17.4 Filtering

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

Another common intermediate stream operation is *filtering* elements to select those that match a condition—known as a *predicate*. For example, the following code selects the even integers in the range 1–10, multiplies each by 3 and sums the results:

```
int total = 0;

for (int x = 1; x <= 10; x++) {
   if (x % 2 == 0) { // if x is even
      total += x * 3;
   }
}
```

Figure 17.7 reimplements this loop using streams.

```
// Fig. 17.7: StreamFilterMapReduce.java
// Triple the even ints from 2 through 10 then s
import java.util.stream.IntStream;

public class StreamFilterMapReduce {
   public static void main(String[] args) {
      // sum the triples of the even integers fr
```

```
 8          System.out.printf(
 9             "Sum of the triples of the even ints fr
10             IntStream.rangeClosed(1, 10)
11                      .filter(x -> x % 2 == 0)
12                      .map(x -> x * 3)
13                      .sum());
14       }
15    }
```

```
Sum of the triples of the even ints from 2 through 10
```

Fig. 17.7

Triple the even `int`s from 2 through 10 then sum them with `IntStream`.

The stream pipeline in lines 10–13 performs four chained method calls:

- Line 10 creates the data source—an `IntStream` for the closed range 1 through 10.
- Line 11, which we'll discuss in detail momentarily, filters the stream's elements by selecting only the elements that are divisible by 2 (that is, the even integers), producing a stream of the even integers from 2, 4, 6, 8 and 10.
- Line 12 maps each element (`x`) in the stream to that element times 3, producing a stream of the even integers from 6, 12, 18, 24 and 30.
- Line 13 reduces the stream to the sum of its elements (90).

The new feature here is the filtering operation in line 11.

`IntStream` method `filter` receives as its argument a method that takes one parameter and returns a `boolean` result. If the result is `true` for a given element, that element is included in the resulting stream.

The lambda in line 11:

```
x -> x % 2 == 0
```

determines whether its `int` argument is divisible by `2` (that is, the remainder after dividing by `2` is `0`) and, if so, returns `true`; otherwise, the lambda returns `false`. For each element in the stream, `filter` calls the method that it receives as an argument, passing to the method the current stream element. If the method's return value is `true`, the corresponding element becomes part of the intermediate stream that `filter` returns.

Line 11 creates an intermediate stream representing only the elements that are divisible by `2`. Next, line 12 uses `map` to create an intermediate stream representing the even integers (2, 4, 6, 8 and 10) that are multiplied by 3 (6, 12, 18, 24 and 30). Line 13 initiates the stream processing with a call to the *terminal* operation `sum`. At this point, the combined processing steps are applied to each element, then `sum` returns the total of the elements that remain in the stream. We discuss this further in the next section.

Error-Prevention Tip 17.1

The order of the operations in a stream pipeline matters. For example, `filter`ing the even numbers from 1–10 yields 2, 4, 6, 8, 10, then `map`ping them to twice their values yields 4, 8, 12, 16 and 20. On the other hand, `map`ping the numbers from 1–10 to twice their values yields 2, 4, 6, 8, 10, 12, 14, 16, 18 and 20, then `filter`ing the even numbers gives all of those values, because they're all even before the `filter` operation is performed.

The stream pipeline shown in this example could have been implemented by using only `map` and `sum`. Exercise 17.18 asks you to eliminate the `filter` operation.