Methods

Though this be madness, yet there is a method in't.

—William Shakespeare, *Hamlet,* ca. 1600

The methods in Java are analogous to those you created in Karel. In contrast to the topics of expressions and control statements from earlier in the week, methods require far fewer exegetical details. On the other hand, the ideas are far more important. If you plot overall importance on a scale of 1 to 10, the **`switch`** statement probably weighs in somewhere around 2; methods are definitely a 10.

The common idea that links methods in Karel and Java is that both provide a service to other, higher-level parts of the program and therefore act as tools. In both languages, the **`run`** method can call subsidiary methods to accomplish parts of the overall task. Those methods in turn call other methods that perform simpler operations, and so on. The caller views the method in terms of the effect it accomplishes. The method supplies all the details about how that operation is done. By hiding the details of complex operations, methods simplify the conceptual structure of a program considerably and allow you as a programmer to view it at varying levels of detail.

The fundamental difference between methods in Karel and their counterparts in Java is that Java makes it possible for data to pass back and forth between the caller and the method. Callers supply information to the method by supplying **arguments;** methods give information back to their callers by **returning results.** The entire process of passing this data between the two levels is in many respects the most important issue for you to understand in this context. In particular, you should take note of the following:

- Methods can be applied to other objects. In this case, the syntax of the call is

  receiver`.`*name*`(`*arguments*`)`

  The use of the receiver syntax is discussed in section 5.2.
- Arguments in the calling method are assigned to the corresponding formal parameters in the callee according to their position in the argument list. Thus, the first argument is assigned to the first parameter name, the second to the second, and so on. The names of the variables are completely irrelevant to this process.
- Arguments are copied rather than shared. If you change the value of a formal parameter, the corresponding actual argument—even if it is a variable with the same name—is unaffected.
- The **`return`** statement causes a method to return immediately to its caller and also indicates the value to be returned as a result. Other languages, particularly Pascal and Fortran, use different mechanisms for specifying the method value.

Methods can return values of any of the types you have encountered so far. Methods that return numeric data should pose no conceptual problem because this concept is familiar from high-school algebra. For some reason, methods that return strings or Boolean data seem to pose more of a pedagogical obstacle, although the basic idea is precisely the same. Methods that return Boolean values, called **predicate methods,** are very important to programming. It is worth spending some time on that section that describes them.

See the folder for programming assignment #2 for a solution to the following problem:

> Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of *n* is any divisor less than *n* itself). They called such numbers **perfect numbers**. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14. Write a predicate method **`isPerfect`** that takes an integer **`n`** and returns **`true`** if **`n`** is perfect, and **`false`** otherwise. Test your implementation by writing a main program that uses the **`isPerfect`** method to list the perfect numbers between 1 and some user-specified bound.