DEFINING METHODS IN JAVA

We already understand how to invoke instance methods such as `add`, `pause`, and `getHeight`. Now we'll turn to learning how to define our own methods, creating our own names for them, to do things that we want.

Why would you ever want to create new methods? Sometimes a program becomes unmanageably long, and methods are helpful for breaking it up into bite-sized pieces. And sometimes you want a program to do the same thing in several very different places of a program. In either case, defining a new method can be helpful.

Defining methods

Before defining a method in Java, you first must answer four questions.

- What will the method do? Generally speaking, you should be able to summarize the purpose of a method within a brief sentence.
- What name will the method have? The same rules apply to method names as to variable names. Conventionally, the first word is not capitalized, but subsequent words are, as in `getHeight` and `setFillColor`. Normally, method names are a short summary of the method's purpose, with one to three words, starting with a verb.
- What parameters will the method require, and what will the parameters' types be? For example, the `pause` method requires one parameter, an **`int`**. We'll address parameters later in this chapter (Section 12.2). For this section, we will deal only with methods that require no parameters.
- What type of value, if any, will the method return in response? This is called the method's return value. We have seen some methods that have return values, such as`getHeight`, which responds with an **`int`** value. Some methods do not have return values, such as `add` and `setFilled`; for these methods, we use the special word **`void`** to describe the type of value returned.

The definition of a method follows the following template.

```
public <returnType> <methodName>() {
    <bodyOfMethod>
}
```

Methods must be defined directly within a class definition; Java does not permit nesting one method definition within another method's definition. The order in which methods are listed within the class is not important.

Actually, we have been writing a method in each of our programs all along, using the following:

```
public void run() {
    // body of program
}
```

As you can see by the word `void`, this method doesn't return anything. The method name `run` is treated specially by the `TurtleProgram` and `GraphicsProgram` classes we have been using: As soon as the window is created, these classes invoke the method named `run` to execute the body of the program.

Let us look at a problem where defining an additional method beyond `run` would be useful. In particular, suppose we want to modify the `MovingBalloon` program of Figure 6.4, so that it animates *two* hot air balloons descending to the ground. One way to create two balloons is to simply duplicate all of the code for creating one balloon a second time. But this results in a long program, and the duplication of code makes it more difficult to change the program. For example, we might decide later to enhance the two balloons in some way, such as adding passengers into their baskets. And if we wanted to have an entire balloon festival with several balloons landing, then retyping the balloon creation code for each balloon would be even more problematic.

So we decide to write a method, and we begin by answering our four questions. Our new method's purpose, in a sentence, is to create an object representing a hot air balloon. For a name, we'll use the sensible `createBalloon`. This method will not require any parameters when it is invoked, and it will return in response a `GCompound` object that combines the different components of a balloon into one object.

Having answered our four questions about the method's design, we can turn to writing our program, which appears in Figure 12.1.

Figure 12.1: The `MultipleBalloons` program.

```
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class MultipleBalloons extends GraphicsProgram {
    public void run() {
        GCompound eastbound = createBalloon();
        add(eastbound, 10, 10);
        GCompound westbound = createBalloon();
        add(westbound, getWidth() - westbound.getWidth() - 10, 60);
11
12         while(westbound.getY() + westbound.getHeight() < getHeight()) {
13             pause(40);
14             eastbound.move( 1, 1);
15             westbound.move(-1, 1);
16         }
17         while(eastbound.getY() + eastbound.getHeight() < getHeight()) {
18             pause(40);
19             eastbound.move( 1, 1);
20         }
21     }
22
23     public GCompound createBalloon() {
24         GOval ball = new GOval(0, 0, 50, 50);
25         ball.setFilled(true);
26         ball.setFillColor(new Color(208, 48, 48));
27
28         GRect basket = new GRect(15, 60, 20, 10);
29         basket.setFilled(true);
30         basket.setFillColor(new Color(224, 192, 0));
31
32         GCompound balloon = new GCompound();
33         balloon.add(ball);
34         balloon.add(new GLine(3, 39, 15, 60));  // left line
35         balloon.add(new GLine(48, 39, 35, 60)); // right line
36         balloon.add(basket);
37         return balloon;
38     }
39 }
```

This program defines two methods, the `run` method (line 6) and the `createBalloon` method (line 23). Since `createBalloon` is meant to respond with a `GCompound` value, line 23 mentions that the `createBalloon` method returns a `GCompound` value.

Also, notice line 37 at the end of `createBalloon`'s body. This is a new category of statement that we haven't seen before, the **`return`** statement.

```
return <returnValue>;
```

When the computer executes a method, it will execute the method's body until it reaches a **`return`** statement. Once it reaches such a statement, the computer immediately halts executing the method and uses the value designated after the word `return` as the value returned by the method. Most frequently, each method has just one `return` statement, at its end. But some methods may have other `return` statements, most often nested in `if` statements, to indicate that the method should return prematurely. A method might have this because of an easy request where the method can complete early without executing all of the program

For methods that have a `void` return type, methods can omit the `return` statement altogether, and the computer stops executing the method once it goes off the end of the method's body; this is what we've been doing all along with our `run` methods. If we run into a situation where we want to return early from a method that has a `void` return type, we simply omit listing any return value.

```
return;
```

Methods whose return type is not `void` are *required* to include a `return` statement specifying what to return.

When the computer executes the `MultipleBalloons` program, it proceeds as follows.

- The computer invokes the `run` method just after creating the window and begins executing the body of that method.
- The first thing the `run` method says to do (line 7) is to invoke the `createBalloon` method. At this time, it suspends its work on the `run` method in order to complete`createBalloon`. So, after glancing at line 7, it proceeds to line 24.
- It begins executing statements from line 7 until it reaches the `return` statement in line 37. The expression following `return` in line 37 says to return the `GCompound` object to which `balloon` refers — the `GCompound` which happens to combine all the shapes into a balloon.
- After reaching the `return` statement in `createBalloon`, the computer resumes its work on the `run` method from where it left off, line 7. Line 7 says to assign `eastbound` to refer to the `GCompound` returned by `createBalloon`, and line 8 says to place that `GCompound` object into the window at coordinates (10, 10).
- When the computer reaches line 9, it sees that it must invoke `createBalloon` again to see what the method will return this time. So suspends its work on `run` once more, goes to line 24, and proceeds until it reaches the `return` statement of line 37, which says to return this second `GCompound` object created in line 32. After completing`createBalloon`, the computer resumes its execution of `run` from where it left off, assigning `westbound` to refer to the `GCompound` object returned.
- The computer continues executing the `run` method, placing the second `GCompound` object on the right side of the screen, and proceeding to animate both balloons' descent to the window's bottom.

Source : http://www.toves.org/books/java/ch12-methods/index.html