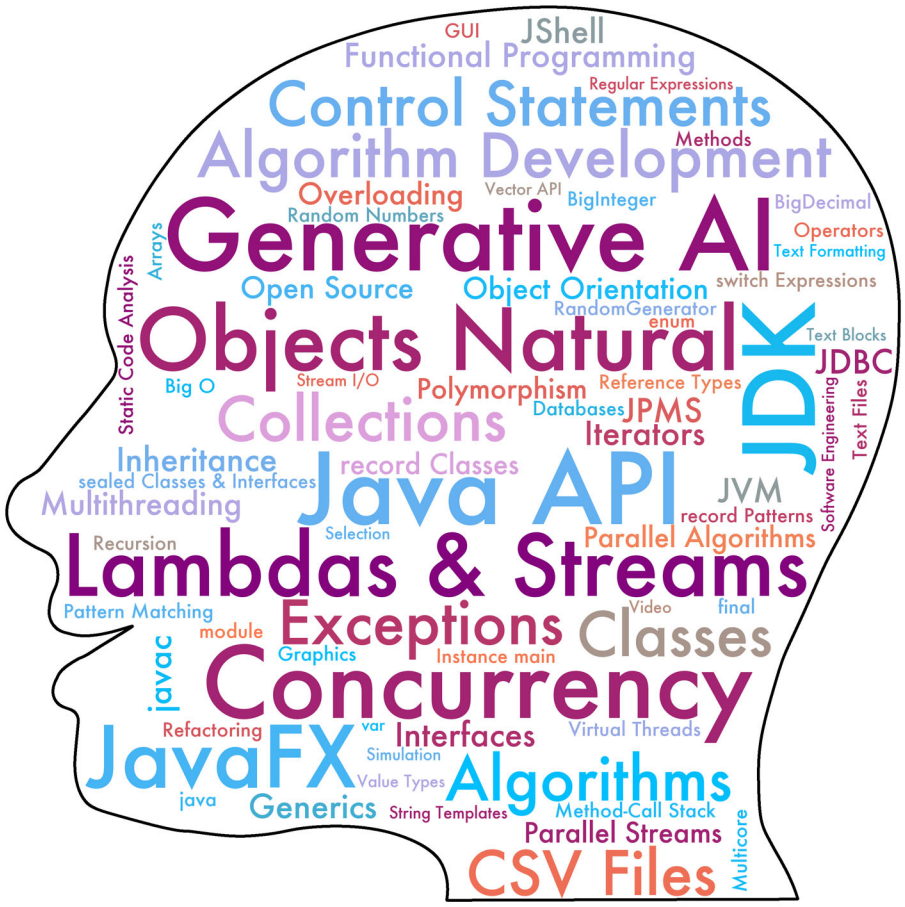
6

Arrays and `ArrayLists`



Objectives

In this chapter, you'll:

- Understand the differences between primitive and reference types.
- Use arrays to store data in and retrieve data from lists and tables of values.
- Declare arrays, initialize arrays and refer to individual elements of arrays.
- Iterate through arrays with the enhanced `for` statement.
- Pass arrays to methods.
- Understand pass-by-value vs. pass-by-reference.
- Declare and manipulate multidimensional arrays.
- Define methods with variable-length argument lists.
- Read command-line arguments into a program.
- Perform predefined array manipulations using the methods of class `Arrays`.
- Use generative AI to leverage your Java learning experience.
- Continue our objects-natural approach with a case study on using the Java API's `ArrayList` class—a dynamically resizable array-like data structure with many capabilities beyond those of built-in Java arrays.

Outline

6.1 Introduction

This chapter introduces **data structures**, which enable you to store **collections** of related data items. **Array** objects are data structures containing data items of the same type and make it convenient to process related groups of values.

We examine the differences between primitive types and reference types and show that arrays can be used to store both. We discuss how to declare, create and initialize arrays, and demonstrate common array manipulations, such as calculating array-element values, totaling array elements, visualizing array data and using array elements as counters.

We introduce Java's exception-handling mechanism, using it to process an error that occurs when a program attempts to access a non-existent array element. We demonstrate the enhanced `for` statement for processing an array's data more conveniently and with fewer potential errors than with the counter-controlled `for` statement.

We introduce two-dimensional arrays for representing tables containing rows and columns of data. We declare and manipulate several two-dimensional arrays, iterating through them using nested counting `for` loops and nested enhanced `for` loops.

We create methods that can be called with varying numbers of arguments and show how to manipulate the `main` method's command-line arguments via its `args` parameter. To avoid "reinventing the wheel" for common array manipulations, we use several methods defined in class `Arrays` (package `java.util`). You'll continue using generative AI to leverage your Java learning experience.

This chapter's Objects-Natural Case Study introduces the `ArrayList` collection class—a reusable, reliable, powerful and efficient data structure that is similar to an array but is **dynamically resizable** to accommodate more or fewer elements.

6.2 Primitive Types vs. Reference Types

Java's types are either primitive types or **reference types**. The eight primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`—you've created variables of

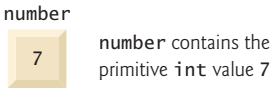
types `int` and `double`. You've also worked with objects of reference types including `Scanner`, `String`, `BigInteger`, `BigDecimal`, `RandomGenerator` and various types from the Java Date/Time API. **All nonprimitive types are reference types.**

Primitive-Type Variables Store Values

A primitive-type variable holds exactly one value of its declared type at a time. Consider the following `int` variable declaration:

```
int number = 7;
```

After this statement executes, `number` contains the value `7`, as in the following diagram:



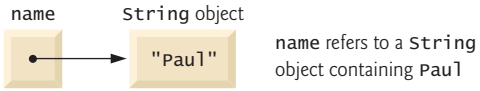
When you assign a new value to `number`, the new value replaces `7`, and the previous value is lost.

Reference-Type Variables Store the Locations of Objects

Reference-type variables store **references** to objects that are located elsewhere in memory. For example, given the following declaration

```
String name = "Paul";
```

the variable `name` refers to a `String` object somewhere in memory containing `"Paul"`, as in the following diagram:



You use the variable `name` to refer to `String` object and call its methods. For example, the following statement calls the `String` object's `length` method and stores the result in the `int` primitive variable `length`:

```
int length = name.length();
```

You can invoke methods via reference-type variables, which refer to objects, but not via primitive-type variables.

Checkpoint

1 *(Fill-In)* Reference-type variables store _____ to objects located elsewhere in memory.
Answer: references.

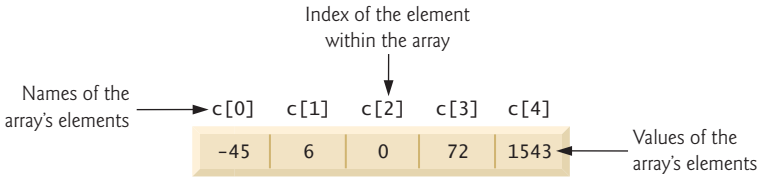
2 *(True/False)* All types in Java are reference types.
Answer: False. All nonprimitive types are reference types. The eight primitive types are not reference types.

6.3 Arrays

An array is a group of variables (called **elements**) containing data of the same type. Arrays are objects, so they're reference types—an array variable contains a reference to an array object in memory. The elements of an array can be primitive types or reference types

(including other arrays, as shown in Section 6.16). To refer to a particular element in an array, we specify the name of the reference to the array and the **index** (that is, position number) of the element in the array.

The following diagram shows a visual representation of an `int` array named `c`:



Array names follow the same conventions as other variable names. This array contains five elements. A program refers to any one of these elements with an **array-access expression** that includes the name of the array followed by the index of the particular element in **square brackets**, `[ ]`—e.g., `c[0]`. The first element in every array has **index 0.** Thus, the elements of the array `c` are `c[0]` (pronounced "`c` sub zero") through `c[4]`. The value of `c[0]` is `-45`, `c[2]` is `0` and `c[4]` is `1543`. An array-access expression can be used on the left side of an assignment to place a new value into an array element. For example, the following statement replaces `c[4]`'s value:

```
c[4] = 87;
```

An index must be a nonnegative integer expression. The highest index (in this case, 4) is always one less than the array's size (in this case, 5). Each array knows its own size (number of elements), which you can access via its **`length` instance variable**, as in

```
c.length
```

An array's length cannot be changed after it's created.

Checkpoint

1 *(Fill-In)* The number used to refer to a particular array element is called the element's _____.

Answer: index.

2 *(True/False)* An array can store many different types of values.

Answer: False. An array can store only values of the same type.

3 *(Code)* Suppose the one-dimensional array `values` contains three elements. The names of those elements are _____, _____ and _____.

Answer: `values[0]`, `values[1]` and `values[2]`.

6.4 Declaring and Creating Arrays

An array object's elements occupy contiguous space in memory, enabling efficient access to the elements by their indices. To create an array object, you use an **array-creation expression** consisting of `new` followed by the array's element type and square brackets specifying the number of elements. Such an expression returns a reference that can be stored in an array variable. The following declaration and array-creation expression create an array object containing five `int` elements and store the array's reference in the array variable named `c`:

```
int[] c = new int[5];
```

When you create an array, each element receives a default value—zero for numeric primitive-type elements, `false` for `boolean` elements and `null` for reference-type elements. The value `null` indicates that a reference-type variable does not currently refer to an object. You'll soon see that you can provide non-default element values when you create an array.

Declaring an Array Variable and Creating an Array in Separate Statements

Creating an array can also be performed in two steps as follows:

```
int[] c; // declare the array variable
c = new int[5]; // create the array; assign to array variable
```

In the declaration, the square brackets following the type indicate that `c` is a variable that will refer to an array—the declaration does not reserve any memory for the array's elements. In the assignment statement, the array variable `c` receives the reference to a new array of five `int` elements.

Creating Multiple Arrays in One Statement

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

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

When the array's element type and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables. In this case, variables `b` and `x` refer to `String` arrays. For readability and clarity, declare only one variable per declaration. The preceding declaration is equivalent to:

```
String[] b = new String[100]; // create array b
String[] x = new String[27]; // create array x
```

Placement of the Square Brackets in Array Variable Declarations

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

```
int[] a, b, c;
```

This declares `a`, `b` and `c` as array variables. Again, placing square brackets directly following the type indicates that all the identifiers in the declaration are array variables. However, if only `a` is intended to be an array variable, and `b` and `c` are intended to be individual `int` variables, then the declaration should be

```
int a[], b, c;
```

In this declaration, `int` distributes to each variable, but the square brackets apply only to the variable `a`. So, `a` can store a reference to an array of `int`s, and `b` and `c` can store individual `int` values. Declaring each variable in a separate variable-declaration statement eliminates this confusion.

Primitive-Type vs. Reference-Type Elements

A program can declare arrays with elements of any type:

- Every element of a primitive-type array is a variable that contains a value of the array's declared element type. For example, every `int` array element is a variable that stores an `int` value.

- Every element of a reference-type array is a variable that can refer to an object of the array's declared element type. For example, every `String` array element is a variable that can refer to a `String` object.

Checkpoint

1 *(True/False)* All array elements are initialized to zero by default.

Answer: False. The default values are zero for numeric primitive-type elements, `false` for `boolean` elements and `null` for reference elements.

2 *(True/False)* An array variable's declaration reserves memory for the array's elements.

Answer: False. You must use an array-creation expression with `new` to reserve memory for the array's elements.

3 *(Code)* How many `String` objects are created by the following statement?

```
String[] colors = new String[7];
```

Answer: No `String` objects are created. The statement creates an array of `String` variables, each initialized to `null` and is capable of eventually holding a reference to a `String` object.

6.5 Creating and Initializing an Array

The program of Fig. 6.1 uses the keyword `new` to create an array of five `int` elements, which are initially zero—the default initial value for `int` array elements. Line 7 declares `array`—a reference-type variable capable of referring to an array of `int` elements—then initializes it with a reference to an array object containing five `int` elements. Line 9 outputs the column headings. The first column contains each array element's index (0–4), and the second column contains each array element's default initial value (0).

```
1  // Fig. 6.1: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // declare variable array and initialize it with an array object
7        int[] array = new int[5]; // create the array object
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; ++counter) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

Fig. 6.1 | Initializing the elements of an array to default values of zero. (Part 1 of 2.)

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

Fig. 6.1 | Initializing the elements of an array to default values of zero. (Part 2 of 2.)

The `for` statement (lines 12–14) outputs the index (represented by `counter`) and value of each array element (represented by `array[counter]`). The control variable `counter` is initially `0` because index values start at 0. Using **zero-based counting** allows the loop to access every element. The `for`'s loop-continuation condition uses the expression `array.length` (line 12) to determine the array's length. In this example, the length is five, so the loop continues executing as long as `counter`'s value is less than 5. A five-element array's highest index value is 4, so using the less-than operator in the loop-continuation condition ensures the loop does not attempt to access an element beyond the end of the array—during the loop's final iteration, `counter` is `4`. We'll soon see what Java does when encountering an out-of-range index at execution time.

Checkpoint

1 *(Fill-In)* Using _____ counting allows a loop to access every element of an array.
Answer: zero-based.

2 *(True/False)* An application that initializes the elements of a 15-element `int` array to zero must contain at least one `for` statement.
Answer: False. Arrays of primitive numeric types are initialized to zero by default when you create them.

3 *(Code)* Find and correct the error(s) in the following code, which should print the 10 elements of the array `values`:

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

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

Answer: The loop-continuation test causes the loop to reference an array element outside the array's bounds (`values[10]`). Change the `<=` operator to `<`.

Generative AI

1 Prompt genAIs with the code in Checkpoint 3 and ask them to find and correct the error(s). Compare the responses with the answer in Checkpoint 3.

6.6 Array Initializers

You can create an array and initialize its elements with an **array initializer**—a comma-separated list of expressions enclosed in braces. In this case, the array length is determined by the number of elements in the array initializer. For example,

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

creates a five-element array with index values 0–4. Element `n[0]` is initialized to 10, `n[1]` to 20, etc. When the compiler encounters an array declaration that includes an array initializer, it counts the number of initializers in the list to determine the array's size and creates a new array of that size for you.

Figure 6.2 initializes an integer array with five values (line 7) and displays the array in tabular format. The code for displaying the array elements (lines 12–14) is identical to that in Fig. 6.1 (lines 12–14).

```
1  // Fig. 6.2: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // array initializer specifies the initial value for each element
7        int[] array = {32, 27, 64, 18, 95};
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; ++counter) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
```

Fig. 6.2 | Initializing the elements of an array with an array initializer.

Rather than using array initializers, as in line 7, array element values are often read into a program. For example, you could input the values from a user or a file on disk, as discussed in Chapter 11, Files and I/O Streams. Reading the data into a program (rather than "hand coding" it into the program) makes the program more reusable because it can be used with different data sets.

Checkpoint

1 *(Fill-In)* You can initialize an array with a(n) _____ in braces ({ and }).
Answer: array initializer.

2 *(Code)* Initialize the five elements of the `int` array `values` to `8`.
Answer: `int values[] = {8, 8, 8, 8, 8};`

3 *(Code)* Write a `for` loop that copies the elements of the `int` array `values` into the `int` array `otherValues`, which has the same number of elements as `values`.
Answer:

```
for (int counter = 0; counter < values.length; ++counter) {
   otherValues[counter] = values[counter];
}
```

Generative AI

1 Prompt genAIs to write the Java code for Checkpoint 3 and compare their responses to Checkpoint 3's answer.

6.7 Calculating Array Element Values

Figure 6.3 creates a five-element array, assigns an even integer from 2 through 10 to each element, then displays the array in tabular format. The `for` statement in lines 10–12 calculates an array element's value by multiplying the current value of the control variable `counter` by 2, then adding 2. Lines 17–19 display each array element's index and value.

```
1  // Fig. 6.3: InitArray.java
2  // Calculating the values to be placed into the elements of an array.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        final int ARRAY_LENGTH = 5; // declare constant
7        int[] array = new int[ARRAY_LENGTH]; // create array
8
9        // calculate value for each array element
10       for (int counter = 0; counter < array.length; ++counter) {
11          array[counter] = 2 + 2 * counter;
12       }
13
14       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
15
16       // output each array element's value
17       for (int counter = 0; counter < array.length; ++counter) {
18          System.out.printf("%5d%8d%n", counter, array[counter]);
19       }
20    }
21 }
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
```

Fig. 6.3 | Calculating the values to be placed into the elements of an array.

In this example, we specified the array's size using the ARRAY_LENGTH constant variable—declared at line 6 with the `final` modifier and initialized to `5`. Constant variables must be initialized before use and cannot be modified thereafter. Attempting to modify a `final` variable after it's initialized causes a compilation error.

Like `enum` constants, constant variables are named constants. They often make programs more readable. A named constant such as ARRAY_LENGTH clearly indicates its purpose, whereas a literal value such as `5` could have different meanings based on its context. Constant variables use all uppercase letters by convention, and multiword names should separate each word from the next with an underscore (_) as in ARRAY_LENGTH.

Checkpoint

1 *(True/False)* Constant variables must be initialized in their declaration.

Answer: False. Constant variables must be initialized before they're used. For example, the following code is allowed if the assignment occurs before ARRAY_SIZE is used:

```
final int ARRAY_SIZE;
ARRAY_SIZE = 5;
```

2 *(Code)* Find and correct the error(s) in the following code:

```
final int ARRAY_SIZE = 5;
ARRAY_SIZE = 10;
```

Answer: Assigning a value to a constant after initializing it is a compilation error. Initialize the constant correctly in its declaration, or declare a separate constant for the value `10`.

Generative AI

1 Prompt genAIs to rewrite the program of Fig. 6.3 to calculate each element's value as the square of that element's index.

6.8 Totaling Array Elements

Array elements often stored values for use in calculations. For example, if an array's elements represent exam grades, a professor may wish to total them and use the sum to calculate the class average. Lines 10–12 in Fig. 6.4 sum the values in the 10-element `int` array `grades` (created and initialized at line 6). In Section 6.13, we'll reimplement this example with the "enhanced `for`" statement, which does not require a counter.

```
1  // Fig. 6.4: SumArray.java
2  // Computing the sum of the elements of an array.
3
4  public class SumArray {
5     public static void main(String[] args) {
6        int[] grades = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
```

Fig. 6.4 | Computing the sum of the elements of an array. (Part 1 of 2.)

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

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

```
Total of array elements: 849
```

Fig. 6.4 | Computing the sum of the elements of an array. (Part 2 of 2.)

Checkpoint

1 *(Code)* Assume the `int` array `values` contains the integers 1–5, and the `int` variable `product` exists and is initialized to 1. Use `product` in a `for` loop that calculates the product of the elements in `values`.

Answer:

```
for (int counter = 0; counter < values.length; ++counter) {
   product *= values[counter];
}
```

Generative AI

1 Prompt genAIs to write Java code that calculates the product of the elements in the array `values` containing the integers 1–100. Ask them to explain the steps they used to create the code in their responses.

6.9 Intro to Visualization: Using a Bar Chart to Display Array Data Graphically

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

Professors often like to review an exam's grade distribution. They might visualize this with a graph showing the number of grades in several categories. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. They include one grade of 100, two grades in the 90s, four in the 80s, two in the 70s, one in the 60s and none below 60. Figure 6.5 stores this grade distribution data in an 11-element array. Each element represents a grade category. For example, `distribution[0]` stores the number of grades in the range 0–9, `distribution[7]` the number of grades in the range 70–79 and `distribution[10]` the number of 100 grades.

```
1  // Fig. 6.5: BarChart.java
2  // Bar chart printing program.
3
4  public class BarChart {
5     public static void main(String[] args) {
6        int[] distribution = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
7
8        System.out.println("Grade distribution:");
9
10       // for each distribution element, output a bar of the chart
11       for (int counter = 0; counter < distribution.length; ++counter) {
12          // output bar label ("00-09: ", ..., "90-99: ", "100: ")
13          if (counter == 10) {
14             System.out.printf("%5d: ", 100);
15          }
16          else {
17             System.out.printf("%02d-%02d: ",
18                counter * 10, counter * 10 + 9);
19          }
20
21          // print bar of asterisks
22          for (int stars = 0; stars < distribution[counter]; ++stars) {
23             System.out.print("*");
24          }
25
26          System.out.println();
27       }
28    }
29 }
```

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

Fig. 6.5 | Bar chart printing program.

The program graphs the array's information as a bar chart with each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 13–19 output a grade range (e.g., `"70-79: "`) based on the current `counter` value. When `counter` is `10`, line 14 outputs `100` with a field width of 5, followed by a colon and a space, to align the label `"100: "` with the other bar labels. In this example, no students received a grade below 60, so `distribution[0]` through `distribution[5]` each contain `0`, and no asterisks are displayed next to the first six grade ranges. In line 17, the format specifier `%02d` indicates that an `int` value should be formatted as a field of two digits. The **0 flag**

in the format specifier displays a leading 0 for values with fewer digits than the field width (2). Appendix B, Formatted Output, discuss format specifiers and format flags.

Generating the Bars

The nested `for` statement (lines 22–24) outputs the bars—the outer loop iterates over the grade ranges, and the nested loop prints the asterisks for each range. Note the loop-continuation condition at line 22 (`stars < distribution[counter]`). When the program reaches the nested `for`, the loop counts from 0 to one less than `distribution[counter]`, thus using a value in `distribution` to determine the number of asterisks to display.

Lines 22–26 in Fig. 6.5 could be replaced with

```
System.out.println("*".repeat(distribution[counter]));
```

Recall that a string literal like `"*"` is a `String` object. The preceding statement calls the **`String` method `repeat`** on the `String` object `"*"` to repeat it the number of times specified by `repeat`'s argument. The method returns a new `String` containing the repeated text.

Checkpoint

1 *(Fill-In)* The _____ flag in a format specifier displays leading 0s for values with fewer digits than the field width.

Answer: 0.

2 *(Code)* Write a statement that displays the following text. Use `String` method `repeat` to create the line of dashes:

```
Arrays
------
```

Answer:

```
System.out.printf("Arrays%n%s%n", "-".repeat(6));
```

Generative AI

1 Prompt genAIs for a list of flags used in Java `System.out.printf` format strings.

6.10 Using Array Elements as Counters

Sometimes, programs use counters to summarize data, such as survey results. In Fig. 5.3, we used separate counters in our die-rolling program to track the frequencies if each face on a six-sided die as the program rolled the die 60,000,000 times. Figure 6.6 presents an array version of Fig. 5.3.

```
1  // Fig. 6.6: RollDie.java
2  // Die-rolling program using arrays instead of switch.
3  import java.util.random.RandomGenerator;
4
5  public class RollDie {
6     public static void main(String[] args) {
7        var randomNumbers = RandomGenerator.getDefault();
```

Fig. 6.6 | Die-rolling program using arrays instead of `switch`. (Part 1 of 2.)

```
8         int[] frequency = new int[7]; // array of frequency counters
9
10        // roll die 60,000,000 times; use die value as frequency index
11        for (int roll = 1; roll <= 60_000_000; ++roll) {
12           ++frequency[randomNumbers.nextInt(1, 7)];
13        }
14
15        System.out.printf("%s%12s%n", "Face", "Frequency");
16
17        // output each array element's value
18        for (int face = 1; face < frequency.length; ++face) {
19           System.out.printf("%4d%12d%n", face, frequency[face]);
20        }
21     }
22  }
```

```
Face   Frequency
   1    10001418
   2     9999022
   3     9996511
   4    10001187
   5    10002132
   6     9999730
```

Fig. 6.6 | Die-rolling program using arrays instead of `switch`. (Part 2 of 2.)

Figure 6.6 uses the array `frequency` (line 8) to count the occurrences of each face of the die. Line 12 of this program replaces lines 19–29 of Fig. 5.3. Line 12 uses the random value to determine which `frequency` element to increment. The calculation in line 12 produces random numbers from 1 to 6, so the array `frequency` must be large enough to store six counters. We chose to use a seven-element array in which we ignore `frequency[0]` because it's clearer to have the face value 1 increment the counter variable `frequency[1]` than `frequency[0]`. Using a seven-element array with indices 0–6 ensures that it has an index for each possible die face (1–6). In line 12, the calculation inside the square brackets evaluates first to determine the array element to increment, then the `++` operator adds one to that element. We also replaced lines 33–35 from Fig. 5.3 by looping through array `frequency` to output the results (lines 18–20). When we study functional programming capabilities in Chapter 17, we'll show how to replace lines 11–13 and 18–20 with a single statement!

Checkpoint

1 *(Code)* Write a program that uses an array to summarize 100 flips of a coin and displays the number of heads and tails.

Answer:

```
1  // Section 6.10, Checkpoint 1: CoinFlipper.java
2  // Summarizing coin flips.
3  import java.util.random.RandomGenerator;
4
```

```
 5  public class CoinFlipper {
 6     public static void main(String[] args) {
 7        RandomGenerator randomNumbers = RandomGenerator.getDefault();
 8
 9        // constants representing heads and tails
10        final int HEADS = 0;
11        final int TAILS = 1;
12
13        // frequency[0] for heads, frequencies[1] for tails
14        int[] frequency = new int[2];
15
16        // simulate 100 coin flips
17        for (int counter = 0; counter < 100; ++counter) {
18           ++frequency[randomNumbers.nextInt(2)];
19        }
20
21        System.out.printf("Number of heads: %d%n", frequency[HEADS]);
22        System.out.printf("Number of tails: %d%n", frequency[TAILS]);
23     }
24  }
```

```
Number of heads: 46
Number of tails: 54
```

Generative AI

1 Prompt genAIs to solve Checkpoint 1 and compare the generated code to Checkpoint 1's answer. If the code is significantly different, ask them to explain their code and the steps used to develop it.

6.11 Analyzing Survey Results

Next, let's use arrays to summarize data collected in a survey. Consider the following problem statement:

> Twenty students were asked to rate the student cafeteria's food quality on a scale of 1 to 5, with 1 being "awful" and 5 being "excellent." Place the responses in an integer array and determine the frequency of each rating.

This is a typical use of arrays. We wish to summarize the responses of each type (that is, 1–5). Array `responses` (Fig. 6.7, lines 7–8) is a 20-element integer array containing the students' survey responses. The array's last value is intentionally an incorrect response (14)—we'll use this value to show that Java validates array index values at runtime.

```
 1  // Fig. 6.7: StudentPoll.java
 2  // Poll analysis program.
 3
 4  public class StudentPoll {
 5     public static void main(String[] args) {
```

Fig. 6.7 | Poll analysis program. (Part 1 of 2.)

```
 6         // student response array (more typically, input at runtime)
 7         int[] responses =
 8            {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 14};
 9         int[] frequency = new int[6]; // array of frequency counters
10
11         // for each answer, select responses element and use that value
12         // as frequency index to determine array element to increment
13         for (int answer = 0; answer < responses.length; ++answer) {
14            try {
15               ++frequency[responses[answer]];
16            }
17            catch (ArrayIndexOutOfBoundsException e) {
18               System.out.println(e); // invokes toString method
19               System.out.printf("   responses[%d] = %d%n%n",
20                  answer, responses[answer]);
21            }
22         }
23
24         System.out.printf("%s%10s%n", "Rating", "Frequency");
25
26         // output each array element's value
27         for (int rating = 1; rating < frequency.length; ++rating) {
28            System.out.printf("%6d%10d%n", rating, frequency[rating]);
29         }
30      }
31   }
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 14 out of bounds for length 6
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

Fig. 6.7 | Poll analysis program. (Part 2 of 2.)

The frequency Array

We use the six-element array `frequency` (line 9) to count the number of occurrences of each response. Each element (except element 0) is used as a counter for one of the possible survey-response values—`frequency[1]` counts the number of students who rated the food as 1, `frequency[2]` counts the number of students who rated the food as 2, and so on.

Summarizing the Results

The `for` statement (lines 13–22) reads the responses from the array `responses` one at a time and increments one of the counters `frequency[1]` to `frequency[5]`; we ignore `frequency[0]` because the survey responses are limited to the range 1–5. The key statement in the loop is line 15. This statement increments the appropriate `frequency` counter as determined by the value of `responses[answer]`.

Let's step through the first few iterations of the `for` statement:

- When `answer` is `0`, `responses[answer]` is the value of `responses[0]` (that is, `1`—line 8). So, `frequency[responses[answer]]` evaluates to `frequency[1]`, and counter `frequency[1]` is incremented by one. To evaluate the expression, we begin with the value in the innermost set of brackets (`answer`, currently `0`). The value of `answer` is plugged into the expression, and the next set of brackets (`responses[answer]`) is evaluated. That value is used as the `frequency` array index to determine which counter to increment (in this case, `frequency[1]`).
- The next time through the loop `answer` is `1`, and `responses[answer]` is the value of `responses[1]` (that is, `2`—line 8). So, `frequency[responses[answer]]` is interpreted as `frequency[2]`, causing `frequency[2]` to be incremented.
- When `answer` is `2`, `responses[answer]` is the value of `responses[2]` (that is, `5`—line 8). So, `frequency[responses[answer]]` is interpreted as `frequency[5]`, causing `frequency[5]` to be incremented, and so on.

Regardless of the number of responses processed, only a six-element array (in which we ignore element zero) is required to summarize the results because all the correct responses are from 1 to 5, and the index values for a six-element array are 0–5.

Processing the Incorrect Response

In the program's output, the `Frequency` column summarizes only 19 of the 20 values in the `responses` array. The `responses` array's last element contains an (intentionally) incorrect response that was not counted. All indices must be greater than or equal to 0 and less than the array's length. The process of validating indices at runtime is called **array bounds checking**. Any attempt to access an element outside that range results in a runtime error known as an **`ArrayIndexOutOfBoundsException`**. The output's first line shows the message associated with that error. The message includes the exception's package and class name, and an error message indicating the index value that was out of bounds and the length of the array. The next section introduces Java's exception-handling mechanism used in lines 14–21 to display the first two lines of this program's output when the `ArrayIndexOutOfBoundsException` occurred. Exception handling enables programs to handle unexpected values gracefully without crashing.

Checkpoint

1 *(Fill-In)* The process of validating array indices at runtime is known as array _____.
Answer: bounds checking.

2 *(True/False)* Any attempt to access an element outside an array's range of valid indices results in a runtime error known as an `ArrayIndexOutOfBoundsException`.
Answer: True.

6.12 Exception Handling

An **exception** indicates a problem that occurs while a program executes. **Exception handling** helps you create **fault-tolerant programs** that can resolve (or handle) exceptions. Sometimes, this allows a program to continue executing as if no problems were encountered. For example, the `StudentPoll` program still displays results (Fig. 6.7), even though

one of the responses was out of range. More severe problems might prevent a program from continuing normal execution, instead requiring it to notify the user of the problem, then terminate. When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it **throws an exception**—that is, an exception occurs. Methods in your own classes can also throw exceptions, as you'll see in Chapter 8.

6.12.1 The `try` Statement

To handle an exception, use a **`try` statement** (lines 14–21 of Fig. 6.7). The **`try` block** (lines 14–16) contains the code that might throw an exception, and the **`catch` block** (lines 17–21) contains the code that handles the exception if one occurs. You can have many `catch`es to handle various exceptions that might be thrown in the corresponding `try` block. When line 15 correctly increments a `frequency` array element, lines 17–21 are skipped. The braces that delimit the `try` and `catch` blocks are required.

6.12.2 The `catch` Block

When the program encounters the invalid value `14` in the `responses` array, it attempts to add `1` to `frequency[14]`, which is outside the array's bounds—the `frequency` array has only six elements (with indexes 0–5). The JVM performs array bounds checking at execution time. When line 15 attempts to access an out-of-bounds element, the JVM throws an `ArrayIndexOutOfBoundsException` to notify the program of this problem. At this point, the `try` block terminates, and the `catch` block begins executing. If you declared local variables in the `try` block, they're now out of scope (and no longer exist), so they're not accessible in the `catch` block. The `catch` block declares an `ArrayIndexOutOfBoundsException` exception parameter (`e`) to handle exceptions of that type. You can use the parameter's identifier to interact with a caught exception object inside the `catch` block.

To prevent `ArrayIndexOutOfBoundsException`s when writing code that accesses array elements, ensure that each array index is greater than or equal to 0 and less than the array's length. For example, the code in Fig. 6.7 could have performed data validation on the values in `responses` to ensure they were valid response values before using them as array indices for the `frequency` array.

Systems in industry that have undergone extensive testing are still likely to contain bugs. Our preference for industrial-strength systems is to catch and deal with runtime exceptions, such as `ArrayIndexOutOfBoundsException`s, to ensure that a system either stays up and running or degrades gracefully and to inform the system's developers of the problem.

`toString` Method of the Exception Parameter

When lines 17–21 of Fig. 6.7 catch the exception, the program displays a message indicating the problem that occurred. Every object has a `String` representation. When you display an object using `System.out`'s `print`, `println` and `printf` methods, they implicitly call the object's `toString` method to get its `String` representation.[1] Line 18 implicitly calls the exception object's `toString` method to get the exception object's error message `String` and display it. Lines 19–20 show the `responses[answer]` value that caused the problem. Once the message is displayed in this example, the exception is considered handled, and the program continues with the statement after the `catch` block's closing brace. In this

1. Section 8.6 discusses `toString` in more detail.

example, the end of the `for` statement is reached (line 22), so the program continues with the increment of the control variable in line 13. We discuss exception handling again in Chapter 8 and more deeply in Chapter 10.

Checkpoint

1 *(Fill-In)* A(n) _____ indicates a problem that occurs while a program executes.
Answer: exception.

2 *(Fill-In)* A(n) _____ block contains code that might throw an exception, and a(n) _____ block contains code that handles an exception.
Answer: `try`, `catch`.

3 *(True/False)* To prevent `ArrayIndexOutOfBoundsExceptions` when accessing an array `data`'s elements, ensure that each index is in the range $0 \leq index \leq$ `data.length`.
Answer: False. To prevent `ArrayIndexOutOfBoundsExceptions`, the index must be greater than or equal to `0` and less than `data.length`.

Generative AI

1 Prompt genAIs with the code in Fig. 6.7. Ask them to rewrite the code to remove the `try` statement but still prevent `ArrayIndexOutOfBoundsExceptions`. Ask them to explain the updated code. Execute the generated code to confirm that it runs correctly and produces the same two-column summary of the response frequencies.

6.13 Enhanced for Statement: Totaling Array Elements

Figure 6.8 reimplements Fig. 6.4, which totaled the integers in an `int` array named `grades`. Lines 10–12 introduce the **enhanced for statement** for iterating through an array's elements without using a counter. This eliminates several error possibilities, such as improperly specifying the control variable's initial value, the loop-continuation test and the increment expression, thus avoiding the possibility of "stepping outside" the array.

```
1  // Fig. 6.8: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest {
5     public static void main(String[] args) {
6        int[] grades = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
9        // add each element's value to total
10       for (int grade : grades) {
11          total += grade;
12       }
13
14       System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

Fig. 6.8 | Using the enhanced `for` statement to total integers in an array. (Part 1 of 2.)

```
Total of array elements: 849
```

Fig. 6.8 | Using the enhanced `for` statement to total integers in an array. (Part 2 of 2.)

The enhanced `for` statement's header (line 10) declares the variable `grade` as an `int`, because the `grades` array contains `int` values. The variable declaration and the array are separated by a required colon (`:`). The loop selects one `int` value from `grades` during each iteration and assigns it to `grade`, which the program processes in the loop's body. This enhanced `for` statement's header can be read as "for each `grade` in `grades`." Line 11 adds the `grade` variable's current value to the `total`. The loop automatically terminates after it processes the `grades` array's last element.

Equivalent Counting for Loop

Lines 10–12 are equivalent to the following counter-controlled iteration used in lines 10–12 of Fig. 6.4 to total the integers in `grades`:

```
for (int counter = 0; counter < grades.length; ++counter) {
   total += grades[counter];
}
```

The enhanced `for` statement hides the counting details from you.

When to Use the Enhanced for Statement

The enhanced `for` statement can be used to read array elements' values,[2] but not modify them.[3] If your program needs to modify elements, use a counter-controlled `for` statement. You should use the enhanced `for` statement when code looping through an array does not need access to each array element's index. For example, totaling the integers in an array requires only the element values—each element's index is irrelevant.

Functional Programming

The `for` statement and the enhanced `for` statement each iterate sequentially from a starting value to an ending value. In Chapter 17, Functional Programming with Lambdas & Streams, you'll see an elegant, more concise and less error-prone means for iterating through arrays and other collections in a manner that enables some iterations to occur in parallel with others to achieve better multi-core system performance.

Checkpoint

1 *(Fill-In)* The _____ allows you to iterate through an array's elements without a counter.
Answer: enhanced `for` statement.

2 *(True/False)* An enhanced `for` statement can attempt to access out-of-bounds array elements.
Answer: False. An enhanced `for` statement stops iterating after processing an array's last element.

2. Section 6.20 will show that the enhanced `for` statement can also be used with other collections of data.
3. Modifying an element of a primitive-type array means changing the element's value. Modifying an element of a reference-type array means making the element to refer to a different object.

3 *(Code)* Write a header for an enhanced `for` statement that iterates through the array `colors` containing `String` elements.
Answer: `for (String color : colors)`

Generative AI

1 Prompt genAIs to rewrite the code in Fig. 6.2 to use an enhanced `for` loop and produce the same output. Compare the generated code's enhanced `for` loop to Fig. 6.2's counter-controlled `for` loop to examine how the generated code determines and displays the array elements' index values.

6.14 Passing Arrays to Methods

This section demonstrates how to pass arrays and individual array elements as arguments to methods. To pass an array argument to a method, use the array name without brackets. For example, if `hourlyTemperatures` is declared as

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

then the method call

```
processTemps(hourlyTemperatures);
```

passes the reference of array `hourlyTemperatures` to method `processTemps`. Recall that every array object "knows" its own length via its `length` instance variable, so when we pass an array object's reference into a method, we need not pass the array's length as an additional argument.

Array Parameters

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

```
void processTemps(double[] temps)
```

indicating that parameter `temps` receives the reference of a `double` array. The method call passes array `hourlyTemperature`'s reference, so when the called method uses the array variable `temps`, it refers to the same array object as `hourlyTemperatures` in the caller.

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

Passing Arrays and Individual Array Elements to Methods

Figure 6.9 demonstrates passing an entire array and passing a primitive-type array element to a method. The `main` method calls `static` methods `modifyArray` and `modifyElement` at lines 18 and 30, respectively. Recall that a `static` method can invoke other `static` methods of the same class without using the class name and a dot (`.`) separator.

```
1  // Fig. 6.9: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray {
5     // main creates array and calls modifyArray and modifyElement
6     public static void main(String[] args) {
7        int[] array = {1, 2, 3, 4, 5};
8
9        System.out.printf(
10          "Effects of passing reference to entire array:%n" +
11          "The values of the original array are:%n");
12
13       // output original array elements
14       for (int value : array) {
15          System.out.printf("   %d", value);
16       }
17
18       modifyArray(array); // pass array reference
19       System.out.printf("%n%nThe values of the modified array are:%n");
20
21       // output modified array elements
22       for (int value : array) {
23          System.out.printf("   %d", value);
24       }
25
26       System.out.printf(
27          "%n%nEffects of passing array element value:%n" +
28          "array[3] before calling modifyElement: %d%n", array[3]);
29
30       modifyElement(array[3]); // attempt to modify array[3]
31       System.out.printf(
32          "array[3] after modifyElement completes: %d%n", array[3]);
33    }
34
35    // multiply each element of an array by 2
36    public static void modifyArray(int[] array2) {
37       for (int counter = 0; counter < array2.length; ++counter) {
38          array2[counter] *= 2;
39       }
40    }
41
42    // multiply argument by 2
43    public static void modifyElement(int element) {
44       element *= 2;
45       System.out.printf(
46          "Value of element in modifyElement: %d%n", element);
47    }
48 }
```

Fig. 6.9 | Passing arrays and individual array elements to methods. (Part 1 of 2.)

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

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

Fig. 6.9 | Passing arrays and individual array elements to methods. (Part 2 of 2.)

The enhanced `for` statement in lines 14–16 outputs `array`'s elements. Line 18 invokes `modifyArray` (lines 36–40), passing `array` as an argument. The method receives a copy of `array`'s reference and uses it to multiply each of `array`'s elements by 2. To show that `array`'s elements were modified, lines 22–24 output the elements again. We could not use the enhanced `for` statement in lines 37–39 because we're modifying `array`'s values.

Next, we demonstrate that when a copy of a primitive-type array element is passed to a method, modifying the copy does not affect the original value of that element in the calling method's array. Lines 26–28 output `array[3]`'s value before invoking `modifyElement`. Remember that the value of this element is now 8 after it was modified during the call to `modifyArray`. Line 30 calls `modifyElement` and passes `array[3]` as an argument. This is just one `int` value (8) in `array`, so the program passes a copy of `array[3]`'s value. Method `modifyElement` (lines 43–47) multiplies its argument by 2, stores the result in its parameter `element`, then outputs `element`'s value (16). Since method parameters, like local variables, cease to exist when the method in which they're declared completes execution, the method parameter `element` is destroyed when `modifyElement` terminates. When the program returns control to `main`, lines 31–32 output the unmodified `array[3]` value (i.e., 8).

Checkpoint

1 *(True/False)* When passing an array to a method, you must also pass the array's length.

Answer: False. Arrays know their own size, so you do not need to pass the array's length.

2 *(Code)* Write a method named `displayStrings` that `main` can call to display an array of `String`s in two-column format with the headings `Index` and `Value`.

Answer:

```
public static void displayStrings(String[] array) {
   System.out.printf("%5s  %s%n", "Index", "Value");
   int index = 0;

   for (String value : array) {
      System.out.printf("%5d  %s%n", index, value);
      ++index;
   }
}
```

Generative AI

1 Prompt genAIs to create a Java method named `sum` that `main` can call to calculate and return the sum of an `int` array's elements, then test the method's code in a program. Execute the generated code to ensure that it works.

6.15 Pass-By-Value vs. Pass-By-Reference

The preceding example demonstrated passing arrays and primitive-type array elements as arguments to methods. Let's take a closer look at the two ways to pass arguments to methods in many programming languages—**pass-by-value** and **pass-by-reference**.

When an argument is passed by value, a copy of the argument's value is passed to the called method. The called method works exclusively with the copy. Changes to the called method's copy do not affect the original variable's value in the caller.

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

Unlike some other languages, Java passes all arguments to methods by value, meaning it passes a copy of each argument. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type `int` and `double`) and copies of references to objects. Objects themselves cannot be passed to methods.

When a method modifies a primitive-type parameter, changes to the parameter have no effect on the original argument value in the calling method. For example, in Fig. 6.9, when line 30 in `main` passed `array[3]` to method `modifyElement`, the statement in line 44 that multiplied the parameter `element` by `2` did not affect `array[3]`'s value in `main`. This is also true for reference-type parameters. If you modify a reference-type parameter so it refers to another object, only the method's parameter refers to the new object—the reference stored in the caller's variable still refers to the original object.

When passing reference-type variables, the method can modify the referenced object by calling its methods but cannot change the reference itself. Since the reference stored in the parameter is a copy of the reference passed as an argument, the parameter in the called method and the argument in the calling method refer to the same object in memory. For example, in Fig. 6.9, parameter `array2` in method `modifyArray` and variable `array` in `main` refer to the same array object in memory. Any changes made using the parameter `array2` are carried out on the object that `array` references in the calling method. In Fig. 6.9, the changes made in `modifyArray` using `array2` affect the contents of the array object referenced by `array` in `main`. Thus, the called method can use the reference it receives to manipulate the caller's object directly.

Passing references to arrays instead of the array objects themselves makes sense for performance reasons. Because arguments are passed by value, if array objects were passed, a copy of each element would be passed. For large arrays, this would waste time and consume considerable storage for the element copies.

Checkpoint

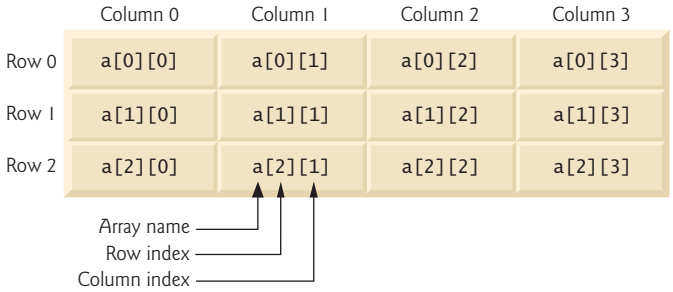
1 *(True/False)* If you modify a reference-type parameter in a called method so it refers to another object, the caller's variable will refer to that other object when the method returns.

Answer: False. If you modify a reference-type parameter so that it refers to another object, only the parameter refers to the new object. The reference stored in the caller's variable still refers to the original object.

2 *(True/False)* When passing a reference-type variable to a method, the method can modify the referenced object by calling the object's `public` methods.
Answer: True.

6.16 Multidimensional Arrays

Multidimensional arrays with two dimensions often represent tables of values with data arranged in rows and columns. To identify a particular table element, you specify two indices. By convention, the first identifies the element's row and the second its column. Arrays that require two indices to identify each element are called **two-dimensional arrays**. Multidimensional arrays can have two or more dimensions. Java's multidimensional arrays are actually one-dimensional arrays whose elements refer to one-dimensional arrays. The following diagram illustrates a two-dimensional array named `a` with three rows and four columns (i.e., a three-by-four array). An array with *m* rows and *n* columns is called an ***m*-by-*n* array**.



Every element in array `a` is identified by an array-access expression of the form `a[`*row*`][`*column*`]` in which `a` is the array's name, and *row* and *column* are indices that uniquely identify each element by row index and column index. The element names in row 0 all have a first index of `0`, and the element names in column 3 all have a second index of `3`.

Arrays of One-Dimensional Arrays

Like one-dimensional arrays, multidimensional arrays can be initialized with array initializers in declarations. A two-dimensional array `b` with two rows and two columns could be declared and initialized with **nested array initializers** as follows:

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

The initial values are grouped by row in braces. So `1` and `2` initialize `b[0][0]` and `b[0][1]`, respectively, and `3` and `4` initialize `b[1][0]` and `b[1][1]`, respectively. The compiler counts the number of nested array initializers (represented by sets of braces within the outer braces) to determine the number of rows in array `b`. The compiler counts the initializer values in a row's nested array initializer to determine its number of columns. As we'll see, this means that rows can have different lengths, enabling "jagged arrays."

Multidimensional arrays are maintained as arrays of one-dimensional arrays. Therefore, array `b`'s rows in the preceding declaration are one-dimensional arrays—one contain-

ing the values in the first nested array initializer `{1, 2}` and one containing the values in the second nested array initializer `{3, 4}`. So, array `b` is a one-dimensional array of two elements, each of which refers to a one-dimensional array of two `int` values.

Two-Dimensional Arrays with Rows of Different Lengths

The manner in which multidimensional arrays are represented makes them quite flexible. In fact, the rows in an array are not required to be the same length. For example,

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

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

Creating Two-Dimensional Arrays with Array-Creation Expressions

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

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

The elements of a multidimensional array are initialized when the array object is created.

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

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

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

Two-Dimensional Array Example: Displaying Element Values

Figure 6.10 demonstrates initializing two-dimensional arrays with array initializers and using nested `for` loops to **traverse the arrays** (i.e., manipulate every element of each array). Class `InitArray`'s `main` declares two arrays. The declaration of `array1` (line 7) uses nested array initializers of the same length to initialize the first row to the values 1, 2 and 3, and the second row to the values 4, 5 and 6. The declaration of `array2` (line 8) uses nested initializers of different lengths—the first row is initialized to two elements with the values 1 and 2; the second row is initialized to one element with the value 3; and the third row is initialized to three elements with the values 4, 5 and 6.

```
1  // Fig. 6.10: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray {
5     // create and output two-dimensional arrays
6     public static void main(String[] args) {
```

Fig. 6.10 | Initializing two-dimensional arrays. (Part 1 of 2.)

```
7         int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
8         int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
9
10        System.out.println("Values in array1 by row are");
11        outputArray(array1); // displays array1 by row
12
13        System.out.printf("%nValues in array2 by row are%n");
14        outputArray(array2); // displays array2 by row
15     }
16
17     // output rows and columns of a two-dimensional array
18     public static void outputArray(int[][] array) {
19        // loop through array's rows
20        for (int row = 0; row < array.length; ++row) {
21           // loop through columns of current row
22           for (int column = 0; column < array[row].length; ++column) {
23              System.out.printf("%d  ", array[row][column]);
24           }
25
26           System.out.println();
27        }
28     }
29  }
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Fig. 6.10 | Initializing two-dimensional arrays. (Part 2 of 2.)

Lines 11 and 14 call method `outputArray` (lines 18–28) to output the elements of `array1` and `array2`, respectively. Method `outputArray`'s parameter—`int[][] array`—indicates that the method receives a two-dimensional array. The nested `for` statement (lines 20–27) outputs the rows of a two-dimensional array. In the loop-continuation condition of the outer `for` statement (line 20), the expression `array.length` determines the number of rows in the array. In the inner `for` statement, the line 22 expression `array[row].length` determines the number of columns in the current row of the array. The inner `for` statement's condition enables the loop to determine the number of columns in each row.

Nested Enhanced for Loops

We used counter-controlled loops in this example to demonstrate array access expressions with two sets of square brackets (e.g., `array[row][column]` in line 23). You can also implement lines 20–27 using enhanced `for` statements:

```
// loop through array's rows using enhanced for loop
for (int[] row : array) {
   // loop through columns of current row using enhanced for loop
   for (int element : row) {
      System.out.printf("%d  ", element);
   }

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

In the outer loop, the variable `row` is a one-dimensional `int` array representing the current row of the two-dimensional `array` object—you can read this `for` header as, "for each `row` in `array`." In the inner loop, the variable `element` is an `int` representing an element from the current `row`—you can read this `for` header as, "for each `element` in the current `row`." When the inner loop finishes processing the current `row`, the outer loop outputs a newline then processes the next `row` (if any).

Common Multidimensional-Array Manipulations

Let's consider several other common manipulations using the three-row and four-column array named `a` from the diagram at the beginning of Section 6.16. The following `for` statement sets all the elements in row `2` of array `a` to zero:

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

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

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

The following nested enhanced `for` statement totals the values of all the elements in array `a`:

```
int total = 0;
for (int[] row : a) {
   for (int element : row) {
      total += element;
   }
}
```

These nested `for` statements total the array elements one row at a time:

- The outer `for` statement initially selects row 0, so the first row's elements can be totaled by the inner `for` statement.
- Next, the outer `for` selects row `1` so the second row can be totaled.
- Then, the outer `for` selects row `2` so the third row can be totaled.

The variable `total` can be displayed when the outer `for` statement terminates.

Checkpoint

1 *(Code)* Assume the `int` array `table` has three rows and two columns. Use a `for` statement to assign to each element the sum of its row and column indices. Use the integer variables `row` and `column` as control variables.
Answer:

```
for (int row = 0; row < table.length; ++row) {
   for (int column = 0; column < table[row].length; ++column) {
      table[row][column] = row + column;
   }
}
```

2 *(Code)* Using the array `table` that was initialized in Checkpoint 1, write array-access expressions that show the order in which `table`'s elements were initialized.
Answer: `table[0][0]`, `table[0][1]`, `table[1][0]`, `table[1][1]`, `table[2][0]`, `table[2][1]`.

3 *(Code)* Write nested counting `for` loops to determine and display the smallest value in the two-dimensional array `values`, which contains elements of type `double`.
Answer:

```
double smallest = values[0][0];

for (int row = 0; row < values.length; ++row) {
   for (int column = 0; column < values[row].length; ++column) {
      if (values[row][column] < smallest) {
         smallest = values[row][column];
      }
   }
}

System.out.printf("Smallest is %f%n", smallest);
```

Generative AI

1 Prompt genAIs to write nested `for` loops that determine and display the largest value in the two-dimensional array of `double`s named `values`. Do the genAIs use nested counting `for` loops or nested enhanced `for` loops?

6.17 Variable-Length Argument Lists

With **variable-length argument lists**, you can create methods that receive an unspecified number of arguments. In a method's parameter list, a type followed by an **ellipsis (`...`)** indicates the method can receive a variable number of arguments of that type. This use of the ellipsis may occur only once in a parameter list. Such a parameter must be placed at the end of the parameter list. While you can use method overloading and array passing to accomplish much of what variable-length argument lists do, using an ellipsis in a method's parameter list is more concise.

Figure 6.11 demonstrates the method `average` (lines 6–15), which receives a variable-length sequence of `double`s. Java treats the variable-length argument list as an array. So, the method body manipulates the parameter `numbers` as an array of `double`s. Lines 10–12 use the enhanced `for` loop to calculate the total of the `double`s in the array. Line 14 uses `numbers.length` to obtain the size of the `numbers` array for use in the averaging calcula-

tion. Lines 27, 29 and 31 in `main` call method `average` with two, three and four arguments, respectively. Method `average` has a variable-length argument list (line 6), so it can average as many `double` arguments as the caller passes. The output shows that each call to the method `average` returns the correct value.

```
1  // Fig. 6.11: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest {
5     // calculate average
6     public static double average(double... numbers) {
7        double total = 0.0;
8
9        // calculate total using the enhanced for statement
10       for (double d : numbers) {
11          total += d;
12       }
13
14       return total / numbers.length;
15    }
16
17    public static void main(String[] args) {
18       double d1 = 10.0;
19       double d2 = 20.0;
20       double d3 = 30.0;
21       double d4 = 40.0;
22
23       System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
24          d1, d2, d3, d4);
25
26       System.out.printf("Average of d1 and d2 is %.1f%n",
27          average(d1, d2));
28       System.out.printf("Average of d1, d2 and d3 is %.1f%n",
29          average(d1, d2, d3));
30       System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
31          average(d1, d2, d3, d4));
32    }
33 }
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 6.11 | Using variable-length argument lists.

Checkpoint

1 *(Fill-In)* A(n) _____ in a method's parameter list indicates that the method can receive a variable number of arguments.
Answer: ellipsis (...).

2 *(True/False)* A parameter for a variable-length argument list can appear anywhere in a method's parameter list.
Answer: False. Such a parameter must appear last.

Generative AI

1 If you call Fig. 6.11's `average` method with no arguments it would attempt to calculate `0.0 / 0`, which does not have a numeric representation. Prompt genAIs with Fig. 6.11's `average` method and ask them to modify the Java code so `average` requires at least one argument—calling it with no arguments should result in a compilation error. Call the generated `average` method with no arguments to confirm that the compiler generates an error.

6.18 Command-Line Arguments

You can pass arguments from the command line to a program via method `main`'s `String[]` parameter, which is named `args` by convention. Typical uses of command-line arguments include passing configuration options, data, filenames and folder paths to programs. When you execute a program with the `java` command, the JVM passes the command-line arguments (that appear after the class name) to the program's `main` method as `String`s in the array `args`. The array's `length` attribute stores the number of command-line arguments.

In Fig. 6.12, we use command-line arguments to determine an array's size, its first element value and the increment for calculating its remaining element values. The command

```
java InitArray 5 0 4
```

passes three arguments—`5`, `0` and `4`—to the `InitArray` program. Command-line arguments are separated by whitespace. When this command executes, `InitArray`'s `main` method's `args` parameter receives a three-element array (`args.length` is `3`) in which:

- `args[0]` contains the `String` `"5"`,
- `args[1]` contains the `String` `"0"` and
- `args[2]` contains the `String` `"4"`.

The program determines how to use these arguments. For this example, we convert them to `int` values and use them to initialize an array. When the program executes, if `args.length` is not `3`, the program prints an error message and terminates by returning from `main` immediately (lines 7–12). Otherwise, lines 15–32 initialize and display the array based on the command-line arguments.

```
1  // Fig. 6.12: InitArray.java
2  // Initializing an array using command-line arguments.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // check number of command-line arguments
7        if (args.length != 3) {
```

Fig. 6.12 | Initializing an array using command-line arguments. (Part 1 of 2.)

```
            System.out.println("""
               Error: Please re-execute the program and include
               an array size, initial value and increment.""");
            return;
         }

         // get array size from first command-line argument
         int arrayLength = Integer.parseInt(args[0]);
         int[] array = new int[arrayLength];

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

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

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

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

```
> java InitArray
Error: Please re-execute the program and include
an array size, initial value and increment.""");
```

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

```
> java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Fig. 6.12 | Initializing an array using command-line arguments. (Part 2 of 2.)

Line 15 gets `args[0]`—a `String` that specifies the array size—and converts it to an `int` value that the program uses to create the array in line 16. The **`Integer` class `static` method `parseInt`** converts its `String` argument to an `int`. This method can fail—in Generative AI question 1, you'll prompt genAIs to deal with this possibility.

Lines 19–20 convert the `args[1]` and `args[2]` command-line arguments to `int` values and store them in `initialValue` and `increment`, respectively. Lines 23–25 calculate each array element's value. Lines 27–32 display the array's contents.

In the three sample executions:

- The first shows that the program received an insufficient number of command-line arguments.
- The second uses the command-line arguments 5, 0 and 4 to specify the array's size (5), the first element's value (0) and the increment for calculating each subsequent element's value (4). The output shows that these values create an array containing 0, 4, 8, 12 and 16.
- The third uses the command-line arguments 8, 1 and 2 to produce an array whose 8 elements are the nonnegative odd integers from 1 to 15.

Checkpoint

1 *(Fill-In)* Command-line arguments are stored in a(n) _____.
Answer: array of `String`s (called `args` by convention).

2 *(Code)* Assume that command-line arguments are stored in the `String` array `args`. Write an expression that determines the total number of command-line arguments.
Answer: `args.length`.

3 *(Code)* Write a program that receives integers as command-line arguments and computes and displays their sum.
Answer:

```
1  // Section 6.18, Checkpoint 3: Sum.java
2  // Totaling command-line arguments.
3  public class Sum {
4     public static void main(String[] args) {
5        // check if any command-line arguments were provided
6        if (args.length == 0) {
7           System.out.println(
8              "Provide some integer command-line arguments.");
9           return;
10       }
11
12       int sum = 0;
13
14       // compute the sum of the command-line arguments
15       for (String arg : args) {
16          sum += Integer.parseInt(arg);
17       }
18
19       System.out.printf("Sum is: %d%n", sum); // display the sum
20    }
21 }
```

```
java Sum 10 20 30
Sum is 60
```

Generative AI

1 Prompt genAIs with Fig. 6.12's code and ask them to handle the exception that would occur if the user enters a non-integer command-line argument.

6.19 Class Arrays

The **`Arrays` class** helps you avoid reinventing the wheel by providing `static` methods for common array manipulations, such as:

- **`sort`** for sorting an array—that is, arranging elements into ascending or descending order,
- **`binarySearch`** for searching a sorted array—that is, determining whether an array contains a specific value and, if so, where the value is located,
- **`equals`** for determining whether two arrays have the same contents,
- **`fill`** for filling an array's elements with the same value, and
- **`toString`** for creating a `String` representation of an array's contents without having to iterate through the elements using a loop.

These and many other `Arrays` methods are overloaded for arrays of primitive-type and reference-type elements. Chapter 23, Computer Science Thinking: Recursion, Searching, Sorting and Big O, shows how to implement searching and sorting algorithms, a subject of great interest to computer-science researchers and students, as well as developers of high-performance systems. Figure 6.13 uses `Arrays` methods `sort`, `binarySearch`, `equals`, `fill` and `toString`, and shows how to copy arrays with the **`System` class's `static` `arraycopy` method**.

```
1  // Fig. 6.13: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations {
6     public static void main(String[] args) {
7        // sort doubles into ascending order
8        double[] doubles = {8.4, 9.3, 0.2, 7.9, 3.4};
9        Arrays.sort(doubles);
10       System.out.printf("doubles: %s%n", Arrays.toString(doubles));
11
12       // fill 10-element array with 7s
13       int[] filledInts = new int[10];
14       Arrays.fill(filledInts, 7);
15       System.out.printf("filledInts: %s%n", Arrays.toString(filledInts));
16
```

Fig. 6.13 | `Arrays` class methods and `System.arraycopy`. (Part 1 of 2.)

```
17         // copy array ints into array intsCopy
18         int[] ints = {1, 2, 3, 4, 5, 6};
19         int[] intsCopy = new int[ints.length];
20         System.arraycopy(ints, 0, intsCopy, 0, ints.length);
21         System.out.printf("ints: %s%n", Arrays.toString(ints));
22         System.out.printf("intsCopy: %s%n", Arrays.toString(intsCopy));
23
24         // compare ints and intsCopy for equality
25         boolean b = Arrays.equals(ints, intsCopy);
26         System.out.printf("%nints %s intsCopy%n",
27             (b ? "equals" : "does not equal"));
28
29         // compare ints and filledInts for equality
30         b = Arrays.equals(ints, filledInts);
31         System.out.printf("ints %s filledInts%n",
32             (b ? "equals" : "does not equal"));
33
34         // search ints for the value 5
35         int location = Arrays.binarySearch(ints, 5);
36
37         if (location >= 0) {
38            System.out.printf("Found 5 at element %d in ints%n", location);
39         }
40         else {
41            System.out.println("5 not found in ints");
42         }
43
44         // search intArray for the value 8763
45         location = Arrays.binarySearch(ints, 8763);
46
47         if (location >= 0) {
48            System.out.printf(
49               "Found 8763 at element %d in ints%n", location);
50         }
51         else {
52            System.out.println("8763 not found in ints");
53         }
54      }
55   }
```

```
doubles: [0.2, 3.4, 7.9, 8.4, 9.3]
filledInts: [7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
ints: [1, 2, 3, 4, 5, 6]
intsCopy: [1, 2, 3, 4, 5, 6]

ints equals intsCopy
ints does not equal filledInts
Found 5 at element 4 in ints
8763 not found in ints
```

Fig. 6.13 | `Arrays` class methods and `System.arraycopy`. (Part 2 of 2.)

Sorting an Array's Elements

In `main`, line 9 sorts the elements of array `doubles`. The `static` method `sort` of class `Arrays` orders the array's elements in ascending order by default. Overloaded versions of `sort` allow you to sort a specific range of elements within the array. Line 10 displays the sorted array. **`Arrays` `static` method `toString`** returns a string representation of an array's elements in a comma-separated list delimited by square brackets.

Filling an Array's Elements with a Specified Value

Line 14 calls `static` method `fill` of class `Arrays` to populate the 10 elements of array `filledInts` with `7`s. Overloaded versions of `fill` allow you to populate a specific range of elements with the same value. Line 15 displays the filled array.

Copying Arrays

Line 20 copies the elements of `ints` into `intsCopy`. The first argument (`ints`) passed to `System` method `arraycopy` is the array from which elements are to be copied. The second argument (`0`) is the index specifying the starting point in the range of elements to copy from the array. This value can be any valid array index. The third argument (`intsCopy`) specifies the destination array that will store the copy. The fourth argument (`0`) specifies the index in the destination array where the first copied element should be stored. The last argument specifies the number of elements to copy from the array in the first argument. In this case, we copy all the elements in the array.

Comparing Arrays

Lines 25 and 30 call `static` method `equals` of class `Arrays` to determine whether all the elements of two arrays are equivalent. If the arrays contain the same elements in the same order, the method returns `true` (line 25); otherwise, it returns `false` (line 30). When comparing array contents, always use `Arrays.equals(array1, array2)`, which compares the two arrays' contents, rather than `array1.equals(array2)`, which compares whether `array1` and `array2` refer to the same array object.

Searching Arrays for a Specified Value

Lines 35 and 45 call `Arrays` `static` method `binarySearch` to perform a binary search on `ints`, using the second argument (`5` and `8763`, respectively). The value you are searching for is known as the **key** or **search key**. If the key is found (line 35), `binarySearch` returns the element's index; otherwise, `binarySearch` returns a negative value (line 45). Passing an unsorted array to `binarySearch` is a logic error—the value returned is undefined. Chapter 23 discusses binary searching in detail.

Class `Arrays` Method `parallelSort`

The `Arrays` class has several "parallel" methods that can use multi-core hardware to significantly improve performance. `Arrays` method `parallelSort` can sort large arrays more efficiently on multi-core systems. In Section 19.12, we create a 100,000,000-element array and use features of the Date/Time API to compare how long it takes to sort the array with `sort` and `parallelSort`.

Checkpoint

1 *(Fill-In)* Arrays `static` method _____ arranges array elements into ascending order.
Answer: `sort`.

2 *(True/False)* `Arrays` `static` method `toString` converts an array to a `String` representation delimited by square brackets and with commas separating the elements.
Answer: True.

3 *(Research)* Investigate class `Arrays` in the Java SE API documentation to learn about other methods the class provides for common array manipulations.

Generative AI

1 Prompt genAIs to write a tutorial for programming novices showing how to use the top methods of Java class `Arrays`.

6.20 Objects-Natural Case Study: Intro to Collections and Class `ArrayList`

We now continue our objects-natural approach with a case study on creating and manipulating objects of the **Java Collection Framework's `ArrayList<E>` collection class**. The Java API provides many predefined collections (that is, data structures) that store groups of related objects in memory. These classes provide efficient, proven methods that organize, store and retrieve your data portably without requiring knowledge of how the data is being stored. This reduces program development time and helps create more robust applications. Chapter 12, Generic Collections, covers the Java Collection Framework in detail.

You've used arrays to store sequences of objects. Arrays are fixed in size, so they cannot automatically change their size at execution time to accommodate additional or fewer elements. The collection class `ArrayList<E>` (package `java.util`) provides a convenient solution to this problem. The `E` is a placeholder for the `ArrayList`'s element type. When declaring a new `ArrayList`, you replace `E` with the element type. For example,

```
ArrayList<String> list;
```

declares `list` as an `ArrayList` collection that can store only `String`s. Once the collection's element type is specified, the compiler ensures all operations using that collection are **type-safe**. For example, the compiler ensures that all elements you add to an `ArrayList<String>` are `String`s; otherwise, compilation errors occur. As with arrays, collection indices start at zero. In fact, `ArrayList` is implemented using a conventional array behind the scenes. The following table shows some key methods of class `ArrayList<E>`:

Method	Description
`add`	Overloaded to add an element to the end of an `ArrayList` or at a specific index in an `ArrayList`.
`addFirst`	Adds an element to the beginning of an `ArrayList` .
`addLast`	Adds an element to the end of an `ArrayList` .
`clear`	Removes all the elements from an `ArrayList`.
`contains`	Returns a `boolean` indicating whether an `ArrayList` contains the specified value.
`get`	Returns the element at the specified index.
`getFirst`	Returns the first element of an `ArrayList` .

Method (Cont.)	Description
`getLast`	Returns the last element of an `ArrayList` .
`indexOf`	Returns the index of the first occurrence of the specified value in an `ArrayList`.
`remove`	Overloaded to remove the specified value's first occurrence or the element at the specified index.
`removeFirst`	Removes the first element of an `ArrayList`
`removeLast`	Removes the last element of an `ArrayList`
`size`	Returns the number of elements stored in an `ArrayList`.

Boxing and Unboxing

Classes with type placeholders that can be used with any type are called **generic classes**. Only reference types can be used to declare variables and create objects of generic classes. Java provides a mechanism—known as **boxing**—that allows primitive values to be wrapped as objects for use with generic classes. For example, an `ArrayList<Integer>` can store only `Integer` objects. When you place an `int` value into an `ArrayList<Integer>`, the `int` value is boxed (wrapped) as an `Integer` object, and when you get an `Integer` object from an `ArrayList<Integer>` then assign the object to an `int` variable, the `int` value inside the object is **unboxed** (unwrapped). This boxing/unboxing process is performed transparently to the programmer.

Demonstrating an `ArrayList<String>`

Figure 6.14 demonstrates some common `ArrayList` capabilities. Line 8 creates a new empty `ArrayList` of `String`s with an initial capacity of three elements. The capacity indicates how many items the `ArrayList` can hold without growing. When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array. This operation is time-consuming. It would be inefficient for the `ArrayList` to grow by one element each time an element is added. Instead, it grows by approximately 50% only when an element is added and the number of elements is equal to the capacity—i.e., there's no space for the new element. If you do not pass a capacity when you create the `ArrayList`, the default initial capacity is 10. We chose the initial capacity 3 (line 8) in this example to show that the `ArrayList` grows once we attempt to add an element beyond its initial capacity.

```
1  // Fig. 6.14: ArrayListCollection.java
2  // Generic ArrayList<E> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection {
6     public static void main(String[] args) {
7        // create a new ArrayList of Strings with an initial capacity of 3
8        var items = new ArrayList<String>(3);
9        System.out.printf(
10          "After creating list with initial capacity of 3, size is: %d%n",
11          items.size());
12
```

Fig. 6.14 | Generic `ArrayList<E>` collection demonstration. (Part 1 of 3.)

```
13         items.add("red"); // append an item to the list
14         items.add(0, "yellow"); // insert "yellow" at index 0
15
16         System.out.print(
17            "Display list contents with counter-controlled for loop:");
18
19         // display the colors in the list
20         for (int i = 0; i < items.size(); ++i) {
21            System.out.printf(" %s", items.get(i));
22         }
23
24         // display colors using enhanced for in the display method
25         display(items,
26            "%nDisplay list contents with enhanced for loop:");
27         System.out.printf("list size: %d%n", items.size());
28
29         items.add("green"); // add "green" to the end of the list
30         items.add("yellow"); // add "yellow" to the end of the list
31         display(items, "\nList with two new elements:");
32         System.out.println("List has grown beyond initial capacity of 3");
33         System.out.printf("list size: %d%n", items.size());
34
35         items.remove("yellow"); // remove the first "yellow"
36         display(items, "\nRemove first instance of yellow:");
37         System.out.printf("list size: %d%n", items.size());
38
39         items.remove(1); // remove item at index 1
40         display(items, "\nRemove second list element (green):");
41         System.out.printf("list size: %d%n", items.size());
42
43         // check if a value is in the List
44         System.out.printf("\n\"red\" is %sin the list%n",
45            items.contains("red") ? "": "not ");
46
47         // get the index of an item in the list
48         System.out.printf("\"red\" is located at index: %d%n",
49            items.indexOf("red"));
50
51         // clear the ArrayList and display its size
52         items.clear();
53         System.out.printf("\nSize after clear: %s%n", items.size());
54      }
55
56      // display the ArrayList's elements on the console
57      public static void display(ArrayList<String> items, String header) {
58         System.out.printf(header); // display header
59
60         // display each element in items with enhanced for
61         for (String item : items) {
62            System.out.printf(" %s", item);
63         }
64
```

Fig. 6.14 | Generic `ArrayList<E>` collection demonstration. (Part 2 of 3.)

```
65          System.out.println();
66       }
67    }
```

```
After creating list with initial capacity of 3, size is: 0
Display list contents with counter-controlled for loop: yellow red
Display list contents with enhanced for loop: yellow red
list size: 2

List with two new elements: yellow red green yellow
List has grown beyond initial capacity of 3
list size: 4

Remove first instance of yellow: red green yellow
list size: 3

Remove second list element (green): red yellow
list size: 2

"red" is in the list
"red" is located at index: 0

Size after clear: 0
```

Fig. 6.14 | Generic `ArrayList<E>` collection demonstration. (Part 3 of 3.)

The **`add` method** adds elements to the `ArrayList` (lines 13–14). The version with one argument (line 13) appends it to the end of the `ArrayList`. The version with two arguments (line 14) inserts a new element at the specified position. The first argument is an index. The second argument is the value to insert at that index. The indices of all subsequent elements are incremented by one. Inserting an element is usually slower than adding an element to the end of the `ArrayList`.

Lines 20–22 display the `ArrayList`'s items. The **`size` method** returns the `ArrayList`'s current number of elements—unlike arrays, `ArrayLists` do not have a `length` instance variable. The **`get` method** (line 21) obtains the element at the specified index. Lines 25–26 display the elements again by invoking method `display` (lines 57–66), which uses an enhanced `for` loop to iterate through the `ArrayList`'s elements, just as we did with built-in arrays earlier in this chapter. Then, line 27 displays the `ArrayList`'s current size. Lines 29–30 add two more elements to the `ArrayList`, then line 31 displays the elements again to confirm that the two elements were added to the end of the collection, and line 33 displays the `ArrayList`'s new size. Note that the size is now larger than the `ArrayList`'s initial capacity.

The **`remove` method** call in line 35 deletes the first element with a specific value (`"yellow"`). If no such element is in the `ArrayList`, `remove` does nothing. Lines 36–37 display the `ArrayList`'s current elements and size to show that the first `"yellow"` was removed and the size decreased by one. Line 39 uses an overloaded version of `remove` that deletes the element at the specified index (in this case, `1`). When an element is deleted, the indices of any elements after the removed element decrease by one—a time-consuming operation. Lines 40–41 display the `ArrayList`'s current elements and size to show that `"green"` was removed and the size decreased by one again.

Lines 44–45 use the **`contains` method** to check whether `"red"` is in the `ArrayList`. The `contains` method returns `true` if the element is found in the `ArrayList`; otherwise, it returns `false`. Lines 48–49 use the **`indexOf` method** to get the index of `"red"` in the `ArrayList` or `-1` if `"red"` is not found. Methods `contains` and `indexOf` compare their arguments to each `ArrayList` element in order—known as a linear search—so using them on large `ArrayList`s can be inefficient. We'll discuss the linear search algorithm in Chapter 23, Computer Science Thinking: Recursion, Searching, Sorting and Big O.

Line 52 removes all the `ArrayList`'s remaining elements by calling the **`clear` method**. Then, line 53 shows that the `ArrayList`'s size is now 0.

Diamond (<>) Notation for Creating an Object of a Generic Class

Figure 6.14's `ArrayList` is a local variable, so we used local-variable type inference in line 8 to determine the type of variable `items` from its initializer. In older code, line 8 might appear as:

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

The **diamond (<>) notation** in the `new` expression tells the compiler to infer the `ArrayList` object's element type (`String`) from the declaration of variable `items`. The preceding statement shortens statements like

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

in which `ArrayList<String>` appears in both the variable declaration and the `new` expression.

Checkpoint

1 *(Fill-In)* The compiler ensures all operations on a generic collection are _____.
Answer: type-safe.

2 *(True/False)* Java's automatic processes of converting a primitive value into an object and getting a primitive value from an object are known as _____ and _____.
Answer: boxing, unboxing.

3 *(Code)* Write a program that creates an `ArrayList<Integer>`, then adds the `int` values 1–5 to it and displays the `ArrayList`'s contents.
Answer: In the following code, note that line 14 converts `list` to a `String` with an implicit call to **`ArrayList`'s `toString` method**. The result is a `String` in which the elements are placed in square brackets and separated by commas, as shown in the first line of the output.

```
1  // Section 6.20, Checkpoint 3: ArrayListExample.java
2  // ArrayList with integers.
3  import java.util.ArrayList;
4
5  public class ArrayListExample {
6     public static void main(String[] args) {
7        var list = new ArrayList<Integer>();
8
9        // add 1-5 to the ArrayList
10       for (int i = 1; i <= 5; ++i) {
11          list.add(i); // auto-boxes the ints
12       }
```

(continued…)

```
13
14          System.out.printf("list contains: %s%n", list);
15
16          // total the list's elements
17          int total = 0;
18
19          for (int value : list) { // auto-unboxes the Integers
20             total += value;
21          }
22
23          System.out.printf("sum of list's elements: %d%n", total);
24       }
25    }
```

```
list contains: [1, 2, 3, 4, 5]
sum of list's elements: 15
```

Generative AI

1 Prompt genAIs with the code in Fig. 6.14 and ask them to rewrite the code to remove the calls to `displayArray` and replace them with calls to `System.out.printf` that implicitly use `ArrayList`'s `toString` method. Run the updated code and compare the output to Fig. 6.14's output.

6.21 Wrap-Up

This chapter began our introduction to data structures, using arrays to store data in and retrieve data from lists and tables of values. We discussed the differences between primitive-type variables, which store values, and reference-type variables, which store references to objects elsewhere in memory. We introduced arrays, discussing how to declare, create and initialize them, and explaining how to access and manipulate their elements.

You saw that array elements are initialized by default and that you can use array initializers to provide custom initializers for them. You also used loops to assign calculated values to array elements. We showed common array manipulations, including totaling array elements, presenting array data visually with a bar chart, and using array elements as counters to summarize die-roll frequencies and survey results. We used the `try` statement to handle an `ArrayIndexOutOfBoundsException` when a program attempted to access an array element outside an array's bounds.

You iterated through arrays using counting `for` loops. We also used the enhanced `for` loop to iterate over an array's elements without a counter, eliminating various counting-error possibilities and preventing attempts to access elements outside an array's bounds. We passed arrays and individual array elements to methods to demonstrate the difference between pass-by-value and pass-by-reference.

We introduced multidimensional arrays and used two-dimensional arrays to represent tables containing rows and columns of data. You declared, created and manipulated two-dimensional arrays. You iterated through two-dimensional arrays using nested counting `for` loops and nested enhanced `for` loops. We showed how to write methods that use variable-length argument lists and how to read arguments passed to a program from the command line. You performed various array manipulations with the `Arrays` class's `static` methods.

The Objects-Natural Case Study introduced the `ArrayList<E>` generic collection, which provides the functionality and performance of arrays, as well as dynamic resizing to accommodate more elements. You used the `add` and `remove` methods to add new items to and remove items from an `ArrayList`. We showed `ArrayList` method `size` returns the current number of items in the `ArrayList`. We also used `contains` to determine whether an item was in the `ArrayList`.

We continue our coverage of data structures in Chapter 12, Generic Collections, which introduces several collections from the Java Collections Framework and additional methods of class `Arrays`. In Chapter 17, Functional Programming with Lambdas & Streams, we introduce Java's functional programming capabilities, which enable you to write more concise code that can have fewer errors and that can be easier to read, debug and modify. Chapter 17 is keyed to appropriate earlier sections of the book so that you can conveniently use lambdas and streams sooner if you'd like. Now that you've completed Chapter 6, you can read Sections 17.1–17.7, which introduce the lambdas and streams concepts and use them to rework examples from Chapters 5 and 6. After completing Chapter 17, you'll be able to reimplement many of Chapter 6's examples more concisely and elegantly, and in a way that makes them easier to parallelize to improve performance on today's multi-core systems. In Chapter 19, Concurrency and Parallelism, we'll demonstrate that improved performance by timing various operations using features of the Date/Time API.

In the next chapter, we discuss class `String` and its methods. We introduce regular expressions for pattern matching in strings and use them to validate user input.

Exercises

6.1 *(Die Rolling Modification: Command-Line Argument)* Modify Fig. 6.6's die-rolling program to receive one command-line argument representing the number of dice to roll.

6.2 *(Die Rolling Modification: `ArrayList<Integer>`)* Modify Exercise 6.1's die-rolling program to use an `ArrayList<Integer>` rather than an `int` array.

6.3 *(Sales Commissions)* Use a one-dimensional array to solve the following problem: A company pays its salespeople commissions based on their gross sales. The salespeople receive $200 per week plus 9% of their weekly gross sales. For example, a salesperson who grosses $5,000 in sales receives $200 plus 9% of $5,000—$650 for that week. Write a program (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) $1,000 and over

Summarize the results in tabular format.

6.4 *(Find the Smallest and Largest)* Write a program that uses one `for` loop to determine the smallest and largest values in a 20-element `int` array that's filled with random values from 0–100. Display the array's contents and the smallest and largest values.

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

6.6 *(Variable-Length Argument List)* Write a program that calculates the product of a series of integers passed to method `product` using a variable-length argument list. Test your `product` method with several calls that receive different numbers of arguments.

6.7 *(Command-Line Arguments)* Rewrite Fig. 6.1 so a command-line argument specifies the array's size. If no command-line argument is supplied, use 10 as the default array size.

6.8 *(Using the Enhanced `for` Statement)* Write a program that uses an enhanced `for` statement to sum the `double` values passed by the command-line arguments. Use the `static` method `parseDouble` of class `Double` to convert a `String` to a `double` value.

6.9 *(Dice Rolling)* Write a program to simulate the rolling of two dice. The program should use an object of class `RandomGenerator` once to roll the first die and again to roll the second die. The sum of the two values should then be calculated. Each die can show an integer value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 the least frequent. The following diagram shows the 36 possible dice combinations:

	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

Your program should roll the dice 36,000,000 times. Use a one-dimensional array to tally the number of times each possible sum appears. Display the results in tabular format.

6.10 *(Game of Craps)* Write a program that runs 1,000,000 games of craps (Fig. 5.4) and answers the following questions:

a) How many games are won on the first roll, second roll, …, twentieth roll and after the twentieth roll?
b) How many games are lost on the first roll, second roll, …, twentieth roll and after the twentieth roll?
c) What are the chances of winning at craps? You should discover that craps is one of the fairest casino games. What do you suppose this means?
d) What is the average length of a game of craps?
e) Do the chances of winning improve with the length of the game?

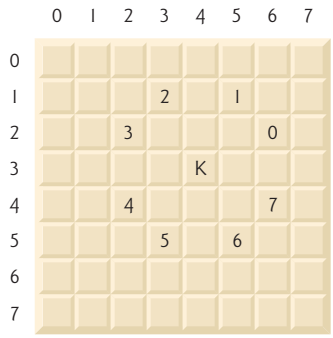
6.11 *(Total Sales)* Use a two-dimensional array to solve the following problem: A company has four salespeople (1 to 4) selling five different products (1 to 5). Once a day, each salesperson passes in a slip for each type of product sold. Each slip contains the following:

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

Thus, each salesperson submits between 0 and 5 sales slips per day. Assume that the information from all the slips for last month is available. Write a program that inputs this information for last month's sales, then summarize the total sales by salesperson and product. Store all totals in the two-dimensional array `sales`. After processing all the information for last month, display the results in tabular format, with each column representing a salesperson and each row representing a particular product. Total each row to get the total sales of each product. Total each column to get the total sales by salesperson for last month. Your output should include these totals to the right of the totaled rows and to the bottom of the totaled columns.

6.12 *(Knight's Tour)* An interesting puzzler for chess buffs is the Knight's Tour problem, originally proposed by the mathematician Euler. Can the knight piece move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

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



a) Draw an eight-by-eight chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a `1` in the starting square, a `2` in the second square, a `3` in the third, and so on. Before starting the tour, estimate how far you think you'll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?
b) Let's develop a program that moves the knight around a chessboard. The board is represented by an eight-by-eight two-dimensional array `board`. Each square is initialized to zero. We describe each of the eight possible moves in terms of its horizontal and vertical components. For example, a move of type 0, as shown in the previous diagram, consists of moving two squares horizontally to the right and one square vertically upward. A move of type 2 consists of moving

one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

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

The variables `currentRow` and `currentColumn` indicate the knight's current row and column, respectively. To make a move of type `moveNumber`, where `moveNumber` is 0–7, your program should use the statements

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

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

c) You've probably gained some valuable insights after attempting to write and run a Knight's Tour program. We'll use these insights to develop a heuristic (i.e., a common-sense rule) for moving the knight. Heuristics do not guarantee success, but they can greatly improve the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. The most problematic 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 reach so that when the board gets congested near the end of the tour, there will be a greater chance of success.

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

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

Write a new version of the Knight's Tour using the accessibility heuristic. The knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. 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 tied square for which the next move would arrive at a square with the lowest accessibility number.

6.13 ***(Knight's Tour: Brute-Force Approaches)*** In part (c) of Exercise 6.12, we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue to increase in power, we'll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let's call this approach "brute-force" problem-solving.

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

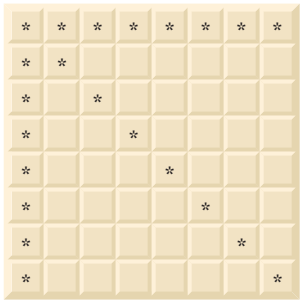
b) The program in part (a) most likely produced a relatively short tour. Modify your program to attempt 1,000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your program finishes attempting the 1,000 tours, it should display this information in a neat tabular format. What was the best result?

c) The program in part (b) most likely gave you some "respectable" tours but no full tours. Now, let your program run until it produces a full tour. This version of the program could run for hours. Keep a table of the number of tours of each length and display the results 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 challenging to develop? Which required more computing power? Can we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Can 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.

6.14 ***(Knight's Tour: Closed-Tour Test)*** In the Knight's Tour (Exercise 6.12), 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 where the knight started the tour. Modify the program you wrote in Exercise 6.12 to test for a closed tour if a full tour has occurred.

6.15 ***(Eight Queens)*** Another puzzler for chess buffs is the Eight Queens problem, which asks: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other—that is, no two queens are in the same row, the same column or the same diagonal? Use the thinking developed in Exercise 6.12 to formulate a heuristic for solving the Eight Queens problem. Run your program. It's possible to assign a value to each square of the chessboard to indicate how many squares of an empty chessboard are "eliminated" if a queen is placed in that square. Each of the corners would be assigned the value 22, as shown in the following diagram:



Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be to place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

6.16 ***(Eight Queens: Brute-Force Approaches)*** In this exercise, you'll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 6.15.

a) Use the random brute-force technique developed in Exercise 6.13 to solve the Eight Queens problem.
b) Use an exhaustive technique to solve the Eight Queens problem. Try all possible combinations of eight queens on the chessboard.
c) Why might the exhaustive brute-force approach not be appropriate for solving the Knight's Tour problem?
d) Compare and contrast the random brute-force and exhaustive brute-force approaches.

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

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

When this process completes, the array elements that are `true` indicate that the corresponding index is a prime number. These indices can be displayed. Write a program that uses a 1,000-element array to determine and display the prime numbers between 2 and 999. Ignore elements 0 and 1.

6.18 ***(Fibonacci Series)*** 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 method `fibonacci(n)` that calculates the *n*th Fibonacci number. The method should receive an `int` and return a `long`. Incorporate this method into a program that enables the user to enter the value of `n`.
b) Determine the largest Fibonacci number that can be displayed on your system.
c) Modify the program you wrote in part (a) to use `BigInteger` instead of `long` to calculate and return Fibonacci numbers. What is the first Fibonacci number greater than `Long.MAX_VALUE` (the largest value a `long` can store)?

Special Section: Building Your Own Computer via Software Simulation

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

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

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

Before running an SML program, we must load the program into memory. Every SML program's first instruction (or statement) is 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 (so instructions are signed four-digit decimal numbers). We assume an SML instruction's sign is always plus, but a data word's sign may be plus or minus. Each Simpletron memory location may contain an instruction or a data value used by a program. Some memory locations may remain unused. Each SML instruction's first two digits are the operation code specifying the operation to perform. SML operation codes are summarized in the following table:

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

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.

SML Addition Program

The following SML program reads two numbers from the user and computes and displays their sum:

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)

Location (Cont.)	Number	Instruction
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

The program operates as follows:

- The instruction +1007 reads the first number from the user and places it into location 07 (which has been initialized to 0).
- Then, +1008 reads the next number into location 08.
- The load instruction, +2007, puts the first number into the accumulator.
- 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 in memory location 09.
- The write instruction, +1109, takes the number from memory location 09 and displays it (as a signed four-digit decimal number).
- The halt instruction, +4300, terminates execution.

SML Comparison Program

The following SML program reads two numbers from the user and determines and displays the larger value. The instruction +4107 is a conditional transfer of control, like Java's `if` statement:

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)

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

a) Use a sentinel-controlled loop to read 10 positive numbers. Compute and display their sum.

b) Use a counter-controlled loop to read seven positive and negative numbers, then compute and display their average.

c) Read a series of numbers and determine and display the largest. The first number read indicates how many numbers the program should process.

6.20 ***(Computer Simulator)*** In this problem, you're going to build your own computer. No, you'll not be soldering components together. Instead, you'll use the powerful technique of software-based simulation to create an object-oriented software model of the Simpletron of Exercise 6.19. Your Simpletron simulator will turn your computer into a Simpletron, and you'll be able to run, test and debug the SML programs you wrote in Exercise 6.19.

When you run your Simpletron simulator, it should display:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction    ***
*** (or data word) at a time. I will display     ***
*** the location number and a question mark (?). ***
*** You then enter the word for that location.   ***
*** Input -99999 to stop entering your program.  ***
```

Your program should simulate the Simpletron's memory with a one-dimensional array `memory` that has 100 elements. Now assume that the simulator is running, and let's examine the dialog as we enter the second SML program in Exercise 6.19:

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

Your program should display the memory location followed by a question mark. The user enters each value to the right of a question mark in the preceding dialogue. When the user enters the sentinel value `-99999`, the program should display:

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

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

Use the variable `accumulator` to represent the accumulator register. Use the variable `instructionCounter` to keep track of the location in memory that contains the instruction being performed. Use the variable `operationCode` to indicate the current operation (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 current instruction. Do not execute instructions directly from memory. Instead, transfer the next instruction from memory to `instructionRegister`. Then, pick off the left two digits and place them in `operationCode`, and pick off the

right two digits and place them in `operand`. When the Simpletron begins execution, the special registers are all initialized to zero.

Let's walk through the execution of the first SML instruction, `+1009` in memory location `00`. This procedure is called an instruction-execution cycle.

The `instructionCounter` tells us the location of the next instruction to be performed. We fetch the contents of that location from `memory` by using the Java statement

```
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 a read (versus a write, a load, and so on). A `switch` statement or `switch` expression differentiates among the 12 SML operations. In the `switch`, simulate various SML instructions' behavior as shown in the following table. We discuss branch instructions shortly and leave the others to you:

Instruction	Description
read:	Display the prompt `"Enter an integer"`, then input the integer and store it in location `memory[operand]`.
load:	`accumulator = memory[operand];`
add:	`accumulator += memory[operand];`
halt:	Terminate the SML program's execution and display `*** Simpletron execution terminated ***`

When the SML program completes execution, each register's name and contents as well as the complete memory contents should be displayed. Such a printout is often called a computer dump (no, a computer dump is not a place where old computers go). To help you program your `dump` method, the following is a sample dump format:

```
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
```

After executing a Simpletron program, a dump would show the values in the registers and memory when execution terminated.

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

At this point, the simulation of the first instruction is complete. 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 increment the instruction-counter register:

```
++instructionCounter;
```

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

Now, let's consider how the Simpletron simulates branching instructions—the transfers of control. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated in the switch as

```
instructionCounter = operand;
```

Similarly, 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 SML program you wrote in Exercise 6.19. If you desire, you may embellish SML with additional features and provide these features in your simulator.

Your simulator should check for various types of errors. During the program-loading phase, for example, each number the user enters 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 re-enter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, and accumulator overflows—arithmetic operations that produce values outside the range +9999 to -9999. Such serious errors are called fatal errors. When a fatal error is detected, your simulator should display an error message, such as

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

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

6.21 ***(Simpletron Simulator Modifications)*** In Exercise 6.20, 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 the exercises of Chapter 12, 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 1,000 memory locations to enable the Simpletron to handle larger programs.
b) Allow the simulator to perform remainder calculations. This modification requires an additional SML instruction.
c) Allow the simulator to perform exponentiation calculations. This modification requires an additional SML instruction.
d) Modify the simulator to use hexadecimal rather than integer values to represent SML instructions.
e) Modify the simulator to allow output of a newline. This modification requires an additional SML instruction.
f) Modify the simulator to process floating-point values in addition to integers.
g) Modify the simulator to handle string input. Each Simpletron word can be divided into two groups, each holding a two-digit integer representing the ASCII decimal equivalent of a character (`https://www.ascii-code.com/characters`). Add an SML instruction that inputs a string and stores it 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 enable output of the strings in part (g). Add an SML instruction that displays 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 SML instruction checks the length and displays the string by translating each two-digit number into its equivalent character.

Special Section: Generative AI Exercises

For each of the following questions, try multiple genAIs to determine which performs best. Always verify the results. Keep in mind that they sometimes make mistakes and even hallucinate, creating fake information.

6.22 ***(Generative AI: Research the Next Version of Java)*** New Java versions are released every March and September. Prompt generative AIs with the current version of Java and ask them to write a tutorial covering the significant new language and library features scheduled to be released in the next version.

6.23 ***(Generative AI: One-Dimensional Array Animation Demos)*** Prompt generative AIs to locate animations that demonstrate iterating through a one-dimensional array.

6.24 ***(Generative AI: Two-Dimensional Array Animation Demos)*** Prompt generative AIs to locate animations that demonstrate iterating through a two-dimensional array.

6.25 ***(Generative AI: One-Dimensional Array Practice Exercises)*** Prompt generative AIs to create practice novice-level exercises with answers for one-dimensional Java array manipulations.

6.26 ***(Generative AI: Two-Dimensional Array Practice Exercises)*** Prompt generative AIs to create practice novice-level exercises with answers for two-dimensional Java array manipulations.

6.27 ***(Generative AI: What Does This One-Dimensional Array Code Do?)*** Prompt generative AIs to create cryptic code examples for one-dimensional Java array manipulations so you can study them and determine what they do.

6.28 ***(Generative AI: What Does This Two-Dimensional Array Code Do?)*** Prompt generative AIs to create cryptic code examples for two-dimensional Java array manipulations so you can study them and determine what they do.

6.29 ***(Generative AI: Failing Fast vs. Exception Handling)*** Some developers believe that during program development, it's better to allow a program to fail when an exception occurs (known as "failing fast") rather than catching and handling the exception. Prompt generative AIs to discuss the benefits and disadvantages of failing fast vs. exception handling.

6.30 ***(Generative AI: Improving Fault Tolerance)*** Prompt generative AIs with a program and ask them to improve the code's fault tolerance by adding exception handling for methods that throw exceptions.

6.31 ***(Generative AI: Making a Difference)*** Prompt generative AIs for suggested Java programs and projects you can build to "make a difference" in the world.

6.32 ***(Generative AI: Checking Java Idiom)*** Prompt generative AIs with code from your exercise solutions and ask them to evaluate the code for proper idiom in the context of the latest Java version and what you've learned so far in this book.

6.33 ***(Generative AI: Primitive vs. Reference Types)*** Prompt generative AIs to write a tutorial on primitive types vs. reference types in Java.

6.34 ***(Generative AI: Passing Arguments to Methods)*** Prompt generative AIs to write a tutorial comparing pass-by-value and pass-by-reference in Java.

6.35 ***(Generative AI: Modernizing Java Code)*** Because of the vast amounts of Java legacy code that have been written since Java appeared in 1995, many developers are concerned with modernizing their organizations' code bases. Prompt generative AIs to write a tutorial on modernizing Java code.