17.3 Mapping and Lambdas

[This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 5, Control Statements: Part 2; Logical Operators.]

The preceding example specified a stream pipeline containing only a data source and a terminal operation that produced a result. Most stream pipelines also contain **intermediate operations** that specify tasks to perform on a stream's elements before a terminal operation produces a result.

In this example, we introduce a common intermediate operation called **mapping**, which transforms a stream's elements to new values. The result is a stream with the same number of elements containing the transformation's results. Sometimes the mapped elements are of different types from the original stream's elements.

To demonstrate mapping, let's revisit the program of Fig. 5.5 in which we calculated the sum of the even integers from 2 through 20 using external iteration, as follows:

```
int total = 0;

for (int number = 2; number <= 20; number += 2) {
   total += number;
}
```

Figure 17.4 reimplements this task using streams and internal iteration.

```
1   // Fig. 17.4: StreamMapReduce.java
2   // Sum the even integers from 2 through 20 with
3   import java.util.stream.IntStream;
4
5   public class StreamMapReduce {
6      public static void main(String[] args) {
7         // sum the even integers from 2 through 20
8         System.out.printf("Sum of the even ints fr
9            IntStream.rangeClosed(1, 10)
10                  .map((int x) -> {return x * 2;
11                  .sum());
12      }
13   }
```

```
Sum of the even ints from 2 through 20 is: 110
```

Fig. 17.4

Sum the even integers from 2 through 20 with `IntStream`.

The stream pipeline in lines 9–11 performs three chained method calls:

- Line 9 creates the data source—an `IntStream` containing the elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` and `10`.
- Line 10, which we'll discuss in detail momentarily, performs a processing step that maps each element (`x`) in the stream to that element multiplied by

2. The result is a stream of the even integers `2`, `4`, `6`, `8`, `10`, `12`, `14`, `16`, `18` and `20`.

- Line 11 reduces the stream's elements to a single value—the `sum` of the elements. This is the terminal operation that initiates the pipeline's processing, then `sum`s the stream's elements.

The new feature here is the mapping operation in line 10, which in this case multiplies each stream element by 2. `IntStream` method `map` receives as its argument (line 10)

```
   (int x) -> {return x * 2;}
```

which you'll see in the next section is an alternate notation for "a *method* that receives an `int` parameter `x` and returns that value multiplied by `2`." For each element in the stream, `map` calls this method, passing to it the current stream element. The method's return value becomes part of the new stream that `map` returns.

17.3.1 Lambda Expressions

As you'll see throughout this chapter, many intermediate and terminal stream operations receive methods as arguments. Method `map`'s argument in line 10

```
   (int x) -> {return x * 2;}
```

is called a **lambda expression** (or simply a **lambda**), which

represents an *anonymous method*—that is, a *method without a name*. Though a lambda expression's syntax does not look like the methods you've seen previously, the left side does look like a method parameter list and the right side does look like a method body. We explain the syntax details shortly.

Lambda expressions enable you to create methods that can be treated as data. You can:

- pass lambdas as arguments to other methods (like `map`, or even other lambdas)
- assign lambda expressions to variables for later use and
- return lambda expressions from methods.

You'll see that these are powerful capabilities.

Software Engineering Observation 17.4

Lambdas and streams enable you to combine many benefits of functional-programming techniques with the benefits of object-oriented programming.

17.3.2 Lambda Syntax

A lambda consists of a *parameter list* followed by the **arrow token (`->`)** and a body, as in:

```
(parameterList) -> {statements}
```

The lambda in line 10

```
(int x) -> {return x * 2;}
```

receives an `int`, multiplies its value by `2` and returns the result. In this case, the body is a *statement block* that may contain statements enclosed in curly braces. The compiler *infers* from the lambda that it returns an `int`, because the parameter `x` is an `int` and the literal `2` is an `int`—multiplying an `int` by an `int` yields an `int` result. As in a method declaration, lambdas specify parameters in a comma-separated list. The preceding lambda is similar to the method

```
int multiplyBy2(int x) {
   return x * 2;
}
```

but the lambda does not have a name and the compiler infers its return type. There are several variations of the lambda syntax.

Eliminating a Lambda's Parameter Type(s)

A lambda's parameter type(s) usually may be omitted, as in:

```
(x) -> {return x * 2;}
```

in which case, the compiler infers the parameter and return types by the lambda's context—we'll say more about this later. If for any reason the compiler cannot infer the parameter or return types (e.g., if there are multiple type possibilities), it generates an error.

Simplifying the Lambda's Body

If the body contains only one expression, the `return` keyword, curly braces and semicolon may be omitted, as in:

```
(x) -> x * 2
```

In this case, the lambda *implicitly* returns the expression's value.

Simplifying the Lambda's Parameter List

If the parameter list contains only one parameter, the

parentheses may be omitted, as in:

```
x -> x * 2
```

Lambdas with Empty Parameter Lists

To define a lambda with an empty parameter list, use empty parentheses to the left of the arrow token (`->`), as in:

```
() -> System.out.println("Welcome to lambdas!")
```

Method References

In addition, to the preceding lambda-syntax variations, there are specialized shorthand forms of lambdas that are known as *method references*, which we introduce in Section 17.6.

17.3.3 Intermediate and Terminal Operations

In the stream pipeline shown in lines 9–11, `map` is an intermediate operation and `sum` is a terminal operation.

Method `map` is one of many intermediate operations that specify tasks to perform on a stream's elements.

Lazy and Eager Operations

Intermediate operations use **lazy evaluation**—each intermediate operation results in a new stream object, but does not perform any operations on the stream's elements until a terminal operation is called to produce a result. This allows library developers to optimize stream-processing performance. For example, if you have 1,000,000 `Person` objects and you're looking for the *first* one with the last name `"Jones"`, rather than processing all 1,000,000 elements, stream processing can terminate as soon as the first matching `Person` object is found.

Performance Tip 17.1

Lazy evaluation helps improve performance by ensuring that operations are performed only if necessary.

Terminal operations are **eager**—they perform the requested operation when they're called. We say more about lazy and eager operations as we encounter them throughout the chapter. You'll see how lazy operations can improve performance in Section 17.5, which discusses how a stream pipeline's intermediate operations are applied to each stream element.

Figures 17.5 and 17.6 show some common intermediate and terminal operations, respectively.

Common intermediate stream operations	
`filter`	Returns a stream containing only the elements that satisfy a condition (known as a *predicate*). The new stream often has fewer elements than the original stream.
`distinct`	Returns a stream containing only the unique elements—duplicates are eliminated.
`limit`	Returns a stream with the specified number of elements from the beginning of the original stream.
`map`	Returns a stream in which each of the original stream's elements is mapped to a new value (possibly of a different type)—for example, mapping numeric values to the squares of the numeric values or mapping numeric grades to letter grades (A, B C, D or F). The new stream has the same number of elements as the original stream.
`sorted`	Returns a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream. We'll show how to specify both ascending and descending order.

Fig. 17.5

Common intermediate stream operations.

Common terminal stream operations	
`forEach`	Performs processing on every element in a stream (for example, display each element).
Reduction operations—*Take all values in the stream and return a single value*	

`average`	Returns the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Returns the *maximum* value in a stream.
`min`	Returns the *minimum* value in a stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (for example, a lambda that adds two elements and returns the sum).

Fig. 17.6

Common terminal stream operations.