77

Here is a possible definition of the Point class we have been discussing:

```
/** Represents a Cartesian (x,y) point */
public class Point {
     // The coordinates of the point
     public double x, y;
     public Point(double x, double y) {     // A constructor that
         this.x = x; this.y = y;            // initializes the fields
     }

     public double distanceFromOrigin() {   // A method that operates
         return Math.sqrt(x*x + y*y);       // on the x and y fields
     }
}
```

This class definition is stored in a file named *Point.java* and compiled to a file named *Point.class*, where it is available for use by Java programs and other classes. This class definition is provided here for completeness and to provide context, but don't expect to understand all the details just yet; most of Chapter 3 is devoted to the topic of defining classes.

Now that we have defined the Point class as a new data type, we can use the following line to declare a variable that holds a Point object:

```
Point p;
```

Declaring a variable to hold a Point object does not create the object itself, however. To actually create an object, you must use the new operator. This keyword is followed by the object's class (i.e., its type) and an optional argument list in parentheses. These arguments are passed to the constructor for the class, which initializes internal fields in the new object:

```
// Create a Point object representing (2,-3.5).
// Declare a variable p and store a reference to the new Point object
Point p = new Point(2.0, -3.5);

// Create some other objects as well
// An object that represents the current time
LocalDateTime d = new LocalDateTime();
// A HashSet object to hold a set of strings
Set<String> words = new HashSet<>();
```

78

Now that we've seen how to define classes and instantiate them by creating objects, we need to look at the Java syntax that allows us to use those objects. Recall that a class defines a collection of fields and methods. Each object has its own copies of those fields and has access to those methods. We use the dot character (.) to access the named fields and methods of an object. For example:

```
Point p = new Point(2, 3);          // Create an object
double x = p.x;                     // Read a field of the object
p.y = p.x * p.x;                    // Set the value of a field
double d = p.distanceFromOrigin(); // Access a method of the object
```

This syntax is very common when programming in object-oriented languages, and Java is no exception, frequently. Note, in particular, the expressions p.distanceFromOrigin(). This tells the Java compiler to look up a method named distanceFromOrigin() (which is defined by the class Point) and use that method to perform a computation on the fields of the object p. We'll cover the details of this operation in Chapter 3.

The String class represents text as a string of characters. Because programs usually communicate with their users through the written word, the ability to manipulate strings of text is quite important in any programming language. In Java, strings are objects; the data type used to represent text is the String class. Modern Java programs usually use more string data than anything else.

Accordingly, because strings are such a fundamental data type, Java allows you to include text literally in programs by placing it between double-quote (") characters. For example:

```
String name = "David";
System.out.println("Hello, " + name);
```

Excerpts from Flanagan, D. (2019). *Java in a Nutshell*. Sebastopol, CA. O'Reilly.

79

An *array* is a special kind of object that holds zero or more primitive values or references. These values are held in the *elements* of the array, which are unnamed variables referred to by their position or *index*. The type of an array is characterized by its *element type*, and all elements of the array must be of that type.

Array elements are numbered starting with zero, and valid indexes range from zero to the number of elements minus one. The array element with index 1, for example, is the *second* element in the array. The number of elements in an array is its *length*. The length of an array is specified when the array is created, and it never changes.

The element type of an array may be any valid Java type, including array types. This means that Java supports arrays of arrays, which provide a kind of multidimensional array capability. Java does not support the matrix-style multidimensional arrays found in some languages.

Array types are reference types, just as classes are. Instances of arrays are objects, just as the instances of a class are.[4] Unlike classes, array types do not have to be defined. Simply place square brackets after the element type. For example, the following code declares three variables of array type:

```
byte b;                         // byte is a primitive type
byte[] arrayOfBytes;            // byte[] is an array of byte values
byte[][] arrayOfArrayOfBytes;   // byte[][] is an array of byte[]
String[] points;                // String[] is an array of strings
```

The length of an array is not part of the array type. It is not possible, for example, to declare a method that expects an array of exactly four `int` values. If a method parameter is of type `int[]`, a caller can pass an array with any number (including zero) of elements.

To create an array value in Java, you use the `new` keyword, just as you do to create an object. Array types don't have constructors, but you are required to specify a length whenever you create an array. Specify the desired size of your array as a nonnegative integer between square brackets:

```
// Create a new array to hold 1024 bytes
byte[] buffer = new byte[1024];
// Create an array of 50 references to strings
String[] lines = new String[50];
```

When you create an array with this syntax, each of the array elements is automatically initialized to the same default value that is used for the fields of a class: `false` for `boolean` elements, `\u0000` for `char` elements, `0` for integer elements, `0.0` for floating-point elements, and `null` for elements of reference type.

Array creation expressions can also be used to create and initialize a multidimensional array of arrays. This syntax is somewhat more complicated and is explained later in this section.

80

The elements of an array are variables. When an array element appears in an expression, it evaluates to the value held in the element. And when an array element appears on the lefthand side of an assignment operator, a new value is stored into that element. Unlike a normal variable, however, an array element has no name, only a number. Array elements are accessed using a square bracket notation. If `a` is an expression that evaluates to an array reference, you index that array and refer to a specific element with `a[i]`, where `i` is an integer literal or an expression that evaluates to an `int`. For example:

```
// Create an array of two strings
String[] responses = new String[2];
responses[0] = "Yes";  // Set the first element of the array
responses[1] = "No";   // Set the second element of the array

// Now read these array elements
System.out.println(question + " (" + responses[0] + "/" +
                   responses[1] + " ): ");

// Both the array reference and the array index may be more complex
double datum = data.getMatrix()[data.row() * data.numColumns() +
                   data.column()];
```

The array index expression must be of type `int`, or a type that can be widened to an `int`: `byte`, `short`, or even `char`. It is obviously not legal to index an array with a `boolean`, `float`, or `double` value. Remember that the `length` field of an array is an `int` and that arrays may not have more than `Integer.MAX_VALUE` elements. Indexing an array with an expression of type `long` generates a compile-time error, even if the value of that expression at runtime would be within the range of an `int`.

Remember that the first element of an array `a` is `a[0]` , the second element is `a[1]`, and the last is `a[a.length-1]`.

A common bug involving arrays is use of an index that is too small (a negative index) or too large (greater than or equal to the array `length`). In languages like C or C++, accessing elements before the beginning or after the end of an array yields unpredictable behavior that can vary from invocation to invocation and platform to platform. Such bugs may not always be caught, and if a failure occurs, it may be at some later time. While it is just as easy to write faulty array indexing code in Java, Java guarantees predictable results by checking every array access at runtime. If an array index is too small or too large, Java immediately throws an `ArrayIndexOutOfBoundsException`.

Excerpts from Flanagan, D. (2019). *Java in a Nutshell*. Sebastopol, CA. O'Reilly.