17.2 Streams and Reduction

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

In counter-controlled iteration, you typically determine *what* you want to accomplish then specify precisely *how* to accomplish it using a `for` loop. In this section, we'll investigate that approach, then show you a better way to accomplish the same tasks.

17.2.1 Summing the Integers from 1 through 10 with a `for` Loop

Let's assume that *what* you want to accomplish is to sum the integers from 1 through 10. In Chapter 5, you saw that you can do this with a counter-controlled loop:

```
int total = 0;

for (int number = 1; number <= 10; number++) {
   total += number;
```

```
   }
```

This loop specifies precisely *how* to perform the task—with a `for` statement that processes each value of control variable `number` from 1 through 10, adding `number`'s current value to `total` once per loop iteration and incrementing number after each addition operation. This is known as **external iteration**, because *you* specify all the iteration details.

17.2.2 External Iteration with `for` Is Error Prone

Let's consider potential problems with the preceding code. As implemented, the loop requires two variables (`total` and `number`) that the code *mutates* (that is, modifies) during each loop iteration. Every time you write code that modifies a variable, it's possible to introduce an error into your code. There are several opportunities for error in the preceding code. For example, you could:

- initialize the variable `total` incorrectly
- initialize the `for` loop's control variable `number` incorrectly
- use the wrong loop-continuation condition
- increment control variable `number` incorrectly or
- incorrectly add each value of `number` to the `total`.

In addition, as the tasks you perform get more complicated,

understanding *how* the code works gets in the way of understanding *what* it does. This makes the code harder to read, debug and modify, and more likely to contain errors.

17.2.3 Summing with a Stream and Reduction

Now let's take a different approach, specifying *what* to do rather than *how* to do it. In Fig. 17.3, we specify only *what* we want to accomplish—that is, sum the integers from 1 through 10—then simply let Java's `IntStream` class (package `java.util.stream`) deal with *how* to do it. The key to this program is the following expression in lines 9–10

```
IntStream.rangeClosed(1, 10)
         .sum()
```

which can be read as, "for the stream of `int` values in the range 1 through 10, calculate the sum" or more simply "sum the numbers from 1 through 10." In this code, notice that there is neither a counter-control variable nor a variable to store the total—this is because `IntStream` conveniently defines `rangeClosed` and `sum`.

```
1  // Fig. 17.3: StreamReduce.java
2  // Sum the integers from 1 through 10 with IntSt
3  import java.util.stream.IntStream;
4
```

```
 5   public class StreamReduce {
 6      public static void main(String[] args) {
 7         // sum the integers from 1 through 10
 8         System.out.printf("Sum of 1 through 10 is:
 9            IntStream.rangeClosed(1, 10)
10                     .sum());
11      }
12   }
```

```
Sum of 1 through 10 is: 55
```

Fig. 17.3

Sum the integers from 1 through 10 with `IntStream`.

Streams and Stream Pipelines

The chained method calls in lines 9–10 create a **stream pipeline**. A **stream** is a sequence of elements on which you perform tasks, and the stream pipeline moves the stream's elements through a sequence of tasks (or *processing steps*).

Good Programming

Practice 17.1

When using chained method calls, align the dots (`.`) vertically for readability as we did in lines 9–10 of Fig. 17.3.

Specifying the Data Source

A stream pipeline typically begins with a method call that creates the stream—this is known as the *data source*. Line 9 specifies the data source with the method call

```
IntStream.rangeClosed(1, 10)
```

which creates an `IntStream` representing an ordered range of `int` values.

Here, we use the `static` method `rangeClosed` to create an `IntStream` containing the ordered sequence of `int` elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` and `10`. The method is named `rangeClosed`, because it produces a *closed range* of values—that is, a range of elements that includes *both* of the method's arguments (`1` and `10`). `IntStream` also provides method `range`, which produces a *half-open range* of values from its first argument up to, but not including, its second argument—for example,

```
IntStream.range(1, 10)
```

produces an `IntStream` containing the ordered sequence of `int` elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` and `9`, but *not* 10.

Calculating the Sum of the `IntStream`'s Elements

Next, line 10 completes the stream pipeline with the processing step

```
   .sum()
```

This invokes the `IntStream`'s `sum` instance method, which returns the sum of all the `int`s in the stream—in this case, the sum of the integers from 1 through 10.

The processing step performed by method `sum` is known as a **reduction**—it reduces the stream of values to a *single* value (the sum). This is one of several predefined `IntStream` reductions—Section 17.7 presents the predefined reductions `count`, `min`, `max`, `average` and `summaryStatistics`, as well as the `reduce` method for defining your own reductions.

Processing the Stream Pipeline

A **terminal operation** initiates a stream pipeline's processing and produces a result. `IntStream` method `sum` is a terminal operation that produces the sum of the stream's elements. Similarly, the reductions `count`, `min`, `max`, `average`, `summaryStatistics` and `reduce` are all terminal operations. You'll see other terminal operations throughout this chapter. Section 17.3.3 discusses terminal operations in more detail.

17.2.4 Internal Iteration

The key to the preceding example is that it specifies *what* we want the task to accomplish—calculating the sum of the integers from 1 through 10—rather than *how* to accomplish it. This is an example of **declarative programming** (specifying *what*) vs. **imperative programming** (specifying *how*). We broke the goal into two simple tasks—producing the numbers in a closed range (1–10) and calculating their sum. Internally, the `IntStream` (that is, the data source itself) already knows how to perform each of these tasks. We did *not* need to specify *how* to iterate through the elements or declare and use *any* mutable variables. This is known as **internal iteration**, because `IntStream` handles all the iteration details—a key aspect of **functional programming**. Unlike external iteration with the `for` statement, the primary potential for error in line 9 of Fig. 17.3 is specifying the incorrect starting and/or ending values as arguments. Once you're used to it, stream pipeline code also can be easier to read.

Software Engineering Observation 17.2

Functional-programming techniques enable you to write higher-level code, because many of the details are implemented for you by the Java streams library. Your code becomes more concise, which improves productivity and can help you rapidly prototype programs.

Software Engineering Observation 17.3

Functional-programming techniques eliminate large classes of errors, such as off-by-one errors (because iteration details are hidden from you by the libraries) and incorrectly modifying variables (because you focus on immutability and thus do not modify data). This makes it easier to write correct programs.