7.4 Examples Using Arrays

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

7.4.1 Creating and Initializing an Array

The application of Fig. 7.2 uses keyword `new` to create an array of 10 `int` elements, which are initially zero (the default initial value for `int` variables). Line 7 declares `array`—a reference capable of referring to an array of `int` elements—then initializes the variable with a reference to an array object containing 10 `int` elements. Line 9 outputs the column headings. The first column contains the index (0–9) of each array element, and the second column contains the default initial value (0) of each array element.

```
1   // Fig. 7.2: InitArray.java
2   // Initializing the elements of an array to defa
3
4   public class InitArray {
5      public static void main(String[] args) {
6         // declare variable array and initialize i
7         int[] array = new int[10]; // create the a
8
```

```
 9          System.out.printf("%s%8s%n", "Index", "Val
                       10
 11          // output each array element's value
 12         for (int counter = 0; counter < array.leng
 13            System.out.printf("%5d%8d%n", counter,
                    14         }
                      15      }
                        16   }
```

Index	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Fig. 7.2

Initializing the elements of an array to default values of zero.

The `for` statement (lines 12–14) outputs the index

(represented by `counter`) and value of each array element (represented by `array[counter]`). Control variable `counter` is initially `0`—index values start at 0, so using **zero-based counting** allows the loop to access every element of the array. The `for`'s loop-continuation condition uses the expression `array.length` (line 12) to determine the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable `counter` is less than 10. The highest index value of a 10-element array is 9, so using the less-than operator in the loop-continuation condition guarantees that the loop does not attempt to access an element *beyond* the end of the array (i.e., during the final iteration of the loop, `counter` is `9`). We'll soon see what Java does when it encounters such an *out-of-range index* at execution time.

7.4.2 Using an Array Initializer

You can create an array and initialize its elements with an **array initializer**—a comma-separated list of expressions (called an **initializer list**) enclosed in braces. In this case, the array length is determined by the number of elements in the initializer list. 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]` is initialized to `20`, and so on. When the compiler encounters an array declaration that includes an initializer list, it *counts* the number of initializers in the list to determine the size of the array, then sets up the appropriate `new` operation “behind the scenes.”

The application in Fig. 7.3 initializes an integer array with 10 values (line 7) and displays the array in tabular format. The code for displaying the array elements (lines 12–14) is identical to that in Fig. 7.2 (lines 12–14).

```
          1   // Fig. 7.3: InitArray.java
   2   // Initializing the elements of an array with an
                          3
             4   public class InitArray {
    5      public static void main(String[] args) {
    6         // initializer list specifies the initial
    7         int[] array = {32, 27, 64, 18, 95, 14, 90,
                          8
    9         System.out.printf("%s%8s%n", "Index", "Val
                         10
    11         // output each array element's value
   12         for (int counter = 0; counter < array.leng
   13            System.out.printf("%5d%8d%n", counter,
                     14         }
                       15      }
                         16   }
```

Index	Value
0	32
1	27

2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Fig. 7.3

Initializing the elements of an array with an array initializer.

7.4.3 Calculating the Values to Store in an Array

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

```
1   // Fig. 7.4: InitArray.java
```

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

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18

```
9                                    20
```

Fig. 7.4

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

Line 6 uses the modifier `final` to declare the constant variable `ARRAY_LENGTH` with the value `10`. Constant variables must be initialized *before* they're used and *cannot* be modified thereafter. If you attempt to *modify* a `final` variable after it's initialized in its declaration, the compiler issues an error message like

```
cannot assign a value to final variable variableName
```

Good Programming Practice 7.2

Constant variables also are called ***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* `10` *could have different meanings based on its context.*

Good Programming Practice 7.3

Constants use all uppercase letters by convention and multiword named constants should have each word separated from the next with an underscore (_) as in `ARRAY_LENGTH`.

Common Programming Error 7.4

Assigning a value to a previously initialized `final` *variable is a compilation error. Similarly, attempting to access the value of a* `final` *variable before it's initialized results in a compilation error like, "*`variable` *variableName* `might not have been initialized`*."*

7.4.4 Summing the Elements of an Array

Often, array elements represent values for use in a calculation. If, for example, they're exam grades, a professor may wish to total the array elements and use that sum to calculate the class average. The `GradeBook` examples in Figs. 7.14 and 7.18 use this technique.

Figure 7.5 sums the values contained in a 10-element array. The program declares, creates and initializes the array at line 6. Lines 10–12 perform the calculations.

```
1   // Fig. 7.5: 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[] array = {87, 68, 94, 100, 83, 78, 85
7         int total = 0;
8
9         // add each element's value to total
10        for (int counter = 0; counter < array.leng
11           total += array[counter];
12        }
13
14        System.out.printf("Total of array elements
15     }
16  }
```

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

Fig. 7.5

Computing the sum of the elements of an array.

Note that the values supplied as array initializers are often read into a program rather than specified in an initializer list. For example, you could input the values from a user or from a file

on disk, as discussed in Chapter 15, Files, Input/Output Streams, NIO and XML Serialization. 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* sets of data.

7.4.5 Using Bar Charts to Display Array Data Graphically

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

Professors often like to examine the grade distribution on an exam. A professor might visualize this with a graph of the number of grades in each of 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 grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Figure 7.6 stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, `array[0]` indicates the number of grades in the range 0–9, `array[7]` the number of grades in the range 70–79 and `array[10]` the number of 100 grades.

```
1   // Fig. 7.6: BarChart.java
2   // Bar chart printing program.
3
4   public class BarChart {
5      public static void main(String[] args) {
6         int[] array = {0, 0, 0, 0, 0, 0, 1, 2, 4,
7
8         System.out.println("Grade distribution:");
9
10        // for each array element, output a bar of
11        for (int counter = 0; counter < array.leng
12           // output bar label ("00-09: ", …, "90-
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 < array[count
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. 7.6

Bar chart printing program.

The `GradeBook` classes later in the chapter (Figs. 7.14 and 7.18) contain code that calculates these grade frequencies based on a set of grades. For now, we manually create the array with the given grade frequencies. The application reads the numbers from the array and graphs the information as a bar chart. It displays 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. The nested `for` statement (lines 22–24) outputs the bars. Note the loop-continuation condition at line 22 (`stars < array[counter]`). Each time the program reaches the inner `for`, the loop counts from `0` up to `array[counter]`, thus using a value in `array` to determine the number of asterisks to display. In this example, *no* students received a grade below 60, so `array[0]`–`array[5]` contain zeroes, 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).

7.4.6 Using the Elements of an Array as Counters

Sometimes, programs use counter variables to summarize data, such as the results of a survey. In Fig. 6.7, we used separate counters in our die-rolling program to track the number of occurrences of each side of a six-sided die as the program rolled the die 60,000,000 times. An array version of this application is shown in Fig. 7.7.

```
1   // Fig. 7.7: RollDie.java
2   // Die-rolling program using arrays instead of s
3   import java.security.SecureRandom;
4
5   public class RollDie {
6      public static void main(String[] args) {
7         SecureRandom randomNumbers = new SecureRan
8         int[] frequency = new int[7]; // array of
9
10        // roll die 60,000,000 times; use die valu
11        for (int roll = 1; roll <= 60_000_000; rol
12           ++frequency[1 + randomNumbers.nextInt(6
13        }
14
15        System.out.printf("%s%10s%n", "Face", "Fre
16
```

```
 17          // output each array element's value
18          for (int face = 1; face < frequency.length
19             System.out.printf("%4d%10d%n", face, fr
                  20          }
                    21       }
                      22    }
```

Face	Frequency
1	9995532
2	10003079
3	10000564
4	10000726
5	9998994
6	10001105

Fig. 7.7

Die-rolling program using arrays instead of `switch`.

8

Figure 7.7 uses the array `frequency` (line 8) to count the occurrences of each side of the die. *The single statement in line 12 of this program replaces lines 19–41 of Fig. 6.7*. 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. However, we use a seven-element array in which we ignore `frequency[0]`—it's more logical to have the face value 1 increment `frequency[1]` than `frequency[0]`. Thus, each face value is used as an index for array `frequency`. In line 12, the calculation inside the square brackets evaluates first to determine which element of the array to increment, then the `++` operator adds one to that element. We also replaced lines 45–47 from Fig. 6.7 by looping through array `frequency` to output the results (lines 18–20). When we study Java SE 8's functional programming capabilities in Chapter 17, we'll show how to replace lines 11–13 and 18–20 with a *single* statement!

7.4.7 Using Arrays to Analyze Survey Results

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

Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.

This is a typical array-processing application (Fig. 7.8). We wish to summarize the number of responses of each type (that is, 1–5). Array `responses` (lines 7–8) is a 20-element integer array containing the students' survey responses. The last value in the array *is intentionally* an incorrect response

(14). When a Java program executes, array element indices are checked for validity—all indices must be greater than or equal to 0 and less than the length of the array. Any attempt to access an element outside that range of indices results in a runtime error that's known as an `ArrayIndexOutOfBoundsException`. At the end of this section, we'll discuss the invalid response value, demonstrate array **bounds checking** and introduce Java's *exception-handling* mechanism, which can be used to detect and handle an `ArrayIndexOutOfBoundsException`.

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.

```
1   // Fig. 7.8: StudentPoll.java
2   // Poll analysis program.
3
4   public class StudentPoll {
5      public static void main(String[] args) {
6         // student response array (more typically,
7         int[] responses =
8            {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3,
9         int[] frequency = new int[6]; // array of
10
11        // for each answer, select responses eleme
```

```
12          // as frequency index to determine element
13          for (int answer = 0; answer < responses.le
                14             try {
 15                ++frequency[responses[answer]];
                  16             }
17              catch (ArrayIndexOutOfBoundsException e
18                System.out.println(e); // invokes to
19                System.out.printf(" responses[%d] =
 20                   answer, responses[answer]);
                  21             }
                    22          }
                         23
24          System.out.printf("%s%10s%n", "Rating", "F
                         25
 26          // output each array element's value
27          for (int rating = 1; rating < frequency.le
28             System.out.printf("%6d%10d%n", rating,
                    29          }
                      30      }
                        31    }
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
responses[19] = 14
```

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

Fig. 7.8

Poll analysis program.

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 appears in 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`—see 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 index for the `frequency` array to determine which counter to increment (in this case, `frequency[1]`).
- The next time through the loop `answer` is `1`, `responses[answer]` is the value of `responses[1]` (that is, `2`—see 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`—see 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. In the program's output, the `Frequency` column summarizes only 19 of the 20 values in the `responses` array —the last element of the array `responses` contains an (intentionally) incorrect response that was not counted. Section 7.5 discusses what happens when the program in Fig. 7.8 encounters the invalid response (`14`) in the last element of array `responses`.