Programming via Java Defining methods

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.

12.1. 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 ofFigure 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.

```
1  import acm.program.*;
2  import acm.graphics.*;
3  import java.awt.*;
4
5  public class MultipleBalloons extends GraphicsProgram {
6      public void run() {
7          GCompound eastbound = createBalloon();
8          add(eastbound, 10, 10);
```

```
9           GCompound westbound = createBalloon();
10          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.

12.2. Parameters

Now suppose we want to modify our `MultipleBalloons` program so that the balloons have different colors. The best way to accomplish this is to use *parameters* for passing additional information — in this case, a color — into the method.

To write a method that takes a parameter, you list the type and name of the parameter in parentheses.

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

For example, if we decide to name our parameter `balloonColor`, we would modify our `createBalloon` method declaration to be:

```
public GCompound createBalloon(Color balloonColor) {
```

The name `balloonColor` will be the name of a variable available within `createBalloon`, which will refer to whatever value is designated when invoking the method. The first few lines of `createBalloon` will be the following.

```
GOval ball = new GOval(0, 0, 50, 50);
ball.setFilled(true);
ball.setFillColor(balloonColor);
```

Notice how in the last line, we use the `balloonColor` variable.

When we invoke the method within `run`, we will need to include the balloon's color in parentheses. For example, the `run` method might start with the following lines.

```
GCompound eastbound = createBalloon(new Color(255, 0, 0));
add(eastbound, 10, 10);
GCompound westbound = createBalloon(new Color(0, 0, 255));
add(westbound, getWidth() - westbound.getWidth() - 10, 60);
```

This will end up executing `createBalloon` the first time with the `balloonColor` variable referring to the red color; and the second time with the `balloonColor` variable referring to the blue color. As a result, the balloon proceeding southeast will be a red balloon, while the balloon proceeding southwest will be blue.

If you want a method with multiple parameters, list each parameter's type and name in the method declaration's parentheses, separated by commas. For example, if we want parameters for customizing the new balloon's height and basket color as well, we would want two additional parameters to `createBalloon`.

```
public GCompound createBalloon(Color balloonColor, int height,
        Color basketColor) {
```

Java permits defining multiple methods with the same name, as long as they can be distinguished based on their parameters: That is, any two methods of the same name must have a different number of parameters — or, if they have the same number of parameters, one of the methods must have a parameter whose type is incompatible with the other method's parameter in the same position.

Having multiple methods of the same name is useful when you want multiple methods that do very similar things, but we don't want to have to remember two different names. For example, if we have our `createBalloon` method where the balloon color, height, and basket color can all be specified as parameters, we may still want a `createBalloon` method where only the balloon color is specified. We can accomplish this by adding the following method.

```
public GCompound createBalloon(Color balloonColor) {
    return createBalloon(balloonColor, 70, new Color(224, 192, 0));
}
```

When we want to create a balloon with the default height and basket color, we can just invoke this one-parameter method, and it will promptly invoke the three-parameter `createBalloon` method, including the default height of 70, and the default basket color of tan. The **`return`** statement in the one-parameter `createBalloon`says that the method should return whichever `GCompound` object that the three-parameter `createBalloon` method returns.

12.3. Methods and variables

Each method has its own set of variables, completely separate from other method's variables, even if two happen to have variables with the same name. Consider the following.

```
public void run() {
    GCompound balloon;
    createBalloon();
    add(balloon);       // Illegal!
}

public void createBalloon() {
    GCompound balloon = new GCompound();
    // ... code to initialize the balloon
}
```

You might be tempted to think that when `createBalloon` assigns to its `balloon` variable, the `run` method's `balloon`'s variable would also be initialized. But these are actually two completely different variables, and so this assignment in `createBalloon` has no effect on the `balloon` variable in `run`. In fact, the compiler will reject the program, complaining that the `run` method's `balloon` variable will never have been initialized when it is used as a parameter for the `add` method.

So what can we do? If we want Method *A* to know about a value created by Method *B*, our only choices are to pass the value as a parameter (if *B* invokes *A*) or to return the value back (if *A* invokes *B*). This severely limits the amount of communication between methods, though it does accord with the good programming practice of keeping each method's purpose simple. (Actually, there is a third way for communicating values, the *instance variable*, which we will study in Chapter 14. Good programmers would not use instance variables to communicate between methods anyway.)

Similarly, when you use a variable *x* for a method's parameter *y*, a subsequent change to the variable *y* will not alter the value associated with *x*. Consider the following.

```
public void run() {
    GCompound eastbound = new GCompound();
    createBalloon(eastbound);
    add(eastbound);
}

public void createBalloon(GCompound balloon) {
    // This method accomplishes nothing!!
    balloon = new GCompound();
    balloon.add(new GOval(0, 0, 50, 50));
    balloon.add(new GRect(15, 70, 21, 10));
}
```

In `run`, we invoke `createBalloon`, passing a variable `eastbound` for the `balloon` parameter. Passing the parameter simply copies the value of `eastbound` into `balloon`, so that `balloon` and `eastbound` reference the same object. There is no link established between the variables `balloon` and `eastbound`, however: They just both happen to reference the same object.

As a result, when the `createBalloon` method assigns `balloon` to reference a *different* `GCompound` object, this has no influence on `eastbound`. The `createBalloon` method will add an oval and a rectangle into this second `GCompound` object, and `createBalloon` returns. But `eastbound` still references the first `GCompound` object (instantiated in the first line of `run`), which remains empty. Adding `eastbound` into the window simply adds an empty compound object, and the window will appear empty.

Computer scientists call this technique for passing parameters call by value: When invoking a method, the *value* designated for the parameter is copied into the parameter variable. (The primary alternative to call by value is *call by reference*, where changes to the parameter variable also affect whatever variable is specified in the parentheses. Java does not have any support for call by reference, but some other programming languages do.)

In our example above, we can repair the code by removing the line `balloon = new GCompound()` from `createBalloon`. With this line removed, the `balloon` variable in `createBalloon` continues to refer to the same `GCompound` object as `eastbound`. The remaining lines of `createBalloon` will add shapes into the this `GCompound` object, which the `run` method will add into the screen. Thus, the above fragment would display a circle and rectangle if the `balloon =...` line is omitted; but nothing will appear if that line is included.

(Even though the program could be repaired by removing the `balloon =...` line, this is not the best design. A better design would avoid having `balloon` as a parameter altogether, and to instead return the `GCompound` back to `run`, as we originally did in Figure 6.4. We've been discussing this alternative implementation just to clarify how variables interact between methods.)

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