DEFINING METHODS IN JAVA - II

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.

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