17.5 How Elements Move Through Stream Pipelines

Section 17.3 mentioned that each intermediate operation results in a new stream. Each new stream is simply an object representing the processing steps that have been specified to that point in the pipeline. Chaining intermediate-operation method calls adds to the set of processing steps to perform on each stream element. The last stream object in the stream pipeline contains all the processing steps to perform on each stream element.

When you initiate a stream pipeline with a terminal operation, the intermediate operations' processing steps are applied for a given stream element *before* they are applied to the next stream element. So the stream pipeline in Fig. 17.7 operates as follows:

```
For each element
    If the element is an even integer
          Multiply the element by 3 and add the resul
```

To prove this, consider a modified version of Fig. 17.7's stream pipeline in which each lambda displays the intermediate operation's name and the current stream element's value:

```
IntStream.rangeClosed(1, 10)
         .filter(
            x -> {
               System.out.printf("%nfilter: %d%n", x)
               return x % 2 == 0;
            })
         .map(
            x -> {
               System.out.println("map: " + x);
               return x * 3;
            })
         .sum()
```

The modified pipeline's output below (we added the comments) clearly shows that each even integer's `map` step is applied *before* the next stream element's `filter` step:

```
filter: 1 // odd so no map step is performed for this

filter: 2 // even so a map step is performed next
map: 2

filter: 3 // odd so no map step is performed for this

filter: 4 // even so a map step is performed next
map: 4

filter: 5 // odd so no map step is performed for this

filter: 6 // even so a map step is performed next
map: 6

filter: 7 // odd so no map step is performed for this

filter: 8 // even so a map step is performed next
map: 8
```

```
filter: 9 // odd so no map step is performed for this

filter: 10 // even so a map step is performed next
map: 10
```

For the odd elements, the `map` step was *not* performed. When a `filter` step returns `false`, the element's remaining processing steps are *ignored* because that element is not included in the results. (This version of Fig. 17.7 is located in a subfolder with that example.)