The question is how to change the computation of the right answer. It would be great if we could say:

```
int answer = x text y;
```

If Java somehow filled in "+" or "*" appropriately and then used the corresponding operator for the computation, then it would work. But Java doesn't work this way. The way that we usually get around this in Java is by introducing an `if/else` structure that tests whether the string is a plus or an asterisk. But then the code works for only those two operators, and additional branches must be added to the code later to make it support subtraction and other operations.

What we really want is the ability to pass an additional parameter that specifies the calculation to perform. We want to say, "Use the addition operation the first time and the multiplication operation the second time." A functional programmer would say that what we want to be able to do is to pass in a function. This is an example of what we mean by elevating functions to first class status in the language. We want to be able to introduce a fourth parameter that specifies the function to use for computing the right answer. That requires thinking of the function as a thing in the language that can be passed as a parameter.

Lambda Expressions

Java 8 provides a nice mechanism for doing exactly that. We can form a *lambda expression*.

> **Lambda Expression (Lambda)**
>
> An expression that describes a function by specifying its parameters and the value that it returns.

The term "lambda" was coined by a logician named Alonzo Church in the 1930s. The term is used consistently across many programming languages, so it is worth becoming familiar with it. The Python programming language, for example, uses "lambda" as a keyword for forming this type of anonymous function.

Lambda expressions are formed in Java by specifying the parameters of the function and an expression that represents the value to return separated by the special operator "->".

```
<parameters> -> <expression>
```

For example, we can use the following lambda expression to represent a function that adds together two arguments:

```
(int x, int y) -> x + y
```

Notice that the parameters are enclosed in parentheses. In reading this expression, we typically describe it as, "Given the parameters x and y of type int, we return x + y."

We can also write this as a method with a name, as in:

```
public static int sum(int x, int y) {
    return x + y;
}
```

Notice how the lambda expression takes the parenthesized parameter list from the method header and the expression used in the return statement to form a simple expression. Once you get used to reading lambda expressions, you will find that it is a concise way to read and reason about the underlying computation being performed.

It is also often possible to eliminate the types for the parameters because they can usually be inferred by the surrounding context. For our sample code, we will be able to use this lambda expression to describe addition:

```
(x, y) -> x + y
```

And this expression to describe multiplication:

```
(x, y) -> x * y
```

Given this new option, we can rewrite our client code as follows to perform 3 each of addition and multiplication problems.

```
Scanner console = new Scanner(System.in);
giveProblems(console, 3, "+", (x, y) -> x + y);
giveProblems(console, 3, "*", (x, y) -> x * y);
```

Below is a sample log of execution.

```
9 + 1 = 10
you got it right
4 + 4 = 8
you got it right
6 + 2 = 9
incorrect...the answer was 8
2 of 3 correct

10 * 11 = 110
you got it right
9 * 6 = 64
incorrect...the answer was 54
5 * 7 = 45
incorrect...the answer was 35
1 of 3 correct
```

This ability to pass a lambda expression as a parameter points out the benefit of treating functions as first-class elements of the language. Just as we can provide a different number of problems to perform or a different text to use for displaying the problems, we can also provide a different function for computing the right answer. This is a much more flexible approach than having to write tedious `if/else` constructs that say exactly what to do for each different possibility. Instead we provide a simple definition of the function we want to use and the function is stored in a parameter of the method.

It is more important that you learn how to become a client of these functional programming features of Java than to learn how to implement them yourself. But for those who are interested in seeing the implementation, the following is the revised method code:

```
public static void giveProblems(Scanner console, int numProblems,
                                String text, IntBinaryOperator operator) {
    Random r = new Random();
    int numRight = 0;
    for (int i = 1; i <= numProblems; i++) {
        int x = r.nextInt(12) + 1;
        int y = r.nextInt(12) + 1;
        System.out.print(x + " " + text + " " + y + " = ");
        int answer = operator.applyAsInt(x, y);
        int response = console.nextInt();
        if (response == answer) {
            System.out.println("you got it right");
            numRight++;
        } else {
            System.out.println("incorrect...the answer was " + answer);
        }
    }
    System.out.println(numRight + " of " + numProblems + " correct");
    System.out.println();
}
```

There are other variations on lambda expression syntax. For example, if a lambda expression accepts only a single parameter, the parentheses around it are not required. The following is a lambda expression that accepts an integer and returns that integer plus 1:

```
n -> n + 1
```

Another syntax variation is that if the computation is not a simple expression, you can include multiple statements enclosed in curly braces, such as:

```
x -> { int z = x * x; System.out.println(z); return z; }
```

Our discussion of first-class functions is a little generous to Java because it turns out that functions in Java 8 are not truly first-class. The language designers have done some fancy work behind the scenes to make it feel like Java has first-class functions, but they aren't really first-class because you can't do basic things like storing them directly in a variable. Instead Java takes advantage of interfaces that have a single abstract method in them (known as *functional interfaces*) and constructs an object for you that implements the interface's method using the elements of the lambda expression. As a result, Java's implementation of functional programming is more clunky and restrictive than in a true functional programming language. But for our purposes, the lambda expressions and functional interfaces act enough like first-class functions that we can explore the concept, even though it is a bit of an illusion.

19.3 Streams

You don't truly appreciate the benefits of functional programming in Java until you explore streams, which are the primary mechanism that Java provides for this style of programming. We have seen this concept before when we studied files in Chapter 6. We saw input streams that are a source of data and output streams that are a destination for results. The streams in this chapter are a generalization of that idea. Oracle describes a Java stream as a sequence of elements of data on which various functional programming operations can be performed.

Stream

A sequence of elements from a data source that supports aggregate operations.

Basic Idea

The best way to think about a stream is to visualize it as a flow of data from a source to a terminator with possible modifiers in between, as shown in Figure 19.1.

There is always one source and one terminator, but there can be any number of modifiers (including none) in between. As the diagram indicates, think of each modifier as transforming the stream in some way. One sequence of values flows in and a different sequence of values flows out. This way, we solve a complex programming task by identifying the source of the data to process, the final result we want to compute, and a series of transformations in between that move us closer to completing the task. Each of the modifiers will be specified by a function, which means that we

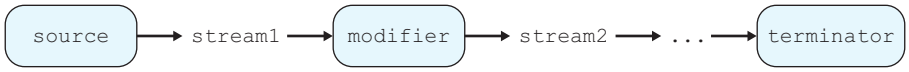

Figure 19.1 Streams

are decomposing the overall task into a series of subtasks that each involve a single transformation specified by a function.

As a first example, suppose we want to find the sum of the squares of the integers 1 through 5. We could use a classic cumulative sum to accomplish this:

```
// compute the sum of the squares of integers 1-5
int sum = 0;
for (int i = 1; i <= 5; i++) {
    sum = sum + i * i;
}
```

This code specifies exactly how to perform this computation, using a loop variable called `i` that varies from 1 to 5 and accumulating the final answer in a variable called `sum`. You will see that when we use streams, we describe more *what* we want computed instead of specifying *how* to compute it. This can make the coding itself simpler, but more importantly, it gives the computer more flexibility to decide how to implement the computation. As we will see in the case study at the end of this chapter, this can allow the computer to optimize the solution to run faster.

Using a stream approach, we first have to identify a source of data. We don't have a convenient source for the squares of the positive integers, but there is a static method called `IntStream.range` that produces a stream of sequential integers in a particular range. As with the `substring` method of the `String` class, the `range` method has a first parameter that is inclusive and a second parameter that is exclusive. So we will make the call `IntStream.range(1, 6)` to produce a stream with the integers [1, 2, 3, 4, 5]. There is a variation of the method called `rangeClosed` that would allow us to pass `(1, 5)` as parameters, but the `range` method uses the same convention we have studied in Java for substrings and it is also more commonly used in other programming languages. Python, for example, has a `range` function that works the same way.

We also need to pick an appropriate terminator. In this case Java provides one for us in the form of a method called `sum` that adds up the values in a stream of numbers. For now, let's just add up the integers and store the result in a variable. So we would write this line of code:

```
int sum = IntStream.range(1, 6).sum();
```

This sets the variable `sum` to 15 (1 + 2 + 3 + 4 + 5). In this case, we have the required source of data and the required terminator, but there are no modifications along the way. Figure 19.2 shows a diagram of what is going on.

```
IntStream.range(1, 6) -> [1, 2, 3, 4, 5] -> sum -> 15
```

Figure 19.2 Stream operations on range of integers