

7

Arrays

Objectives

- To introduce the array data structure.
- To understand the use of arrays to store, sort and search lists and tables of values.
- To understand how to declare an array, initialize an array and refer to individual elements of an array.
- To be able to pass arrays to methods.
- To understand basic sorting techniques.
- To be able to declare and manipulate multiple-subscript arrays.

With sobs and tears he sorted out

Those of the largest size ...

Lewis Carroll

Attempt the end, and never stand to doubt;

Nothing's so hard, but search will find it out.

Robert Herrick

Now go, write it before them in a table,

and note it in a book.

Isaiah 30:8

'Tis in my memory lock'd,

And you yourself shall keep the key of it.

William Shakespeare



Outline

- 7.1 Introduction**
- 7.2 Arrays**
- 7.3 Declaring and Allocating Arrays**
- 7.4 Examples Using Arrays**
 - 7.4.1 Allocating an Array and Initializing Its Elements**
 - 7.4.2 Using an Initializer List to Initialize Elements of an Array**
 - 7.4.3 Calculating the Value to Store in Each Array Element**
 - 7.4.4 Summing the Elements of an Array**
 - 7.4.5 Using Histograms to Display Array Data Graphically**
 - 7.4.6 Using the Elements of an Array as Counters**
 - 7.4.7 Using Arrays to Analyze Survey Results**
- 7.5 References and Reference Parameters**
- 7.6 Passing Arrays to Methods**
- 7.7 Sorting Arrays**
- 7.8 Searching Arrays: Linear Search and Binary Search**
 - 7.8.1 Searching an Array with Linear Search**
 - 7.8.2 Searching a Sorted Array with Binary Search**
- 7.9 Multiple-Subscripted Arrays**
- 7.10 (Optional Case Study) Thinking About Objects: Collaboration Among Objects**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises • Recursion Exercises • Special Section: Building Your own Computer

7.1 Introduction

This chapter serves as an introduction to the important topic of data structures. *Arrays* are data structures consisting of related data items of the same type. Arrays are “static” entities, in that they remain the same size once they are created, although an array reference may be reassigned to a new array of a different size. Chapter 19, “Data Structures,” introduces dynamic data structures, such as lists, queues, stacks and trees, that can grow and shrink as programs execute. Chapter 20, “Java Utilities Package and Bit Manipulation,” discusses class **Vector**, which is an array-like class whose objects can grow and shrink in response to a Java program’s changing storage requirements. Chapter 21, “The Collections API,” introduces Java’s predefined data structures that enable the programmer to use existing data structures for lists, queues, stacks and trees rather than “reinventing the wheel.” The Collections API also provides class **Arrays**, which defines a set of utility methods for array manipulation.

7.2 Arrays

An array is a group of contiguous memory locations that all have the same name and the same type. To refer to a particular location or element in the array, we specify the name of the array and the *position number* (or *index* or *subscript*) of the particular element in the array.

Figure 7.1 shows an integer array called **c**. This array contains 12 *elements*. A program refers to any one of these elements by giving the name of the array followed by the position number of the particular element in square brackets (**[]**). The first element in every array has *position number zero* (sometimes called the *zeroth element*). Thus, the first element of array **c** is **c[0]**, the second element of array **c** is **c[1]**, the seventh element of array **c** is **c[6]**, and, in general, the *i*th element of array **c** is **c[i - 1]**. Array names follow the same conventions as other variable names.

Formally, the position number in square brackets is called a subscript (or an index). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, the program evaluates the expression to determine the subscript. For example, if we assume that variable **a** is **5** and that variable **b** is **6**, then the statement

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

adds 2 to array element **c[11]**. Note that a subscripted array name is an *lvalue*—it can be used on the left side of an assignment to place a new value into an array element.

Name of array (Note that all elements of this array have the same name, c)	c[0]	-45
	c[1]	6
	c[2]	0
	c[3]	72
	c[4]	1543
	c[5]	-89
	c[6]	0
	c[7]	62
	c[8]	-3
	c[9]	1
	c[10]	6453
Position number (index or subscript) of the element within array c	c[11]	78

Fig. 7.1 A 12-element array.

Let us examine array **c** in Figure 7.1 more closely. The *name* of the array is **c**. Every array in Java *knows* its own length and maintains this information in a variable called **length**. The expression **c.length** accesses array **c**'s **length** variable to determine the length of the array. The 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

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

To divide the value of the seventh element of array **c** by **2** and assign the result to the variable **x**, we would write

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



Common Programming Error 7.1

It is important to note the difference between the “seventh element of the array” and “array element seven.” Because array subscripts begin at 0, the “seventh element of the array” has a subscript of 6, while “array element seven” has a subscript of 7 and is actually the eighth element of the array. This confusion is a source of “off-by-one” errors.

The brackets used to enclose the subscript of an array are one of Java's many operators. Brackets are in the highest level of precedence in Java. The chart in Fig. 7.2 shows the precedence and associativity of the operators introduced so far. They are shown top to bottom in decreasing order of precedence with their associativity and type. See Appendix C for the complete operator precedence chart.

Operators	Associativity	Type
() [] .	left to right	highest
++ --	right to left	unary postfix
++ -- + - ! (type)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
	left to right	boolean logical inclusive OR
&&	left to right	logical AND
	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 7.2 Precedence and associativity of the operators discussed so far.

7.3 Declaring and Allocating Arrays

Arrays are objects that occupy space in memory. All objects in Java (including arrays) must be allocated dynamically with operator **new**. For an array, the programmer specifies the type of the array elements and the number of elements as part of the **new** operation. To allocate 12 elements for integer array **c** in Fig. 7.1, use the declaration

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

The preceding declaration also can be performed in two steps as follows:

```
int c[];           // declares the array
c = new int[ 12 ]; // allocates the array
```

When allocating an array, each element of the array receives a default value—zero for the numeric primitive-data-type elements, **false** for **boolean** elements or **null** for references (any nonprimitive type).



Common Programming Error 7.2

*Unlike array declarations in several other programming languages (such as C and C++), Java array declarations must not specify the number of array elements in the square brackets after the array name; otherwise, a syntax error occurs. For example, the declaration **int c[12]**; causes a syntax error.*

A program can allocate memory for several arrays with a single declaration. The following declaration reserves 100 elements for **String** array **b** and 27 elements for **String** array **x**:

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

When declaring an array, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all identifiers in the declaration represent arrays, as in

```
double[] array1, array2;
```

which declares both **array1** and **array2** as arrays of **double** values. As shown previously, the declaration and initialization of the array can be combined in the declaration. The following declaration reserves 10 elements for **array1** and 20 elements for **array2**:

```
double[] array1 = new double[ 10 ],
        array2 = new double[ 20 ];
```

A program can declare arrays of any data type. It is important to remember that every element of a primitive data type array contains one value of the declared data type. For example, every element of an **int** array contains an **int** value. However, in an array of a nonprimitive type, every element of the array is a reference to an object of the array's declared data type. For example, every element of a **String** array is a reference to a **String**. In arrays that store references, the references have the value **null** by default.

7.4 Examples Using Arrays

This section presents several examples using arrays that demonstrate declaring arrays, allocating arrays, initializing arrays and manipulating array elements in various ways. For

simplicity, the examples in this section use arrays that contain elements of type `int`. Please remember that a program can declare an array of any data type.

7.4.1 Allocating an Array and Initializing Its Elements

The application of Figure 7.3 uses operator `new` to allocate dynamically an array of 10 `int` elements, which are initially zero (the default value in an array of type `int`). The program displays the array elements in tabular format in a `JTextArea`.

Line 12 declares `array`—a reference capable of referring to an array of `int` elements. Line 14 allocates the 10 elements of the array with `new` and initializes the reference. The program builds its output in the `String` called `output` that will be displayed in a `JTextArea` on a message dialog. Line 16 appends to `output` the headings for the columns displayed by the program. The columns represent the subscript for each array element and the value of each array element, respectively.

Lines 19–20 use a `for` structure to append the subscript number (represented by `counter`) and value of each array element to `output`. Note the use of zero-based counting (remember, subscripts start at 0), so that the loop can access every array element. Also, note the expression `array.length` in the `for` structure condition 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. For a 10-element array, the subscript values are 0 through 9, so using the less than operator `<` guarantees that the loop does not attempt to access an element beyond the end of the array.

```

1  // Fig. 7.3: InitArray.java
2  // Creating an array.
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class InitArray {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int array[];           // declare reference to an array
13
14         array = new int[ 10 ]; // dynamically allocate array
15
16         String output = "Subscript\tValue\n";
17
18         // append each array element's value to String output
19         for ( int counter = 0; counter < array.length; counter++ )
20             output += counter + "\t" + array[ counter ] + "\n";
21
22         JTextArea outputArea = new JTextArea();
23         outputArea.setText( output );
24     }

```

Fig. 7.3 Initializing the elements of an array to zeros (part 1 of 2).

```

25     JOptionPane.showMessageDialog( null, outputArea,
26         "Initializing an Array of int Values",
27         JOptionPane.INFORMATION_MESSAGE );
28
29     System.exit( 0 );
30 }
31 }

```

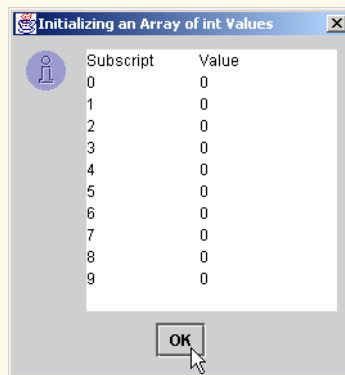


Fig. 7.3 Initializing the elements of an array to zeros (part 2 of 2).

7.4.2 Using an Initializer List to Initialize Elements of an Array

A program can allocate and initialize the elements of an array in the array declaration by following the declaration with an equal sign and a comma-separated *initializer list* enclosed in braces (`{` and `}`). In this case, the array size is determined by the number of elements in the initializer list. For example, the declaration

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

creates a five-element array with subscripts of **0**, **1**, **2**, **3** and **4**. Note that the preceding declaration does not require the **new** operator to create the array object. When the compiler encounters an array declaration that includes an initializer list, the compiler counts the number of initializers in the list and sets up a new operation to allocate the appropriate number of array elements.

The application of Fig. 7.4 initializes an integer array with 10 values (line 14) and displays the array in tabular format in a **JTextArea** on a message dialog. The code for displaying the array elements is identical to that of Fig. 7.3.

```

1  // Fig. 7.4: InitArray.java
2  // Initializing an array with a declaration.
3
4  // Java extension packages
5  import javax.swing.*;
6

```

Fig. 7.4 Initializing the elements of an array with a declaration (part 1 of 2).

```

7  public class InitArray {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         // initializer list specifies number of elements and
13         // value for each element
14         int array[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
15
16         String output = "Subscript\tValue\n";
17
18         // append each array element's value to String output
19         for ( int counter = 0; counter < array.length; counter++ )
20             output += counter + "\t" + array[ counter ] + "\n";
21
22         JTextArea outputArea = new JTextArea();
23         outputArea.setText( output );
24
25         JOptionPane.showMessageDialog( null, outputArea,
26             "Initializing an Array with a Declaration",
27             JOptionPane.INFORMATION_MESSAGE );
28
29         System.exit( 0 );
30     }
31 }

```

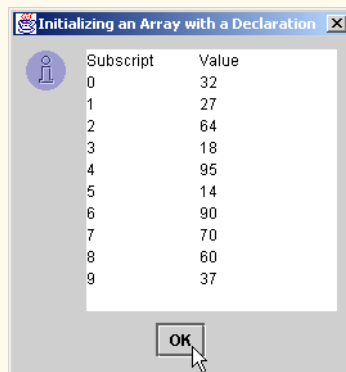


Fig. 7.4 Initializing the elements of an array with a declaration (part 2 of 2).

7.4.3 Calculating the Value to Store in Each Array Element

Some programs calculate the value stored in each array element. The application of Fig. 7.5 initializes the elements of a 10-element array to the even integers from 2 to 20 (2, 4, 6, ..., 20) and displays the array in tabular format. The **for** structure at lines 18–19 generates an array element's value by multiplying the current value of **counter** (the control variable of the loop) by 2 and adding 2.

```

1  // Fig. 7.5: InitArray.java
2  // Initialize array with the even integers from 2 to 20.
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class InitArray {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         final int ARRAY_SIZE = 10;
13         int array[];           // reference to int array
14
15         array = new int[ ARRAY_SIZE ]; // allocate array
16
17         // calculate value for each array element
18         for ( int counter = 0; counter < array.length; counter++ )
19             array[ counter ] = 2 + 2 * counter;
20
21         String output = "Subscript\tValue\n";
22
23         for ( int counter = 0; counter < array.length; counter++ )
24             output += counter + "\t" + array[ counter ] + "\n";
25
26         JTextArea outputArea = new JTextArea();
27         outputArea.setText( output );
28
29         JOptionPane.showMessageDialog( null, outputArea,
30             "Initializing to Even Numbers from 2 to 20",
31             JOptionPane.INFORMATION_MESSAGE );
32
33         System.exit( 0 );
34     }
35 }

```

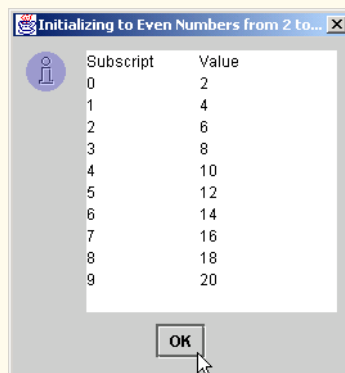


Fig. 7.5 Generating values to be placed into elements of an array.

Line 12 uses the **final** qualifier to declare constant variable **ARRAY_SIZE**, whose value is **10**. Remember that constant variables must be initialized before they are used and cannot be modified thereafter. If an attempt is made to modify a **final** variable after it is declared as shown on line 12, the compiler issues a message like

cannot assign a value to final variable *variableName*

If an attempt is made to use a **final** variable before it is initialized, the compiler issues the error message

Variable *variableName* **may not have been initialized**

Constant variables also are called *named constants* or *read-only variables*. Such variables often can make programs more readable. Note that the term “constant variable” is an oxymoron—a contradiction in terms—like “jumbo shrimp” or “freezer burn.”



Common Programming Error 7.3

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

7.4.4 Summing the Elements of an Array

Often, the elements of an array represent a series of values to use in a calculation. For example, if the elements of an array represent the grades for an exam in a class, the professor may wish to total the elements of an array, then calculate the class average for the exam.

The application of Fig. 7.6 sums the values contained in the 10-element integer array. The program declares, allocates and initializes the array at line 12. The **for** structure at lines 16–17 performs the calculations. [Note: It is important to remember that the values being supplied as initializers for **array** normally would be read into the program. For example, in an applet the user could enter the values through a **JTextField**, or in an application the values could be read from a file on disk (discussed in Chapter 16). Reading the data into a program makes the program more flexible, because it can be used with different sets of data.]

```

1  // Fig. 7.6: SumArray.java
2  // Total the values of the elements of an array.
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class SumArray {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13         int total = 0;
14     }

```

Fig. 7.6 Computing the sum of the elements of an array (part 1 of 2).

```

15      // add each element's value to total
16      for ( int counter = 0; counter < array.length; counter++ )
17          total += array[ counter ];
18
19      JOptionPane.showMessageDialog( null,
20          "Total of array elements: " + total,
21          "Sum the Elements of an Array",
22          JOptionPane.INFORMATION_MESSAGE );
23
24      System.exit( 0 );
25  }
26  }

```

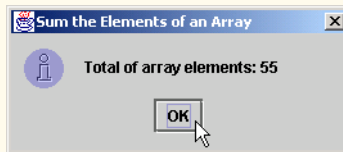


Fig. 7.6 Computing the sum of the elements of an array (part 2 of 2).

7.4.5 Using Histograms 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 larger numeric values. One simple way to display numeric data graphically is with a *histogram* that shows each numeric value as a bar of asterisks (*).

Our next application (Fig. 7.7) reads numbers from an array and graphs the information in the form of a bar chart (or histogram). The program displays each number followed by a bar consisting of that many asterisks. The nested **for** loop (lines 17–24) appends the bars to the **String** that will be displayed in **JTextArea outputArea** on a message dialog. Note the loop continuation condition of the inner **for** structure at line 22 (**stars <= array[counter]**). Each time the program reaches the inner **for** structure, the loop counts from **1** to **array[counter]**, thus using a value in **array** to determine the final value of the control variable **stars** and the number of asterisks to display.

```

1  // Fig. 7.7: Histogram.java
2  // Histogram printing program.
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class Histogram {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int array[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
13
14         String output = "Element\tValue\tHistogram";

```

Fig. 7.7 A program that prints histograms (part 1 of 2).

```

15
16     // for each array element, output a bar in histogram
17     for ( int counter = 0; counter < array.length; counter++ ) {
18         output +=
19             "\n" + counter + "\t" + array[ counter ] + "\t";
20
21         // print bar of asterisks
22         for ( int stars = 0; stars < array[ counter ]; stars++ )
23             output += " ";
24     }
25
26     JTextArea outputArea = new JTextArea();
27     outputArea.setText( output );
28
29     JOptionPane.showMessageDialog( null, outputArea,
30         "Histogram Printing Program",
31         JOptionPane.INFORMATION_MESSAGE );
32
33     System.exit( 0 );
34 }
35 }

```

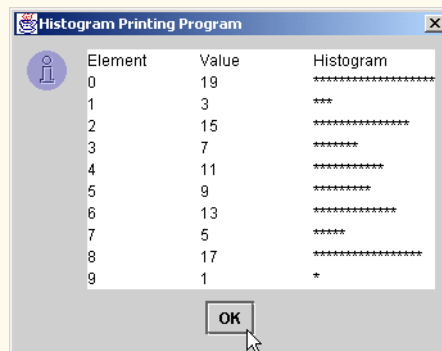


Fig. 7.7 A program that prints histograms (part 2 of 2).

7.4.6 Using the Elements of an Array as Counters

Sometimes programs use a series of counter variables to summarize data, such as the results of a survey. In Chapter 6, we used a series of counters in our die-rolling program to track the number of occurrences of each side on a six-sided die as the program rolled the die 6000 times. We also indicated that there is a more elegant method of writing the program of Fig. 6.8. An array version of this application is shown in Fig. 7.8.

```

1  // Fig. 7.8: RollDie.java
2  // Roll a six-sided die 6000 times
3
4  // Java extension packages
5  import javax.swing.*;
6

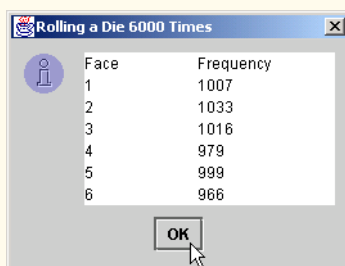
```

Fig. 7.8 Die-rolling program using arrays instead of **switch** (part 1 of 2).

```

7  public class RollDie {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int face, frequency[] = new int[ 7 ];
13
14         // roll die 6000 times
15         for ( int roll = 1; roll <= 6000; roll++ ) {
16             face = 1 + ( int ) ( Math.random() * 6 );
17
18             // use face value as subscript for frequency array
19             ++frequency[ face ];
20         }
21
22         String output = "Face\tFrequency";
23
24         // append frequencies to String output
25         for ( face = 1; face < frequency.length; face++ )
26             output += "\n" + face + "\t" + frequency[ face ];
27
28         JTextArea outputArea = new JTextArea();
29         outputArea.setText( output );
30
31         JOptionPane.showMessageDialog( null, outputArea,
32             "Rolling a Die 6000 Times",
33             JOptionPane.INFORMATION_MESSAGE );
34
35         System.exit( 0 );
36     }
37 }

```



Face	Frequency
1	1007
2	1033
3	1016
4	979
5	999
6	966

Fig. 7.8 Die-rolling program using arrays instead of **switch** (part 2 of 2).

The program uses the seven-element array **frequency** to count the occurrences of each side of the die. This program replaces lines 20–46 of Fig. 6.8 with line 19 of this program. Line 19 uses the random **face** value as the subscript for array **frequency** to determine which element the program should increment during each iteration of the loop. Since the random number calculation on line 19 produces numbers from 1 to 6 (the values for a six-sided die), the **frequency** array must be large enough to store six counters. However, in this program, we chose to use a seven-element array. We ignore the first array element, **frequency[0]**, because it is more logical to have the face value 1 increment

frequency[1] than **frequency[0]**. This allows us to use each face value directly as a subscript for the **frequency** array.



Good Programming Practice 7.1

Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.



Performance Tip 7.1

Sometimes performance considerations far outweigh clarity considerations.

Also, lines 25–26 of this program replace lines 52–55 from Fig. 6.8. We can loop through array **frequency**, so we do not have to enumerate each line of text to display in the **JTextArea** as we did in Fig. 6.8.

7.4.7 Using Arrays to Analyze Survey Results

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

Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (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. 7.9). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array **responses** is a 40-element integer array of the students' responses to the survey. We use an 11-element array **frequency** to count the number of occurrences of each response. As in Fig. 7.8, we ignore the first element (**frequency[0]**) because it is more logical to have the response 1 increment **frequency[1]** than **frequency[0]**. This allows us to use each response directly as a subscript on the **frequency** array. Each element of the array is used as a counter for one of the survey responses.

```

1  // Fig. 7.9: StudentPoll.java
2  // Student poll program
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class StudentPoll {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
13                             1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
14                             6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
15                             5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
16         int frequency[] = new int[ 11 ];
17     }

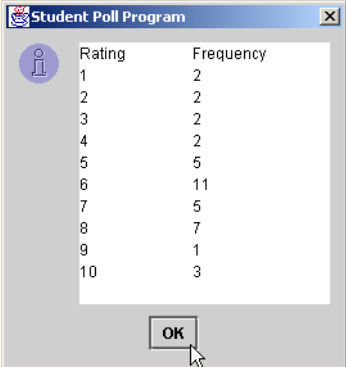
```

Fig. 7.9 A simple student-poll analysis program (part 1 of 2).

```

18      // for each answer, select value of an element of
19      // responses array and use that value as subscript in
20      // frequency array to determine element to increment
21      for ( int answer = 0; answer < responses.length; answer++ )
22          ++frequency[ responses[ answer ] ];
23
24      String output = "Rating\tFrequency\n";
25
26      // append frequencies to String output
27      for ( int rating = 1; rating < frequency.length; rating++ )
28          output += rating + "\t" + frequency[ rating ] + "\n";
29
30      JTextArea outputArea = new JTextArea();
31      outputArea.setText( output );
32
33      JOptionPane.showMessageDialog( null, outputArea,
34          "Student Poll Program",
35          JOptionPane.INFORMATION_MESSAGE );
36
37      System.exit( 0 );
38  }
39  }

```



Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

Fig. 7.9 A simple student-poll analysis program (part 2 of 2).

The **for** loop (lines 21–22) 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 22, which increments the appropriate **frequency** counter, depending on the value of **responses[answer]**.

Let's consider several iterations of the **for** loop. When counter **answer** is 0, **responses[answer]** is the value of the first element of array **responses** (i.e., 1), so the program interprets **++frequency[responses[answer]]** as

```
++frequency[ 1 ];
```

which increments the value in array element one. To evaluate the expression, start with the value in the innermost set of square brackets (**answer**). Once you know the value of **answer**, plug that value into the expression and evaluate the next outer set of square brackets

(**responses** [**answer**]). Then, use that value as the subscript for the **frequency** array to determine which counter to increment.

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

```
++frequency[ 2 ];
```

which increments array element two (the third element of the array).

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

```
++frequency[ 6 ];
```

which increments array element six (the seventh element of the array), 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 subscript values for an 11-element array are 0 through 10. Also, note that the summarized results are correct, because the elements of array **frequency** were initialized to zero when the array was allocated with **new**.

If the data contained invalid values, such as 13, the program would attempt to add 1 to **frequency** [13]. This is outside the bounds of the array. In the C and C++ programming languages, such a reference would be allowed by the compiler and at execution time. The program would “walk” past the end of the array to where it thought element number 13 was located and add 1 to whatever happened to be at that location in memory. This could potentially modify another variable in the program or even result in premature program termination. Java provides mechanisms to prevent accessing elements outside the bounds of the array.



Common Programming Error 7.4

Referring to an element outside the array bounds is a logic error.



Testing and Debugging Tip 7.1

When a Java program executes, the Java interpreter checks array element subscripts to be sure they are valid (i.e., all subscripts must be greater than or equal to 0 and less than the length of the array). If there is an invalid subscript, Java generates an exception.



Testing and Debugging Tip 7.2

*Exceptions indicate that an error occurred in a program. A programmer can write code to recover from an exception and continue program execution instead of abnormally terminating the program. When an invalid array reference occurs, Java generates an **ArrayIndexOutOfBoundsException**. Chapter 14 covers exception handling in detail.*



Testing and Debugging Tip 7.3

When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (one less than the size of the array). The loop terminating condition should prevent accessing elements outside this range.



Testing and Debugging Tip 7.4

Programs should validate all input values to prevent erroneous information from affecting a program's calculations.

7.5 References and Reference Parameters

Two ways to pass arguments to methods (or functions) in many programming languages (like C and C++) 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 made and passed to the called method.



Testing and Debugging Tip 7.5

With pass-by-value, changes to the called method's copy do not affect the original variable's value in the calling method. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.

When an argument is passed by reference, the caller gives the called method the ability to access the caller's data directly and to modify that data if the called method so chooses. Pass-by-reference improves performance, because it eliminates the overhead of copying large amounts of data.



Software Engineering Observation 7.1

Unlike other languages, Java does not allow the programmer to choose whether to pass each argument by value or by reference. Primitive data type variables are always passed by value. Objects are not passed to methods; rather, references to objects are passed to methods. The references themselves are passed by value—a copy of a reference is passed to a method. When a method receives a reference to an object, the method can manipulate the object directly.



Software Engineering Observation 7.2

*When returning information from a method via a **return** statement, primitive-data-type variables are always returned by value (i.e., a copy is returned) and objects are always returned by reference (i.e., a reference to the object is returned).*

To pass a reference to an object into a method, simply specify the reference name in the method call. Mentioning the reference by its parameter name in the body of the called method actually refers to the original object in memory, and the original object can be accessed directly by the called method.

Arrays are treated as objects by Java; therefore, arrays are passed to methods by reference—a called method can access the elements of the caller's original arrays. The name of an array is actually a reference to an object that contains the array elements and the **length** instance variable, which indicates the number of elements in the array. In the next section, we demonstrate pass-by-value and pass-by-reference using arrays.



Performance Tip 7.2

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 would consume considerable storage for the copies of the arrays.

7.6 Passing Arrays 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

```
int hourlyTemperatures[] = new int[ 24 ];
```

the method call

```
modifyArray( hourlyTemperatures );
```

passes array **hourlyTemperatures** to method **modifyArray**. In Java, every array object “knows” its own size (via the **length** instance variable). Thus, when we pass an array object into a method, we are not required to pass the size of the array as an additional argument.

Although entire arrays and objects referred to by individual elements of a nonprimitive-type array are passed by reference, individual array elements of primitive data types are passed by value exactly as simple variables are. Such simple single pieces of data are called *scalars* or *scalar quantities*. To pass an array element to a method, use the subscripted name of the array element as an argument in the method call.

For a method to receive an array through a method call, the method’s parameter list must specify an array parameter (or several if more than one array is to be received). For example, the method header for method **modifyArray** might be written as

```
void modifyArray( int b[] )
```

indicating that **modifyArray** expects to receive an integer array in parameter **b**. Since arrays are passed by reference, when the called method uses the array name **b**, it refers to the actual array in the caller (array **hourlyTemperatures** in the preceding call).

The applet of Fig. 7.10 demonstrates the difference between passing an entire array and passing an array element. Once again, we use an applet here, because we have not yet defined an application that contains methods other than **main**. We are still taking advantage of some of the features provided for free in an applet (such as the automatic creation of an applet object and the automatic calls to **init**, **start** and **paint** by the applet container). In Chapter 9, Object-Oriented Programming, we introduce applications that execute in their own windows. At that point, we will begin to see application classes containing several methods.

Lines 15–17 in method **init** define the **JTextArea** called **outputArea** and attach it to the applet’s content pane. The **for** structure at lines 26–27 appends the five elements of **array** (an array of **int** values) to the **String** called **output**. Line 29 invokes method **modifyArray** and passes it **array** as an argument. Method **modifyArray** (lines 50–54) multiplies each element by 2. To illustrate that **array**’s elements were modified, the **for** structure at lines 34–35 appends the five elements of **array** to **output** again. As the screen capture shows, method **modifyArray** did change the value of each element.

```
1  // Fig. 7.10: PassArray.java
2  // Passing arrays and individual array elements to methods
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
```

Fig. 7.10 Passing arrays and individual array elements to methods (part 1 of 3).

```

10 public class PassArray extends JApplet {
11
12     // initialize applet
13     public void init()
14     {
15         JTextArea outputArea = new JTextArea();
16         Container container = getContentPane();
17         container.add( outputArea );
18
19         int array[] = { 1, 2, 3, 4, 5 };
20
21         String output =
22             "Effects of passing entire array by reference:\n" +
23             "The values of the original array are:\n";
24
25         // append original array elements to String output
26         for ( int counter = 0; counter < array.length; counter++ )
27             output += "    " + array[ counter ];
28
29         modifyArray( array ); // array passed by reference
30
31         output += "\n\nThe values of the modified array are:\n";
32
33         // append modified array elements to String output
34         for ( int counter = 0; counter < array.length; counter++ )
35             output += "    " + array[ counter ];
36
37         output += "\n\nEffects of passing array " +
38             "element by value:\n" +
39             "a[3] before modifyElement: " + array[ 3 ];
40
41         // attempt to modify array[ 3 ]
42         modifyElement( array[ 3 ] );
43
44         output += "\na[3] after modifyElement: " + array[ 3 ];
45         outputArea.setText( output );
46
47     } // end method init
48
49     // multiply each element of an array by 2
50     public void modifyArray( int array2[] )
51     {
52         for ( int counter = 0; counter < array2.length; counter++ )
53             array2[ counter ] *= 2;
54     }
55
56     // multiply argument by 2
57     public void modifyElement( int element )
58     {
59         element *= 2;
60     }
61
62 } // end class PassArray

```

Fig. 7.10 Passing arrays and individual array elements to methods (part 2 of 3).

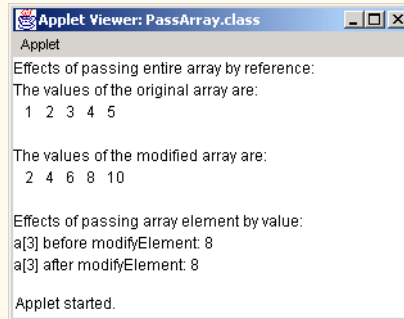


Fig. 7.10 Passing arrays and individual array elements to methods (part 3 of 3).

Next, the program demonstrates that individual elements of primitive-type arrays are passed to methods by value. To show the value of `array[3]` before calling method `modifyElement`, lines 37–39 append the value of `array[3]` (and other information) to `output`. Line 42 invokes method `modifyElement` and passes `array[3]` as an argument. Remember that `array[3]` is actually one `int` value in `array`. Also, remember that values of primitive types are passed to methods by value. Therefore, the program passes a copy of `array[3]`. Method `modifyElement` multiplies its argument by 2 and stores the result in its parameter `element`. Method parameters are local variables, so when `modifyElement` terminates, the local variable `element` is destroyed. Thus, when the program returns control to `init`, line 44 appends the unmodified value of `array[3]` to `output`. Line 45 displays the results in the `JTextArea`.

7.7 Sorting Arrays

Sorting data (i.e., placing the data into some particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name, to make it easy to find phone numbers. Virtually every organization must sort some data and in many cases massive amounts of data. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science. In this chapter, we discuss one of the simplest sorting schemes. In the exercises in this chapter, Chapter 19 and Chapter 21, we investigate more complex schemes that yield superior performance.



Performance Tip 7.3

Sometimes, the simplest algorithms perform poorly. Their virtue is that they are easy to program, test and debug. Sometimes, more complex algorithms are required to realize maximum performance.

Figure 7.11 sorts the values of `array` (a the 10-element array of `int` values) into ascending order. The technique we use is called the *bubble sort* or the *sinking sort*, because the smaller values gradually “bubble” their way to the top of the array (i.e., toward the first element) like air bubbles rising in water, while the larger values sink to the bottom (end) of the array. The technique uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values

are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the array. The applet contains methods **init**, **bubbleSort** and **swap**. Method **init** (lines 13–36) initializes the applet. Method **bubbleSort** (lines 39–58) is called from **init** to sort the elements of **array**. Method **bubbleSort** calls method **swap** (lines 61–68) as necessary to exchange two elements of the array.

Lines 24–25 append the original values of **array** to the **String** called **output**. Line 27 invokes method **bubbleSort** and passes **array** as the array to sort.

Method **bubbleSort** receives the array as parameter **array2**. The nested **for** structure at lines 42–56 performs the sort. The outer loop controls the number of passes of the array. The inner loop controls the comparisons and swapping (if necessary) of the elements during each pass.

```

1  // Fig. 7.11: BubbleSort.java
2  // Sort an array's values into ascending order.
3
4  // Java core packages
5  import java.awt.*;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class BubbleSort extends JApplet {
11
12     // initialize applet
13     public void init()
14     {
15         JTextArea outputArea = new JTextArea();
16         Container container = getContentPane();
17         container.add( outputArea );
18
19         int array[] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
20
21         String output = "Data items in original order\n";
22
23         // append original array values to String output
24         for ( int counter = 0; counter < array.length; counter++ )
25             output += "    " + array[ counter ];
26
27         bubbleSort( array ); // sort array
28
29         output += "\n\nData items in ascending order\n";
30
31         // append sorted\ array values to String output
32         for ( int counter = 0; counter < array.length; counter++ )
33             output += "    " + array[ counter ];
34
35         outputArea.setText( output );
36     }
37

```

Fig. 7.11 Sorting an array with bubble sort (part 1 of 2).

```

38 // sort elements of array with bubble sort
39 public void bubbleSort( int array2[] )
40 {
41     // loop to control number of passes
42     for ( int pass = 1; pass < array2.length; pass++ ) {
43
44         // loop to control number of comparisons
45         for ( int element = 0;
46             element < array2.length - 1;
47             element++ ) {
48
49             // compare side-by-side elements and swap them if
50             // first element is greater than second element
51             if ( array2[ element ] > array2[ element + 1 ] )
52                 swap( array2, element, element + 1 );
53
54         } // end loop to control comparisons
55
56     } // end loop to control passes
57
58 } // end method bubbleSort
59
60 // swap two elements of an array
61 public void swap( int array3[], int first, int second )
62 {
63     int hold; // temporary holding area for swap
64
65     hold = array3[ first ];
66     array3[ first ] = array3[ second ];
67     array3[ second ] = hold;
68 }
69
70 } // end class BubbleSort

```

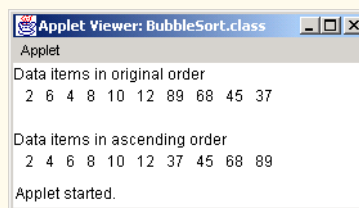


Fig. 7.11 Sorting an array with bubble sort (part 2 of 2).

Method `bubbleSort` first compares `array2[0]` to `array2[1]`, then `array2[1]` to `array2[2]`, then `array2[2]` to `array2[3]` so on until it completes the pass by comparing `array2[8]` to `array2[9]`. Although there are 10 elements, the comparison loop performs only nine comparisons. The comparisons performed in a bubble sort could cause a large value to move down the array (sink) many positions on a single pass. However, a small value may move up (bubble) only one position per pass. On the first pass, the largest value is guaranteed to sink to the bottom element of the array,

array2[9]. On the second pass, the second largest value is guaranteed to sink to **array2[8]**. On the ninth pass, the ninth largest value sinks to **array2[1]**. This leaves the smallest value in **array2[0]**, so only nine passes are required to sort a 10-element array.

If a comparison reveals that the two elements are in descending order, **bubbleSort** calls method **swap** to exchange the two elements, so they will be in ascending order in the array. Method **swap** receives a reference to the array (which it calls **array3**) and two integers representing the subscripts of the two elements of the array to exchange. The exchange is performed by the three assignments

```
hold = array3[ first ];
array3[ first ] = array3[ second ];
array3[ second ] = hold;
```

where the extra variable **hold** temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```
array3[ first ] = array3[ second ];
array3[ second ] = array3[ first ];
```

If **array3[first]** is 7 and **array3[second]** is 5, after the first assignment both array elements contain 5 and the value 7 is lost, hence the need for the extra variable **hold**.

The chief virtue of the bubble sort is that it is easy to program. However, the bubble sort runs slowly. This becomes apparent when sorting large arrays. In Exercise 7.11, we ask you to develop more efficient versions of the bubble sort. Other exercises investigate some sorting algorithms that are far more efficient than the bubble sort. More advanced courses (often titled “Data Structures,” “Algorithms” or “Computational Complexity”) investigate sorting and searching in greater depth.

7.8 Searching Arrays: Linear Search and Binary Search

Often, a programmer will be working with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain *key value*. The process of locating a particular element value in an array is called *searching*. In this section, we discuss two searching techniques—the simple *linear search* technique and the more efficient *binary search* technique. Exercise 7.31 and Exercise 7.32 at the end of this chapter ask you to implement recursive versions of the linear search and the binary search.

7.8.1 Searching an Array with Linear Search

In the applet of Fig. 7.12, method **linearSearch** (defined at lines 52–62) uses a **for** structure (lines 55–59) containing an **if** structure to compare each element of an array with a *search key*. If the search key is found, the method returns the subscript value for the element to indicate the exact position of the search key in the array. If the search key is not found, the method returns **-1** to indicate that the search key was not found. We return **-1** because it is not a valid subscript number. If the array being searched is not in any particular order, it is just as likely that the search key will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array.

Figure 7.12 contains a 100-element array filled with the even integers from 0 to 198. The user types the search key in a **JTextField** and presses *Enter* to start the search. [Note: We pass the array to **linearSearch** even though the array is an instance variable of the class. We do this because an array normally is passed to a method of another class for sorting. For example, class **Arrays** (see Chapter 21) contains a variety of **static** methods for sorting arrays, searching arrays, comparing the contents of arrays and filling arrays of all the primitive types, **Objects** and **Strings**.]

```

1  // Fig. 7.12: LinearSearch.java
2  // Linear search of an array
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class LinearSearch extends JApplet
12     implements ActionListener {
13
14     JLabel enterLabel, resultLabel;
15     JTextField enterField, resultField;
16     int array[];
17
18     // set up applet's GUI
19     public void init()
20     {
21         // get content pane and set its layout to FlowLayout
22         Container container = getContentPane();
23         container.setLayout( new FlowLayout() );
24
25         // set up JLabel and JTextField for user input
26         enterLabel = new JLabel( "Enter integer search key" );
27         container.add( enterLabel );
28
29         enterField = new JTextField( 10 );
30         container.add( enterField );
31
32         // register this applet as enterField's action listener
33         enterField.addActionListener( this );
34
35         // set up JLabel and JTextField for displaying results
36         resultLabel = new JLabel( "Result" );
37         container.add( resultLabel );
38
39         resultField = new JTextField( 20 );
40         resultField.setEditable( false );
41         container.add( resultField );
42
43         // create array and populate with even integers 0 to 198
44         array = new int[ 100 ];

```

Fig. 7.12 Linear search of an array (part 1 of 2).

```

45
46     for ( int counter = 0; counter < array.length; counter++ )
47         array[ counter ] = 2 * counter;
48
49 } // end method init
50
51 // Search array for specified key value
52 public int linearSearch( int array2[], int key )
53 {
54     // loop through array elements
55     for ( int counter = 0; counter < array2.length; counter++ )
56
57         // if array element equals key value, return location
58         if ( array2[ counter ] == key )
59             return counter;
60
61     return -1; // key not found
62 }
63
64 // obtain user input and call method linearSearch
65 public void actionPerformed((ActionEvent actionEvent) )
66 {
67     // input also can be obtained with enterField.getText()
68     String searchKey = actionEvent.getActionCommand();
69
70     // Array a is passed to linearSearch even though it
71     // is an instance variable. Normally an array will
72     // be passed to a method for searching.
73     int element =
74         linearSearch( array, Integer.parseInt( searchKey ) );
75
76     // display search result
77     if ( element != -1 )
78         resultField.setText( "Found value in element " +
79                             element );
80     else
81         resultField.setText( "Value not found" );
82 }
83
84 } // end class LinearSearch

```

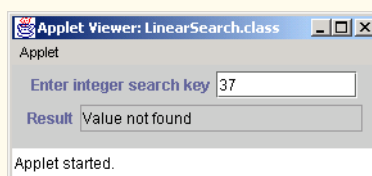
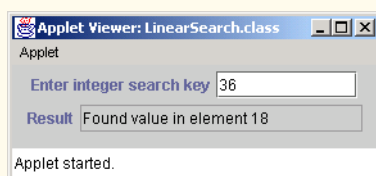


Fig. 7.12 Linear search of an array (part 2 of 2).

The linear search method works well for small arrays or for unsorted arrays. However, for large arrays, linear searching is inefficient. If the array is sorted, the high-speed *binary search* technique presented in the next section can be used.

7.8.2 Searching a Sorted Array with Binary Search

The binary search algorithm eliminates half of the elements in the array being searched after each comparison. The algorithm locates the middle array element and compares it to the search key. If they are equal, the search key has been found and binary search returns the subscript of that element. Otherwise, binary search reduces the problem to searching half of the array. If the search key is less than the middle array element, the first half of the array will be searched; otherwise, the second half of the array will be searched. If the search key is not the middle element in the specified subarray (piece of the original array), the algorithm repeats on one quarter of the original array. The search continues until the search key is equal to the middle element of a subarray or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).

In a worst-case scenario, searching a sorted array of 1024 elements will take only 10 comparisons using a binary search. Repeatedly dividing 1024 by 2 (because after each comparison we are able to eliminate half of the array) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 (2^{10}) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 (2^{20}) elements takes a maximum of 20 comparisons to find the key. An array of one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous increase in performance over the linear search that required comparing the search key to an average of half the elements in the array. For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

Figure 7.13 presents the iterative version of method **binarySearch** (lines 85–116). The method receives two arguments—an integer array called **array2** (the array to search) and an integer **key** (the search key). The program passes the array to **binarySearch** even though the array is an instance variable of the class. Once again, we do this because an array normally is passed to a method of another class for sorting. If **key** matches the **middle** element of a subarray, **binarySearch** returns **middle** (the subscript of the current element) to indicate that the value was found and the search is complete. If **key** does not match the **middle** element of a subarray, **binarySearch** adjusts the **low** subscript or **high** subscript (both declared in the method), to continue the search using a smaller subarray. If **key** is less than the middle element, the **high** subscript is set to **middle - 1** and the search continues on the elements from **low** to **middle - 1**. If **key** is greater than the middle element, the **low** subscript is set to **middle + 1** and the search continues on the elements from **middle + 1** to **high**. Method **binarySearch** performs these comparisons in the nested **if/else** structure at lines 102–111.

```

1  // Fig. 7.13: BinarySearch.java
2  // Binary search of an array
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.text.*;
```

Fig. 7.13 Binary search of a sorted array (part 1 of 5)

```
8
9 // Java extension packages
10 import javax.swing.*;
11
12 public class BinarySearch extends JApplet
13     implements ActionListener {
14
15     JLabel enterLabel, resultLabel;
16     JTextField enterField, resultField;
17     JTextArea output;
18
19     int array[];
20     String display = "";
21
22     // set up applet's GUI
23     public void init()
24     {
25         // get content pane and set its layout to FlowLayout
26         Container container = getContentPane();
27         container.setLayout( new FlowLayout() );
28
29         // set up JLabel and JTextField for user input
30         enterLabel = new JLabel( "Enter integer search key" );
31         container.add( enterLabel );
32
33         enterField = new JTextField( 10 );
34         container.add( enterField );
35
36         // register this applet as enterField's action listener
37         enterField.addActionListener( this );
38
39         // set up JLabel and JTextField for displaying results
40         resultLabel = new JLabel( "Result" );
41         container.add( resultLabel );
42
43         resultField = new JTextField( 20 );
44         resultField.setEditable( false );
45         container.add( resultField );
46
47         // set up JTextArea for displaying comparison data
48         output = new JTextArea( 6, 60 );
49         output.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
50         container.add( output );
51
52         // create array and fill with even integers 0 to 28
53         array = new int[ 15 ];
54
55         for ( int counter = 0; counter < array.length; counter++ )
56             array[ counter ] = 2 * counter;
57
58     } // end method init
59
```

Fig. 7.13 Binary search of a sorted array (part 2 of 5)

```

60 // obtain user input and call method binarySearch
61 public void actionPerformed((ActionEvent) actionEvent )
62 {
63     // input also can be obtained with enterField.getText()
64     String searchKey = actionEvent.getActionCommand();
65
66     // initialize display string for new search
67     display = "Portions of array searched\n";
68
69     // perform binary search
70     int element =
71         binarySearch( array, Integer.parseInt( searchKey ) );
72
73     output.setText( display );
74
75     // display search result
76     if ( element != -1 )
77         resultField.setText(
78             "Found value in element " + element );
79     else
80         resultField.setText( "Value not found" );
81
82 } // end method actionPerformed
83
84 // method to perform binary search of an array
85 public int binarySearch( int array2[], int key )
86 {
87     int low = 0; // low element subscript
88     int high = array.length - 1; // high element subscript
89     int middle; // middle element subscript
90
91     // loop until low subscript is greater than high subscript
92     while ( low <= high ) {
93
94         // determine middle element subscript
95         middle = ( low + high ) / 2;
96
97         // display subset of array elements used in this
98         // iteration of binary search loop
99         buildOutput( array2, low, middle, high );
100
101         // if key matches middle element, return middle location
102         if ( key == array2[ middle ] )
103             return middle;
104
105         // if key less than middle element, set new high element
106         else if ( key < array2[ middle ] )
107             high = middle - 1;
108
109         // key greater than middle element, set new low element
110         else
111             low = middle + 1;
112     }

```

Fig. 7.13 Binary search of a sorted array (part 3 of 5)

```

113
114     return -1;    // key not found
115
116 } // end method binarySearch
117
118 // build row of output showing subset of array elements
119 // currently being processed
120 void buildOutput( int array3[],
121                 int low, int middle, int high )
122 {
123     // create 2-digit integer number format
124     DecimalFormat twoDigits = new DecimalFormat( "00" );
125
126     // loop through array elements
127     for ( int counter = 0; counter < array3.length;
128         counter++ ) {
129
130         // if counter outside current array subset, append
131         // padding spaces to String display
132         if ( counter < low || counter > high )
133             display += "    ";
134
135         // if middle element, append element to String display
136         // followed by asterisk (*) to indicate middle element
137         else if ( counter == middle )
138             display +=
139                 twoDigits.format( array3[ counter ] ) + "* ";
140
141         // append element to String display
142         else
143             display +=
144                 twoDigits.format( array3[ counter ] ) + " ";
145
146     } // end for structure
147
148     display += "\n";
149
150 } // end method buildOutput
151
152 } // end class BinarySearch

```

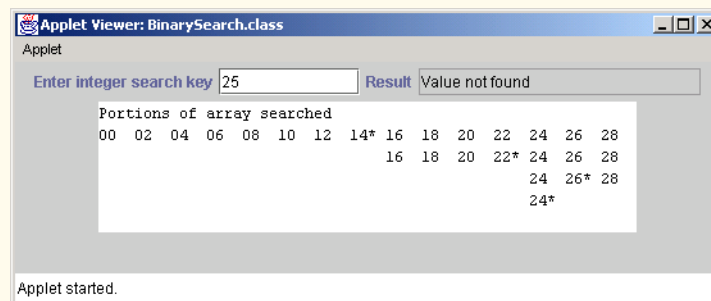


Fig. 7.13 Binary search of a sorted array (part 4 of 5)

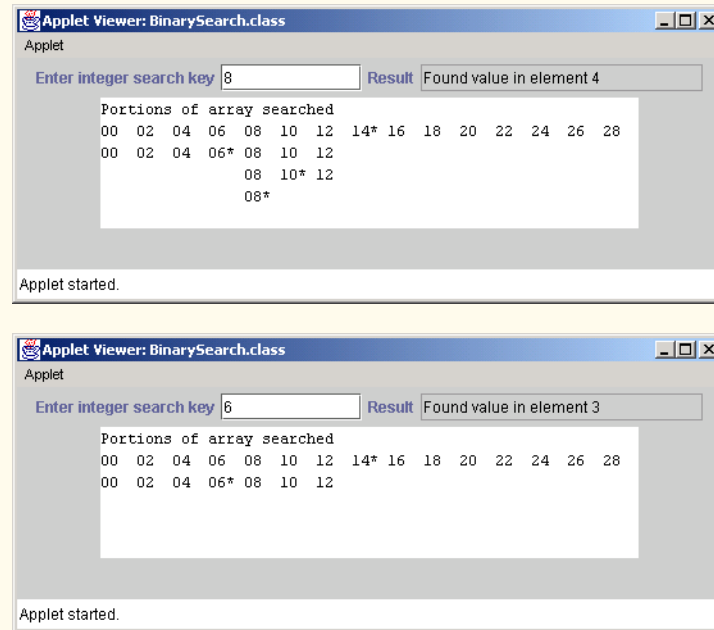


Fig. 7.13 Binary search of a sorted array (part 5 of 5)

The program uses a 15-element array. The first power of 2 greater than the number of array elements is 16 (2^4); therefore, **binarySearch** requires at most four comparisons to find the **key**. To illustrate this, line 99 of method **binarySearch** calls method **buildOutput** (defined at lines 120–150) to output each subarray during the binary search process. Method **buildOutput** marks the middle element in each subarray with an asterisk (*) to indicate the element to which the **key** is compared. Each search in this example results in a maximum of four lines of output—one per comparison.

JTextArea output uses *Monospaced* font (a *fixed-width font*—i.e., all characters are the same width) to help align the displayed text in each line of output. Line 49 uses method **setFont** to change the font displayed in **output**. Method **setFont** can change the font of text displayed on most GUI components. The method requires a **Font** (package **java.awt**) object as its argument. A **Font** object is initialized with three arguments—the **String** name of the font ("**Monospaced**"), an **int** representing the style of the font (**Font.PLAIN** is a constant integer defined in class **Font** that indicates plain font) and an **int** representing the point size of the font (**12**). Java provides generic names for several fonts available on every Java platform. *Monospaced* font is also called *Courier*. Other common fonts include *Serif* (also called *TimesRoman*) and *SansSerif* (also called *Helvetica*). Java 2 actually provides access to all fonts on your system through methods of class **GraphicsEnvironment**. The style can also be **Font.BOLD**, **Font.ITALIC** or **Font.BOLD + Font.ITALIC**. The point size represents the size of the font—there are 72 points to an inch. The actual size of the text as it appears on the screen may vary based on the size of the screen and the screen resolution. We discuss font manipulation again in Chapter 11.

7.9 Multiple-Subscripted Arrays

Multiple-subscripted arrays with two subscripts 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 the two subscripts—by convention, the first identifies the element's row and the second identifies the element's column. Arrays that require two subscripts to identify a particular element are called *double-subscripted arrays* or *two-dimensional arrays*. Note that multiple-subscripted arrays can have more than two subscripts. Java does not support multiple-subscripted arrays directly, but does allow the programmer to specify single-subscripted arrays whose elements are also single-subscripted arrays, thus achieving the same effect. Figure 7.14 illustrates a double-subscripted array, **a**, containing three rows and four columns (i.e., a 3-by-4 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. 7.14 by an element name of the form **a[row][column]**; **a** is the name of the array and **row** and **column** are the subscripts that uniquely identify the row and column of each element in **a**. Notice that the names of the elements in the first row all have a first subscript of **0**; the names of the elements in the fourth column all have a second subscript of **3**.

Multiple-subscripted arrays can be initialized with initializer lists in declarations like a single-subscripted array. A double-subscripted array **b[2][2]** could be declared and initialized with

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

The values are grouped by row in braces. So, **1** and **2** initialize **b[0][0]** and **b[0][1]**, and **3** and **4** initialize **b[1][0]** and **b[1][1]**. The compiler determines the number of rows by counting the number of initializer sublists (represented by sets of braces) in the initializer list. The compiler determines the number of columns in each row by counting the number of initializer values in the initializer sublist for that row.

Multiple-subscripted arrays are maintained as arrays of arrays. The declaration

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

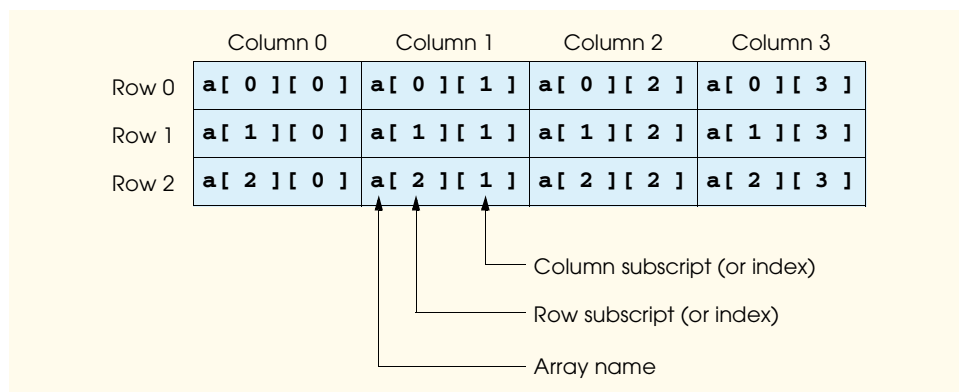


Fig. 7.14 A double-subscripted array with three rows and four columns.

creates integer array **b** with row **0** containing two elements (**1** and **2**) and row **1** containing three elements (**3**, **4** and **5**).

A multiple-subscripted array with the same number of columns in every row can be allocated dynamically. For example, a 3-by-4 array is allocated as follows:

```
int b[][];
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. Note that programs also can use variables to specify array dimensions. As with single-subscripted arrays, the elements of a double-subscripted array are initialized when **new** creates the array object.

A multiple-subscripted array in which each row has a different number of columns can be allocated dynamically as follows:

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

The preceding code creates a two-dimensional array with two rows. Row **0** has five columns and row **1** has three columns.

The applet of Fig. 7.15 demonstrates initializing double-subscripted arrays in declarations and using nested **for** loops to traverse the arrays (i.e., manipulate every element of the array).

```
1 // Fig. 7.15: InitArray.java
2 // Initializing multidimensional arrays
3
4 // Java core packages
5 import java.awt.Container;
6
7 // Java extension packages
8 import javax.swing.*;
9
10 public class InitArray extends JApplet {
11     JTextArea outputArea;
12
13     // set up GUI and initialize applet
14     public void init()
15     {
16         outputArea = new JTextArea();
17         Container container = getContentPane();
18         container.add( outputArea );
19
20         int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
21         int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
22
23         outputArea.setText( "Values in array1 by row are\n" );
24         buildOutput( array1 );
```

Fig. 7.15 Initializing multidimensional arrays (part 1 of 2).

```

25
26     outputArea.append( "\nValues in array2 by row are\n" );
27     buildOutput( array2 );
28 }
29
30 // append rows and columns of an array to outputArea
31 public void buildOutput( int array[][] )
32 {
33     // loop through array's rows
34     for ( int row = 0; row < array.length; row++ ) {
35
36         // loop through columns of current row
37         for ( int column = 0;
38              column < array[ row ].length;
39              column++ )
40             outputArea.append( array[ row ][ column ] + " " );
41
42         outputArea.append( "\n" );
43     }
44 }
45 }

```

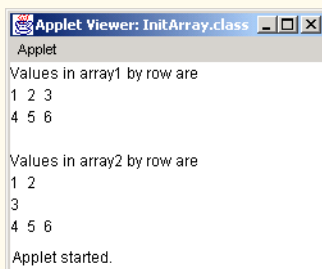


Fig. 7.15 Initializing multidimensional arrays (part 2 of 2).

The program declares two arrays in method **init**. The declaration of **array1** (line 20) provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3. The second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of **array2** (line 21) provides six initializers in three sublists. The sublist for the first row explicitly initializes the first row to have two elements with values 1 and 2, respectively. The sublist for the second row initializes the second row to have one element with value 3. The sublist for the third row initializes the third row to the values 4, 5 and 6.

Line 24 of method **init** calls method **buildOutput** (defined at lines 31–44) to append each array's elements to **outputArea** (a **JTextArea**). Method **buildOutput** specifies the array parameter as **int array[][]** to indicate that the method receives a double-subscripted array as an argument. Note the use of a nested **for** structure (lines 34–43) to output the rows of a double-subscripted array. In the outer **for** structure, the expression **array.length** determines the number of rows in the array. In the inner **for** structure, 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.

Many common array manipulations use **for** repetition structures. For example, the following **for** structure sets all the elements in the third row of array **a** in Fig. 7.14 to zero:

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

We specified the *third* row; therefore, we know that the first subscript is always **2** (**0** is the first row and **1** is the second row). The **for** loop varies only the second subscript (i.e., the column subscript). The preceding **for** structure is equivalent to the assignment statements

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

The following nested **for** structure totals the values of all the elements in array **a**.

```
int total = 0;

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

The **for** structure totals the elements of the array one row at a time. The outer **for** structure begins by setting the **row** subscript to **0** so that the elements of the first row may be totaled by the inner **for** structure. The outer **for** structure then increments **row** to **1** so that the second row can be totaled. Then, the outer **for** structure increments **row** to **2** so that the third row can be totaled. The result can be displayed when the nested **for** structure terminates.

The applet of Fig. 7.16 performs several other common array manipulations on 3-by-4 array **grades**. Each row of the array represents a student, and each column represents a grade on one of the four exams the students took during the semester. Four methods perform the array manipulations. Method **minimum** (lines 52–69) determines the lowest grade of any student for the semester. Method **maximum** (lines 72–89) determines the highest grade of any student for the semester. Method **average** (lines 93–103) determines a particular student’s semester average. Method **buildString** (lines 106–121) appends the double-subscripted array to **String output** in a tabular format.

Methods **minimum**, **maximum** and **buildString** each use array **grades** and the variables **students** (number of rows in the array) and **exams** (number of columns in the array). Each method loops through array **grades** by using nested **for** structures—for example, the nested **for** structure from the definition of method **minimum** (lines 58–66). The outer **for** structure sets **row** (the row subscript) to **0** so that the elements of the first row can be compared to variable **lowGrade** in the body of the inner **for** structure. The inner **for** structure loops through the four grades of a particular row and compares each grade to **lowGrade**. If a grade is less than **lowGrade**, **lowGrade** is set to that grade. The outer **for** structure then increments the row subscript by **1**. The elements of the second row are compared to variable **lowGrade**. The outer **for** structure then increments the row subscript to **2**. The elements of the third row are compared to variable **lowGrade**. When

execution of the nested structure is complete, **lowGrade** contains the smallest grade in the double-subscripted array. Method **maximum** works similarly to method **minimum**.

```

1  // Fig. 7.16: DoubleArray.java
2  // Double-subscripted array example
3
4  // Java core packages
5  import java.awt.*;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class DoubleArray extends JApplet {
11     int grades[][] = { { 77, 68, 86, 73 },
12                       { 96, 87, 89, 81 },
13                       { 70, 90, 86, 81 } };
14
15     int students, exams;
16     String output;
17     JTextArea outputArea;
18
19     // initialize instance variables
20     public void init()
21     {
22         students = grades.length;    // number of students
23         exams = grades[ 0 ].length;  // number of exams
24
25         // create JTextArea and attach to applet
26         outputArea = new JTextArea();
27         Container container = getContentPane();
28         container.add( outputArea );
29
30         // build output string
31         output = "The array is:\n";
32         buildString();
33
34         // call methods minimum and maximum
35         output += "\n\nLowest grade: " + minimum() +
36                 "\n\nHighest grade: " + maximum() + "\n";
37
38         // call method average to calculate each student's average
39         for ( int counter = 0; counter < students; counter++ )
40             output += "\n\nAverage for student " + counter + " is " +
41                     average( grades[ counter ] );
42
43         // change outputArea's display font
44         outputArea.setFont(
45             new Font( "Courier", Font.PLAIN, 12 ) );
46
47         // place output string in outputArea
48         outputArea.setText( output );
49     }

```

Fig. 7.16 Example of using double-subscripted arrays (part 1 of 3).

```
50
51 // find minimum grade
52 public int minimum()
53 {
54     // assume first element of grades array is smallest
55     int lowGrade = grades[ 0 ][ 0 ];
56
57     // loop through rows of grades array
58     for ( int row = 0; row < students; row++ )
59
60         // loop through columns of current row
61         for ( int column = 0; column < exams; column++ )
62
63             // Test if current grade is less than lowGrade.
64             // If so, assign current grade to lowGrade.
65             if ( grades[ row ][ column ] < lowGrade )
66                 lowGrade = grades[ row ][ column ];
67
68     return lowGrade; // return lowest grade
69 }
70
71 // find maximum grade
72 public int maximum()
73 {
74     // assume first element of grades array is largest
75     int highGrade = grades[ 0 ][ 0 ];
76
77     // loop through rows of grades array
78     for ( int row = 0; row < students; row++ )
79
80         // loop through columns of current row
81         for ( int column = 0; column < exams; column++ )
82
83             // Test if current grade is greater than highGrade.
84             // If so, assign current grade to highGrade.
85             if ( grades[ row ][ column ] > highGrade )
86                 highGrade = grades[ row ][ column ];
87
88     return highGrade; // return highest grade
89 }
90
91 // determine average grade for particular
92 // student (or set of grades)
93 public double average( int setOfGrades[] )
94 {
95     int total = 0; // initialize total
96
97     // sum grades for one student
98     for ( int count = 0; count < setOfGrades.length; count++ )
99         total += setOfGrades[ count ];
100 }
```

Fig. 7.16 Example of using double-subscripted arrays (part 2 of 3).

```

101     // return average of grades
102     return ( double ) total / setOfGrades.length;
103 }
104
105 // build output string
106 public void buildString()
107 {
108     output += "          "; // used to align column heads
109
110     // create column heads
111     for ( int counter = 0; counter < exams; counter++ )
112         output += "[" + counter + " ] ";
113
114     // create rows/columns of text representing array grades
115     for ( int row = 0; row < students; row++ ) {
116         output += "\ngrades[" + row + " ] ";
117
118         for ( int column = 0; column < exams; column++ )
119             output += grades[ row ][ column ] + " ";
120     }
121 }
122 }

```

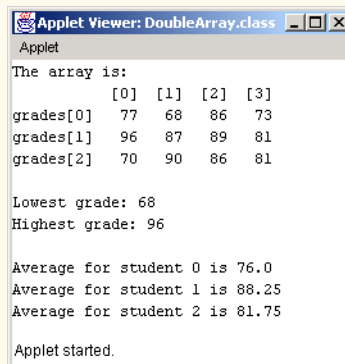


Fig. 7.16 Example of using double-subscripted arrays (part 3 of 3).

Method **average** takes one argument—a single-subscripted array of test results for a particular student. When line 41 calls **average**, the argument is **grades[counter]**, which specifies that a particular row of the double-subscripted array **grades** should be passed to **average**. For example, the argument **grades[1]** represents the four values (a single-subscripted array of grades) stored in the second row of the double-subscripted array **grades**. Remember that, in Java, a double-subscripted array is an array with elements that are single-subscripted arrays. Method **average** 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.

7.10 (Optional Case Study) Thinking About Objects: 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. We say that objects send *messages* to other objects. We explain how messages are sent and received in Java in Section 10.22. A *collaboration* consists of an object of one class sending a particular message to an object of another class.

The message sent by the first object invokes an operation of the second object. In “Thinking About Objects” Section 6.17, we determined many of the operations of the classes in our system. In this section, we concentrate on the messages that invoke these operations. Figure 7.22 is the table of classes and verb phrases from Section 6.17. We have removed all the verb phrases in classes **Elevator** and **Person** that do not correspond to operations. The remaining phrases are our first estimate of the collaborations in our system. As we proceed through this and the remaining “Thinking About Objects” sections, we will discover additional collaborations.

We examine the list of verb phrases to determine the collaborations in our system. For example, class **Elevator** lists the phrase “resets elevator button.” To accomplish this task, an object of class **Elevator** sends the **resetButton** message to an object of class **ElevatorButton** (invoking the **resetButton** operation of **ElevatorButton**). Figure 7.23 lists all the collaborations that can be gleaned from our table of verb phrases. According to Fig. 7.22, the **Elevator**¹ rings the **Bell** and opens (and closes) the **ElevatorDoor**, so we include a **ringBell**, **openDoor** and **closeDoor** message in Fig. 7.23. However, we must consider how the **FloorDoors** open and close. According to the class diagram of Fig. 3.23, **ElevatorShaft** is the only class that associates with **FloorDoor**. The **Elevator** signals its arrival (we assume to the **ElevatorShaft**) by sending an **elevatorArrived** message. The **ElevatorShaft** responds to this message by resetting the appropriate **FloorButton** and turning on the appropriate **Light**—the **ElevatorShaft** sends **resetButton** and **turnOnLight** messages. At this point in the design, we may assume that the **Elevator** also signals its departure—that is, the **ElevatorShaft** sends an **elevatorDeparted** message to the **ElevatorShaft**, which then turns off the appropriate **Light** by sending it a **turnOffLight** message.

Class	Verb phrases
Elevator	Resets elevator button, rings elevator bell, signals its arrival, signals its departure, opens its door, closes its door.
ElevatorShaft	Turns off light, turns on light, resets floor button.

Fig. 7.22 Verb phrases for each class exhibiting behaviors in simulation (part 1 of 2).

1. We refer to an object by using that object’s class name preceded by an article (“a,” “an” or “the”)—for example, the **Elevator** refers to the object of class **Elevator**. Our syntax avoids redundancy—i.e., we avoid repeating the phrase “an object of class....”

Class	Verb phrases
Person	Presses floor button, presses elevator button, rides elevator, enters elevator, exits elevator.
FloorButton	Summons (requests) elevator.
ElevatorButton	Signals elevator to move to opposite floor.
FloorDoor	Signals person to enter elevator (by opening).
ElevatorDoor	Signals person to exit elevator (by opening), opens floor door, closes floor door.
ElevatorModel	Creates person.

Fig. 7.22 Verb phrases for each class exhibiting behaviors in simulation (part 2 of 2).

An object of class...	Sends the message...	To an object of class...
Elevator	resetButton ringBell elevatorArrived elevatorDeparted openDoor closeDoor	ElevatorButton Bell ElevatorShaft ElevatorShaft ElevatorDoor ElevatorDoor
ElevatorShaft	resetButton turnOnLight turnOffLight	FloorButton Light Light
Person	pressButton enterElevator exitElevator	FloorButton, ElevatorButton Elevator Elevator
FloorButton	requestElevator	Elevator
ElevatorButton	moveElevator	Elevator
FloorDoor	doorOpened doorClosed	Person Person
ElevatorDoor	doorOpened doorClosed openDoor closeDoor	Person Person FloorDoor FloorDoor

Fig. 7.23 Collaborations in the elevator system.

A **Person** may press either a **FloorButton** or the **ElevatorButton**. A **Person** object may enter and exit the **Elevator**. Therefore, a **Person** may send **pressButton**, **enterElevator** and **exitElevator** messages. A **FloorButton** requests, or summons, the **Elevator**, so a **FloorButton** may send a **requestElevator** message. The **ElevatorButton** signals the **Elevator** to begin moving to the other floor, so the **ElevatorButton** may send a **moveElevator** message.

Both a **FloorDoor** and the **ElevatorDoor** inform a **Person** that they have opened or closed, so both objects send **doorOpened** and **doorClosed** messages.² Finally, the **ElevatorModel** creates a **Person**, but the **ElevatorModel** cannot send a **personCreated** message, because that **Person** does not yet exist. We discover, in Section 8.6, how to use a special method that creates, or *instantiates*, new objects. This method is referred to as a *constructor*—the **ElevatorModel** will create a **Person** by calling that **Person**’s constructor (which is similar to sending the **personCreated** message).

Lastly, the **ElevatorDoor** must send **openDoor** and **closeDoor** messages to a **FloorDoor** to guarantee that these doors open and close together.

Collaboration Diagrams

Now let us consider the objects that must interact so that people in our simulation can enter and exit the elevator when it arrives on a floor. The UML provides the *collaboration diagram* to model such interactions. Collaboration diagrams are a type of *interaction diagram*; they model the behavioral aspects of the system by providing information about how objects interact. Collaboration diagrams emphasize which objects participate in the interactions. The other type of interaction diagram is the *sequence diagram*, which we present in Chapter 15. Figure 7.24 shows a collaboration diagram that models a person who is pressing a floor button. An object in a collaboration diagram is represented as a rectangle that encloses the object’s name. We write object names in the collaboration diagram by using the convention we introduced in the object diagram of Fig. 3.24—objects are written in the form **objectName : ClassName**. In this example, we disregard the object name, because we care only about the object type. Collaborating 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 a method belonging to the receiving object—think of the name as a “service” that the receiving object (a “server”) provides for its sending objects (its “clients”).

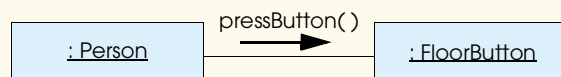


Fig. 7.24 Collaboration diagram of a person pressing a floor button

The arrow in Fig. 7.24 represents a message in the UML and a method—or *synchronous call*—in Java. This arrow indicates that the flow of control is from the sending object to the receiving object—the sending object may not send another message until the receiving object processes the message and returns control to the sending object. For example, in Fig. 7.24, a **Person** calls method **pressButton** of a **FloorButton** and

- Note that most of the messages perform some specific action on the receiving object; for example, the **Elevator** resets the **ElevatorButton**. However, other messages inform receiving objects of *events* that have already happened; for example, the **FloorDoor** informs the **Person** that the **FloorDoor** has opened. In Section 10.22, we elaborate on the topic of events—for now, however, we proceed as if the two types of messages are indistinguishable.

may not send another message to an object until **pressButton** has finished and returns control to that **Person**. If our program contains a **FloorButton** object called **firstFloorButton**, and we assume that the **Person** manages the flow of control, the Java code implementation in class **Person** that represents this collaboration diagram is

```
firstFloorButton.pressButton();
```

Figure 7.25 shows a collaboration diagram that models the interactions among objects in the system while objects of class **Person** enter and exit the elevator. The collaboration begins when the **Elevator** arrives on a **Floor**. The number to the left of the message name indicates the order in which the message is passed. The *sequence of messages* in a collaboration diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message **1** and ends with message **4.2**. The sequence of passing messages follows a nested structure—for example, message **1.1** is the *first message* nested in message **1**, and message **3.2** is the *second message* nested in message **3**. Message **3.2.1** would be the *first message* nested in message **3.2**. A message may be passed only when all nested messages from the previous message have been passed. For example, in Fig. 7.25, the **Elevator** passes message **4** after message **3**, message **3.1**, message **3.1.1**, message **3.1.1.1**, message **3.2** and message **3.2.1** are passed.

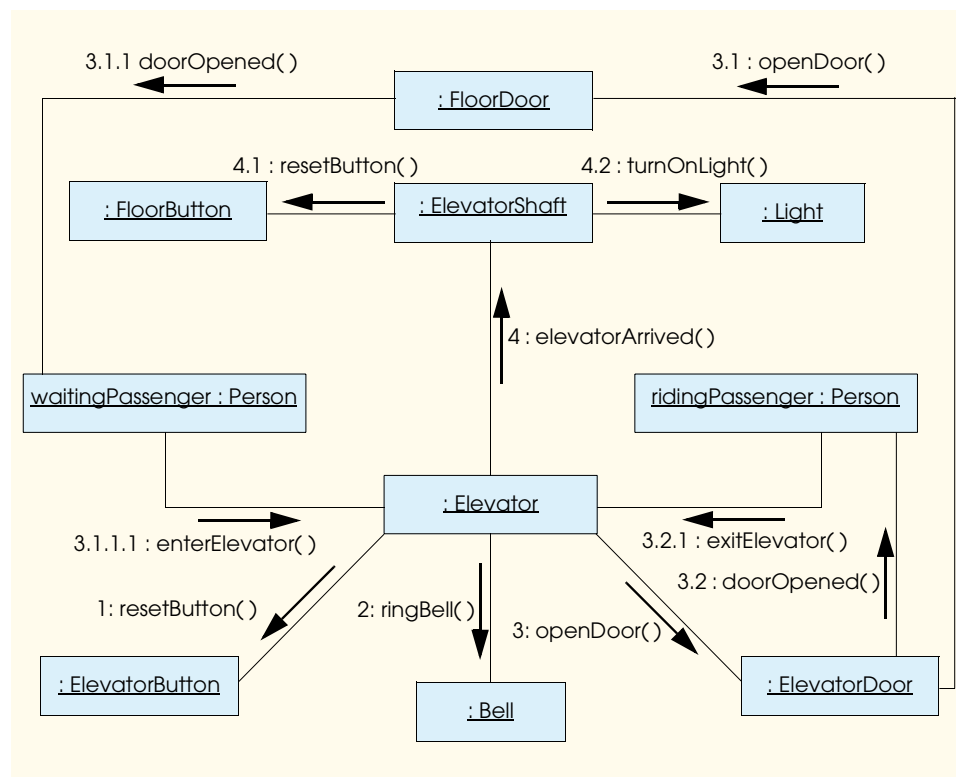


Fig. 7.25 Collaboration diagram for passengers exiting and entering the elevator.

The **Elevator** sends the **resetButton** message (message 1) to the **ElevatorButton** to reset the **ElevatorButton**. The **Elevator** sends the **ringBell** message (message 2) to the **Bell**, then opens the **ElevatorDoor** by passing the **openDoor** message (message 3). The **ElevatorDoor** then opens the **FloorDoor** by sending an **openDoor** message (message 3.1 at the top of the diagram) to that **FloorDoor**. The **FloorDoor** informs the **waitingPassenger** that the **FloorDoor** has opened (message 3.1.1), and the **waitingPassenger** enters the **Elevator** (message 3.1.1.1). The **ElevatorDoor** then informs the **ridingPassenger** that the **ElevatorDoor** has opened (message 3.2), so that the **ridingPassenger** may exit the **Elevator** (message 3.2.1). Lastly, the **Elevator** informs the **ElevatorShaft** of the arrival (message 4), so that the **ElevatorShaft** can reset the **FloorButton** (message 4.1) and turn on the **Light** (message 4.2).

Unfortunately, this design creates a problem. According to the diagram, the **waitingPassenger** enters the **Elevator** (message 3.1.1.1) before the **ridingPassenger** (message 3.2.1) exits. In “Thinking About Objects” Section 15.12, we apply multithreading, synchronization and active classes to our collaboration diagram, to force the **waitingPassenger** to wait for the **ridingPassenger** to exit the **Elevator**. Before we correct this problem, we modify this diagram to indicate more accurately the message passing in Section 10.22 when we discuss *event handling*.

SUMMARY

- Java stores lists of values in arrays. An array is a contiguous group of related memory locations. These locations are related by the fact that they all have the same name and the same type. To refer to a particular location or element within the array, we specify the name of the array and the subscript (or index or position number) of the element.
- Each array has a **length** member that is set to the number of elements in the array at the time the program creates the array object.
- A subscript may be an integer or an integer expression. If a program uses an expression as a subscript, the program evaluates the expression to determine the particular element of the array.
- Java arrays always begin with element 0; thus, it is important to note the difference when referring to the “seventh element of the array” as opposed to “array element seven.” The seventh element has a subscript of **6**, while array element seven has a subscript of **7** (actually the eighth element of the array).
- Arrays occupy space in memory and are considered to be objects. Operator **new** must be used to reserve space for an array. For example, the following creates an array of 100 **int** values:

```
int b[] = new int[ 100 ];
```

- When declaring an array, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all identifiers in the declaration represent arrays, as in

```
double[] array1, array2;
```

- The elements of an array can be initialized with initializer lists in a declaration and by input.
- Java prevents referencing elements beyond the bounds of an array. If this occurs during program execution, an **ArrayIndexOutOfBoundsException** occurs.
- Constant variables must be initialized with a constant expression before they are used and cannot be modified thereafter.

- To pass an array to a method, pass the name of the array. To pass a single element of an array to a method, simply pass the name of the array followed by the subscript of the particular element.
- Arrays are passed to methods as references—therefore, the called methods can modify the element values in the caller's original arrays. Single elements of primitive-data-type arrays are passed to methods by value.
- To receive an array argument, the method must specify an array parameter in the parameter list.
- An array can be sorted by using the bubble-sort technique. Several passes of the array are made. On each pass, successive pairs of elements are compared. If a pair is in order (or the values are identical), it is left as is. If a pair is out of order, the values are swapped. For small arrays, the bubble sort is acceptable, but for larger arrays it is inefficient compared to more sophisticated sorting algorithms.
- The linear search compares each element of the array with the search key. If the array is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array. Linear search works well for small arrays and is acceptable even for large unsorted arrays.
- For sorted arrays, the binary search eliminates from consideration one half of the elements in the array after each comparison. The algorithm locates the middle element of the array and compares it with the search key. If they are equal, the search key is found and the array subscript of that element is returned. Otherwise, the problem is reduced to searching one half of the array that is still under consideration. In a worst-case scenario, searching a sorted array of 1024 elements will take only 10 comparisons using a binary search.
- Most GUI components have method **setFont** to change the font of the text on the GUI component. The method requires a **Font** (package **java.awt**) object as its argument.
- A **Font** object is initialized with three arguments—a **String** representing the name of the font, an **int** representing the style of the font and an **int** representing the point size of the font. The style can be **Font.PLAIN**, **Font.BOLD**, **Font.ITALIC** or **Font.BOLD + Font.ITALIC**. The point size represents the size of the font—there are 72 points to an inch. The actual screen size may vary based on the size of the screen and the screen resolution.
- Arrays may be used to represent tables of values consisting of information arranged in rows and columns. To identify a particular element of a table, two subscripts are specified: The first identifies the row in which the element is contained, and the second identifies the column in which the element is contained. Tables or arrays that require two subscripts to identify a particular element are called double-subscripted arrays.
- A double-subscripted array can be initialized with an initializer list of the form


```
arrayType arrayName[][] = { { row1 sublist }, { row2 sublist }, ... };
```
- To dynamically create an array with a fixed number of rows and columns, use


```
arrayType arrayName[][] = new arrayType[ numRows ][ numColumns ];
```
- To pass one row of a double-subscripted array to a method that receives a single-subscripted array, simply pass the name of the array followed by only the row subscript.

TERMINOLOGY

a[i]	bubble sort
a[i][j]	column subscript
array	constant variable
array initializer list	declare an array
binary search of an array	double-subscripted array
bounds checking	element of an array

final	passing arrays to methods
Font class from java.awt	position number
Font.BOLD	row subscript
Font.ITALIC	search key
Font.PLAIN	searching an array
index	setFont method
initialize an array	single-subscripted array
initializer	sinking sort
linear search of an array	sorting
<i>lvalue</i>	sorting an array
m-by-n array	square brackets, []
multiple-subscripted array	subscript
name of an array	table of values
named constant	tabular format
off-by-one error	temporary area for exchange of values
pass of a bubble sort	value of an element
pass-by-reference	zeroth element
pass-by-value	

SELF-REVIEW EXERCISES

- 7.1** Fill in the blank(s) in each of the following statements:
- Lists and tables of values can be stored in _____.
 - The elements of an array are related by the fact that they have the same _____ and _____.
 - The number used to refer to a particular element of an array is called its _____.
 - The process of placing the elements of an array in order is called _____ the array.
 - Determining if an array contains a certain key value is called _____ the array.
 - An array that uses two subscripts is referred to as a _____ array.
- 7.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- An array can store many different types of values.
 - An array subscript should normally be of data type **float**.
 - 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.
- 7.3** Answer the following questions regarding an array called **fractions**:
- Define a constant variable **ARRAY_SIZE** initialized to 10.
 - Declare an array with **ARRAY_SIZE** elements of type **float** and initialize the elements to 0.
 - Name the fourth element of the array.
 - Refer to array element 4.
 - Assign the value **1.667** to array element 9.
 - Assign the value **3.333** to the seventh element of the array.
 - Sum all the elements of the array using a **for** repetition structure. Define the integer variable **x** as a control variable for the loop.
- 7.4** Answer the following questions regarding an array called **table**:
- Declare and create the array as an integer array and with 3 rows and 3 columns. Assume the constant variable **ARRAY_SIZE** has been defined to be 3.
 - How many elements does the array contain?
 - Use a **for** repetition structure to initialize each element of the array to the sum of its subscripts. Assume the integer variables **x** and **y** are declared as control variables.

7.5 Find the error in each of the following program segments and correct the error:

- 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) name, type. c) subscript (or index or position number). d) sorting. e) searching. f) double-subscripted (or two-dimensional).

- 7.2**
- a) False. An array can store only values of the same type.
 - b) False. An array subscript must be an integer or an integer expression.
 - c) False for individual primitive-data-type elements of an array because they are passed by value. If a reference to an array is passed, then modifications to the array elements are reflected in the original. Also, an individual element of a nonprimitive type is passed by reference, and changes to the object will be reflected in the original array element.

- 7.3**
- a) `final int ARRAY_SIZE = 10;`
 - b) `float fractions[] = new float[ ARRAY_SIZE ];`
 - c) `fractions[ 3 ]`
 - d) `fractions[ 4 ]`
 - e) `fractions[ 9 ] = 1.667;`
 - f) `fractions[ 6 ] = 3.333;`
 - g) `float total = 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 The solutions to Exercise 7.5 are as follows:

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

EXERCISES

7.6 Fill in the blanks in each of the following:

- a) Java stores lists of values in _____.
- b) The elements of an array are related by the fact that they _____.
- c) When referring to an array element, the position number contained within brackets is called a _____.

- d) The names of the four elements of array **p** are _____, _____, _____ and _____.
- e) Naming an array, stating its type and specifying the number of dimensions in the array is called _____ the array.
- f) The process of placing the elements of an array into either ascending or descending order is called _____.
- g) In a double-subscripted array, the first subscript identifies the _____ of an element and the second subscript identifies the _____ of an element.
- h) An m-by-n array contains _____ rows, _____ columns and _____ elements.
- i) The name of the element in row 3 and column 5 of array **d** is _____.

7.7 State 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.
- b) An array declaration reserves space for the array.
- c) To indicate that 100 locations should be reserved for integer array **p**, the programmer writes the declaration
`p[ 100 ] ;`
- d) A Java program that initializes the elements of a 15-element array to zero must contain at least one **for** statement.
- e) A Java program that totals the elements of a double-subscripted array must contain nested **for** statements.

7.8 Write Java statements to accomplish each of the following:

- a) Display the value of the seventh element of character array **f**.
- b) Initialize each of the five elements of single-subscripted integer array **g** to 8.
- c) Total the elements of floating-point array **c** of 100 elements.
- d) Copy 11-element array **a** into the first portion of array **b**, containing 34 elements.
- e) Determine and print the smallest and largest values contained in 99-element floating-point array **w**.

7.9 Consider a 2-by-3 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 the names of all the elements in the second row of **t**.
- f) Write the names of all the elements in the third column of **t**.
- g) Write a single statement that sets the element of **t** in row 1 and column 2 to zero.
- h) Write a series of statements that initializes each element of **t** to zero. Do not use a repetition structure.
- i) Write a nested **for** structure that initializes each element of **t** to zero.
- j) Write a nested **for** structure that inputs the values for the elements of **t** from the keyboard.
- k) Write a series of statements that determines and prints the smallest value in array **t**.
- l) Write a statement that displays the elements of the first row of **t**.
- m) Write a statement that totals the elements of the fourth column of **t**.
- n) Write a series of statements that prints the array **t** in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

7.10 Use a single-subscripted 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 applet (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
- g) \$800-\$899
- h) \$900-\$999
- i) \$1000 and over

The applet should use the GUI techniques introduced in Chapter 6. Display the results in a **JTextArea**. Use **JTextArea** method **setText** to update the results after each value input by the user.

7.11 The bubble sort presented in Fig. 7.11 is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort:

- a) After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are “in place”; and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass and so on.
- b) The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass if any swaps have been made. If none have been made, the data must already be in the proper order, so the program should terminate. If swaps have been made, at least one more pass is needed.

7.12 Write statements that perform the following single-subscripted array operations:

- a) Set the 10 elements of integer array **counts** to zeros.
- b) Add 1 to each of the 15 elements of integer array **bonus**.
- c) Print the five values of integer array **bestScores** in column format.

7.13 Use a single-subscripted array to solve the following problem: Write an applet that inputs 20 numbers, each of which is 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 20 numbers are different. Use the smallest possible array to solve this problem. The applet should use the GUI techniques introduced in Chapter 6. Display the results in a **JTextArea**. Use **JTextArea** method **setText** to update the results after each value input by the user.

7.14 Label the elements of 3-by-5 double-subscripted array **sales** to indicate the order in which they are set to zero by the following program segment:

```
for ( int row = 0; row < sales.length; row++ )

    for ( int col = 0; col < sales[ row ].length; col++ )

        sales[ row ][ col ] = 0;
```

7.15 Write an applet to simulate the rolling of two dice. The program should use **Math.random** to roll the first die and should use **Math.random** again to roll the second die. The sum of the two values should then be calculated. [Note: 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 being the least frequent sums. Figure 7.26 shows the 36 possible combinations of the two dice. Your program should

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.26 The 36 possible outcomes of rolling two dice.

roll the dice 36,000 times. Use a single-subscripted array to tally the numbers of times each possible sum appears. Display the results in a **JTextArea** in tabular format. Also, determine whether the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one sixth of all the rolls should be 7). The applet should use the GUI techniques introduced in Chapter 6. Provide a **JButton** to allow the user of the applet to roll the dice another 36,000 times. The applet should reset the elements of the single-subscripted array to 0 before rolling the dice again.

7.16 What does the program of Fig. 7.27 do?

```

1  // Exercise 7.16: WhatDoesThisDo.java
2
3  // Java core packages
4  import java.awt.*;
5
6  // Java extension packages
7  import javax.swing.*;
8
9  public class WhatDoesThisDo extends JApplet {
10     int result;
11
12     public void init()
13     {
14         int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15
16         result = whatIsThis( array, array.length );
17
18         Container container = getContentPane();
19         JTextArea output = new JTextArea();
20         output.setText( "Result is: " + result );
21         container.add( output );
22     }
23
24     public int whatIsThis( int array2[], int size )
25     {
26         if ( size == 1 )
27             return array2[ 0 ];

```

Fig. 7.27 Determine what this program does.

```

28     else
29         return array2[ size - 1 ] +
30             whatIsThis( array2, size - 1 );
31     }
32 }

```

Fig. 7.27 Determine what this program does.

7.17 Write a program that runs 1000 games of craps (Fig. 6.9) and answers the following questions:

- How many games are won on the first roll, second roll, ..., twentieth roll and after the twentieth roll?
- How many games are lost on the first roll, second roll, ..., twentieth roll and after the twentieth roll?
- 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?]
- What is the average length of a game of craps?
- Do the chances of winning improve with the length of the game?

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

Your program should display the following alternatives:

```

Please type 1 for "smoking"
Please type 2 for "nonsmoking"

```

If the person types 1, your program should assign a seat in the smoking section (seats 1-5). If the person types 2, your program should assign a seat in the nonsmoking section (seats 6-10). Your program should then print a boarding pass indicating the person's seat number and whether it is in the smoking or nonsmoking section of the plane.

Use a single-subscripted 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 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 program should, of course, never assign a seat that has already been assigned. When the smoking section is full, your program should ask the person if it is acceptable to be placed in the nonsmoking section (and vice versa). If yes, make the appropriate seat assignment. If no, print the message **"Next flight leaves in 3 hours."**

7.19 What does the program of Fig. 7.28 do?

```

1  // Exercise 7.19: WhatDoesThisDo2.java
2
3  // Java core packages
4  import java.awt.*;
5
6  // Java extension packages
7  import javax.swing.*;
8

```

Fig. 7.28 Determine what this program does.

```

9  public class WhatDoesThisDo2 extends JApplet {
10
11      public void init()
12      {
13          int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14          JTextArea outputArea = new JTextArea();
15
16          someFunction( array, 0, outputArea );
17
18          Container container = getContentPane();
19          container.add( outputArea );
20      }
21
22      public void someFunction( int array2[], int x, JTextArea out )
23      {
24          if ( x < array2.length ) {
25              someFunction( array2, x + 1, out );
26              out.append( array2[ x ] + " " );
27          }
28      }
29  }

```

Fig. 7.28 Determine what this program does.

7.20 Use a double-subscripted 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 different 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 of the slips for last month is available. Write an applet that will read all this information for last month's sales and summarize the total sales by salesperson by product. All totals should be stored in the double-subscripted array **sales**. After processing all the information for last month, display the results in tabular format with each of the columns representing a particular salesperson and each of the rows 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 printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns. Display the results in a **JTextArea**.

7.21 (*Turtle Graphics*) The Logo language, which is popular among young computer users, made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a Java program. 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; 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 as well.

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 program must process are shown in Fig. 7.29.

Command	Meaning
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5,10	Move forward 10 spaces (or a number other than 10)
6	Print the 20-by-20 array
9	End of data (sentinel)

Fig. 7.29 Turtle graphics commands.

Suppose that the turtle is somewhere near the center of the floor. The following “program” would draw and print 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 **1**s. When the **6** command (print) is given, wherever there is a **1** in the array, display an asterisk or some other character you choose. Wherever there is a zero, display a blank. Write a Java applet to implement the turtle graphics capabilities discussed here. The applet should display the turtle graphics in a **JTextArea**, using Monospaced font. 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. The question is this: 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 L-shaped moves (over two in one direction and then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 7.30.

- Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a **1** in the first square you move to, a **2** in the second square, a **3** in the third, etc. 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?

	0	1	2	3	4	5	6	7
0								
1				2		1		
2			3				0	
3					K			
4			4				7	
5				5		6		
6								
7								

Fig. 7.30 The eight possible moves of the knight.

- b) Now let us develop an applet that will move the knight around a chessboard. The board is represented by an 8-by-8 double-subscripted array **board**. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their horizontal and vertical components. For example, a move of type 0 as shown in Fig. 7.30 consists of moving two squares horizontally to the right and one square vertically upward. Move 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 single-subscripted arrays, **horizontal** and **vertical**, as follows:

```

horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1

```

Let the variables **currentRow** and **currentColumn** indicate the row and column of the knight's current position. To make a move of type **moveNumber**, where **moveNumber** is between 0 and 7, your program uses the statements

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

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 already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?

- c) After attempting to write and run a Knight's Tour program, you have probably developed some valuable insights. We will use these to develop a *heuristic* (or strategy) 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 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 they are, then always moving the knight (using the knight's L-shaped moves) to the most inaccessible square. We label a double-subscripted array **accessibility** with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each center square 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
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 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 program 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 program. Did you get a full tour? Modify the program 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 program 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 program should move to the 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 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 increasing 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, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?
- b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a single-subscripted array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?
- c) Most likely, the preceding program gave you some "respectable" tours, but no full tours. Now "pull all the stops out" and simply let your program run until it produces a full tour. (*Caution:* This version of the program could run for hours on a powerful computer.) Once again, keep a table of the number of tours of each length and print this table when the first full tour is found. How many tours did your program 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. Simply stated: 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 program. (*Hint:* It is possible to assign a value to each square of the chessboard indicating 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 in Figure 7.31.) Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: 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) Solve the Eight Queens exercise, using the random brute force technique developed in Exercise 7.23.
- b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard).
- c) Why do you suppose the exhaustive brute force approach may 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, 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 program you wrote in Exercise 7.22 to test for a closed tour if a full tour has occurred.

7.27 (*The Sieve of Eratosthenes*) A prime integer is any integer that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

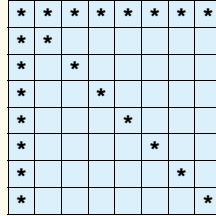


Fig. 7.31 The 22 squares eliminated by placing a queen in the upper left corner.

- Create a primitive type **boolean** array with all elements initialized to **true**. Array elements with prime subscripts will remain **true**. All other array elements will eventually be set to **false**.
- Starting with array subscript 2 (subscript 1 must be prime), determine whether a given element is **true**. If so, loop through the remainder of the array and set to **false** every element whose subscript is a multiple of the subscript for the element with value **true**. Then, continue the process with the next element with value **true**. For array subscript 2, all elements beyond element 2 in the array that have subscripts which are multiples of 2 will be set to **false** (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond element 3 in the array that have subscripts which are multiples of 3 will be set to **false** (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still **true** indicate that the subscript is a prime number. These subscripts can be displayed. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

7.28 (Bucket Sort) A bucket sort begins with a single-subscripted array of positive integers to be sorted and a double-subscripted array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to $n - 1$, where n is the number of values in the array to be sorted. Each row of the double-subscripted array is referred to as a bucket. Write an applet containing a method called **bucketSort** that takes an integer array as an argument and performs as follows:

- Place each value of the single-subscripted array into a row of the bucket array based on the value's ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This is called a "distribution pass."
- Loop through the bucket array row by row and copy the values back to the original array. This is called a "gathering pass." The new order of the preceding values in the single-subscripted array is 100, 3 and 97.
- Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the single-subscripted array is 100, 3 and 97. On the third pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After the last gathering pass, the original array is now in sorted order.

Note that the double-subscripted array of buckets is ten times the size of the integer array being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory. The bubble sort requires space for only one additional element of data. This is an example of the space-time trade-off: The bucket sort uses more memory than the bubble sort, but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second double-subscripted bucket array and repeatedly swap the data between the two bucket arrays.

RECURSION EXERCISES

7.29 (*Selection Sort*) A selection sort searches an array looking for the smallest element in the array, then swaps that element with the first element of the array. The process is repeated for the subarray beginning with the second element. Each pass of the array places one element in its proper location. For an array of n elements, $n - 1$ passes must be made, and for each subarray, $n - 1$ comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write recursive method **selectionSort** to perform this algorithm.

7.30 (*Palindromes*) A palindrome is a string that is spelled the same way forward and backward. Some examples of palindromes are “radar,” “able was i ere i saw elba” and (if blanks are ignored) “a man a plan a canal panama.” Write a recursive method **testPalindrome** that returns **boolean** value **true** if the string stored in the array is a palindrome and **false** otherwise. The method should ignore spaces and punctuation in the string. [Hint: Use **String** method **toCharArray**, which takes no arguments, to get a **char** array containing the characters in the **String**. Then, pass the array to method **testPalindrome**.]

7.31 (*Linear Search*) Modify Figure 7.12 to use recursive method **linearSearch** to perform a linear search of the array. The method should receive an integer array, the array size and the search key as arguments. If the search key is found, return the array subscript; otherwise, return -1 .

7.32 (*Binary Search*) Modify the program of Figure 7.13 to use a recursive method **binarySearch** to perform the binary search of the array. The method should receive an integer array and the starting subscript and ending subscript as arguments. If the search key is found, return the array subscript; otherwise, return -1 .

7.33 (*Eight Queens*) Modify the Eight Queens program you created in Exercise 7.24 to solve the problem recursively.

7.34 (*Print an array*) Write a recursive method **printArray** that takes an array of **int** values and the size of the array as arguments and returns nothing. The method should stop processing and return when it receives an array of size 0.

7.35 (*Print a string backward*) Write a recursive method **stringReverse** that takes a character array containing a string as an argument, prints the string backward and returns nothing.

7.36 (*Find the minimum value in an array*) Write a recursive method **recursiveMinimum** that takes an integer array and the array size as arguments and returns the smallest element of the array. The method should stop processing and return when it receives an array of one element.

7.37 (*Quicksort*) In the examples and exercises of this chapter, we discussed the sorting techniques bubble sort, bucket sort and selection sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

- a) *Partitioning Step*: Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.
- b) *Recursive Step*: Perform step 1 on each unsorted subarray.

Each time step 1 is performed on a subarray, another element is placed in its final location of the sorted array and two unsorted subarrays are created. When a subarray consists of one element, it must be sorted therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

- a) Starting from the rightmost element of the array, compare each element to **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The new array is

12 2 6 4 89 8 10 **37** 68 45

Element 12 is in *italic* to indicate that it was just swapped with **37**.

- b) Starting from the left of the array, but beginning with the element after 12, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The new array is

12 2 6 4 **37** 8 10 89 68 45

- c) Starting from the right, but beginning with the element before 89, compare each element to **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The new array is

12 2 6 4 10 8 **37** 89 68 45

- d) Starting from the left, but beginning with the element after 10, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** to itself we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied on the previous array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive method **quickSort** to sort a single-subscripted integer array. The method should receive as arguments an integer array, a starting subscript and an ending subscript. Method **partition** should be called by **quickSort** to perform the partitioning step.

7.38 (*Maze Traversal*) The following grid of #s and dots (.) is a double-subscripted array representation of a maze.

```
# # # # # # # # # # #
# . . . # . . . . . #
. . # . # . # # # . #
# # # . # . . . # . #
# . . . . # # # . # .
# # # # . # . # . #
# . . # . # . # . #
# # . # . # . # . #
# . . . . . . . # . #
# # # # # . # # # . #
# . . . . . # . . #
# # # # # # # # # #
```

In the preceding double-subscripted array, the #s represent the walls of the maze, and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand

from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

Write recursive method **mazeTraverse** to walk through the maze. The method should receive as arguments a 12-by-12 character array representing the maze and the starting location of the maze. As **mazeTraverse** attempts to locate the exit from the maze, it should place the character **X** in each square in the path. The method should display the maze after each move so the user can watch as the maze is solved.

7.39 (*Generating Mazes Randomly*) Write a method **mazeGenerator** that takes as an argument a double-subscripted 12-by-12 character array and randomly produces a maze. The method should also provide the starting and ending locations of the maze. Try your method **mazeTraverse** from Exercise 7.38, using several randomly generated mazes.

7.40 (*Mazes of Any Size*) Generalize methods **mazeTraverse** and **mazeGenerator** of Exercise 7.38 and Exercise 7.39 to process mazes of any width and height.

7.41 (*Simulation: The Tortoise and the Hare*) In this problem, you will recreate one of the truly great moments in history, namely 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.

There is a clock that ticks once per second. With each tick of the clock, your applet should adjust the position of the animals according to the rules in Fig. 7.32

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the “starting gate”). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in the preceding table 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.

Begin the race by printing

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

Then, for each tick of the clock (i.e., each repetition of a loop), print 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 program should print **OUCH!!!** beginning at that position. All print positions other than the **T**, the **H** or the **OUCH!!!** (in case of a tie) should be blank.

After each line is printed, test for whether either animal has reached or passed square 70. If so, print the winner and terminate the simulation. If the tortoise wins, print **TORTOISE WINS!!! YAY!!!** If the hare wins, print **Hare wins. Yuch.** If both animals win on the same tick of the clock, you may want to favor the turtle (the “underdog”) or you may want to print **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 program, 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.

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

Fig. 7.32 Rules for adjusting the positions of the tortoise and the hare.

SPECIAL SECTION: BUILDING YOUR OWN COMPUTER

In the next several problems, we take a temporary diversion away from the world of high-level language programming. We “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!

7.42 (*Machine-Language Programming*) Let us create a computer we will call the Simpletron. As its name implies, it is a simple machine, but as we will soon see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands, that is, Simpletron Machine Language or SML for short.

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 information in the Simpletron is handled in terms of *words*. A word is a signed four-digit decimal number such as **+3364**, **-1293**, **+0007**, **-0001**, etc. 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 either 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. 7.33.

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.

Operation code	Meaning
<i>Input/output operations:</i>	
<code>final int READ = 10;</code>	Read a word from the keyboard into a specific location in memory.
<code>final int WRITE = 11;</code>	Write a word from a specific location in memory to the screen.
<i>Load/store operations:</i>	
<code>final int LOAD = 20;</code>	Load a word from a specific location in memory into the accumulator.
<code>final int STORE = 21;</code>	Store a word from the accumulator into a specific location in memory.
<i>Arithmetic operations:</i>	
<code>final int ADD = 30;</code>	Add a word from a specific location in memory to the word in the accumulator (leave result in the accumulator).
<code>final int SUBTRACT = 31;</code>	Subtract a word from a specific location in memory from the word in the accumulator (leave result in the accumulator).
<code>final int DIVIDE = 32;</code>	Divide a word from a specific location in memory into the word in the accumulator (leave result in the accumulator).
<code>final int MULTIPLY = 33;</code>	Multiply a word from a specific location in memory by the word in the accumulator (leave result in the accumulator).
<i>Transfer of control operations:</i>	
<code>final int BRANCH = 40;</code>	Branch to a specific location in memory.
<code>final int BRANCHNEG = 41;</code>	Branch to a specific location in memory if the accumulator is negative.
<code>final int BRANCHZERO = 42;</code>	Branch to a specific location in memory if the accumulator is 0.
<code>final int HALT = 43;</code>	Halt—the program has completed its task.

Fig. 7.33 Simpletron Machine Language (SML) operation codes .

The first SML program (Fig. 7.34) reads two numbers from the keyboard and computes and prints 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 prints it (as a signed four-digit decimal number). The *halt* instruction, **+4300**, terminates execution.

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 7.34 SML program that reads two integers and computes their sum.

The second SML program (Fig. 7.35) reads two numbers from the keyboard and determines and prints the larger value. Note the use of the instruction **+4107** as a conditional transfer of control, much the same as Java's **if** statement.

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

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

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

Fig. 7.35 SML program that reads two integers and determines which is larger.

7.43 (*A Computer Simulator*) It may at first seem outrageous, but 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. You will not be disappointed. 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 7.42. Your Simpletron will be an event-driven applet—you will click a button to execute each SML instruction and you will be able to see the instruction “in action.”

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. ***
```

The program should display an **input JTextField** in which the user will type each instruction one at a time and a **Done** button for the user to click when the complete SML program has been entered. Simulate the memory of the Simpletron with a single-subscripted 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. 7.35 (Exercise 7.42):

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

Your program should use a **JTextField** to display the memory location followed by a question mark. Each of the values to the right of a question mark is typed by the user into the **input JTextField**. When the **Done** button is clicked, the program should display:

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

The SML program has now been placed (or loaded) in array **memory**. The Simpletron should provide an “**Execute next instruction**” button the user can click to execute each instruction in your SML program. Execution begins with the instruction in location **00** and, like 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 **operation-**

Code and “pick off” the right two digits and place them in **operand**. Each of the preceding registers should have a corresponding **TextField** in which its current value can be displayed at all times. When Simpletron begins execution, the special registers are all initialized to 0.

Now, let us “walk through” execution of the first SML instruction, **+1009** in memory location **00**. This 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

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

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

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

Now the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, etc.). A **switch** differentiates among the twelve operations of SML.

In the **switch** structure, the behavior of various SML instructions is simulated as shown in Fig. 7.36. We discuss branch instructions shortly and leave the others to the reader.

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* (and, 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. 7.37. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. The sample dump assumes the output will be sent to the display screen with a series of **System.out.print** and **System.out.println** method calls. However, we encourage you to experiment with a version that can be displayed on the applet using a **TextArea** or an array of **TextField** objects.

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 by prompting the user to enter a value into the input dialog, reading the value, converting the value to an integer and storing it in memory location **memory[operand]**. Since your Simpletron is event driven, it waits for the user to type a value into the **input TextField** and press the *Enter* key. 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:

```
++instructionCounter;
```

Instruction	Description
<i>read</i> :	Display an input dialog with the prompt “ Enter an integer. ” Convert the input value to an integer and store it in location memory[operand] .
<i>load</i> :	accumulator = memory[operand];
<i>add</i> :	accumulator += memory[operand];

Fig. 7.36 Behavior of several SML instructions in the Simpletron.

```

REGISTERS:
accumulator      +0000
instructionCounter 00
instructionRegister +0000
operationCode     00
operand          00

MEMORY:
      0      1      2      3      4      5      6      7      8      9
0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

```

Fig. 7.37 A sample dump.

This completes the simulated execution of the first instruction. When the user clicks the **Execute next instruction** button, the entire process (i.e., the instruction execution cycle) begins again with the fetch of the next instruction to be executed.

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 7.42. You may embellish SML with additional features and provide for these 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, accumulator overflows (i.e., arithmetic operations resulting in values larger than **+9999** or smaller than **-9999**) and the like. Such serious errors are called *fatal errors*. When a fatal error is detected, your simulator should print an error message such as:

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

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

7.44 (*Modifications to the Simpletron Simulator*) In Exercise 7.43, 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 Exercise 19.26 and Exercise 19.27, 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 modulus calculations. This requires an additional Simpletron Machine Language instruction.
- c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.
- d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.
- e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language 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 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 print 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 prints the string by translating each two-digit number into its equivalent character.]]

7.45 The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

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 *nonrecursive* method **fibonacci(n)** that calculates the *n*th Fibonacci number. Incorporate this method into an applet that enables the user to enter the value of **n**.
- b) Determine the largest Fibonacci number that can be printed on your system.
- c) Modify the program of part a) to use **double** instead of **int** to calculate and return Fibonacci numbers and use this modified program to repeat part b).