17.15 Infinite Streams

A data structure, such as an array or a collection, always represents a finite number of elements—all the elements are stored in memory, and memory is finite. Of course, any stream created from a finite data structure will have a finite number of elements, as has been the case in this chapter's prior examples.

Lazy evaluation makes it possible to work with **infinite streams** that represent an unknown, potentially infinite, number of elements. For example, you could define a method `nextPrime` that produces the next prime number in sequence every time you call it. You could then use this to define an infinite stream that *conceptually* represents all prime numbers. However, because streams are lazy until you perform a terminal operation, you can use intermediate operations to restrict the total number of elements that are actually calculated when a terminal operation is performed. Consider the following pseudocode stream pipeline:

```
Create an infinite stream representing all prime numb
If the prime number is less than 10,000
Display the prime number
```

Even though we begin with an infinite stream, only the finite set of primes less than 10,000 would be displayed.

You create infinite streams with the stream-interfaces methods `iterate` and `generate`. For the purpose of this discussion, we'll use the `IntStream` version of these methods.

`IntStream` Method `iterate`

Consider the following infinite stream pipeline:

```
IntStream.iterate(1, x -> x + 1)
         .forEach(System.out::println);
```

`IntStream` method `iterate` generates an ordered sequence of values starting with the seed value (`1`) in its first argument. Each subsequent element is produced by applying to the preceding value in the sequence the `IntUnaryOperator` specified as `iterate`'s second argument. The preceding pipeline generates the infinite sequence 1, 2, 3, 4, 5, …, but this pipeline has a problem. We did not specify how many elements to produce, so this is the equivalent of an infinite loop.

Limiting an Infinite Stream's Number of Elements

One way to limit the total number of elements that an infinite

stream produces is the short-circuiting terminal operation `limit`, which specifies the maximum number of elements to process from a stream. In the case of an infinite stream, `limit` terminates the infinite generation of elements. So, the following stream pipeline

```
IntStream.iterate(1, x -> x + 1)
         .limit(10)
         .forEach(System.out::println);
```

begins with an infinite stream, but limits the total number of elements produced to `10`, so it displays the numbers from 1 through 10. Similarly, the pipeline

```
IntStream.iterate(1, x -> x + 1)
         .map(x -> x * x)
         .limit(10)
         .sum()
```

starts with an infinite stream, but sums only the squares of the integers from 1 through 10.

Error-Prevention Tip 17.4

Ensure that stream pipelines using methods that produce infinite streams `limit` *the number of elements to produce.*

`IntStream` Method `generate`

You also may create unordered infinite streams using method `generate`, which receives an `IntSupplier` representing a method that takes no arguments and returns an `int`. For example, if you have a `SecureRandom` object named `random`, the following stream pipeline generates and displays 10 random integers:

```
IntStream.generate(() -> random.nextInt())
         .limit(10)
         .forEach(System.out::println);
```

This is equivalent to using `SecureRandom`'s no-argument `ints` method (Section 17.14):

```
SecureRandom.ints()
            .limit(10)
            .forEach(System.out::println);
```