Chapter 8

Arrays

Up to this point, the only variables we have used were for individual values such as numbers or strings. In this chapter, we'll learn how to store multiple values of the same type using a single variable. This language feature will enable you to write programs that manipulate larger amounts of data.

8.1 Creating arrays

An **array** is a sequence of values; the values in the array are called **elements**. You can make an array of `int`s, `double`s, or any other type, but all the values in an array must have the same type.

To create an array, you have to declare a variable with an *array type* and then create the array itself. Array types look like other Java types, except they are followed by square brackets (`[]`). For example, the following lines declare that `counts` is an "integer array" and `values` is a "double array":

```
int[] counts;
double[] values;
```

To create the array itself, you have to use the `new` operator, which we first saw in Section 3.2:

```
counts = new int[4];
values = new double[size];
```

The first assignment makes `count` refer to an array of four integers. The second makes `values` refer to an array of `double`, where the number of elements in `values` depends on the value of `size`.

Of course, you can also declare the variable and create the array in a single line of code:

```
int[] counts = new int[4];
double[] values = new double[size];
```

You can use any integer expression for the size of an array, as long as the value is nonnegative. If you try to create an array with -4 elements, for example, you will get a `NegativeArraySizeException`. An array with zero elements is allowed, and there are special uses for such arrays that we'll see later on.

8.2 Accessing elements

When you create an array of `int`s, the elements are initialized to zero. Figure 8.1 shows a state diagram of the `counts` array so far.

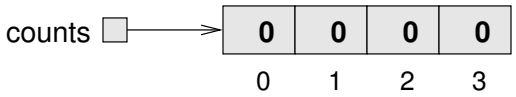

Figure 8.1: State diagram of an `int` array.

The arrow indicates that the value of `counts` is a **reference** to the array. You should think of *the array* and *the variable* that refers to it as two different things. As we'll soon see, we can assign a different variable to refer to the same array, and we can change the value of `counts` to refer to a different array.

The large numbers inside the boxes are the elements of the array. The small numbers outside the boxes are the **indexes** (or indices) used to identify each location in the array. Notice that the index of the first element is 0, not 1, as you might have expected.

The `[]` operator selects elements from an array:

```
System.out.println("The zeroth element is " + counts[0]);
```

You can use the [] operator anywhere in an expression:

```
counts[0] = 7;
counts[1] = counts[0] * 2;
counts[2]++;
counts[3] -= 60;
```

Figure 8.2 shows the result of these statements.

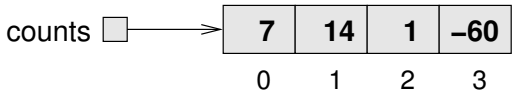

Figure 8.2: State diagram after several assignment statements.

You can use any expression as an index, as long as it has type `int`. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;
while (i < 4) {
    System.out.println(counts[i]);
    i++;
}
```

This `while` loop counts from 0 up to 4. When `i` is 4, the condition fails and the loop terminates. So the body of the loop is only executed when `i` is 0, 1, 2, and 3.

Each time through the loop we use `i` as an index into the array, displaying the `i`th element. This type of array processing is often written using a `for` loop.

```
for (int i = 0; i < 4; i++) {
    System.out.println(counts[i]);
}
```

For the `counts` array, the only legal indexes are 0, 1, 2, and 3. If the index is negative or greater than 3, the result is an `ArrayIndexOutOfBoundsException`.

8.3 Displaying arrays

You can use `println` to display an array, but it probably doesn't do what you would like. For example, the following fragment (1) declares an array variable,

(2) makes it refer to an array of four elements, and (3) attempts to display the contents of the array using `println`:

```
int[] a = {1, 2, 3, 4};
System.out.println(a);
```

Unfortunately, the output is something like:

```
[I@bf3f7e0
```

The bracket indicates that the value is an array, `I` stands for "integer", and the rest represents the address of the array. If we want to display the elements of the array, we can do it ourselves:

```
public static void printArray(int[] a) {
    System.out.print("{" + a[0]);
    for (int i = 1; i < a.length; i++) {
        System.out.print(", " + a[i]);
    }
    System.out.println("}");
}
```

Given the previous array, the output of this method is:

```
{1, 2, 3, 4}
```

The Java library provides a utility class `java.util.Arrays` that provides methods for working with arrays. One of them, `toString`, returns a string representation of an array. We can invoke it like this:

```
System.out.println(Arrays.toString(a));
```

And the output is:

```
[1, 2, 3, 4]
```

As usual, we have to import `java.util.Arrays` before we can use it. Notice that the string format is slightly different: it uses square brackets instead of curly braces. But it beats having to write the `printArray` method.

8.4 Copying arrays

As explained in Section 8.2, array variables contain *references* to arrays. When you make an assignment to an array variable, it simply copies the reference. But it doesn't copy the array itself! For example:

```
double[] a = new double[3];
double[] b = a;
```

These statements create an array of three `double`s and make two different variables refer to it, as shown in Figure 8.3.

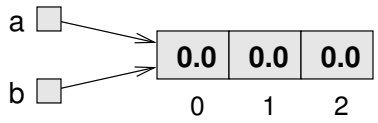

Figure 8.3: State diagram showing two variables that refer to the same array.

Any changes made through either variable will be seen by the other. For example, if we set `a[0] = 17.0`, and then display `b[0]`, the result is `17.0`. Because `a` and `b` are different names for the same thing, they are sometimes called **aliases**.

If you actually want to copy the array, not just a reference, you have to create a new array and copy the elements from the old to the new, like this:

```
double[] b = new double[3];
for (int i = 0; i < 3; i++) {
    b[i] = a[i];
}
```

Another option is to use `java.util.Arrays`, which provides a method named `copyOf` that copies an array. You can invoke it like this:

```
double[] b = Arrays.copyOf(a, 3);
```

The second parameter is the number of elements you want to copy, so you can also use `copyOf` to copy just part of an array.

8.5 Array length

The examples in the previous section only work if the array has three elements. It would be better to generalize the code to work with arrays of any size. We can do that by replacing the magic number, 3, with `a.length`:

```
double[] b = new double[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

All arrays have a built-in constant, `length`, that stores the number of elements. The expression `a.length` may look like a method invocation, but there are no parentheses and no arguments.

The last time this loop gets executed, `i` is `a.length - 1`, which is the index of the last element. When `i` is equal to `a.length`, the condition fails and the body is not executed – which is a good thing, because trying to access `a[a.length]` would throw an exception.

You can also use `a.length` with `Arrays.copyOf`:

```
double[] b = Arrays.copyOf(a, a.length);
```

8.6 Array traversal

Many computations can be implemented by looping through the elements of an array and performing an operation on each element. For example, the following loop squares the elements of a `double` array:

```
for (int i = 0; i < a.length; i++) {
    a[i] = Math.pow(a[i], 2.0);
}
```

Looping through the elements of an array is called a **traversal**. Another common pattern is a **search**, which involves traversing an array looking for a particular element. For example, the following method takes an `int` array and an integer value, and it returns the index where the value appears:

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

If we find the target value in the array, we return its index immediately. If the loop exits without finding the target, it returns `-1`, a special value chosen to indicate a failed search.

Another common traversal is a **reduce** operation, which "reduces" an array of values down to a single value. Examples include the sum or product of the elements, the minimum, and the maximum. The following method takes a `double` array and returns the sum of the elements:

```
public static int sum(double[] a) {
    double total = 0.0;
    for (int i = 0; i < a.length; i++) {
        total += a[i];
    }
    return total;
}
```

Before the loop, we initialize `total` to zero. Each time through the loop, we update `total` by adding one element from the array. At the end of the loop, `total` contains the sum of the elements. A variable used this way is sometimes called an **accumulator**.

8.7 Random numbers

Most computer programs do the same thing every time they run; programs like that are **deterministic**. Usually determinism is a good thing, since we expect the same calculation to yield the same result. But for some applications, we want the computer to be unpredictable. Games are an obvious example, but there are many others.

Making a program **nondeterministic** turns out to be hard, because it's hard for a computer to generate truly random numbers. But there are algorithms that generate unpredictable sequences called **pseudorandom** numbers. For most applications, they are as good as random.

If you did Exercise 3.4, you have already seen `java.util.Random`, which generates pseudorandom numbers. The method `nextInt` takes an integer argument, `n`, and returns a random integer between `0` and `n - 1` (inclusive).

If you generate a long series of random numbers, every value should appear, at least approximately, the same number of times. One way to test this behavior of `nextInt` is to generate a large number of values, store them in an array, and count the number of times each value occurs.

The following method creates an `int` array and fills it with random numbers between 0 and 99. The argument specifies the size of the array, and the return value is a reference to the new array.

```
public static int[] randomArray(int size) {
    Random random = new Random();
    int[] a = new int[size];
    for (int i = 0; i < a.length; i++) {
        a[i] = random.nextInt(100);
    }
    return a;
}
```

The following fragment generates an array and displays it using `printArray` from Section 8.3:

```
int numValues = 8;
int[] array = randomArray(numValues);
printArray(array);
```

The output looks like this:

```
{15, 62, 46, 74, 67, 52, 51, 10}
```

If you run it, you will probably get different values.

8.8 Traverse and count

If these values were exam scores – and they would be pretty bad exam scores – the teacher might present them to the class in the form of a **histogram**. In statistics, a histogram is a set of counters that keeps track of the number of times each value appears.

For exam scores, we might have ten counters to keep track of how many students scored in the 90s, the 80s, etc. To do that, we can traverse the array and count the number of elements that fall in a given range.

The following method takes an array and two integers, `low` and `high`. It returns the number of elements that fall in the range from `low` to `high`.

```
public static int inRange(int[] a, int low, int high) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] >= low && a[i] < high) {
            count++;
        }
    }
    return count;
}
```

This pattern should look familiar: it is another reduce operation. Notice that `low` is included in the range (`>=`), but `high` is excluded (`<`). This detail keeps us from counting any scores twice.

Now we can count the number of scores in each grade range:

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

8.9 Building a histogram

The previous code is repetitious, but it is acceptable as long as the number of ranges is small. But suppose we wanted to keep track of the number of times

each score appears. We would have to write 100 lines of code:

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
...
int count99 = inRange(scores, 99, 100);
```

What we need is a way to store 100 counters, preferably so we can use an index to access them. In other words, we need another array!

The following fragment creates an array of 100 counters, one for each possible score. It loops through the scores and uses `inRange` to count how many times each score appears. Then it stores the results in the array:

```
int[] counts = new int[100];
for (int i = 0; i < counts.length; i++) {
    counts[i] = inRange(scores, i, i + 1);
}
```

Notice that we are using the loop variable `i` three times: as an index into the `counts` array, and as two arguments for `inRange`. The code works, but it is not as efficient as it could be. Every time the loop invokes `inRange`, it traverses the entire array.

It would be better to make a single pass through the array, and for each score, compute which range it falls in and increment the corresponding counter. This code traverses the array of scores *only once* to generate the histogram:

```
int[] counts = new int[100];
for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

Each time through the loop, it selects one element from `scores` and uses it as an index to increment the corresponding element of `counts`. Because this code only traverses the array of scores once, it is much more efficient.

8.10 The enhanced for loop

Since traversing arrays is so common, Java provides an alternative syntax that makes the code more compact. For example, consider a `for` loop that displays the elements of an array on separate lines:

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

We could rewrite the loop like this:

```
for (int value : values) {
    System.out.println(value);
}
```

This statement is called an **enhanced for loop**. You can read it as, "for each `value` in `values`". It's conventional to use plural nouns for array variables and singular nouns for element variables.

Using the enhanced `for` loop, and removing the temporary variable, we can write the histogram code from the previous section more concisely:

```
int[] counts = new int[100];
for (int score : scores) {
    counts[score]++;
}
```

Enhanced `for` loops often make the code more readable, especially for accumulating values. But they are not helpful when you need to refer to the index, as in search operations.

8.11 Vocabulary

array: A collection of values, where all the values have the same type, and each value is identified by an index.

element: One of the values in an array. The `[]` operator selects elements.

index: An integer variable or value used to indicate an element of an array.

reference: A value that indicates another value, like an array. In a state diagram, a reference appears as an arrow.

alias: A variable that refers to the same object as another variable.

traversal: Looping through the elements of an array (or other collection).

search: A traversal pattern used to find a particular element of an array.

reduce: A traversal pattern that combines the elements of an array into a single value.

accumulator: A variable used to accumulate results during a traversal.

deterministic: A program that does the same thing every time it is invoked.

nondeterministic: A program that always behaves differently, even when run multiple times with the same input.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

enhanced for loop: An alternative syntax for traversing the elements (values) of an array.

8.12 Exercises

The code for this chapter is in the `ch08` directory of `ThinkJavaCode`. See page xv for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 8.1 The goal of this exercise is to practice encapsulation with some of the examples in this chapter.

1. Starting with the code in Section 8.6, write a method called `powArray` that takes a `double` array, `a`, and returns a new array that contains the elements of `a` squared. Generalize it to take a second argument and raise the elements of `a` to the given power.

2. Starting with the code in Section 8.10, write a method called `histogram` that takes an `int` array of scores from 0 to (but not including) 100, and returns a histogram of 100 counters. Generalize it to take the number of counters as an argument.

Exercise 8.2 The purpose of this exercise is to practice reading code and recognizing the traversal patterns in this chapter. The following methods are hard to read, because instead of using meaningful names for the variables and methods, they use names of fruit.

```
public static int banana(int[] a) {
    int kiwi = 1;
    int i = 0;
    while (i < a.length) {
        kiwi = kiwi * a[i];
        i++;
    }
    return kiwi;
}
```

```
public static int grapefruit(int[] a, int grape) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == grape) {
            return i;
        }
    }
    return -1;
}
```

```
public static int pineapple(int[] a, int apple) {
    int pear = 0;
    for (int pine: a) {
        if (pine == apple) {
            pear++;
        }
    }
    return pear;
}
```

For each method, write one sentence that describes what the method does,

without getting into the details of how it works. For each variable, identify the role it plays.

Exercise 8.3 What is the output of the following program? Draw a stack diagram that shows the state of the program just before `mus` returns. Describe in a few words what `mus` does.

```
public static int[] make(int n) {
    int[] a = new int[n];
    for (int i = 0; i < n; i++) {
        a[i] = i + 1;
    }
    return a;
}
```

```
public static void dub(int[] jub) {
    for (int i = 0; i < jub.length; i++) {
        jub[i] *= 2;
    }
}
```

```
public static int mus(int[] zoo) {
    int fus = 0;
    for (int i = 0; i < zoo.length; i++) {
        fus += zoo[i];
    }
    return fus;
}
```

```
public static void main(String[] args) {
    int[] bob = make(5);
    dub(bob);
    System.out.println(mus(bob));
}
```

Exercise 8.4 Write a method called `indexOfMax` that takes an array of integers and returns the index of the largest element. Can you write this method using an enhanced `for` loop? Why or why not?

Exercise 8.5 The Sieve of Eratosthenes is "a simple, ancient algorithm for

finding all prime numbers up to any given limit," which you can read about at https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

Write a method called `sieve` that takes an integer parameter, `n`, and returns a `boolean` array that indicates, for each number from `0` to `n - 1`, whether the number is prime.

Exercise 8.6 Write a method named `areFactors` that takes an integer `n` and an array of integers, and that returns `true` if the numbers in the array are all factors of `n` (which is to say that `n` is divisible by all of them).

Exercise 8.7 Write a method named `arePrimeFactors` that takes an integer `n` and an array of integers, and that returns `true` if the numbers in the array are all prime *and* their product is `n`.

Exercise 8.8 Many of the patterns we have seen for traversing arrays can also be written recursively. It is not common, but it is a useful exercise.

1. Write a method called `maxInRange` that takes an array of integers and two indexes, `lowIndex` and `highIndex`, and finds the maximum value in the array, but only considering the elements between `lowIndex` and `highIndex`, including both.

   This method should be recursive. If the length of the range is 1, that is, if `lowIndex == highIndex`, we know immediately that the sole element in the range must be the maximum. So that's the base case.

   If there is more than one element in the range, we can break the array into two pieces, find the maximum in each of the pieces, and then find the maximum of the maxima.

2. Methods like `maxInRange` can be awkward to use. To find the largest element in an array, we have to provide the range for the entire array.

   ```
   double max = maxInRange(a, 0, a.length - 1);
   ```

   Write a method called `max` that takes an array and uses `maxInRange` to find and return the largest element.