Defining Methods – Why?

In the preceding sections, we put Java to work in solving interesting problems. The demo programs grew to a healthy size, however. The `CountWords2.java` program, as an example, took about 40 lines of code and had six levels of indentation at the deepest spot. Coping with this increase in complexity is the topic of the next several sections. We'll learn how to define custom *methods* to subdivide complex programming tasks into smaller tasks that are easy to understand and easy to manage.

A method is a *function*. The term "function" has a heritage in computer science dating back several decades; however the term is not used in Java. If you have programming experience with a language like C or Pascal, and you are just learning Java, you may be more comfortable with the term "function". If you think "function" when you read "method", it may help ease you into the world of Java. Make it a personal goal, however, to wean yourself of the term "function". Once you begin to think in terms of methods — constructors, class methods, and instance methods — you are well underway to becoming a Java programmer.

All the methods presented in this chapter are examples of *class methods*, as opposed to *instance methods*. The distinction between the two was elaborated in a preceding section, but will re-surface here. We will learn to defined instance methods later when we learn to define classes.

Any programming problem that can be solved using methods, can be solved without methods. So, what's the big deal? Are methods just another of those tools, like the `break` statement or the selection operator, that exist for our convenience but aren't really necessary? Not at all. Methods are an essential Java programming tool. It would not be possible in practice to solve "large" programming problems without methods. The task would grow in complexity until it was impossible for a mere mortal to understand how all the parts worked together.

There are at least three reasons why we use methods: (i) to allow us to cope with complex problems, (ii) to hide low-level details that otherwise obscure and confuse, and (iii) to reuse portions of code. Let's examine each of these points in turn.

Complexity

As programs gain in complexity, they generally gain in size, and, sooner or later, they become unruly. When confronting hundreds of lines of code, it is easy to confuse how pieces fit together in achieving an overall goal. One of most powerful techniques to cope with complexity in computer programs is affectionately known as "divide and conquer" — that is, to take a complex task and divide it into smaller, more easily understood tasks. In Java, those "smaller, more easily understood tasks" are implemented as *methods*. Once a method is defined, it is "put aside" and used. The task of the method is thereafter "solved", and we needn't concern ourselves with the details any longer.

Hiding Details

Another important use of methods is to create "black boxes". Once a method is defined, it can be used in a program. At the level of using it, we needn't concern ourselves with how the method's task is performed. The details are hidden, and that's just great because our thoughts are focused "higher up" — in solving a big problem that exploits the solutions to small problems in achieving a goal. This is analogous to management hierarchies in corporations: Senior managers delegate tasks to junior managers because solving those tasks themselves is distracting. When we use a method, we are "delegating" the task of solving a low-level problem to a method. To solve it

oneself is distracting. We are, in a sense, treating the method as a black box, because we accept the result without concern for the details. This is illustrated in Figure 1.

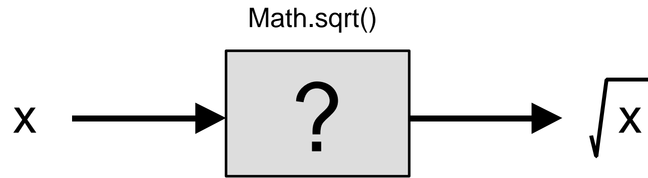

Figure 1. A method as a black box

The question mark in Figure 1 emphasizes that the details of calculating a square root are hidden from us. Delegating tasks to methods is a great luxury, and we want to exploit it to the fullest. The `Math.sqrt()` method is an excellent example because it is a common task that we call on frequently in solving high-level problems, such as finding the length of the hypotenuse (*z*) of a right-angle triangle:

$$z = \sqrt{x^2 + y^2}$$

The low-level details of calculating a square root are hidden, and that's fine because we want to focus our efforts higher up, in solving a problem that just happens to use a square root operation as part of the solution. We treat the `Math.sqrt()` method as a black box.

Reuse

Once a task is packaged in a method, that method is available to be accessed, or "called", from anywhere in a program. The method can be reused. It can be called more than once in a program, and it can be called from other programs. This is illustrated in Figure 2.

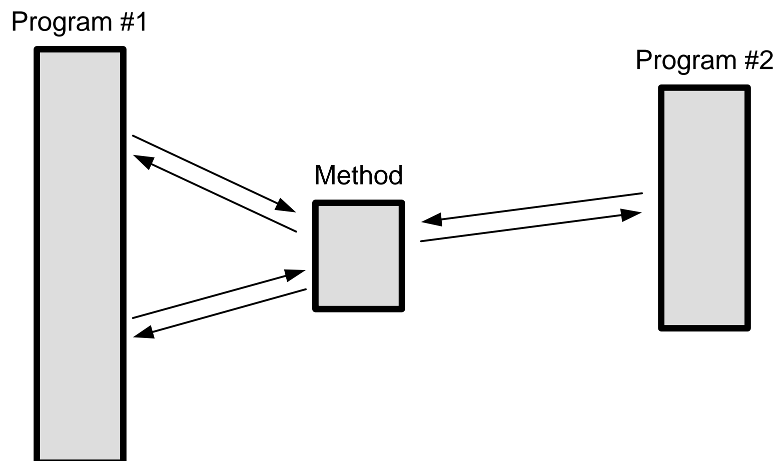

Figure 2. Method reuse

Reuse is an important concept in object-oriented programming languages like Java. The practice is sometimes called, "Write once, use many". Rather than "reinventing the wheel", we build on previous efforts or on the efforts of other programmers — we reuse what precedes us.

In the next section we'll learn how to define simple methods in the Java language.