–31) to output the elements of `array1` and `array2`, respectively. Method `outputArray` specifies the array parameter as `int array[][]` to indicate that the method receives a two-dimensional array. The for statement (lines 23–30) outputs the rows of a two-dimensional array. In the loop-continuation condition of the outer for statement, the expression `array.length` determines the number of rows in the array. In the inner for statement, the expression `array[row].length` determines the number of columns in the current row of the array. This condition enables the loop to determine the exact number of columns in each row.

**Common Multidimensional-Array Manipulations Performed with for Statements**

Many common array manipulations use `for` statements. As an example, the following `for` statement sets all the elements in row 2 of array `a` in Fig. 7.16 to zero:

```java
for ( int column = 0; column < a[ 2 ].length; column++ )
    a[ 2 ][ column ] = 0;
```

We specified row 2; therefore, we know that the first index is always 2 (0 is the first row, and 1 is the second row). This for loop varies only the second index (i.e., the column index). If row 2 of array `a` contains four elements, then the preceding `for` statement is equivalent to the assignment statements

```java
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

---

Fig. 7.17 | Initializing two-dimensional arrays. (Part 2 of 2.)
The following nested for statement totals the values of all the elements in array `a`:

```java
int total = 0;
for ( int row = 0; row < a.length; row++ )
    for ( int column = 0; column < a[row].length; column++ )
        total += a[row][column];
```

This nested for statements total the array elements one row at a time. The outer for statement begins by setting the `row` index to 0 so that the first row's elements can be totaled by the inner for statement. The outer for then increments `row` to 1 so that the second row can be totaled. Then, the outer for increments `row` to 2 so that the third row can be totaled. The variable `total` can be displayed when the outer for statement terminates.

### 7.10 Case Study: Class GradeBook Using a Two-Dimensional Array

In Section 7.8, we presented class GradeBook (Fig. 7.14), which used a one-dimensional array to store student grades on a single exam. In most semesters, students take several exams. Professors are likely to want to analyze grades across the entire semester, both for a single student and for the class as a whole.

**Storing Student Grades in a Two-Dimensional Array in Class GradeBook**

Figure 7.18 contains a version of class GradeBook that uses a two-dimensional array to store the grades of a number of students on multiple exams. Each row of the array represents a single student's grades for the entire course, and each column represents a grade on one of the exams the students took during the course. An application such as GradeBookTest (Fig. 7.19) passes the array as an argument to the GradeBook constructor. In this example, we use a ten-by-three array containing ten students' grades on three exams. Five methods perform array manipulations to process the grades. Each method is similar to its counterpart in the earlier one-dimensional array version of class GradeBook (Fig. 7.14).

- **getMinimum** (lines 52–70) determines the lowest grade of any student for the semester. Method **getMaximum** (lines 73–91) determines the highest grade of any student for the semester. Method **getAverage** (lines 94–104) determines a particular student's semester average. Method **outputBarChart** (lines 107–137) outputs a bar chart of the distribution of all student grades for the semester. Method **outputGrades** (lines 140–164) outputs the two-dimensional array in a tabular format, along with each student's semester average.

Methods `getMinimum`, `getMaximum`, `outputBarChart` and `outputGrades` each loop through array grades by using nested for statements—for example, the nested enhanced for statement from the declaration of method `getMinimum` (lines 58–67). The outer enhanced for statement iterates through the two-dimensional array grades, assigning successive rows to parameter `studentGrades` on successive iterations. The square brackets following the parameter name indicate that `studentGrades` refers to a one-dimensional `int` array—namely, a row in array `grades` containing one student's grades. To find the lowest overall grade, the inner for statement compares the elements of the current one-dimen-
7.10 Case Study: Class GradeBook Using a Two-Dimensional Array

```java
// Fig. 7.18: GradeBook.java
// Grade book using a two-dimensional array to store grades.

public class GradeBook {
    private String courseName; // name of course this grade book represents
    private int grades[] []; // two-dimensional array of student grades

    // two-argument constructor initializes courseName and grades array
    public GradeBook( String name, int gradesArray[][] )
    {
        courseName = name; // initialize courseName
        grades = gradesArray; // store grades
    } // end two-argument GradeBook constructor

    // method to set the course name
    public void setCourseName( String name )
    {
        courseName = name; // store the course name
    } // end method setCourseName

    // method to retrieve the course name
    public String getCourseName()
    {
        return courseName;
    } // end method getCourseName

    // display a welcome message to the GradeBook user
    public void displayMessage()
    {
        // getCourseName gets the name of the course
        System.out.printf( "Welcome to the grade book for %s!
\n", getCourseName() );
    } // end method displayMessage

    // perform various operations on the data
    public void processGrades()
    {
        // output grades array
        outputGrades();

        // call methods getMinimum and getMaximum
        System.out.printf( "\n\n" + "Lowest grade in the grade book is", getMinimum() );
        System.out.printf( "\n\n" + "Highest grade in the grade book is", getMaximum() );

        // output grade distribution chart of all grades on all tests
        outputBarChart();
    } // end method processGrades

    // find minimum grade
    public int getMinimum()
    {
    }
}
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 1 of 4.)
Chapter 7 Arrays

```java
// assume first element of grades array is smallest
int lowGrade = grades[0][0];

// loop through rows of grades array
for (int studentGrades[] : grades) {
    // loop through columns of current row
    for (int grade : studentGrades) {
        // if grade less than lowGrade, assign it to lowGrade
        if (grade < lowGrade)
            lowGrade = grade;
    }
}
return lowGrade; // return lowest grade
}

// find maximum grade
public int getMaximum() {
    int highGrade = grades[0][0];
    // loop through rows of grades array
    for (int studentGrades[] : grades) {
        // loop through columns of current row
        for (int grade : studentGrades) {
            // if grade greater than highGrade, assign it to highGrade
            if (grade > highGrade)
                highGrade = grade;
        }
    }
    return highGrade; // return highest grade
}

// determine average grade for particular set of grades
public double getAverage(int setOfGrades[]) {
    int total = 0; // initialize total
    // sum grades for one student
    for (int grade : setOfGrades)
        total += grade;
    // return average of grades
    return (double) total / setOfGrades.length;
}
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 2 of 4.)
// output bar chart displaying overall grade distribution
public void outputBarChart()
{
    System.out.println( "Overall grade distribution:" );
    // stores frequency of grades in each range of 10 grades
    int frequency[] = new int[ 11 ];
    // for each grade in GradeBook, increment the appropriate frequency
    for ( int studentGrades[] : grades )
    {
        for ( int grade : studentGrades )
            ++frequency[ grade / 10 ];
    } // end outer for
    // for each grade frequency, print bar in chart
    for ( int count = 0; count < frequency.length; count++ )
    {
        // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
        if ( count == 10 )
            System.out.printf( "%5d: ", 100 );
        else
            System.out.printf( "%02d-%02d: ",
                               count * 10, count * 10 + 9 );
        // print bar of asterisks
        for ( int stars = 0; stars < frequency[ count ]; stars++ )
            System.out.print( "*" );
        System.out.println(); // start a new line of output
    } // end inner for
    System.out.println(); // start a new line of output
    // end method outputBarChart

public void outputGrades()
{
    System.out.println( "The grades are:
" );
    System.out.print( "            "); // align column heads
    // create a column heading for each of the tests
    for ( int test = 0; test < grades[ 0 ].length; test++ )
        System.out.printf( "Test %d  ", test + 1 );
    System.out.println( "Average" ); // student average column heading
    // create rows/columns of text representing array grades
    for ( int student = 0; student < grades.length; student++ )
    {
        System.out.printf( "Student %2d", student + 1 );
        for ( int test : grades[ student ] ) // output student's grades
            System.out.printf( "%8d", test );
    } // end inner for
} // end method outputGrades

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 3 of 4.)
sional array studentGrades to variable lowGrade. For example, on the first iteration of the outer for, row 0 of grades is assigned to parameter studentGrades. The inner enhanced for statement then loops through studentGrades and compares each grade value with lowGrade. If a grade is less than lowGrade, lowGrade is set to that grade. On the second iteration of the outer enhanced for statement, row 1 of grades is assigned to studentGrades, and the elements of this row are compared with variable lowGrade. This repeats until all rows of grades have been traversed. When execution of the nested statement is complete, lowGrade contains the lowest grade in the two-dimensional array. Method getMaximum works similarly to method getMinimum.

Method outputBarChart in Fig. 7.18 is nearly identical to the one in Fig. 7.14. However, to output the overall grade distribution for a whole semester, the method here uses a nested enhanced for statement (lines 115–119) to create the one-dimensional array frequency based on all the grades in the two-dimensional array. The rest of the code in each of the two outputBarChart methods that displays the chart is identical.

Method outputGrades (lines 140–164) also uses nested for statements to output values of the array grades and each student's semester average. The output in Fig. 7.19 shows the result, which resembles the tabular format of a professor's physical grade book. Lines 146–147 print the column headings for each test. We use a counter-controlled for statement here so that we can identify each test with a number. Similarly, the for statement in lines 152–163 first outputs a row label using a counter variable to identify each student (line 154). Although array indices start at 0, note that lines 147 and 154 output test + 1 and student + 1, respectively, to produce test and student numbers starting at 1 (see Fig. 7.19). The inner for statement in lines 156–157 uses the outer for statement's counter variable student to loop through a specific row of array grades and output each student's test grade. Note that an enhanced for statement can be nested in a counter-controlled for statement, and vice versa. Finally, line 161 obtains each student's semester average by passing the current row of grades (i.e., grades[ student ]) to method getAverage.

Method getAverage (lines 94–104) takes one argument—a one-dimensional array of test results for a particular student. When line 161 calls getAverage, the argument is grades[ student ], which specifies that a particular row of the two-dimensional array grades should be passed to getAverage. For example, based on the array created in Fig. 7.19, the argument grades[ 1 ] represents the three values (a one-dimensional array of grades) stored in row 1 of the two-dimensional array grades. Recall that a two-dimensional array is an array whose elements are one-dimensional arrays. Method getAverage calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result as a double value (line 103).
Class GradeBookTest That Demonstrates Class GradeBook

The application in Fig. 7.19 creates an object of class GradeBook (Fig. 7.18) using the two-dimensional array of ints named gradesArray (declared and initialized in lines 10–19). Lines 21–22 pass a course name and gradesArray to the GradeBook constructor. Lines 23–24 then invoke myGradeBook's displayMessage and processGrades methods to display a welcome message and obtain a report summarizing the students' grades for the semester, respectively.

```java
// Fig. 7.19: GradeBookTest.java
// Creates GradeBook object using a two-dimensional array of grades.

public class GradeBookTest {
    public static void main( String args[] ) {
        // main method begins program execution
        int gradesArray[][] = { { 87, 96, 70 },
                               { 68, 87, 90 },
                               { 94, 100, 90 },
                               { 100, 81, 82 },
                               { 83, 65, 85 },
                               { 78, 87, 65 },
                               { 85, 75, 83 },
                               { 91, 94, 100 },
                               { 76, 72, 84 },
                               { 87, 93, 73 } };

        GradeBook myGradeBook = new GradeBook(
            "CS101 Introduction to Java Programming", gradesArray );
        myGradeBook.displayMessage();
        myGradeBook.processGrades();
    } // end main
} // end class GradeBookTest
```

Welcome to the grade book for CS101 Introduction to Java Programming!

The grades are:

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student 1</td>
<td>87</td>
<td>96</td>
<td>70</td>
<td>84.33</td>
</tr>
<tr>
<td>Student 2</td>
<td>68</td>
<td>87</td>
<td>90</td>
<td>81.67</td>
</tr>
<tr>
<td>Student 3</td>
<td>94</td>
<td>100</td>
<td>90</td>
<td>94.67</td>
</tr>
<tr>
<td>Student 4</td>
<td>100</td>
<td>81</td>
<td>82</td>
<td>87.67</td>
</tr>
<tr>
<td>Student 5</td>
<td>83</td>
<td>65</td>
<td>85</td>
<td>77.67</td>
</tr>
<tr>
<td>Student 6</td>
<td>78</td>
<td>87</td>
<td>65</td>
<td>76.67</td>
</tr>
<tr>
<td>Student 7</td>
<td>85</td>
<td>75</td>
<td>83</td>
<td>81.00</td>
</tr>
<tr>
<td>Student 8</td>
<td>91</td>
<td>94</td>
<td>100</td>
<td>95.00</td>
</tr>
<tr>
<td>Student 9</td>
<td>76</td>
<td>72</td>
<td>84</td>
<td>77.33</td>
</tr>
<tr>
<td>Student 10</td>
<td>87</td>
<td>93</td>
<td>73</td>
<td>84.33</td>
</tr>
</tbody>
</table>

(continued…)

Fig. 7.19 | Creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 1 of 2.)
7.11 Variable-Length Argument Lists

With variable-length argument lists, you can create methods that receive an unspecified number of arguments. An argument type followed by an ellipsis (…) in a method’s parameter list indicates that the method receives a variable number of arguments of that particular type. This use of the ellipsis can occur only once in a parameter list, and the ellipsis, together with its type, must be placed at the end of the parameter list. While programmers can use method overloading and array passing to accomplish much of what is accomplished with “varargs,” or variable-length argument lists, using an ellipsis in a method’s parameter list is more concise.

Figure 7.20 demonstrates method average (lines 7–16), which receives a variable-length sequence of doubles. Java treats the variable-length argument list as an array whose elements are all of the same type. Hence, the method body can manipulate the parameter numbers as an array of doubles. Lines 12–13 use the enhanced for loop to walk through the array and calculate the total of the doubles in the array. Line 15 accesses numbers.length to obtain the size of the numbers array for use in the averaging calculation. Lines 29, 31 and 33 in main call method average with two, three and four arguments, respectively. Method average has a variable-length argument list (line 7), so it can average as many double arguments as the caller passes. The output shows that each call to method average returns the correct value.

```java
// Fig. 7.20: VarargsTest.java
// Using variable-length argument lists.
public class VarargsTest {
    // calculate average
    public static double average(double... numbers) {
        // Fig. 7.20 | Using variable-length argument lists. (Part 1 of 2.)
    }
```
7.12 Using Command-Line Arguments

Common Programming Error 7.6
Placing an ellipsis indicating a variable-length argument list in the middle of a method parameter list is a syntax error. An ellipsis may be placed only at the end of the parameter list.

7.12 Using Command-Line Arguments

On many systems it is possible to pass arguments from the command line (these are known as command-line arguments) to an application by including a parameter of type String[] (i.e., an array of Strings) in the parameter list of main, exactly as we have done in every application in the book. By convention, this parameter is named args. When an application is executed using the java command, Java passes the command-line arguments that appear after the class name in the java command to the application’s main method as Strings in the array args. The number of arguments passed in from the command line is
obtained by accessing the array’s length attribute. For example, the command “java MyClass a b” passes two command-line arguments, a and b, to application MyClass. Note that command-line arguments are separated by white space, not commas. When this command executes, MyClass’s main method receives the two-element array args (i.e., args.length is 2) in which args[0] contains the String ”a” and args[1] contains the String ”b”. Common uses of command-line arguments include passing options and file names to applications.

Figure 7.21 uses three command-line arguments to initialize an array. When the program executes, if args.length is not 3, the program prints an error message and terminates (lines 9–12). Otherwise, lines 14–32 initialize and display the array based on the values of the command-line arguments.

The command-line arguments become available to main as Strings in args. Line 16 gets args[0]—a String that specifies the array size—and converts it to an int value that the program uses to create the array in line 17. The static method parseInt of class Integer converts its String argument to an int.

```
// Fig. 7.21: InitArray.java
// Using command-line arguments to initialize an array.

public class InitArray
{
    public static void main(  )
    {
        // check number of command-line arguments
        if ( args.length != 3 )
            System.out.println(
                "Error: Please re-enter the entire command, including\n" +
                "an array size, initial value and increment."
            );
        else
        {
            // get array size from first command-line argument
            int arrayLength = Integer.parseInt( args[ 0 ]);
            int array[] = new int[ arrayLength ]; // create array

            // get initial value and increment from command-line arguments
            int initialValue = Integer.parseInt( args[ 1 ]);
            int increment = Integer.parseInt( args[ 2 ]);

            // calculate value for each array element
            for ( int counter = 0; counter < array.length; counter++ )
                array[ counter ] = initialValue + increment * counter;

            System.out.printf( "%s%8s
", "Index", "Value" );

            // display array index and value
            for ( int counter = 0; counter < array.length; counter++ )
                System.out.printf( "%5d%8d
", counter, array[ counter ] );
        } // end else
    } // end main
} // end class InitArray
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 1 of 2.)
7.13 (Optional) GUI and Graphics Case Study: Drawing Arcs  

Using Java’s graphics features, we can create complex drawings that would be more tedious to code line by line. In Fig. 7.22 and Fig. 7.23, we use arrays and repetition statements to draw a rainbow by using Graphics method fillArc. Drawing arcs in Java is similar to drawing ovals—an arc is simply a section of an oval.

Figure 7.22 begins with the usual import statements for creating drawings (lines 3–5). Lines 9–10 declare and create two new colors—VIOLET and INDIGO. As you may know, the colors of a rainbow are red, orange, yellow, green, blue, indigo and violet. Java has pre-

```java
java InitArray
Error: Please re-enter the entire command, including an array size, initial value and increment.
```

```java
java InitArray 5 0 4
Index | Value  
--- | ---  
0    | 0  
1    | 4  
2    | 8  
3    | 12 
4    | 16 
```

```java
java InitArray 10 1 2
Index | Value  
--- | ---  
0    | 1  
1    | 3  
2    | 5  
3    | 7  
4    | 9  
5    | 11 
6    | 13 
7    | 15 
8    | 17 
9    | 19 
```

Fig. 7.21 Initializing an array using command-line arguments. (Part 2 of 2.)
defined constants only for the first five colors. Lines 15–17 initialize an array with the
colors of the rainbow, starting with the innermost arcs first. The array begins with two
Color.WHITE elements, which, as you will soon see, are for drawing the empty arcs at the
center of the rainbow. Note that the instance variables can be initialized when they are declared, as shown in lines 10–17. The constructor (lines 20–23) contains a single statement that calls method setBackground (which is inherited from class JPanel) with the parameter Color.WHITE. Method setBackground takes a single Color argument and sets the background of the component to that color.

Line 30 in paintComponent declares local variable radius, which determines the radius of each arc. Local variables centerX and centerY (lines 33–34) determine the location of the midpoint on the base of the rainbow. The loop at lines 37–46 uses control variable counter to count backward from the end of the array, drawing the largest arcs first and placing each successive smaller arc on top of the previous one. Line 40 sets the color to draw the current arc from the array. The reason we have Color.WHITE entries at the beginning of the array is to create the empty arc in the center. Otherwise, the center of the rainbow would just be a solid violet semicircle. [Note: You can change the individual colors and the number of entries in the array to create new designs.]

The fillArc method call at lines 43–45 draws a filled semicircle. Method fillArc requires six parameters. The first four represent the bounding rectangle in which the arc will be drawn. The first two of these specify the coordinates for the upper-left corner of

---

```
// Fig. 7.23: DrawRainbowTest.java
// Test application to display a rainbow.
import javax.swing.JFrame;

public class DrawRainbowTest
{
    public static void main( String args[] )
    {
        DrawRainbow panel = new DrawRainbow();
        JFrame application = new JFrame();

        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        application.add( panel );
        application.setSize( 400, 250 );
        application.setVisible( true );
    } // end main
} // end class DrawRainbowTest
```

---

**Fig. 7.23** | Creating JFrame to display a rainbow.
the bounding rectangle, and the next two specify its width and height. The fifth parameter is the starting angle on the oval, and the sixth specifies the sweep, or the amount of arc to cover. The starting angle and sweep are measured in degrees, with zero degrees pointing right. A positive sweep draws the arc counterclockwise, while a negative sweep draws the arc clockwise. A method similar to `fillArc` is `drawArc`—it requires the same parameters as `fillArc`, but draws the edge of the arc rather than filling it.

Class `DrawRainbowTest` (Fig. 7.23) creates and sets up a `JFrame` to display the rainbow. Once the program makes the `JFrame` visible, the system calls the `paintComponent` method in class `DrawRainbow` to draw the rainbow on the screen.

**GUI and Graphics Case Study Exercise**

**7.1** *(Drawing Spirals)* In this exercise, you will draw spirals with methods `drawLine` and `drawArc`.

a) Draw a square-shaped spiral (as in the left screen capture of Fig. 7.24), centered on the panel, using method `drawLine`. One technique is to use a loop that increases the line length after drawing every second line. The direction in which to draw the next line should follow a distinct pattern, such as down, left, up, right.

b) Draw a circular spiral (as in the right screen capture of Fig. 7.24), using method `drawArc` to draw one semicircle at a time. Each successive semicircle should have a larger radius (as specified by the bounding rectangle’s width) and should continue drawing where the previous semicircle finished.

---

**7.14 (Optional) Software Engineering Case Study: Collaboration Among Objects**

In this section, we concentrate on the collaborations (interactions) among objects. When two objects communicate with each other to accomplish a task, they are said to collaborate—objects do this by invoking one another’s operations. A collaboration consists of an object of one class sending a **message** to an object of another class. Messages are sent in Java via method calls.

In Section 6.14, we determined many of the operations of the classes in our system. In this section, we concentrate on the messages that invoke these operations. To identify
7.14 Collaboration Among Objects

the collaborations in the system, we return to the requirements document in Section 2.9. Recall that this document specifies the range of activities that occur during an ATM session (e.g., authenticating a user, performing transactions). The steps used to describe how the system must perform each of these tasks are our first indication of the collaborations in our system. As we proceed through this and the remaining Software Engineering Case Study sections, we may discover additional collaborations.

Identifying the Collaborations in a System

We identify the collaborations in the system by carefully reading the sections of the requirements document that specify what the ATM should do to authenticate a user and to perform each transaction type. For each action or step described in the requirements document, we decide which objects in our system must interact to achieve the desired result. We identify one object as the sending object and another as the receiving object. We then select one of the receiving object’s operations (identified in Section 6.14) that must be invoked by the sending object to produce the proper behavior. For example, the ATM displays a welcome message when idle. We know that an object of class Screen displays a message to the user via its displayMessage operation. Thus, we decide that the system can display a welcome message by employing a collaboration between the ATM and the Screen in which the ATM sends a displayMessage message to the Screen by invoking the displayMessage operation of class Screen. [Note: To avoid repeating the phrase “an object of class…,” we refer to an object by using its class name preceded by an article (e.g., “a,” “an” or “the”)—for example, “the ATM” refers to an object of class ATM.]

Figure 7.25 lists the collaborations that can be derived from the requirements document. For each sending object, we list the collaborations in the order in which they first occur during an ATM session (i.e., the order in which they are discussed in the require-

<table>
<thead>
<tr>
<th>An object of class...</th>
<th>sends the message...</th>
<th>to an object of class...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>getInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>authenticateUser</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>BalanceInquiry</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>Withdrawal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deposit</td>
</tr>
<tr>
<td>BalanceInquiry</td>
<td>getAvailableBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>getTotalBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>displayMessage</td>
<td>Screen</td>
</tr>
<tr>
<td></td>
<td>getInput</td>
<td>Keypad</td>
</tr>
<tr>
<td></td>
<td>getAvailableBalance</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>isSufficientCashAvailable</td>
<td>CashDispenser</td>
</tr>
<tr>
<td></td>
<td>debit</td>
<td>BankDatabase</td>
</tr>
<tr>
<td></td>
<td>dispenseCash</td>
<td>CashDispenser</td>
</tr>
</tbody>
</table>

Fig. 7.25 | Collaborations in the ATM system. (Part 1 of 2.)
ments document). We list each collaboration involving a unique sender, message and recipient only once, even though the collaborations may occur at several different times throughout an ATM session. For example, the first row in Fig. 7.25 indicates that the ATM collaborates with the Screen whenever the ATM needs to display a message to the user.

Let’s consider the collaborations in Fig. 7.25. Before allowing a user to perform any transactions, the ATM must prompt the user to enter an account number, then to enter a PIN. It accomplishes each of these tasks by sending a displayMessage message to the Screen. Both of these actions refer to the same collaboration between the ATM and the Screen, which is already listed in Fig. 7.25. The ATM obtains input in response to a prompt by sending a getInput message to the Keypad. Next, the ATM must determine whether the user-specified account number and PIN match those of an account in the database. It does so by sending an authenticateUser message to the BankDatabase. Recall that the BankDatabase cannot authenticate a user directly—only the user’s Account (i.e., the Account that contains the account number specified by the user) can access the user’s PIN on record to authenticate the user. Figure 7.25 therefore lists a collaboration in which the BankDatabase sends a validatePIN message to an Account.

After the user is authenticated, the ATM displays the main menu by sending a series of displayMessage messages to the Screen and obtains input containing a menu selection by sending a getInput message to the Keypad. We have already accounted for these collaborations, so we do not add anything to Fig. 7.25. After the user chooses a type of transaction to perform, the ATM executes the transaction by sending an execute message to an object of the appropriate transaction class (i.e., a BalanceInquiry, a Withdrawal or a Deposit). For example, if the user chooses to perform a balance inquiry, the ATM sends an execute message to a BalanceInquiry.

Further examination of the requirements document reveals the collaborations involved in executing each transaction type. A BalanceInquiry retrieves the amount of money available in the user’s account by sending a getAvailableBalance message to the BankDatabase, which responds by sending the same message back to the user’s Account. Similarly, the BalanceInquiry retrieves the amount of money on deposit by sending a getTotalBalance message to the BankDatabase, which sends the same message back to the ATM.
7.14 Collaboration Among Objects

to the user’s Account. To display both measures of the user’s balance at the same time, the
BalanceInquiry sends a displayMessage message to the Screen.

A Withdrawal sends a series of displayMessage messages to the Screen to display a
menu of standard withdrawal amounts (i.e., $20, $40, $60, $100, $200). The Withdrawal
sends a getInput message to the Keypad to obtain the user’s menu selection. Next, the
Withdrawal determines whether the requested withdrawal amount is less than or equal to
the user’s account balance. The Withdrawal can obtain the amount of money available in
the user’s account by sending a getAvailableBalance message to the BankDatabase. The
Withdrawal then tests whether the cash dispenser contains enough cash by sending an
isSufficientCashAvailable message to the CashDispenser. A Withdrawal sends a
debit message to the BankDatabase to decrease the user’s account balance. The BankDa-
base in turn sends the same message to the appropriate Account. Recall that debiting
funds from an Account decreases both the totalBalance and the availableBalance. To
dispense the requested amount of cash, the Withdrawal sends a dispenseCash message to
the CashDispenser. Finally, the Withdrawal sends a displayMessage message to the
Screen, instructing the user to take the cash.

A Deposit responds to an execute message first by sending a displayMessage mes-
gage to the Screen to prompt the user for a deposit amount. The Deposit sends a get-
Input message to the Keypad to obtain the user’s input. The Deposit then sends a
displayMessage message to the Screen to tell the user to insert a deposit envelope. To
determine whether the deposit slot received an incoming deposit envelope, the Deposit
sends an isEnvelopeReceived message to the DepositSlot. The Deposit updates the
user’s account by sending a credit message to the BankDatabase, which subsequently
sends a credit message to the user’s Account. Recall that crediting funds to an Account
increases the totalBalance but not the availableBalance.

Interaction Diagrams

Now that we have identified a set of possible collaborations between the objects in our
ATM system, let us graphically model these interactions using the UML. The UML pro-
vides several types of interaction diagrams that model the behavior of a system by model-
ing how objects interact. The communication diagram emphasizes which objects
participate in collaborations. [Note: Communication diagrams were called collaboration
diagrams in earlier versions of the UML.] Like the communication diagram, the sequence
diagram shows collaborations among objects, but it emphasizes when messages are sent be-
tween objects over time.

Communication Diagrams

Figure 7.26 shows a communication diagram that models the ATM executing a
BalanceInquiry. Objects are modeled in the UML as rectangles containing names in the
form objectName : ClassName. In this example, which involves only one object of each
type, we disregard the object name and list only a colon followed by the class name. [Note:
Specifying the name of each object in a communication diagram is recommended when
modeling multiple objects of the same type.] Communicating objects are connected with
solid lines, and messages are passed between objects along these lines in the direction
shown by arrows. The name of the message, which appears next to the arrow, is the name
of an operation (i.e., a method in Java) belonging to the receiving object—think of the
name as a “service” that the receiving object provides to sending objects (its “clients”).
The solid filled arrow in Fig. 7.26 represents a message—or synchronous call—in the UML and a method call in Java. This arrow indicates that the flow of control is from the sending object (the ATM) to the receiving object (a BalanceInquiry). Since this is a synchronous call, the sending object may not send another message, or do anything at all, until the receiving object processes the message and returns control to the sending object. The sender just waits. For example, in Fig. 7.26, the ATM calls method execute of a BalanceInquiry and may not send another message until execute has finished and returns control to the ATM. [Note: If this were an asynchronous call, represented by a stick arrowhead, the sending object would not have to wait for the receiving object to return control—it would continue sending additional messages immediately following the asynchronous call. Asynchronous calls are implemented in Java using a technique called multithreading, which is discussed in Chapter 23, Multithreading.]

**Sequence of Messages in a Communication Diagram**

Figure 7.27 shows a communication diagram that models the interactions among objects in the system when an object of class BalanceInquiry executes. We assume that the object’s accountNumber attribute contains the account number of the current user. The collaborations in Fig. 7.27 begin after the ATM sends an execute message to a BalanceInquiry (i.e., the interaction modeled in Fig. 7.26). The number to the left of a message name indicates the order in which the message is passed. The sequence of messages in a communication diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message 1 and ends with message 3. The BalanceInquiry first sends a getAvailableBalance message to the BankDatabase (message 1), then sends a getTotalBalance message to the BankDatabase (message 2). Within the parentheses following a message name, we can specify a comma-separated list of the names of the parameters sent with the message (i.e., arguments in a Java method call)—the BalanceInquiry passes attribute accountNumber with its messages to the BankDatabase to indicate which Account’s balance information to retrieve. Recall from Fig. 6.22 that operations getAvailableBalance and getTotalBalance of class BankDatabase each require a parameter to identify an account. The BalanceInquiry next displays the availableBalance and the totalBalance to the user by passing a displayMessage message to the Screen (message 3) that includes a parameter indicating the message to be displayed.

Note that Fig. 7.27 models two additional messages passing from the BankDatabase to an Account (message 1.1 and message 2.1). To provide the ATM with the two balances of the user’s Account (as requested by messages 1 and 2), the BankDatabase must pass a getAvailableBalance and a getTotalBalance message to the user’s Account. Such messages passed within the handling of another message are called nested messages. The UML recommends using a decimal numbering scheme to indicate nested messages. For example, message 1.1 is the first message nested in message 1—the BankDatabase passes a getAvailableBalance message during BankDatabase’s processing of a message by the same name. [Note: If the BankDatabase needed to pass a second nested message while pro-
cessing message 1, the second message would be numbered 1.2. A message may be passed only when all the nested messages from the previous message have been passed. For example, the BalanceInquiry passes message 3 only after messages 2 and 2.1 have been passed, in that order.

The nested numbering scheme used in communication diagrams helps clarify precisely when and in what context each message is passed. For example, if we numbered the messages in Fig. 7.27 using a flat numbering scheme (i.e., 1, 2, 3, 4, 5), someone looking at the diagram might not be able to determine that BankDatabase passes the getAvailableBalance message (message 1.1) to an Account during the BankDatabase’s processing of message 1, as opposed to after completing the processing of message 1. The nested decimal numbers make it clear that the second getAvailableBalance message (message 1.1) is passed to an Account within the handling of the first getAvailableBalance message (message 1) by the BankDatabase.

**Sequence Diagrams**

Communication diagrams emphasize the participants in collaborations, but model their timing a bit awkwardly. A sequence diagram helps model the timing of collaborations more clearly. Figure 7.28 shows a sequence diagram modeling the sequence of interactions that occur when a withdrawal executes. The dotted line extending down from an object’s rectangle is that object’s lifeline, which represents the progression of time. Actions occur along an object’s lifeline in chronological order from top to bottom—an action near the top happens before one near the bottom.

Message passing in sequence diagrams is similar to message passing in communication diagrams. A solid arrow with a filled arrowhead extending from the sending object to the receiving object represents a message between two objects. The arrowhead points to an activation on the receiving object’s lifeline. An activation, shown as a thin vertical rectangle, indicates that an object is executing. When an object returns control, a return mes-

---

**Fig. 7.27 | Communication diagram for executing a balance inquiry.**
sage, represented as a dashed line with a stick arrowhead, extends from the activation of the object returning control to the activation of the object that initially sent the message. To eliminate clutter, we omit the return-message arrows—the UML allows this practice to make diagrams more readable. Like communication diagrams, sequence diagrams can indicate message parameters between the parentheses following a message name.

The sequence of messages in Fig. 7.28 begins when a Withdrawal prompts the user to choose a withdrawal amount by sending a displayMessage message to the Screen. The Withdrawal then sends a getInput message to the Keypad, which obtains input from the user. We have already modeled the control logic involved in a Withdrawal in the activity diagram of Fig. 5.31, so we do not show this logic in the sequence diagram of Fig. 7.28.
Instead, we model the best-case scenario in which the balance of the user’s account is
greater than or equal to the chosen withdrawal amount, and the cash dispenser contains a
sufficient amount of cash to satisfy the request. For information on how to model control
logic in a sequence diagram, please refer to the web resources and recommended readings
listed at the end of Section 2.9.

After obtaining a withdrawal amount, the Withdrawal sends a getAvailableBalance
message to the BankDatabase, which in turn sends a getAvailableBalance message to the
user’s Account. Assuming that the user’s account has enough money available to permit
the transaction, the Withdrawal next sends an isSufficientCashAvailable message to the
CashDispenser. Assuming that there is enough cash available, the Withdrawal
decreases the balance of the user’s account (i.e., both the totalBalance and the avail-
ableBalance) by sending a debit message to the BankDatabase. The BankDatabase
responds by sending a debit message to the user’s Account. Finally, the Withdrawal sends a
dispenseCash message to the CashDispenser and a displayMessage message to the
Screen, telling the user to remove the cash from the machine.

We have identified the collaborations among objects in the ATM system and modeled
some of these collaborations using UML interaction diagrams—both communication dia-
grams and sequence diagrams. In the next Software Engineering Case Study section
(Section 8.19), we enhance the structure of our model to complete a preliminary object-
oriented design, then we begin implementing the ATM system in Java.

Software Engineering Case Study Self-Review Exercises

7.1 A(n) ________ consists of an object of one class sending a message to an object of another
class.
   a) association
   b) aggregation
   c) collaboration
   d) composition

7.2 Which form of interaction diagram emphasizes what collaborations occur? Which form em-
phasizes when collaborations occur?

7.3 Create a sequence diagram that models the interactions among objects in the ATM system
that occur when a Deposit executes successfully, and explain the sequence of messages modeled by
the diagram.

Answers to Software Engineering Case Study Self-Review Exercises

7.1 c.

7.2 Communication diagrams emphasize what collaborations occur. Sequence diagrams em-
phasize when collaborations occur.

7.3 Figure 7.29 presents a sequence diagram that models the interactions between objects in the
ATM system that occur when a Deposit executes successfully. Figure 7.29 indicates that a Deposit
first sends a displayMessage message to the Screen to ask the user to enter a deposit amount. Next
the Deposit sends a getInput message to the Keypad to receive input from the user. The Deposit then
instructs the user to enter a deposit envelope by sending a displayMessage message to the Screen.
The Deposit next sends an isEnvelopeReceived message to the DepositSlot to confirm that the de-
posit envelope has been received by the ATM. Finally, the Deposit increases the totalBalance at-
tribute (but not the availableBalance attribute) of the user’s Account by sending a credit message
to the BankDatabase. The BankDatabase responds by sending the same message to the user’s Account.
Chapter 7 Arrays

7.15 Wrap-Up

This chapter began our introduction to data structures, exploring the use of arrays to store data in and retrieve data from lists and tables of values. The chapter examples demonstrated how to declare an array, initialize an array and refer to individual elements of an array. The chapter introduced the enhanced for statement to iterate through arrays. We also illustrated how to pass arrays to methods and how to declare and manipulate multidimensional arrays. Finally, the chapter showed how to write methods that use variable-length argument lists and how to read arguments passed to a program from the command line.

We continue our coverage of data structures in Chapter 17, Data Structures, which introduces dynamic data structures, such as lists, queues, stacks and trees, that can grow and shrink as programs execute. Chapter 18, Generics, presents the topic of generics, which provide the means to create general models of methods and classes that can be declared once, but used with many different data types. Chapter 19, Collections, introduces the Java Collections Framework, which uses generics to allow programmers to specify the exact types of objects that a particular data structure will store. Chapter 19 also introduces Java’s predefined data structures, which programmers can use instead of building their own. Chapter 19 discusses many data structures classes, including Vector.

Fig. 7.29 | Sequence diagram that models a Deposit executing.

Java™ How to Program, Seventh Edition
Print Publication Date: 2006/12/27
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.
and ArrayList, which are array-like data structures that can grow and shrink in response to a program’s changing storage requirements. The Collections API also provides class Arrays, which contains utility methods for array manipulation. Chapter 19 uses several static methods of class Arrays to perform such manipulations as sorting and searching the data in an array. You will be able to use some of the Arrays methods discussed in Chapter 19 after reading the current chapter, but some of the Arrays methods require knowledge of concepts presented later in the book.

We have now introduced the basic concepts of classes, objects, control statements, methods and arrays. In Chapter 8, we take a deeper look at classes and objects.

### Summary

**Section 7.1, Introduction**

- Arrays are data structures consisting of related data items of the same type. Arrays are fixed-length entities—they remain the same length once they are created, although an array variable may be reassigned the reference of a new array of a different length.

**Section 7.2, Arrays**

- An array is a group of variables (called elements or components) containing values that all have the same type. Arrays are objects, so they are considered reference types. The elements of an array can be either primitive types or reference types (including arrays).
- To refer to a particular element in an array, we specify the name of the reference to the array and the index (subscript) of the element in the array.
- A program refers to any one of an array’s elements with an array-access expression that includes the name of the array followed by the index of the particular element in square brackets ([]).
- The first element in every array has index zero and is sometimes called the zeroth element.
- An index must be a nonnegative integer. A program can use an expression as an index.
- Every array object knows its own length and maintains this information in a length field. The expression array.length accesses array’s length field to determine the length of the array.

**Section 7.3, Declaring and Creating Arrays**

- To create an array object, the programmer specifies the type of the array elements and the number of elements as part of an array-creation expression that uses keyword new. The following array-creation expression creates an array of 100 int values:

```java
int b[] = new int[100];
```
- When an array is created, each element of the array receives a default value—zero for numeric primitive-type elements, false for boolean elements and null for references (any nonprimitive type).
- When an array is declared, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all the identifiers in the declaration are array variables, as in

```java
double[] array1, array2;
```
- A program can declare arrays of any type. Every element of a primitive-type array contains a variable of the array’s declared type. Similarly, in an array of a reference type, every element is a reference to an object of the array’s declared type.
Section 7.4, Examples Using Arrays

• A program can create an array and initialize its elements with an array initializer (i.e., an initializer list enclosed in braces).

• Constant variables (also called named constants or read-only variables) must be initialized before they are used and cannot be modified thereafter.

• When a Java program executes, the JVM checks array indices to ensure that they are valid (i.e., they must be greater than or equal to 0 and less than the length of the array). If a program uses an invalid index, Java generates a so-called exception to indicate that an error occurred in the program at execution time.

Section 7.6, Enhanced for Statement

• The enhanced for statement allows programmers to iterate through the elements of an array or a collection without using a counter. The syntax of an enhanced for statement is:

```
for (parameter : arrayName) statement
```

where parameter has two parts—a type and an identifier (e.g., int number), and arrayName is the array through which to iterate.

• The enhanced for statement cannot be used to modify elements in an array. If a program needs to modify elements, use the traditional counter-controlled for statement.

Section 7.7, Passing Arrays to Methods

• When an argument is passed by value, a copy of the argument’s value is made and passed to the called method. The called method works exclusively with the copy.

• When an argument is passed by reference, the called method can access the argument’s value in the caller directly and possibly modify it.

• Java does not allow programmers to choose between pass-by-value and pass-by-reference—all arguments are passed by value. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type int and double) and copies of references to objects. Although an object’s reference is passed by value, a method can still interact with the referenced object by calling its public methods using the copy of the object’s reference.

• To pass an object reference to a method, simply specify in the method call the name of the variable that refers to the object.

• When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the array or element’s reference. When an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element’s value.

• To pass an individual array element to a method, use the indexed name of the array as an argument in the method call.

Section 7.9, Multidimensional Arrays

• Multidimensional arrays with two dimensions are often used to represent tables of values consisting of information arranged in rows and columns.

• Arrays that require two indices to identify a particular element are called two-dimensional arrays. An array with m rows and n columns is called an m-by-n array. A two-dimensional array can be initialized with an array initializer of the form

```
arrayType arrayName[][] = { {row1 initializer}, {row2 initializer}, ... };
```
• Multidimensional arrays are maintained as arrays of separate one-dimensional arrays. As a result, the lengths of the rows in a two-dimensional array are not required to be the same.

• A multidimensional array with the same number of columns in every row can be created with an array-creation expression of the form

```java
arrayType arrayName = new arrayType[ numRows ][ numColumns ];
```

• An argument type followed by an ellipsis (... ) in a method’s parameter list indicates that the method receives a variable number of arguments of that particular type. The ellipsis can occur only once in a method’s parameter list, and it must be at the end of the parameter list.

Section 7.11, Variable-Length Argument Lists
• A variable-length argument list is treated as an array within the method body. The number of arguments in the array can be obtained using the array’s `length` field.

Section 7.12, Using Command-Line Arguments
• Passing arguments to `main` in a Java application from the command line is achieved by including a parameter of type `String[]` in the parameter list of `main`. By convention, `main`’s parameter is named `args`.

• Java passes the command-line arguments that appear after the class name in the `java` command to the application’s `main` method as `String`s in the array `args`. The number of arguments passed in from the command line is obtained by accessing the array’s `length` attribute.

### Terminology

- 0 flag (in a format specifier)
- 1 length field of an array
- `a[ i ]` length field of an array
- `a[ i ][ j ]` length field of an array
- array
- array-access expression
- array-creation expression
- array initializer
- bounds checking
- call-by-reference
- call-by-value
- column index
- column of a two-dimensional array
- command-line arguments
- component of an array
- constant variable
- data structure
- declare a constant variable
- declare an array
- element of an array
- ellipsis (... ) in a method’s parameter list
- enhanced `for` statement
- `final` keyword
- index
- index zero
- initialize an array
- initializer list
- `length` field of an array
- `m-by-n` array
- multidimensional array
- name of an array
- named constant
- nested array initializers
- off-by-one error
- one-dimensional array
- `parseInt` method of class `Integer`
- pass-by-reference
- pass-by-value
- passing arrays to methods
- position number
- read-only variable
- row index
- row of a two-dimensional array
- scalar
- scalar quantity
- square brackets, []
- `subscript`
- table of values
- tabular format
- traverse an array
- two-dimensional array
- value of an element
- variable-length argument list
Self-Review Exercises

7.1 Fill in the blank(s) in each of the following statements:
   a) Lists and tables of values can be stored in ________.
   b) An array is a group of ________ (called elements or components) containing values
   that all have the same ________.
   c) The ________ allows programmers to iterate through the elements in an array without
   using a counter.
   d) The number used to refer to a particular element of an array is called the element’s
   ________.
   e) An array that uses two indices is referred to as a(n) ________ array.
   f) Use the enhanced for statement ________ to walk through double array numbers.
   g) Command-line arguments are stored in ________.
   h) Use the expression ________ to receive the total number of arguments in a command
   line. Assume that command-line arguments are stored in String args[ ].
   i) Given the command java MyClass test, the first command-line argument is
   ________.
   j) A(n) ________ in the parameter list of a method indicates that the method can receive a variable number of arguments.

7.2 Determine whether each of the following is true or false. If false, explain why.
   a) An array can store many different types of values.
   b) An array index should normally be of type float.
   c) An individual array element that is passed to a method and modified in that method
   will contain the modified value when the called method completes execution.
   d) Command-line arguments are separated by commas.

7.3 Perform the following tasks for an array called fractions:
   a) Declare a constant ARRAY_SIZE that is initialized to 10.
   b) Declare an array with ARRAY_SIZE elements of type double, and initialize the elements
   to 0.
   c) Refer to array element 4.
   d) Assign the value 1.667 to array element 9.
   e) Assign the value 3.333 to array element 6.
   f) Sum all the elements of the array, using a for statement. Declare the integer variable x
   as a control variable for the loop.

7.4 Perform the following tasks for an array called table:
   a) Declare and create the array as an integer array that has three rows and three columns.
   Assume that the constant ARRAY_SIZE has been declared to be 3.
   b) How many elements does the array contain?
   c) Use a for statement to initialize each element of the array to the sum of its indices. Assume
   that the integer variables x and y are declared as control variables.

7.5 Find and correct the error in each of the following program segments:
   a) final int ARRAY_SIZE = 5;
      ARRAY_SIZE = 10;
   b) Assume int b[] = new int[ 10 ];
      for ( int i = 0; i <= b.length; i++ )
         b[ i ] = 1;
   c) Assume int a[][] = { { 1, 2 }, { 3, 4 } };
      a[ 1, 1 ] = 5;
Answers to Self-Review Exercises

7.1  a) arrays. b) variables, type. c) enhanced for statement. d) index (or subscript or position number). e) two-dimensional. f) for ( double d : numbers ). g) an array of Strings, called args by convention. h) args.length. i) test. j) ellipsis (...).

7.2  a) False. An array can store only values of the same type.
     b) False. An array index must be an integer or an integer expression.
     c) For individual primitive-type elements of an array: False. A called method receives and manipulates a copy of the value of such an element, so modifications do not affect the original value. If the reference of an array is passed to a method, however, modifications to the array elements made in the called method are indeed reflected in the original. For individual elements of a nonprimitive type: True. A called method receives a copy of the reference of such an element, and changes to the referenced object will be reflected in the original array element.
     d) False. Command-line arguments are separated by white space.

7.3  a) final int ARRAY_SIZE = 10;
     b) double fractions[] = new double[ ARRAY_SIZE ];
     c) fractions[ 4 ]
     d) fractions[ 9 ] = 1.667;
     e) fractions[ 6 ] = 3.333;
     f) double total = 0.0;
         for ( int x = 0; x < fractions.length; x++ )
             total += fractions[ x ];

7.4  a) int table[][] = new int[ ARRAY_SIZE ][ ARRAY_SIZE ];
     b) Nine.
     c) for ( int x = 0; x < table.length; x++ )
         for ( int y = 0; y < table[ x ].length; y++ )
             table[ x ][ y ] = x + y;

7.5  a) Error: Assigning a value to a constant after it has been initialized.
     Correction: Assign the correct value to the constant in a final int ARRAY_SIZE declaration or declare another variable.
     b) Error: Referencing an array element outside the bounds of the array (b[10]).
     Correction: Change the <= operator to <.
     c) Error: Array indexing is performed incorrectly.
     Correction: Change the statement to a[ 1 ][ 1 ] = 5;

Exercises

7.6  Fill in the blanks in each of the following statements:

a) One-dimensional array p contains four elements. The names of those elements are ___________ and ___________.

b) Naming an array, stating its type and specifying the number of dimensions in the array is called __________ the array.

c) In a two-dimensional array, the first index identifies the __________ of an element and the second index identifies the __________ of an element.

d) An m-by-n array contains __________ rows, __________ columns and __________ elements.

e) The name of the element in row 3 and column 5 of array d is __________.
7.7 Determine whether each of the following is true or false. If false, explain why.
a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.

7.8 Write Java statements to accomplish each of the following tasks:
a) Display the value of element 6 of array f.
b) Initialize each of the five elements of one-dimensional integer array g to 8.
c) Total the 100 elements of floating-point array c.
d) Copy 11-element array a into the first portion of array b, which contains 34 elements.
e) Determine and display the smallest and largest values contained in 99-element floating-point array w.

7.9 Consider a two-by-three integer array t.
a) Write a statement that declares and creates t.
b) How many rows does t have?
c) How many columns does t have?
d) How many elements does t have?
e) Write access expressions for all the elements in row 1 of t.
f) Write access expressions for all the elements in column 2 of t.
g) Write a single statement that sets the element of t in row 0 and column 1 to zero.
h) Write a series of statements that initializes each element of t to zero. Do not use a repetition statement.
i) Write a nested for statement that initializes each element of t to zero.
j) Write a nested for statement that inputs the values for the elements of t from the user.
k) Write a series of statements that determines and displays the smallest value in t.
l) Write a printf statement that displays the elements of the first row of t. Do not use repetition.
m) Write a statement that totals the elements of the third column of t. Do not use repetition.

7.10 (Sales Commissions) Use a one-dimensional array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9% of $5000, or a total of $650. Write an application (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson’s salary is truncated to an integer amount):
a) $200–299
b) $300–399
c) $400–499
d) $500–599
e) $600–699
f) $700–799
Exercises 353

g) $800–899
h) $900–999
i) $1000 and over

Summarize the results in tabular format.

7.11 Write statements that perform the following one-dimensional-array operations:
   a) Set the 10 elements of integer array `counts` to zero.
   b) Add one to each of the 15 elements of integer array `bonus`.
   c) Display the five values of integer array `bestScores` in column format.

7.12 (Duplicate Elimination) Use a one-dimensional array to solve the following problem: Write an application that inputs five numbers, each between 10 and 100, inclusive. As each number is read, display it only if it is not a duplicate of a number already read. Provide for the “worst case,” in which all five numbers are different. Use the smallest possible array to solve this problem. Display the complete set of unique values input after the user enters each new value.

7.13 Label the elements of three-by-five two-dimensional array `sales` to indicate the order in which they are set to zero by the following program segment:
   ```java
   for ( int row = 0; row < sales.length; row++ )
   {
     for ( int col = 0; col < sales[ row ].length; col++ )
     {
       sales[ row ][ col ] = 0;
     }
   }
   ```

7.14 Write an application that calculates the product of a series of integers that are passed to method `product` using a variable-length argument list. Test your method with several calls, each with a different number of arguments.

7.15 Rewrite Fig. 7.2 so that the size of the array is specified by the first command-line argument. If no command-line argument is supplied, use 10 as the default size of the array.

7.16 Write an application that uses an enhanced for statement to sum the double values passed by the command-line arguments. [Hint: Use the static method `parseDouble` of class `Double` to convert a `String` to a `double` value.]

7.17 (Dice Rolling) Write an application to simulate the rolling of two dice. The application should use an object of class `Random` once to roll the first die and again to roll the second die. The sum of the two values should then be calculated. Each die can show an integer value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent. Figure 7.30 shows the 36 possible combinations of the two dice. Your application should roll the dice 36,000 times. Use a one-dimensional array to tally the number of times each possible sum appears. Display the results in tabular format. Determine whether the totals are reasonable (e.g., there are six ways to roll a 7, so approximately one-sixth of the rolls should be 7).

7.18 (Game of Craps) Write an application that runs 1000 games of craps (Fig. 6.9) and answers the following questions:
   a) How many games are won on the first roll, second roll, …, twentieth roll and after the twentieth roll?
   b) How many games are lost on the first roll, second roll, …, twentieth roll and after the twentieth roll?
   c) What are the chances of winning at craps? [Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?]
   d) What is the average length of a game of craps?
   e) Do the chances of winning improve with the length of the game?
Chapter 7 Arrays

7.19 (Airline Reservations System) A small airline has just purchased a computer for its new automated reservations system. You have been asked to develop the new system. You are to write an application to assign seats on each flight of the airline’s only plane (capacity: 10 seats).

Your application should display the following alternatives:

Please type 1 for First Class and Please type 2 for Economy. If the user types 1, your application should assign a seat in the first-class section (seats 1–5). If the user types 2, your application should assign a seat in the economy section (seats 6–10). Your application should then display a boarding pass indicating the person’s seat number and whether it is in the first-class or economy section of the plane.

Use a one-dimensional array of primitive type boolean to represent the seating chart of the plane. Initialize all the elements of the array to false to indicate that all the seats are empty. As each seat is assigned, set the corresponding elements of the array to true to indicate that the seat is no longer available.

Your application should never assign a seat that has already been assigned. When the economy section is full, your application should ask the person if it is acceptable to be placed in the first-class section (and vice versa). If yes, make the appropriate seat assignment. If no, display the message “Next flight leaves in 3 hours.”

7.20 (Total Sales) Use a two-dimensional array to solve the following problem: A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each type of product sold. Each slip contains the following:

a) The salesperson number
b) The product number
c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all the slips for last month is available. Write an application that will read all this information for last month’s sales and summarize the total sales by salesperson and by product. All totals should be stored in the two-dimensional array sales. After processing all the information for last month, display the results in tabular format, with each column representing a particular salesperson and each row representing a particular product. Cross-total each row to get the total sales of each product for last month. Cross-total each column to get the total sales by salesperson for last month. Your tabular output should include these cross-totals to the right of the totaled rows and to the bottom of the totaled columns.

7.21 (Turtle Graphics) The Logo language made the concept of turtle graphics famous. Imagine a mechanical turtle that walks around the room under the control of a Java application. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves, and while the pen is up, the turtle moves about freely without writing anything. In this problem, you will simulate the operation of the turtle and create a computerized sketchpad.

![Fig. 7.30](image-url) | The 36 possible sums of two dice.
Use a 20-by-20 array `floor` that is initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor with its pen up. The set of turtle commands your application must process are shown in Fig. 7.31.

Suppose that the turtle is somewhere near the center of the floor. The following “program” would draw and display a 12-by-12 square, leaving the pen in the up position:

2 5,12 3 5,12 3 5,12 3 5,12 1 6 9

As the turtle moves with the pen down, set the appropriate elements of array `floor` to 1s. When the 6 command (display the array) is given, wherever there is a 1 in the array, display an asterisk or any character you choose. Wherever there is a 0, display a blank.

Write an application to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

7.22 (Knight’s Tour) One of the more interesting puzzlers for chess buffs is the Knight’s Tour problem, originally proposed by the mathematician Euler. Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). Thus, as shown in Fig. 7.32, from a square near the middle of an empty chessboard, the knight (labeled K) can make eight different moves (numbered 0 through 7).

(a) Draw an eight-by-eight chessboard on a sheet of paper, and attempt a Knight’s Tour by hand. Put a 1 in the starting square, a 2 in the second square, a 3 in the third, and so on. Before starting the tour, estimate how far you think you will get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pen up</td>
</tr>
<tr>
<td>2</td>
<td>Pen down</td>
</tr>
<tr>
<td>3</td>
<td>Turn right</td>
</tr>
<tr>
<td>4</td>
<td>Turn left</td>
</tr>
<tr>
<td>5,10</td>
<td>Move forward 10 spaces (replace 10 for a different number of spaces)</td>
</tr>
<tr>
<td>6</td>
<td>Display the 20-by-20 array</td>
</tr>
<tr>
<td>9</td>
<td>End of data (sentinel)</td>
</tr>
</tbody>
</table>

**Fig. 7.31** Turtle graphics commands.
b) Now let us develop an application that will move the knight around a chessboard. The board is represented by an eight-by-eight two-dimensional array board. Each square is initialized to zero. We describe each of the eight possible moves in terms of its horizontal and vertical components. For example, a move of type 0, as shown in Fig. 7.32, consists of moving two squares horizontally to the right and one square vertically upward. A move of type 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, horizontal and vertical, as follows:

```
horizontal[0] = 2        vertical[0] = -1
```

Let the variables currentRow and currentColumn indicate the row and column, respectively, of the knight’s current position. To make a move of type moveNumber, where moveNumber is between 0 and 7, your application should use the statements

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Write an application to move the knight around the chessboard. Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Test each potential move to see if the knight has already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Run the application. How many moves did the knight make?

c) After attempting to write and run a Knight’s Tour application, you have probably developed some valuable insights. We will use these insights to develop a heuristic (or “rule of thumb”) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome or inaccessible squares are the four corners.
Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so that when the board gets congested near the end of the tour, there will be a greater chance of success.

We could develop an “accessibility heuristic” by classifying each of the squares according to how accessible it is and always moving the knight (using the knight’s L-shaped moves) to the most inaccessible square. We label a two-dimensional array accessibility with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each of the 16 squares nearest the center is rated as 8, each corner square is rated as 2, and the other squares have accessibility numbers of 3, 4 or 6 as follows:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Write a new version of the Knight’s Tour, using the accessibility heuristic. The knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your application should reduce the accessibility numbers as more squares become occupied. In this way, at any given time during the tour, each available square’s accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your application. Did you get a full tour? Modify the application to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

d) Write a version of the Knight’s Tour application that, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the “tied” squares. Your application should move to the tied square for which the next move would arrive at a square with the lowest accessibility number.

7.23 (Knight’s Tour: Brute-Force Approaches) In part (c) of Exercise 7.22, we developed a solution to the Knight’s Tour problem. The approach used, called the “accessibility heuristic,” generates many solutions and executes efficiently.

As computers continue to increase in power, we will be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let us call this approach “brute-force” problem solving.

a) Use random-number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves) at random. Your application should run one tour and display the final chessboard. How far did the knight get?

b) Most likely, the application in part (a) produced a relatively short tour. Now modify your application to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your application finishes attempting the 1000 tours, it should display this information in neat tabular format. What was the best result?

c) Most likely, the application in part (b) gave you some “respectable” tours, but no full tours. Now let your application run until it produces a full tour. [Caution: This version of the application could run for hours on a powerful computer.] Once again, keep a ta-
ble of the number of tours of each length, and display this table when the first full tour is found. How many tours did your application attempt before producing a full tour? How much time did it take?

d) Compare the brute-force version of the Knight’s Tour with the accessibility-heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute-force approach? Argue the pros and cons of brute-force problem solving in general.

7.24 (Eight Queens) Another puzzler for chess buffs is the Eight Queens problem, which asks the following: Is it possible to place eight queens on an empty chessboard so that no queen is “attacking” any other (i.e., no two queens are in the same row, in the same column or along the same diagonal)? Use the thinking developed in Exercise 7.22 to formulate a heuristic for solving the Eight Queens problem. Run your application. [Hint: It is possible to assign a value to each square of the chessboard to indicate how many squares of an empty chessboard are “eliminated” if a queen is placed in that square. Each of the corners would be assigned the value 22, as demonstrated by Fig. 7.33. Once these “elimination numbers” are placed in all 64 squares, an appropriate heuristic might be as follows: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?]

7.25 (Eight Queens: Brute-Force Approaches) In this exercise, you will develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 7.24.

a) Use the random brute-force technique developed in Exercise 7.23 to solve the Eight Queens problem.

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard) to solve the Eight Queens problem.

c) Why might the exhaustive brute-force approach not be appropriate for solving the Knight’s Tour problem?

d) Compare and contrast the random brute-force and exhaustive brute-force approaches.

7.26 (Knight’s Tour: Closed-Tour Test) In the Knight’s Tour (Exercise 7.22), a full tour occurs when the knight makes 64 moves, touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the square in which the knight started the tour. Modify the application you wrote in Exercise 7.22 to test for a closed tour if a full tour has occurred.

![Fig. 7.33](https://example.com/fig733.png)  | The 22 squares eliminated by placing a queen in the upper left corner.
7.27 (Sieve of Eratosthenes) A prime number is any integer greater than 1 that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) Create a primitive type boolean array with all elements initialized to true. Array elements with prime indices will remain true. All other array elements will eventually be set to false.

b) Starting with array index 2, determine whether a given element is true. If so, loop through the remainder of the array and set to false every element whose index is a multiple of the index for the element with value true. Then continue the process with the next element with value true. For array index 2, all elements beyond element 2 in the array that have indices which are multiples of 2 (indices 4, 6, 8, 10, etc.) will be set to false; for array index 3, all elements beyond element 3 in the array that have indices which are multiples of 3 (indices 6, 9, 12, 15, etc.) will be set to false; and so on.

When this process completes, the array elements that are still true indicate that the index is a prime number. These indices can be displayed. Write an application that uses an array of 1000 elements to determine and display the prime numbers between 2 and 999. Ignore array elements 0 and 1.

7.28 (Simulation: The Tortoise and the Hare) In this problem, you will re-create the classic race of the tortoise and the hare. You will use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick of the clock, your application should adjust the position of the animals according to the rules in Fig. 7.34. Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (the “starting gate”). If an animal slips left before square 1, move it back to square 1.

Generate the percentages in Fig. 7.34 by producing a random integer \( i \) in the range \( 1 \leq i \leq 10 \). For the tortoise, perform a “fast plod” when \( 1 \leq i \leq 5 \), a “slip” when \( 6 \leq i \leq 7 \) or a “slow plod” when \( 8 \leq i \leq 10 \). Use a similar technique to move the hare.

<table>
<thead>
<tr>
<th>Animal</th>
<th>Move type</th>
<th>Percentage of the time</th>
<th>Actual move</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tortoise</td>
<td>Fast plod</td>
<td>50%</td>
<td>3 squares to the right</td>
</tr>
<tr>
<td></td>
<td>Slip</td>
<td>20%</td>
<td>6 squares to the left</td>
</tr>
<tr>
<td></td>
<td>Slow plod</td>
<td>30%</td>
<td>1 square to the right</td>
</tr>
<tr>
<td>Hare</td>
<td>Sleep</td>
<td>20%</td>
<td>No move at all</td>
</tr>
<tr>
<td></td>
<td>Big hop</td>
<td>20%</td>
<td>9 squares to the right</td>
</tr>
<tr>
<td></td>
<td>Big slip</td>
<td>10%</td>
<td>12 squares to the left</td>
</tr>
<tr>
<td></td>
<td>Small hop</td>
<td>30%</td>
<td>1 square to the right</td>
</tr>
<tr>
<td></td>
<td>Small slip</td>
<td>20%</td>
<td>2 squares to the left</td>
</tr>
</tbody>
</table>

Fig. 7.34 | Rules for adjusting the positions of the tortoise and the hare.
Begin the race by displaying

BANG !!!!!
AND THEY'RE OFF !!!!!

Then, for each tick of the clock (i.e., each repetition of a loop), display a 70-position line showing the letter T in the position of the tortoise and the letter H in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your application should display OUCH!!!! beginning at that position. All output positions other than the T, the H or the OUCH!!!! (in case of a tie) should be blank.

After each line is displayed, test for whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation. If the tortoise wins, display TORTOISE WINS!!!! YAY!!!! If the hare wins, display Hare wins. Yuch. If both animals win on the same tick of the clock, you may want to favor the tortoise (the "underdog"), or you may want to display It's a tie. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you are ready to run your application, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

Later in the book, we introduce a number of Java capabilities, such as graphics, images, animation, sound and multithreading. As you study those features, you might enjoy enhancing your tortoise-and-hare contest simulation.

7.29 (Fibonacci Series) The Fibonacci series

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \]

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms.

a) Write a method \texttt{fibonacci(n)} that calculates the \(n\)th Fibonacci number. Incorporate this method into an application that enables the user to enter the value of \(n\).

b) Determine the largest Fibonacci number that can be displayed on your system.

c) Modify the application you wrote in part (a) to use \texttt{double} instead of \texttt{int} to calculate and return Fibonacci numbers, and use this modified application to repeat part (b).

\textit{Exercises 7.30—7.33 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.}

7.30 (Card Shuffling and Dealing) Modify the application of Fig. 7.11 to deal a five-card poker hand. Then modify class \texttt{DeckOfCards} of Fig. 7.10 to include methods that determine whether a hand contains

a) a pair
b) two pairs
c) three of a kind (e.g., three jacks)
d) four of a kind (e.g., four aces)
e) a flush (i.e., all five cards of the same suit)
f) a straight (i.e., five cards of consecutive face values)
g) a full house (i.e., two cards of one face value and three cards of another face value)

[Hint: Add methods \texttt{getFace} and \texttt{getSuit} to class \texttt{Card} of Fig. 7.9.]

7.31 (Card Shuffling and Dealing) Use the methods developed in Exercise 7.30 to write an application that deals two five-card poker hands, evaluates each hand and determines which is better.

7.32 (Card Shuffling and Dealing) Modify the application developed in Exercise 7.31 so that it can simulate the dealer. The dealer’s five-card hand is dealt “face down,” so the player cannot see it. The application should then evaluate the dealer’s hand, and, based on the quality of the hand, the
Special Section: Building Your Own Computer

In the next several problems, we take a temporary diversion from the world of high-level language programming to "peel open" a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (through the technique of software-based simulation) on which you can execute your machine-language programs.

7.34 (Machine-Language Programming) Let us create a computer called the Simpletron. As its name implies, it is a simple, but powerful, machine. The Simpletron runs programs written in the only language it directly understands: Simpletron Machine Language (SML).

The Simpletron contains an accumulator—a special register in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All the information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number, such as +3364, -1293, +0007 and -0001. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00, 01, …, 99.

Before running an SML program, we must load, or place, the program into memory. The first instruction (or statement) of every SML program is always placed in location 00. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron's memory (and hence instructions are signed four-digit decimal numbers). We shall assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the operation code specifying the operation to be performed. SML operation codes are summarized in Fig. 7.35.

<table>
<thead>
<tr>
<th>Operation code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{final int READ = 10;}</td>
<td>Read a word from the keyboard into a specific location in memory.</td>
</tr>
<tr>
<td>\texttt{final int WRITE = 11;}</td>
<td>Write a word from a specific location in memory to the screen.</td>
</tr>
</tbody>
</table>

Fig. 7.35 | Simpletron Machine Language (SML) operation codes. (Part 1 of 2.)
The last two digits of an SML instruction are the operand—the address of the memory location containing the word to which the operation applies. Let's consider several simple SML programs.

The first SML program (Fig. 7.36) reads two numbers from the keyboard and computes and displays their sum. The instruction `+1007` reads the first number from the keyboard and places it into location `07` (which has been initialized to 0). Then instruction `+1008` reads the next number into location `08`. The `load` instruction, `+2007`, puts the first number into the accumulator, and the `add` instruction, `+3008`, adds the second number to the number in the accumulator. All SML arithmetic instructions leave their results in the accumulator. The `store` instruction, `+2109`, places the result back into memory location `09`, from which the `write` instruction, `+1109`, takes the number and displays it (as a signed four-digit decimal number). The `halt` instruction, `+4300`, terminates execution.

### Table 7.35

<table>
<thead>
<tr>
<th>Operation code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load/store operations:</strong></td>
<td></td>
</tr>
<tr>
<td>final int LOAD = 20;</td>
<td>Load a word from a specific location in memory into the accumulator.</td>
</tr>
<tr>
<td>final int STORE = 21;</td>
<td>Store a word from the accumulator into a specific location in memory.</td>
</tr>
<tr>
<td><strong>Arithmetic operations:</strong></td>
<td></td>
</tr>
<tr>
<td>final int ADD = 30;</td>
<td>Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).</td>
</tr>
<tr>
<td>final int SUBTRACT = 31;</td>
<td>Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).</td>
</tr>
<tr>
<td>final int DIVIDE = 32;</td>
<td>Divide a word from a specific location in memory into the word in the accumulator (leave result in the accumulator).</td>
</tr>
<tr>
<td>final int MULTIPLY = 33;</td>
<td>Multiply a word from a specific location in memory by the word in the accumulator (leave the result in the accumulator).</td>
</tr>
<tr>
<td><strong>Transfer-of-control operations:</strong></td>
<td></td>
</tr>
<tr>
<td>final int BRANCH = 40;</td>
<td>Branch to a specific location in memory.</td>
</tr>
<tr>
<td>final int BRANCHNEG = 41;</td>
<td>Branch to a specific location in memory if the accumulator is negative.</td>
</tr>
<tr>
<td>final int BRANCHZERO = 42;</td>
<td>Branch to a specific location in memory if the accumulator is zero.</td>
</tr>
<tr>
<td>final int HALT = 43;</td>
<td>Halt. The program has completed its task.</td>
</tr>
</tbody>
</table>
The second SML program (Fig. 7.37) reads two numbers from the keyboard and determines and displays the larger value. Note the use of the instruction +4107 as a conditional transfer of control, much the same as Java’s if statement.

<table>
<thead>
<tr>
<th>Location</th>
<th>Number</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+1007</td>
<td>(Read A)</td>
</tr>
<tr>
<td>01</td>
<td>+1008</td>
<td>(Read B)</td>
</tr>
<tr>
<td>02</td>
<td>+2007</td>
<td>(Load A)</td>
</tr>
<tr>
<td>03</td>
<td>+3008</td>
<td>(Add B)</td>
</tr>
<tr>
<td>04</td>
<td>+2109</td>
<td>(Store C)</td>
</tr>
<tr>
<td>05</td>
<td>+1109</td>
<td>(Write C)</td>
</tr>
<tr>
<td>06</td>
<td>+4300</td>
<td>(Halt)</td>
</tr>
<tr>
<td>07</td>
<td>+0000</td>
<td>(Variable A)</td>
</tr>
<tr>
<td>08</td>
<td>+0000</td>
<td>(Variable B)</td>
</tr>
<tr>
<td>09</td>
<td>+0000</td>
<td>(Result C)</td>
</tr>
</tbody>
</table>

**Fig. 7.36** | SML program that reads two integers and computes their sum.

The second SML program (Fig. 7.37) reads two numbers from the keyboard and determines and displays the larger value. Note the use of the instruction +4107 as a conditional transfer of control, much the same as Java’s if statement.

<table>
<thead>
<tr>
<th>Location</th>
<th>Number</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+1009</td>
<td>(Read A)</td>
</tr>
<tr>
<td>01</td>
<td>+1010</td>
<td>(Read B)</td>
</tr>
<tr>
<td>02</td>
<td>+2009</td>
<td>(Load A)</td>
</tr>
<tr>
<td>03</td>
<td>+3110</td>
<td>(Subtract B)</td>
</tr>
<tr>
<td>04</td>
<td>+4107</td>
<td>(Branch negative to 07)</td>
</tr>
<tr>
<td>05</td>
<td>+1109</td>
<td>(Write A)</td>
</tr>
<tr>
<td>06</td>
<td>+4300</td>
<td>(Halt)</td>
</tr>
<tr>
<td>07</td>
<td>+1110</td>
<td>(Write B)</td>
</tr>
<tr>
<td>08</td>
<td>+4300</td>
<td>(Halt)</td>
</tr>
<tr>
<td>09</td>
<td>+0000</td>
<td>(Variable A)</td>
</tr>
<tr>
<td>10</td>
<td>+0000</td>
<td>(Variable B)</td>
</tr>
</tbody>
</table>

**Fig. 7.37** | SML program that reads two integers and determines the larger.
Chapter 7 Arrays

Now write SML programs to accomplish each of the following tasks:

a) Use a sentinel-controlled loop to read 10 positive numbers. Compute and display their sum.
b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and display their average.
c) Read a series of numbers, and determine and display the largest number. The first number read indicates how many numbers should be processed.

7.35 (Computer Simulator) In this problem, you are going to build your own computer. No, you will not be soldering components together. Rather, you will use the powerful technique of software-based simulation to create an object-oriented software model of the Simpletron of Exercise 7.34. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you will actually be able to run, test and debug the SML programs you wrote in Exercise 7.34.

When you run your Simpletron simulator, it should begin by displaying:

*** Welcome to Simpletron! ***
*** Please enter your program one instruction (or data word) at a time into the input text field. I will display the location number and a question mark (?). You then type the word for that location. Press the Done button to stop entering your program. ***

Your application should simulate the memory of the Simpletron with a one-dimensional array memory that has 100 elements. Now assume that the simulator is running, and let us examine the dialog as we enter the program of Fig. 7.37 (Exercise 7.34):

00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

Your program should display the memory location followed by a question mark. Each value to the right of a question mark is input by the user. When the sentinel value -99999 is input, the program should display the following:

*** Program loading completed ***
*** Program execution begins ***

The SML program has now been placed (or loaded) in array memory. Now the Simpletron executes the SML program. Execution begins with the instruction in location 00 and, as in Java, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable accumulator to represent the accumulator register. Use the variable instructionCounter to keep track of the location in memory that contains the instruction being performed. Use the variable operationCode to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use the variable operand to indicate the memory location on which the current instruction operates. Thus, operand is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather,
transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in `operationCode`, and "pick off" the right two digits and place them in `operand`. When the Simpletron begins execution, the special registers are all initialized to zero.

Now, let us "walk through" execution of the first SML instruction, +1009 in memory location 00. This procedure is called an instruction execution cycle. The `instructionCounter` tells us the location of the next instruction to be performed. We fetch the contents of that location from memory by using the Java statement

```java
instructionRegister = memory[instructionCounter];
```

The operation code and the operand are extracted from the instruction register by the statements

```java
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a `read` (versus a `write`, a `load`, and so on). A `switch` differentiates among the 12 operations of SML. In the `switch` statement, the behavior of various SML instructions is simulated as shown in Fig. 7.38. We discuss branch instructions shortly and leave the others to you.

When the SML program completes execution, the name and contents of each register as well as the complete contents of memory should be displayed. Such a printout is often called a computer dump (no, a computer dump is not a place where old computers go). To help you program your dump method, a sample dump format is shown in Fig. 7.39. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated.

Let us proceed with the execution of our program's first instruction—namely, the +1009 in location 00. As we have indicated, the `switch` statement simulates this task by prompting the user to enter a value, reading the value and storing it in memory location `memory[operand]`. The value is then read into location 09.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Since the instruction just performed was not a transfer of control, we need merely increment the instruction-counter register as follows:

```java
instructionCounter++;
```

This action completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to execute.

### Instruction | Description
--- | ---
read: | Display the prompt "Enter an integer", then input the integer and store it in location `memory[operand]`.
load: | `accumulator = memory[operand];`
add: | `accumulator += memory[operand];`
halt: | This instruction displays the message *** Simpletron execution terminated ***

**Fig. 7.38** | Behavior of several SML instructions in the Simpletron.
Now let us consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the `switch` as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if ( accumulator == 0 )
    instructionCounter = operand;
```

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 7.34. If you desire, you may embellish SML with additional features and provide for these features in your simulator.

Your simulator should check for various types of errors. During the program-loading phase, for example, each number the user types into the Simpletron's `memory` must be in the range -9999 to +9999. Your simulator should test that each number entered is in this range and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, and accumulator overflows (i.e., arithmetic operations resulting in values larger than +9999 or smaller than -9999). Such serious errors are called `fatal errors`. When a fatal error is detected, your simulator should display an error message, such as

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should display a full computer dump in the format we discussed previously. This treatment will help the user locate the error in the program.

**7.36 (Simpletron Simulator Modifications)** In Exercise 7.35, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise,
we propose several modifications and enhancements to the Simpletron Simulator. In Exercises 17.26–17.27, we propose building a compiler that converts programs written in a high-level programming language (a variation of Basic) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler:

a) Extend the Simpletron Simulator's memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.

b) Allow the simulator to perform remainder calculations. This modification requires an additional SML instruction.

c) Allow the simulator to perform exponentiation calculations. This modification requires an additional SML instruction.

d) Modify the simulator to use hexadecimal values rather than integer values to represent SML instructions.

e) Modify the simulator to allow output of a newline. This modification requires an additional SML instruction.

f) Modify the simulator to process floating-point values in addition to integer values.

g) Modify the simulator to handle string input. [Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII (see Appendix B) decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string, beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half-word.]

h) Modify the simulator to handle output of strings stored in the format of part (g). [Hint: Add a machine-language instruction that will display a string, beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and displays the string by translating each two-digit number into its equivalent character.]