17.14 Streams of Random Values

Figure 6.7 summarized 60,000,000 rolls of a six-sided die using *external iteration* (a `for` loop) and a `switch` statement that determined which counter to increment. We then displayed the results using separate statements that performed external iteration. In Fig. 7.7, we reimplemented Fig. 6.7, replacing the entire `switch` statement with a single statement that incremented counters in an array—that version of rolling the die still used external iteration to produce and summarize 60,000,000 random rolls and to display the final results. Both prior versions of this example used mutable variables to control the external iteration and to summarize the results. Figure 17.24 reimplements those programs with a *single statement* that does it all, using lambdas, streams, internal iteration and *no mutable variables* to roll the die 60,000,000 times, calculate the frequencies and display the results.

Performance Tip 17.3

The techniques that `SecureRandom` *uses to produce secure random numbers are significantly slower than those used by* `Random` *(package* `java.util`*). For this reason, Fig. 17.24 may appear to freeze when you run it—on our computers, it* *took over one minute to complete. To save time, you can speed this example's execution by using class* `Random`*. However, industrial-strength applications should use secure random numbers.* Exercise 17.25 *asks you to time* Fig. 17.24*'s stream pipeline, then* Exercise 23.18 *asks you time the pipeline using parallel streams to see if the performance improvems on a multicore system.*

```java
// Fig. 17.24: RandomIntStream.java
// Rolling a die 60,000,000 times with streams
import java.security.SecureRandom;
import java.util.function.Function;
import java.util.stream.Collectors;

public class RandomIntStream {
   public static void main(String[] args) {
      SecureRandom random = new SecureRandom();

      // roll a die 60,000,000 times and summari
      System.out.printf("%-6s%s%n", "Face", "Fre
         random.ints(60_000_000, 1, 7)
               .boxed()
               .collect(Collectors.groupingBy(Funct
                  Collectors.counting()))
               .forEach((face, frequency) ->
                  System.out.printf("%-6d%d%n", fac
   }
}
```

```
Face   Frequency
1      9992993
2      10000363
3      10002272
4      10003810
5      10000321
```

```
                    6      10000241
```

Fig. 17.24

Rolling a die 60,000,000 times with streams.

Class `SecureRandom` has overloaded methods `ints`, `longs` and `doubles`, which it inherits from class `Random` (package `java.util`). These methods return an `IntStream`, a `LongStream` or a `DoubleStream`, respectively, that represent streams of random numbers. Each method has four overloads. We describe the `ints` overloads here—methods `longs` and `doubles` perform the same tasks for streams of `long` and `double` values, respectively:

- `ints()`—creates an `IntStream` for an *infinite stream* (Section 17.15) of random `int` values.
- `ints(long)`—creates an `IntStream` with the specified number of random `int`s.
- `ints(int, int)`—creates an `IntStream` for an *infinite stream* of random `int` values in the half-open range starting with the first argument and up to, but not including, the second argument.
- `ints(long, int, int)`—creates an `IntStream` with the specified number of random `int` values in the range starting with the first argument and up to, but not including, the second argument.

Line 13 uses the last overloaded version of `ints` (which we introduced in Section 17.6) to create an `IntStream` of 60,000,000 random integer values in the range 1–6.

Converting an `IntStream` to a `Stream<Integer>`

We summarize the roll frequencies in this example by collecting them into a `Map<Integer, Long>` in which each `Integer` key is a side of the die and each `Long` value is the frequency of that side. Unfortunately, Java does not allow primitive values in collections, so to summarize the results in a `Map`, we must first convert the `IntStream` to a `Stream<Integer>`. We do this by calling `IntStream` method `boxed`.

Summarizing the Die Frequencies

Lines 15–16 call `Stream` method `collect` to summarize the results into a `Map<Integer, Long>`. The first argument to `Collectors` method `groupingBy` (line 15) calls `static` method `identity` from interface `Function`, which creates a `Function` that simply returns its argument. This allows the actual random values to be used as the `Map`'s keys. The second argument to method `groupingBy` counts the number of occurrences of each key.

Displaying the Results

Lines 17–18 call the resulting `Map`’s `forEach` method to display the summary of the results. This method receives an object that implements the `BiConsumer` functional interface as an argument. Recall that for `Map`s, the first parameter represents the key and the second represents the corresponding value. The lambda in lines 17–18 uses parameter `face` as the key and `frequency` as the value, and displays the face and frequency.