17.7 `IntStream` Operations

[This section demonstrates how lambdas and streams can be used to simplify programming tasks like those you learned in Chapter 7, Arrays and `ArrayList`***s.]***

Figure 17.9 demonstrates additional `IntStream` operations on streams created from arrays. The `IntStream` techniques shown in this and the prior examples also apply to `LongStream`**s** and `DoubleStream`**s** for `long` and `double` values, respectively. We walk through the code in Sections 17.7.1–17.7.4.

```
   1  // Fig. 17.9: IntStreamOperations.java
   2  // Demonstrating IntStream operations.
   3  import java.util.Arrays;
   4  import java.util.stream.Collectors;
   5  import java.util.stream.IntStream;
   6
   7  public class IntStreamOperations {
   8     public static void main(String[] args) {
   9        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9
  10
  11        // display original values
  12        System.out.print("Original values: ");
  13        System.out.println(
  14           IntStream.of(values)
  15              .mapToObj(String::valueOf)
  16              .collect(Collectors.joining("
```

```
                    17
18          // count, min, max, sum and average of the
19          System.out.printf("%nCount: %d%n", IntStre
    20          System.out.printf("Min: %d%n",
21              IntStream.of(values).min().getAsInt());
    22          System.out.printf("Max: %d%n",
23              IntStream.of(values).max().getAsInt());
24          System.out.printf("Sum: %d%n", IntStream.o
 25          System.out.printf("Average: %.2f%n",
26              IntStream.of(values).average().getAsDou
                    27
 28          // sum of values with reduce method
29          System.out.printf("%nSum via reduce method
       30               IntStream.of(values)
31                      .reduce(0, (x, y) -> x + y));
                        32
33          // product of values with reduce method
34          System.out.printf("Product via reduce meth
       35               IntStream.of(values)
36                      .reduce((x, y) -> x * y).getAs
                        37
38          // sum of squares of values with map and s
39          System.out.printf("Sum of squares via map
       40               IntStream.of(values)
    41                        .map(x -> x * x)
        42                        .sum());
                        43
44          // displaying the elements in sorted order
45          System.out.printf("Values displayed in sor
       46               IntStream.of(values)
        47                        .sorted()
48                      .mapToObj(String::valueOf)
49                      .collect(Collectors.joining("
                  50      }
                    51   }
```

```
Original values: 3 10 6 1 4 8 2 5 9 7
```

```
                        Count: 10
                          Min: 1
                          Max: 10
                          Sum: 55
                       Average: 5.50

                Sum via reduce method: 55
           Product via reduce method: 3628800
           Sum of squares via map and sum: 385

   Values displayed in sorted order: 1 2 3 4 5 6 7 8 9 1
```

Fig. 17.9

Demonstrating `IntStream` operations.

17.7.1 Creating an `IntStream` and Displaying Its Values

`IntStream static` method `of` (line 14) receives an `int` array argument and returns an `Int-Stream` for processing the array's values. The stream pipeline in lines 14–16

```
   IntStream.of(values)
            .mapToObj(String::valueOf)
            .collect(Collectors.joining(" ")));
```

displays the stream's elements. First, line 14 creates an `IntStream` for the `values` array, then lines 15–16 use the `mapToObj` and `collect` methods as shown Fig. 17.8 to obtain a `String` representation of the stream's elements separated by spaces. We use this technique several times in this example and subsequent examples to display stream elements.

This example repeatedly creates an `IntStream` from the array `values` using:

```
IntStream.of(values)
```

You might think that we could simply store the stream and reuse it. However, once a stream pipeline is processed with a terminal operation, *the stream cannot be reused*, because it does not maintain a copy of the original data source.

17.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

Class `IntStream` provides various terminal operations for common stream reductions on streams of `int` values:

- `count` (line 19) returns the number of elements in the stream.
- `min` (line 21) returns an `OptionalInt` (package `java.util`) possibly

containing the smallest `int` in the stream. For any stream, it's possible that there are *no elements* in the stream. Returning `OptionalInt` enables method `min` to return the minimum value if the stream contains *at least one element*. In this example, we know the stream has 10 elements, so we call class `OptionalInt`'s `getAsInt` method to obtain the minimum value. If there were no *elements*, the `OptionalInt` would not contain an `int` and `getAsInt` would throw a `NoSuchElementException`. To prevent this, you can instead call method `orElse`, which returns the `OptionalInt`'s value if there is one, or the value you pass to `orElse`, otherwise.

- `max` (line 23) returns an `OptionalInt` possibly containing the largest `int` in the stream. Again, we call the `OptionalInt`'s `getAsInt` method to get the largest value, because we know this stream contains elements.
- `sum` (line 24) returns the sum of all the `int`s in the stream.
- `average` (line 26) returns an `OptionalDouble` (package `java.util`) possibly containing the average of the `int`s in the stream as a value of type `double`. In this example, we know the stream has elements, so we call class `OptionalDouble`'s `getAsDouble` method to obtain the average. If there were no *elements*, the `OptionalDouble` would not contain the average and `getAsDouble` would throw a `NoSuchElementException`. As with `OptionalInt`, to prevent this exception, you can instead call method `orElse`, which returns the `OptionalDouble`'s value if there is one, or the value you pass to `orElse`, otherwise.

Class `IntStream` also provides method `summaryStatistics` that performs the `count`, `min`, `max`, `sum` and `average` operations *in one pass* of an `IntStream`'s elements and returns the results as an `IntSummaryStatistics` object (package `java.util`). This provides a significant performance boost over reprocessing an `IntStream` repeatedly for each individual operation. This object has methods for obtaining each result

and a `toString` method that summarizes all the results. For example, the statement:

```
System.out.println(IntStream.of(values).summaryStatis
```

produces:

```
IntSummaryStatistics{count=10, sum=55, min=1, average
max=10}
```

for the array `values` in Fig. 17.9.

17.7.3 Terminal Operation `reduce`

So far, we've presented various predefined `IntStream` reductions. You can define your own reductions via an `IntStream`'s `reduce` method—in fact, each terminal operation discussed in Section 17.7.2 is a specialized implementation of `reduce`. The stream pipeline in lines 30–31

```
IntStream.of(values)
         .reduce(0, (x, y) -> x + y)
```

shows how to total an `IntStream`'s values using `reduce`,

rather than `sum`.

The first argument to `reduce` (`0`) is the operation's **identity value**—a value that, when combined with any stream element (using the lambda in the `reduce`'s second argument), produces the element's original value. For example, when summing the elements, the identity value is 0, because any `int` value added to 0 results in the original `int` value. Similarly, when getting the product of the elements the identity value is 1, because any `int` value multiplied by 1 results in the original `int` value.

Method `reduce`'s second argument is a method that receives two `int` values (representing the left and right operands of a binary operator), performs a calculation with the values and returns the result. The lambda

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

adds the values. A lambda with two or more parameters *must* enclose them in parentheses.

Error-Prevention Tip 17.2

The operation specified by a `reduce`*'s argument must be associative—that is, the order in which* `reduce` *applies the*

operation to the stream's elements must not matter. This is important, because `reduce` *is allowed to apply its operation to the stream elements in any order. A non-associative operation could yield different results based on the processing order. For example, subtraction is not an associative operation—the expression 7 – (5 – 3) yields 5 whereas the expression (7 – 5) – 3 yields –1. Associative* `reduce` *operations are critical for parallel streams (Chapter 23) that split operations across multiple cores for better performance. Exercise 23.19 explores this issue further.*

Based on the stream's elements

```
3 10 6 1 4 8 2 5 9 7
```

the reduction's evaluation proceeds as follows:

```
0 + 3 --> 3
3 + 10 --> 13
13 + 6 --> 19
19 + 1 --> 20
20 + 4 --> 24
24 + 8 --> 32
32 + 2 --> 34
34 + 5 --> 39
39 + 9 --> 48
48 + 7 --> 55
```

Notice that the first calculation uses the identity value (0) as the left operand and each subsequent calculation uses the result of the prior calculation as the left operand. The

reduction process continues producing a running total of the `IntStream`'s values until they've all been used, at which point the final sum is returned.

Calculating the Product of the Values with Method `reduce`

The stream pipeline in lines 35–36

```
   IntStream.of(values)
            .reduce((x, y) -> x * y).getAsInt()
```

uses the one-argument version of method `reduce`, which returns an `OptionalInt` that, if the stream has elements, contains the product of the `IntStream`'s values; otherwise, the `OptionalInt` does not contain a result.

Based on the stream's elements

```
   3 10 6 1 4 8 2 5 9 7
```

the reduction's evaluation proceeds as follows:

```
   3 * 10 --> 30
   30 * 6 --> 180
```

```
180 * 1 --> 180
180 * 4 --> 720
720 * 8 --> 5,760
5,760 * 2 --> 11,520
11,520 * 5 --> 57,600
57,600 * 9 --> 518,400
518,400 * 7 --> 3,628,800
```

This process continues producing a running product of the `IntStream`'s values until they've all been used, at which point the final product is returned.

We could have used the two-parameter `reduce` method, as in:

```
IntStream.of(values)
         .reduce(1, (x, y) -> x * y)
```

However, if the stream were empty, this version of `reduce` would return the identity value (`1`), which would not be the expected result for an empty stream.

Summing the Squares of the Values

Now consider summing the squares of the stream's elements. When implementing your stream pipelines, it's helpful to break down the processing steps into easy-to-understand tasks. Summing the squares of the stream's elements requires two

distinct tasks:

- squaring the value of each stream element
- summing the resulting values.

Rather than defining this with a `reduce` method call, the stream pipeline in lines 40–42

```
IntStream.of(values)
         .map(x -> x * x)
         .sum());
```

uses the `map` and `sum` methods to compose the sum-of-squares operation. First `map` produces a new `IntStream` containing the original element's squares, then `sum` totals the resulting stream's elements.

17.7.4 Sorting `IntStream` Values

In Section 7.15, you learned how to sort arrays with the `sort static` method of class `Arrays`. You also may sort the elements of a stream. The stream pipeline in lines 46–49

```
IntStream.of(values)
         .sorted()
         .mapToObj(String::valueOf)
         .collect(Collectors.joining(" ")));
```


sorts the stream’s elements and displays each value followed by a space. `IntStream` inter-mediate operation `sorted` orders the elements of the stream into *ascending* order by default. Like `filter`, `sorted` is a *lazy* operation that’s performed only when a terminal operation initiates the stream pipeline’s processing.