CHAPTER 1

The Java SE 8 Stream Library

In this chapter

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation. For example, suppose you want to compute the average of a certain property. You specify the source of data and the property, and the stream library can then optimize the computation, for example by using multiple threads for computing sums and counts and combining the results.

In this chapter, you will learn how to use the Java stream library, which was introduced in Java SE 8, to process collections in a "what, not how" style.

1.1 From Iterating to Stream Operations

When you process a collection, you usually iterate over its elements and do some work with each of them. For example, suppose we want to count all long words in a book. First, let's put them into a list:

```
String contents = new String(Files.readAllBytes(
   Paths.get("alice.txt")), StandardCharsets.UTF_8); // Read file into string
List<String> words = Arrays.asList(contents.split("\\PL+"));
   // Split into words; nonletters are delimiters
```

Now we are ready to iterate:

```
long count = 0;
for (String w : words)
{
   if (w.length() > 12) count++;
}
```

With streams, the same operation looks like this:

```
long count = words.stream()
   .filter(w -> w.length() > 12)
   .count();
```

The stream version is easier to read than the loop because you do not have to scan the code for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, while the loop prescribes the order of operations in complete detail, a stream is able to schedule the operations any way it wants, as long as the result is correct.

Simply changing `stream` into `parallelStream` allows the stream library to do the filtering and counting in parallel.

```
long count = words.parallelStream()
   .filter(w -> w.length() > 12)
   .count();
```

Streams follow the "what, not how" principle. In our stream example, we describe what needs to be done: get the long words and count them. We don't specify in which order, or in which thread, this should happen. In contrast, the loop at the beginning of this section specifies exactly how the computation should work, and thereby forgoes any chances of optimization.

A stream seems superficially similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

1. A stream does not store its elements. They may be stored in an underlying collection or generated on demand.
2. Stream operations don't mutate their source. For example, the `filter` method does not remove elements from a new stream, but it yields a new stream in which they are not present.
3. Stream operations are *lazy* when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of all, the `filter` method will stop filtering after the fifth match. As a consequence, you can even have infinite streams!

Let us have another look at the example. The `stream` and `parallelStream` methods yield a *stream* for the `words` list. The `filter` method returns another stream that contains only the words of length greater than twelve. The `count` method reduces that stream to a result.

This workflow is typical when you work with streams. You set up a pipeline of operations in three stages:

1. Create a stream.
2. Specify *intermediate operations* for transforming the initial stream into others, possibly in multiple steps.
3. Apply a *terminal operation* to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

In the example in Listing 1.1, the stream is created with the `stream` or `parallelStream` method. The `filter` method transforms it, and `count` is the terminal operation.

In the next section, you will see how to create a stream. The subsequent three sections deal with stream transformations. They are followed by five sections on terminal operations.

Listing 1.1 `streams/CountLongWords.java`

```java
package streams;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.List;

public class CountLongWords
{
   public static void main(String[] args) throws IOException
   {
      String contents = new String(Files.readAllBytes(
            Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
      List<String> words = Arrays.asList(contents.split("\\PL+"));

      long count = 0;
      for (String w : words)
      {
         if (w.length() > 12) count++;
      }
      System.out.println(count);

      count = words.stream().filter(w -> w.length() > 12).count();
      System.out.println(count);

      count = words.parallelStream().filter(w -> w.length() > 12).count();
      System.out.println(count);
   }
}
```

java.util.stream.Stream<T> 8

- `Stream<T> filter(Predicate<? super T> p)`

  yields a stream containing all elements of this stream fulfilling `p`.
- `long count()`

  yields the number of elements of this stream. This is a terminal operation.

java.util.Collection<E> 1.2

- `default Stream<E> stream()`
- `default Stream<E> parallelStream()`

  yields a sequential or parallel stream of the elements in this collection.

1.2 Stream Creation

You have already seen that you can turn any collection into a stream with the `stream` method of the `Collection` interface. If you have an array, use the static `Stream.of` method instead.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
   // split returns a String[] array
```

The `of` method has a varargs parameter, so you can construct a stream from any number of arguments:

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Use `Arrays.stream(array, from, to)` to make a stream from array elements between positions `from` (inclusive) and `to` (exclusive).

To make a stream with no elements, use the static `Stream.empty` method:

```
Stream<String> silence = Stream.empty();
   // Generic type <String> is inferred; same as Stream.<String>empty()
```

The `Stream` interface has two static methods for making infinite streams. The `generate` method takes a function with no arguments (or, technically, an object of the `Supplier<T>` interface). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

or a stream of random numbers as

```
Stream<Double> randoms = Stream.generate(Math::random);
```

To produce infinite sequences, such as `0 1 2 3 . . .` , use the `iterate` method instead. It takes a “seed” value and a function (technically, a `UnaryOperator<T>`) and repeatedly applies the function to the previous result. For example,

```
Stream<BigInteger> integers
   = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

The first element in the sequence is the seed `BigInteger.ZERO`. The second element is `f(seed)`, or `1` (as a big integer). The next element is `f(f(seed))`, or `2`, and so on.

NOTE: A number of methods in the Java API yield streams. For example, the `Pattern` class has a method `splitAsStream` that splits a `CharSequence` by a regular expression. You can use the following statement to split a string into words:

```
Stream<String> words = Pattern.compile("\\PL+").splitAsStream(contents);
```

The static `Files.lines` method returns a `Stream` of all lines in a file:

```
try (Stream<String> lines = Files.lines(path))
{
   Process lines
}
```

The example program in Listing 1.2 shows the various ways of creating a stream.

Listing 1.2 `streams/CreatingStreams.java`

```
package streams;

import java.io.IOException;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
import java.util.regex.Pattern;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CreatingStreams
{
   public static <T> void show(String title, Stream<T> stream)
   {
      final int SIZE = 10;
      List<T> firstElements = stream
            .limit(SIZE + 1)
            .collect(Collectors.toList());
      System.out.print(title + ": ");
      for (int i = 0; i < firstElements.size(); i++)
      {
         if (i > 0) System.out.print(", ");
         if (i < SIZE) System.out.print(firstElements.get(i));
         else System.out.print("...");
      }
      System.out.println();
   }

   public static void main(String[] args) throws IOException
   {
      Path path = Paths.get("../gutenberg/alice30.txt");
      String contents = new String(Files.readAllBytes(path),
            StandardCharsets.UTF_8);

      Stream<String> words = Stream.of(contents.split("\\PL+"));
      show("words", words);
      Stream<String> song = Stream.of("gently", "down", "the", "stream");
      show("song", song);
      Stream<String> silence = Stream.empty();
      show("silence", silence);

      Stream<String> echos = Stream.generate(() -> "Echo");
      show("echos", echos);

      Stream<Double> randoms = Stream.generate(Math::random);
      show("randoms", randoms);

      Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
            n -> n.add(BigInteger.ONE));
      show("integers", integers);

      Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(
            contents);
      show("wordsAnotherWay", wordsAnotherWay);

      try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8))
      {
         show("lines", lines);
      }
   }
}
```

***java.util.stream.Stream* 8**

- `static <T> Stream<T> of(T... values)`

  yields a stream whose elements are the given values.
- `static <T> Stream<T> empty()`

  yields a stream with no elements.
- `static <T> Stream<T> generate(Supplier<T> s)`

  yields an infinite stream whose elements are constructed by repeatedly invoking the function `s`.
- `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`

  yields an infinite stream whose elements are `seed`, `f` invoked on `seed`, `f` invoked on the preceding element, and so on.

java.util.Arrays 1.2

- `static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)` 8

  yields a stream whose elements are the specified range of the array.

java.util.regex.Pattern 1.4

- `Stream<String> splitAsStream(CharSequence input)` 8

  yields a stream whose elements are the parts of the input that are delimited by this pattern.

java.nio.file.Files 7

- `static Stream<String> lines(Path path)` 8
- `static Stream<String> lines(Path path, Charset cs)` 8

  yields a stream whose elements are the lines of the specified file, with the UTF-8 charset or the given charset.

***java.util.function.Supplier<T>* 8**

- `T get()`

  supplies a value.

1.3 The `filter`, `map`, and `flatMap` Methods

A stream transformation produces a stream whose elements are derived from those of another stream. You have already seen the `filter` transformation that yields a stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

```
List<String> wordList = . . .;
Stream<String> longWords = wordList.stream().filter(w -> w.length() > 12);
```

The argument of `filter` is a `Predicate<T>`—that is, a function from `T` to `boolean`.

Often, you want to transform the values in a stream in some way. Use the `map` method and pass the function that carries out the transformation. For example, you can transform all words to lowercase like this:

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

Here, we used `map` with a method reference. Often, a lambda expression is used instead:

```
Stream<String> firstLetters = words.stream().map(s -> s.substring(0, 1));
```

The resulting stream contains the first letters of all words.

When you use `map`, a function is applied to each element, and the result is a new stream with the results. Now, suppose you have a function that returns not just one value but a stream of values:

```
public static Stream<String> letters(String s)
{
   List<String> result = new ArrayList<>();
   for (int i = 0; i < s.length(); i++)
      result.add(s.substring(i, i + 1));
   return result.stream();
}
```

For example, `letters("boat")` is the stream `["b", "o", "a", "t"]`.

NOTE: With the `IntStream.range` method in Section 1.13, “Primitive Type Streams,” on p. 36, you can implement this method much more elegantly.

Suppose you map the `letters` method on a stream of strings:

```
Stream<Stream<String>> result = words.stream().map(w -> letters(w));
```

You will get a stream of streams, like `[. . . ["y", "o", "u", "r"], ["b", "o", "a", "t"], . . .]`. To flatten it out to a stream of letters `[. . . "y", "o", "u", "r", "b", "o", "a", "t", . . .]`, use the `flatMap` method instead of `map`:

```
Stream<String> flatResult = words.stream().flatMap(w -> letters(w))
   // Calls letters on each word and flattens the results
```

NOTE: You will find a `flatMap` method in classes other than streams. It is a general concept in computer science. Suppose you have a generic type `G` (such as `Stream`) and functions `f` from some type `T` to `G<U>` and `g` from `U` to `G<V>`. Then you can compose them—that is, first apply `f` and then `g`, by using `flatMap`. This is a key idea in the theory of *monads*. But don't worry—you can use `flatMap` without knowing anything about monads.

java.util.stream.Stream 8

- `Stream<T> filter(Predicate<? super T> predicate)`

  yields a stream containing the elements of this stream that fulfill the predicate.
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`

  yields a stream containing the results of applying `mapper` to the elements of this stream.
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`

  yields a stream obtained by concatenating the results of applying `mapper` to the elements of this stream. (Note that each result is a stream.)

1.4 Extracting Substreams and Concatenating Streams

The call *stream*`.limit(n)` returns a new stream that ends after `n` elements (or when the original stream ends, if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

yields a stream with 100 random numbers.

The call *stream*`.skip(n)` does the exact opposite: It discards the first `n` elements. This is handy when splitting text into words since, due to the way the `split` method works, the first element is an unwanted empty string. We can make it go away by calling `skip`:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

You can concatenate two streams with the static `concat` method of the `Stream` class:

```
Stream<String> combined = Stream.concat(
   letters("Hello"), letters("World"));
   // Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

Of course the first stream should not be infinite—otherwise the second one will never get a chance.

java.util.stream.Stream 8

- `Stream<T> limit(long maxSize)`

  yields a stream with up to `maxSize` of the initial elements from this stream.
- `Stream<T> skip(long n)`

  yields a stream whose elements are all but the initial `n` elements of this stream.
- `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`

  yields a stream whose elements are the elements of `a` followed by the elements of `b`.

1.5 Other Stream Transformations

The `distinct` method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The stream must obviously remember the elements that it has already seen.

```
Stream<String> uniqueWords
   = Stream.of("merrily", "merrily", "merrily", "gently").distinct();
   // Only one "merrily" is retained
```

For sorting a stream, there are several variations of the `sorted` method. One works for streams of `Comparable` elements, and another accepts a `Comparator`. Here, we sort strings so that the longest ones come first:

```
Stream<String> longestFirst =
   words.stream().sorted(Comparator.comparing(String::length).reversed());
```

As with all stream transformations, the `sorted` method yields a new stream whose elements are the elements of the original stream in sorted order.

Of course, you can sort a collection without using streams. The `sorted` method is useful when the sorting process is part of a stream pipeline.

Finally, the `peek` method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
   .peek(e -> System.out.println("Fetching " + e))
   .limit(20).toArray();
```

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by `iterate` is processed lazily.

For debugging, you can have `peek` call a method into which you set a breakpoint.

java.util.stream.Stream 8

- `Stream<T> distinct()`

  yields a stream of the distinct elements of this stream.

- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`

  yields as stream whose elements are the elements of this stream in sorted order. The first method requires that the elements are instances of a class implementing `Comparable`.

- `Stream<T> peek(Consumer<? super T> action)`

  yields a stream with the same elements as this stream, passing each element to `action` as it is consumed.

1.6 Simple Reductions

Now that you have seen how to create and transform streams, we will finally get to the most important point—getting answers from the stream data. The methods that we cover in this section are called *reductions*. Reductions are *terminal operations*. They reduce the stream to a non-stream value that can be used in your program.

You have already seen a simple reduction: The `count` method returns the number of elements of a stream.

Other simple reductions are `max` and `min` that return the largest or smallest value. There is a twist—these methods return an `Optional<T>` value that either wraps the answer or indicates that there is none (because the stream happened to be empty). In the olden days, it was common to return `null` in such a situation. But that can lead to null pointer exceptions when it happens in an incompletely tested program. The `Optional` type is a better way of indicating a missing return value. We discuss the `Optional` type in detail in the next section. Here is how you can get the maximum of a stream:

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.orElse(""));
```

The findFirst returns the first value in a nonempty collection. It is often useful when combined with filter. For example, here we find the first word that starts with the letter Q, if it exists:

```
Optional<String> startsWithQ = words.filter(s -> s.startsWith("Q")).findFirst();
```

If you are OK with any match, not just the first one, use the findAny method. This is effective when you parallelize the stream, since the stream can report any match it finds instead of being constrained to the first one.

```
Optional<String> startsWithQ = words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

If you just want to know if there is a match, use anyMatch. That method takes a predicate argument, so you won't need to use filter.

```
boolean aWordStartsWithQ = words.parallel().anyMatch(s -> s.startsWith("Q"));
```

There are methods allMatch and noneMatch that return true if all or no elements match a predicate. These methods also benefit from being run in parallel.

java.util.stream.Stream **8**

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`

  yields a maximum or minimum element of this stream, using the ordering defined by the given comparator, or an empty Optional if this stream is empty. These are terminal operations.

- `Optional<T> findFirst()`
- `Optional<T> findAny()`

  yields the first, or any, element of this stream, or an empty Optional if this stream is empty. These are terminal operations.

- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

  returns true if any, all, or none of the elements of this stream match the given predicate. These are terminal operations.

1.7 The Optional Type

An Optional<T> object is a wrapper for either an object of type T or no object. In the former case, we say that the value is *present*. The Optional<T> type is intended as a safer alternative for a reference of type T that either refers to an object or is null. However, it is only safer if you use it right. The next section shows you how.

1.7.1 How to Work with Optional Values

The key to using Optional effectively is to use a method that either *produces an alternative* if the value is not present, or *consumes the value* only if it is present.

Let us look at the first strategy. Often, there is a default that you want to use when there was no match, perhaps the empty string:

```
String result = optionalString.orElse("");
   // The wrapped string, or "" if none
```

You can also invoke code to compute the default:

```
String result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
   // The function is only called when needed
```

Or you can throw an exception if there is no value:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
   // Supply a method that yields an exception object
```

You have just seen how to produce an alternative if no value is present. The other strategy for working with optional values is to consume the value only if it is present.

The ifPresent method accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens.

```
optionalValue.ifPresent(v -> Process v);
```

For example, if you want to add the value to a set if it is present, call

```
optionalValue.ifPresent(v -> results.add(v));
```

or simply

```
optionalValue.ifPresent(results::add);
```

When calling ifPresent, no value is returned from the function. If you want to process the function result, use map instead:

```
Optional<Boolean> added = optionalValue.map(results::add);
```

Now added has one of three values: true or false wrapped into an Optional, if optionalValue was present, or an empty Optional otherwise.

NOTE: This `map` method is the analog of the `map` method of the `Stream` interface that you have seen in Section 1.3, “The `filter`, `map`, and `flatMap` Methods,” on p. 9. Simply imagine an optional value as a stream of size zero or one. The result also has size zero or one, and in the latter case, the function has been applied.

`java.util.Optional` 8

- `T orElse(T other)`

  yields the value of this `Optional`, or `other` if this `Optional` is empty.
- `T orElseGet(Supplier<? extends T> other)`

  yields the value of this `Optional`, or the result of invoking `other` if this `Optional` is empty.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`

  yields the value of this `Optional`, or throws the result of invoking `exceptionSupplier` if this `Optional` is empty.
- `void ifPresent(Consumer<? super T> consumer)`

  if this `Optional` is nonempty, passes its value to `consumer`.
- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`

  yields the result of passing the value of this `Optional` to `mapper`, provided this `Optional` is nonempty and the result is not `null`, or an empty `Optional` otherwise.

1.7.2 How Not to Work with Optional Values

If you don’t use `Optional` values correctly, you get no benefit over the “something or `null`” approach of the past.

The `get` method gets the wrapped element of an `Optional` value if it exists, or throws a `NoSuchElementException` if it doesn’t. Therefore,

```
Optional<T> optionalValue = . . .;
optionalValue.get().someMethod();
```

is no safer than

```
T value = . . .;
value.someMethod();
```

The `isPresent` method reports whether an `Optional<T>` object has a value. But

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

is no easier than

```
if (value != null) value.someMethod();
```

java.util.Optional 8

- `T get()`

  yields the value of this `Optional`, or throws a `NoSuchElementException` if it is empty.
- `boolean isPresent()`

  returns `true` if this `Optional` is not empty.

1.7.3 Creating Optional Values

So far, we have discussed how to consume an `Optional` object someone else created. If you want to write a method that creates an `Optional` object, there are several static methods for that purpose, including `Optional.of(result)` and `Optional.empty()`. For example,

```
public static Optional<Double> inverse(Double x)
{
   return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

The `ofNullable` method is intended as a bridge from possibly `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null` and `Optional.empty()` otherwise.

java.util.Optional 8

- `static <T> Optional<T> of(T value)`
- `static <T> Optional<T> ofNullable(T value)`

  yields an `Optional` with the given value. If `value` is `null`, the first method throws a `NullPointerException` and the second method yields an empty `Optional`.
- `static <T> Optional<T> empty()`

  yields an empty `Optional`.

1.7.4 Composing Optional Value Functions with `flatMap`

Suppose you have a method `f` yielding an `Optional<T>`, and the target type `T` has a method `g` yielding an `Optional<U>`. If they were normal methods, you could compose them by calling `s.f().g()`. But that composition doesn't work since `s.f()` has type `Optional<T>`, not `T`. Instead, call

```
Optional<U> result = s.f().flatMap(T::g);
```

If `s.f()` is present, then `g` is applied to it. Otherwise, an empty `Optional<U>` is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield `Optional` values. You can then build a pipeline of steps, simply by chaining calls to `flatMap`, that will succeed only when all steps do.

For example, consider the safe `inverse` method of the preceding section. Suppose we also have a safe square root:

```
public static Optional<Double> squareRoot(Double x)
{
   return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

Then you can compute the square root of the inverse as

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

or, if you prefer,

```
Optional<Double> result = Optional.of(-4.0).flatMap(MyMath::inverse).flatMap(MyMath::squareRoot);
```

If either the `inverse` method or the `squareRoot` returns `Optional.empty()`, the result is empty.

NOTE: You have already seen a `flatMap` method in the `Stream` interface (see Section 1.3, "The `filter`, `map`, and `flatMap` Methods," on p. 9). That method was used to compose two methods that yield streams, by flattening out the resulting stream of streams. The `Optional.flatMap` method works in the same way if you interpret an optional value as a stream of size zero or one.

The example program in Listing 1.3 demonstrates the `Optional` API.

Listing 1.3 `optional/OptionalTest.java`

```
package optional;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;

public class OptionalTest
{
   public static void main(String[] args) throws IOException
   {
```

(Continues)

Listing 1.3 *(Continued)*

```
      String contents = new String(Files.readAllBytes(
            Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
      List<String> wordList = Arrays.asList(contents.split("\\PL+"));

      Optional<String> optionalValue = wordList.stream()
         .filter(s -> s.contains("fred"))
         .findFirst();
      System.out.println(optionalValue.orElse("No word") + " contains fred");

      Optional<String> optionalString = Optional.empty();
      String result = optionalString.orElse("N/A");
      System.out.println("result: " + result);
      result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
      System.out.println("result: " + result);
      try
      {
         result = optionalString.orElseThrow(IllegalStateException::new);
         System.out.println("result: " + result);
      }
      catch (Throwable t)
      {
         t.printStackTrace();
      }

      optionalValue = wordList.stream()
         .filter(s -> s.contains("red"))
         .findFirst();
      optionalValue.ifPresent(s -> System.out.println(s + " contains red"));

      Set<String> results = new HashSet<>();
      optionalValue.ifPresent(results::add);
      Optional<Boolean> added = optionalValue.map(results::add);
      System.out.println(added);

      System.out.println(inverse(4.0).flatMap(OptionalTest::squareRoot));
      System.out.println(inverse(-1.0).flatMap(OptionalTest::squareRoot));
      System.out.println(inverse(0.0).flatMap(OptionalTest::squareRoot));
      Optional<Double> result2 = Optional.of(-4.0)
         .flatMap(OptionalTest::inverse).flatMap(OptionalTest::squareRoot);
      System.out.println(result2);
   }

   public static Optional<Double> inverse(Double x)
   {
      return x == 0 ? Optional.empty() : Optional.of(1 / x);
   }

   public static Optional<Double> squareRoot(Double x)
   {
      return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
   }
}
```

java.util.Optional 8

- `<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)`

  yields the result of applying `mapper` to the value of this `Optional`, or an empty `Optional` if this `Optional` is empty.

1.8 Collecting Results

When you are done with a stream, you will often want to look at its elements. You can call the `iterator` method, which yields an old-fashioned iterator that you can use to visit the elements.

Alternatively, you can call the `forEach` method to apply a function to each element:

```
stream.forEach(System.out::println);
```

On a parallel stream, the `forEach` method traverses elements in arbitrary order. If you want to process them in stream order, call `forEachOrdered` instead. Of course, you might then give up some or all of the benefits of parallelism.

But more often than not, you will want to collect the result in a data structure. You can call `toArray` and get an array of the stream elements.

Since it is not possible to create a generic array at runtime, the expression `stream.toArray()` returns an `Object[]` array. If you want an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
   // stream.toArray() has type Object[]
```

For collecting stream elements to another target, there is a convenient `collect` method that takes an instance of the `Collector` interface. The `Collectors` class provides a large number of factory methods for common collectors. To collect a stream into a list or set, simply call

```
List<String> result = stream.collect(Collectors.toList());
```

or

```
Set<String> result = stream.collect(Collectors.toSet());
```

If you want to control which kind of set you get, use the following call instead:

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));
```

Suppose you want to collect all strings in a stream by concatenating them. You can call

```
String result = stream.collect(Collectors.joining());
```

If you want a delimiter between elements, pass it to the `joining` method:

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

If you want to reduce the stream results to a sum, average, maximum, or minimum, use one of the `summarizing(Int|Long|Double)` methods. These methods take a function that maps the stream objects to a number and yield a result of type `(Int|Long|Double)SummaryStatistics`, simultaneously computing the sum, count, average, minimum, and maximum.

```
IntSummaryStatistics summary = stream.collect(
   Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

java.util.stream.BaseStream 8

- `Iterator<T> iterator()`

  yields an iterator for obtaining the elements of this stream. This is a terminal operation.

The example program in Listing 1.4 shows how to collect elements from a stream.

Listing 1.4 `collecting/CollectingResults.java`

```
package collecting;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class CollectingResults
{
   public static Stream<String> noVowels() throws IOException
   {
      String contents = new String(Files.readAllBytes(
            Paths.get("../gutenberg/alice30.txt")),
            StandardCharsets.UTF_8);
      List<String> wordList = Arrays.asList(contents.split("\\PL+"));
      Stream<String> words = wordList.stream();
      return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
   }

   public static <T> void show(String label, Set<T> set)
   {
      System.out.print(label + ": " + set.getClass().getName());
      System.out.println("["
            + set.stream().limit(10).map(Object::toString)
                  .collect(Collectors.joining(", ")) + "]");
   }

   public static void main(String[] args) throws IOException
   {
      Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10)
            .iterator();
      while (iter.hasNext())
         System.out.println(iter.next());

      Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
      System.out.println("Object array:" + numbers); // Note it's an Object[] array

      try
      {
         Integer number = (Integer) numbers[0]; // OK
         System.out.println("number: " + number);
         System.out.println("The following statement throws an exception:");
         Integer[] numbers2 = (Integer[]) numbers; // Throws exception
      }
      catch (ClassCastException ex)
      {
         System.out.println(ex);
      }

      Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10)
         .toArray(Integer[]::new);
      System.out.println("Integer array: " + numbers3); // Note it's an Integer[] array
```

(Continues)

Listing 1.4 *(Continued)*

```

        Set<String> noVowelSet = noVowels()
              .collect(Collectors.toSet());
        show("noVowelSet", noVowelSet);

       TreeSet<String> noVowelTreeSet = noVowels().collect(
              Collectors.toCollection(TreeSet::new));
        show("noVowelTreeSet", noVowelTreeSet);

        String result = noVowels().limit(10).collect(
              Collectors.joining());
        System.out.println("Joining: " + result);
        result = noVowels().limit(10)
              .collect(Collectors.joining(", "));
        System.out.println("Joining with commas: " + result);

        IntSummaryStatistics summary = noVowels().collect(
              Collectors.summarizingInt(String::length));
        double averageWordLength = summary.getAverage();
        double maxWordLength = summary.getMax();
        System.out.println("Average word length: " + averageWordLength);
        System.out.println("Max word length: " + maxWordLength);
        System.out.println("forEach:");
        noVowels().limit(10).forEach(System.out::println);
    }
}
```

java.util.stream.Stream **8**

- `void forEach(Consumer<? super T> action)`

  invokes `action` on each element of the stream. This is a terminal operation.

- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`

  yields an array of objects, or of type `A` when passed a constructor reference `A[]::new`. These are terminal operations.

- `<R,A> R collect(Collector<? super T,A,R> collector)`

  collects the elements in this stream, using the given collector. The `Collectors` class has factory methods for many collectors.

java.util.stream.Collectors 8

- `static <T> Collector<T,?,List<T>> toList()`
- `static <T> Collector<T,?,Set<T>> toSet()`

  yields collectors that collect elements in a list or set.

- `static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)`

  yields a collector that collects elements into an arbitrary collection. Pass a constructor reference such as `TreeSet::new`.

- `static Collector<CharSequence,?,String> joining()`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter)`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`

  yields a collector that joins strings. The delimiter is placed between strings, and the prefix and suffix before the first and after the last string. When not specified, these are empty.

- `static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)`
- `static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)`
- `static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)`

  yields collectors that produce an `(Int|Long|Double)SummaryStatistics` object, from which you can obtain the count, sum, average, maximum, and minimum of the results of applying `mapper` to each element.

IntSummaryStatistics 8
LongSummaryStatistics 8
DoubleSummaryStatistics 8

- `long getCount()`

  yields the count of the summarized elements.

- `(int|long|double) getSum()`
- `double getAverage()`

  yields the sum or average of the summarized elements, or zero if there are no elements.

- `(int|long|double) getMax()`
- `(int|long|double) getMin()`

  yields the maximum or minimum of the summarized elements, or `(Integer|Long|Double).(MAX|MIN)_VALUE` if there are no elements.

1.9 Collecting into Maps

Suppose you have a Stream<Person> and want to collect the elements into a map so that later you can look up people by their IDs. The Collectors.toMap method has two function arguments that produce the map's keys and values. For example,

```
Map<Integer, String> idToName = people.collect(
   Collectors.toMap(Person::getId, Person::getName));
```

In the common case when the values should be the actual elements, use Function.identity() for the second function.

```
Map<Integer, Person> idToPerson = people.collect(
   Collectors.toMap(Person::getId, Function.identity()));
```

If there is more than one element with the same key, there is a conflict, and the collector will throw an IllegalStateException. You can override that behavior by supplying a third function argument that resolves the conflict and determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, its name in your default locale (such as "German") as key, and its localized name (such as "Deutsch") as value.

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
   Collectors.toMap(
      Locale::getDisplayLanguage,
      l -> l.getDisplayLanguage(l),
      (existingValue, newValue) -> existingValue));
```

We don't care that the same language might occur twice (for example, German in Germany and in Switzerland), so we just keep the first entry.

NOTE: In this chapter, we use the Locale class as a source of an interesting data set. See Chapter 7 for more information on locales.

Now, suppose we want to know all languages in a given country. Then we need a Map<String, Set<String>>. For example, the value for "Switzerland" is the set [French, German, Italian]. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

```
Map<String, Set<String>> countryLanguageSets = locales.collect(
   Collectors.toMap(
         Locale::getDisplayCountry,
         l -> Collections.singleton(l.getDisplayLanguage()),
         (a, b) ->
            { // Union of a and b
               Set<String> union = new HashSet<>(a);
               union.addAll(b);
               return union;
            }));
```

You will see a simpler way of obtaining this map in the next section.

If you want a `TreeMap`, supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a `TreeMap`:

```
Map<Integer, Person> idToPerson = people.collect(
   Collectors.toMap(
      Person::getId,
      Function.identity(),
      (existingValue, newValue) -> { throw new IllegalStateException(); },
      TreeMap::new));
```

NOTE: For each of the `toMap` methods, there is an equivalent `toConcurrentMap` method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps. Note that elements are no longer collected in stream order, but that doesn't usually make a difference.

The example program in Listing 1.5 gives examples of collecting stream results into maps.

Listing 1.5 `collecting/CollectingIntoMaps.java`

```
package collecting;

import java.io.*;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class CollectingIntoMaps
{
   public static class Person
   {
      private int id;
      private String name;
```

(Continues)

Listing 1.5 *(Continued)*

```

      public Person(int id, String name)
      {
         this.id = id;
         this.name = name;
      }

      public int getId()
      {
         return id;
      }

      public String getName()
      {
         return name;
      }

      public String toString()
      {
         return getClass().getName() + "[id=" + id + ",name=" + name + "]";
      }
   }

   public static Stream<Person> people()
   {
      return Stream.of(new Person(1001, "Peter"), new Person(1002, "Paul"),
            new Person(1003, "Mary"));
   }

   public static void main(String[] args) throws IOException
   {
      Map<Integer, String> idToName = people().collect(
            Collectors.toMap(Person::getId, Person::getName));
      System.out.println("idToName: " + idToName);

      Map<Integer, Person> idToPerson = people().collect(
            Collectors.toMap(Person::getId, Function.identity()));
      System.out.println("idToPerson: " + idToPerson.getClass().getName()
            + idToPerson);

      idToPerson = people().collect(
            Collectors.toMap(Person::getId, Function.identity(), (
                  existingValue, newValue) -> {
               throw new IllegalStateException();
            }, TreeMap::new));
      System.out.println("idToPerson: " + idToPerson.getClass().getName()
            + idToPerson);

      Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
      Map<String, String> languageNames = locales.collect(
         Collectors.toMap(
            Locale::getDisplayLanguage,
            l -> l.getDisplayLanguage(l),
            (existingValue, newValue) -> existingValue));
      System.out.println("languageNames: " + languageNames);

      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Set<String>> countryLanguageSets = locales.collect(
         Collectors.toMap(
            Locale::getDisplayCountry,
            l -> Collections.singleton(l.getDisplayLanguage()),
            (a, b) -> { // union of a and b
               Set<String> union = new HashSet<>(a);
               union.addAll(b);
               return union;
            }));
      System.out.println("countryLanguageSets: " + countryLanguageSets);
   }
}
```

java.util.stream.Collectors 8

- `static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- `static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- `static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`
- `static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- `static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- `static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`

yields a collector that produces a map or concurrent map. The `keyMapper` and `valueMapper` functions are applied to each collected element, yielding a key/value entry of the resulting map. By default, an `IllegalStateException` is thrown when two elements give rise to the same key. You can instead supply a `mergeFunction` that merges values with the same key. By default, the result is a `HashMap` or `ConcurrentHashMap`. You can instead supply a `mapSupplier` that yields the desired map instance.

1.10 Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, and the `groupingBy` method supports it directly.

Let's look at the problem of grouping locales by country. First, form this map:

```
Map<String, List<Locale>> countryToLocales = locales.collect(
   Collectors.groupingBy(Locale::getCountry));
```

The function `Locale::getCountry` is the *classifier function* of the grouping. You can now look up all locales for a given country code, for example

```
List<Locale> swissLocales = countryToLocales.get("CH");
   // Yields locales [it_CH, de_CH, fr_CH]
```

NOTE: A quick refresher on locales: Each locale has a language code (such as `en` for English) and a country code (such as `US` for the United States). The locale `en_US` describes English in the United States, and `en_IE` is English in Ireland. Some countries have multiple locales. For example, `ga_IE` is Gaelic in Ireland, and, as the preceding example shows, my JVM knows three locales in Switzerland.

When the classifier function is a predicate function (that is, a function returning a `boolean` value), the stream elements are partitioned into two lists: those where the function returns `true` and the complement. In this case, it is more efficient to use `partitioningBy` instead of `groupingBy`. For example, here we split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(
   Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```

NOTE: If you call the `groupingByConcurrent` method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the `toConcurrentMap` method.

java.util.stream.Collectors 8

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`
- `static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)`

  yields a collector that produces a map or concurrent map whose keys are the results of applying `classifier` to all collected elements, and whose values are lists of elements with the same key.
- `static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)`

  yields a collector that produces a map whose keys are `true`/`false`, and whose values are lists of the elements that fulfill/do not fulfill the predicate.

1.11 Downstream Collectors

The `groupingBy` method yields a map whose values are lists. If you want to process those lists in some way, supply a "downstream collector." For example, if you want sets instead of lists, you can use the `Collectors.toSet` collector that you saw in the previous section:

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(
   groupingBy(Locale::getCountry, toSet()));
```

NOTE: In this and the remaining examples of this section, we assume a static import of `java.util.stream.Collectors.*` to make the expressions easier to read.

Several collectors are provided for reducing grouped elements to numbers:

- `counting` produces a count of the collected elements. For example,

  ```
  Map<String, Long> countryToLocaleCounts = locales.collect(
     groupingBy(Locale::getCountry, counting()));
  ```

  counts how many locales there are for each country.
- `summing(Int|Long|Double)` takes a function argument, applies the function to the downstream elements, and produces their sum. For example,

  ```
  Map<String, Integer> stateToCityPopulation = cities.collect(
     groupingBy(City::getState, summingInt(City::getPopulation)));
  ```

  computes the sum of populations per state in a stream of cities.

- `maxBy` and `minBy` take a comparator and produce maximum and minimum of the downstream elements. For example,

  ```
  Map<String, Optional<City>> stateToLargestCity = cities.collect(
     groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
  ```

  produces the largest city per state.

The `mapping` method yields a collector that applies a function to downstream results and passes the function values to yet another collector. For example,

```
Map<String, Optional<String>> stateToLongestCityName = cities.collect(
   groupingBy(City::getState,
      mapping(City::getName,
         maxBy(Comparator.comparing(String::length)))));
```

Here, we group cities by state. Within each state, we produce the names of the cities and reduce by maximum length.

The `mapping` method also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country.

```
Map<String, Set<String>> countryToLanguages = locales.collect(
   groupingBy(Locale::getDisplayCountry,
      mapping(Locale::getDisplayLanguage,
         toSet())));
```

In the previous section, we used `toMap` instead of `groupingBy`. In this form, you don't need to worry about combining the individual sets.

If the grouping or mapping function has return type `int`, `long`, or `double`, you can collect elements into a summary statistics object, as discussed in Section 1.8, "Collecting Results," on p. 19. For example,

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(
   groupingBy(City::getState,
      summarizingInt(City::getPopulation)));
```

Then you can get the sum, count, average, minimum, and maximum of the function values from the summary statistics objects of each group.

NOTE: There are also three versions of a `reducing` method that apply general reductions, as described in Section 1.12, "Reduction Operations," on p. 33.

Composing collectors is a powerful approach, but it can also lead to very convoluted expressions. Their best use is with `groupingBy` or `partitioningBy` to process the "downstream" map values. Otherwise, simply apply methods such as `map`, `reduce`, `count`, `max`, or `min` directly on streams.

The example program in Listing 1.6 demonstrates downstream collectors.

Listing 1.6 `collecting/DownstreamCollectors.java`

```java
package collecting;

import static java.util.stream.Collectors.*;

import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class DownstreamCollectors
{

   public static class City
   {
      private String name;
      private String state;
      private int population;

      public City(String name, String state, int population)
      {
         this.name = name;
         this.state = state;
         this.population = population;
      }

      public String getName()
      {
         return name;
      }

      public String getState()
      {
         return state;
      }

      public int getPopulation()
      {
         return population;
      }
   }

```

(Continues)

Listing 1.6 *(Continued)*

```
   public static Stream<City> readCities(String filename) throws IOException
   {
      return Files.lines(Paths.get(filename)).map(l -> l.split(", "))
            .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
   }

   public static void main(String[] args) throws IOException
   {
      Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Set<Locale>> countryToLocaleSet = locales.collect(groupingBy(
            Locale::getCountry, toSet()));
      System.out.println("countryToLocaleSet: " + countryToLocaleSet);

      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Long> countryToLocaleCounts = locales.collect(groupingBy(
            Locale::getCountry, counting()));
      System.out.println("countryToLocaleCounts: " + countryToLocaleCounts);

      Stream<City> cities = readCities("cities.txt");
      Map<String, Integer> stateToCityPopulation = cities.collect(groupingBy(
            City::getState, summingInt(City::getPopulation)));
      System.out.println("stateToCityPopulation: " + stateToCityPopulation);

      cities = readCities("cities.txt");
      Map<String, Optional<String>> stateToLongestCityName = cities
            .collect(groupingBy(
                  City::getState,
                  mapping(City::getName,
                        maxBy(Comparator.comparing(String::length)))));

      System.out.println("stateToLongestCityName: " + stateToLongestCityName);

      locales = Stream.of(Locale.getAvailableLocales());
      Map<String, Set<String>> countryToLanguages = locales.collect(groupingBy(
            Locale::getDisplayCountry,
            mapping(Locale::getDisplayLanguage, toSet())));
      System.out.println("countryToLanguages: " + countryToLanguages);

      cities = readCities("cities.txt");
      Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities
            .collect(groupingBy(City::getState,
                  summarizingInt(City::getPopulation)));
      System.out.println(stateToCityPopulationSummary.get("NY"));

      cities = readCities("cities.txt");
      Map<String, String> stateToCityNames = cities.collect(groupingBy(
            City::getState,
            reducing("", City::getName, (s, t) -> s.length() == 0 ? t : s
                  + ", " + t)));

      cities = readCities("cities.txt");
      stateToCityNames = cities.collect(groupingBy(City::getState,
            mapping(City::getName, joining(", "))));
      System.out.println("stateToCityNames: " + stateToCityNames);
   }
}
```

java.util.stream.Collectors 8

- `static <T> Collector<T,?,Long> counting()`

  yields a collector that counts the collected elements.

- `static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)`
- `static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)`
- `static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)`

  yields a collector that computes the sum of the values of applying `mapper` to the collected elements.

- `static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)`
- `static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)`

  yields a collector that computes the maximum or minimum of the collected elements, using the ordering specified by `comparator`.

- `static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)`

  yields a collector that produces a map whose keys are `mapper` applied to the collected elements, and whose values are the result of collecting the elements with the same key using the `downstream` collector.

1.12 Reduction Operations

The `reduce` method is a general mechanism for computing a value from a stream. The simplest form takes a binary function and keeps applying it, starting with the first two elements. It's easy to explain this if the function is the sum:

```
List<Integer> values = . . .;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

In this case, the `reduce` method computes $v_0 + v_1 + v_2 + \ldots$, where the v_i are the stream elements. The method returns an `Optional` because there is no valid result if the stream is empty.

NOTE: In this case, you can write `reduce(Integer::sum)` instead of `reduce((x, y) -> x + y)`.

In general, if the `reduce` method has a reduction operation *op*, the reduction yields v_0 *op* v_1 *op* v_2 *op* $\ldots$, where we write v_i *op* v_{i+1} for the function call $op(v_i, v_{i+1})$. The operation should be *associative*: It shouldn't matter in which order you combine the elements. In math notation, $(x$ *op* $y)$ *op* z must be equal to x *op* $(y$ *op* $z)$. This allows efficient reduction with parallel streams.

There are many associative operations that might be useful in practice, such as sum, product, string concatenation, maximum and minimum, set union and intersection. An example of an operation that is not associative is subtraction. For example, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Often, there is an *identity value e* such that e *op* $x = x$, and you can use that element as the start of the computation. For example, `0` is the identity value for addition. Then call the second form of `reduce`:

```
List<Integer> values = . . .;
Integer sum = values.stream().reduce(0, (x, y) -> x + y)
   // Computes 0 + v0 + v1 + v2 + . . .
```

The identity value is returned if the stream is empty, and you no longer need to deal with the `Optional` class.

Now suppose you have a stream of objects and want to form the sum of some property, such as all lengths in a stream of strings. You can't use the simple form of `reduce`. It requires a function `(T, T) -> T`, with the same types for the arguments and the result. But in this situation, you have two types: The stream elements have type `String`, and the accumulated result is an integer. There is a form of `reduce` that can deal with this situation.

First, you supply an "accumulator" function `(total, word) -> total + word.length()`. That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is

```
int result = words.reduce(0,
   (total, word) -> total + word.length(),
   (total1, total2) -> total1 + total2);
```

NOTE: In practice, you probably won't use the `reduce` method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, max, or min. (We discuss streams of numbers in Section 1.13, "Primitive Type Streams," on p. 36.) In this particular example, you could have called `words.mapToInt(String::length).sum()`, which is both simpler and more efficient since it doesn't involve boxing.

NOTE: There are times when `reduce` is not general enough. For example, suppose you want to collect the results in a `BitSet`. If the collection is parallelized, you can't put the elements directly into a single `BitSet` because a `BitSet` object is not threadsafe. For that reason, you can't use `reduce`. Each segment needs to start out with its own empty set, and `reduce` only lets you supply one identity value. Instead, use `collect`. It takes three arguments:

1. A *supplier* that makes new instances of the target type, for example a constructor for a hash set.
2. An *accumulator* that adds an element to an instance, such as an `add` method.
3. A *combiner* that merges two instances into one, such as `addAll`.

Here is how the `collect` method works for a bit set:

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

java.util.Stream 8

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `T reduce(T identity, BinaryOperator<T> accumulator)`
- `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`

  forms a cumulative total of the stream elements with the given `accumulator` function. If `identity` is provided, then it is the first value to be accumulated. If `combiner` is provided, it can be used to combine totals of segments that are accumulated separately.

- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`

  collects elements in a result of type `R`. On each segment, `supplier` is called to provide an initial result, `accumulator` is called to mutably add elements to it, and `combiner` is called to combine two results.

1.13 Primitive Type Streams

So far, we have collected integers in a `Stream<Integer>`, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types: `double`, `float`, `long`, `short`, `char`, `byte`, and `boolean`. The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers. If you want to store `short`, `char`, `byte`, and `boolean`, use an `IntStream`, and for `float`, use a `DoubleStream`.

To create an `IntStream`, call the `IntStream.of` and `Arrays.stream` methods:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an int[] array
```

As with object streams, you can also use the static `generate` and `iterate` methods. In addition, `IntStream` and `LongStream` have static methods `range` and `rangeClosed` that generate integer ranges with step size one:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bound is excluded
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included
```

The `CharSequence` interface has methods `codePoints` and `chars` that yield an `IntStream` of the Unicode codes of the characters or of the code units in the UTF-16 encoding. (See Chapter 2 for the sordid details.)

```
String sentence = "\uD835\uDD46 is the set of octonions.";
   // \uD835\uDD46 is the UTF-16 encoding of the letter 𝕆, unicode U+1D546

IntStream codes = sentence.codePoints();
   // The stream with hex values 1D546 20 69 73 20 . . .
```

When you have a stream of objects, you can transform it to a primitive type stream with the `mapToInt`, `mapToLong`, or `mapToDouble` methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an `IntStream`:

```
Stream<String> words = . . .;
IntStream lengths = words.mapToInt(String::length);
```

To convert a primitive type stream to an object stream, use the `boxed` method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

- The `toArray` methods return primitive type arrays.

- Methods that yield an optional result return an `OptionalInt`, `OptionalLong`, or `OptionalDouble`. These classes are analogous to the `Optional` class but have methods `getAsInt`, `getAsLong`, and `getAsDouble` instead of the `get` method.
- There are methods `sum`, `average`, `max`, and `min` that return the sum, average, maximum, and minimum. These methods are not defined for object streams.
- The `summaryStatistics` method yields an object of type `IntSummaryStatistics`, `LongSummaryStatistics`, or `DoubleSummaryStatistics` that can simultaneously report the sum, average, maximum, and minimum of the stream.

NOTE: The `Random` class has methods `ints`, `longs`, and `doubles` that return primitive type streams of random numbers.

The program in Listing 1.7 gives examples for the API of primitive type streams.

Listing 1.7 `streams/PrimitiveTypeStreams.java`

```
package streams;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class PrimitiveTypeStreams
{
   public static void show(String title, IntStream stream)
   {
      final int SIZE = 10;
      int[] firstElements = stream.limit(SIZE + 1).toArray();
      System.out.print(title + ": ");
      for (int i = 0; i < firstElements.length; i++)
      {
         if (i > 0) System.out.print(", ");
         if (i < SIZE) System.out.print(firstElements[i]);
         else System.out.print("...");
      }
      System.out.println();
   }
```

(Continues)

Listing 1.7 *(Continued)*

```

   public static void main(String[] args) throws IOException
   {
      IntStream is1 = IntStream.generate(() -> (int) (Math.random() * 100));
      show("is1", is1);
      IntStream is2 = IntStream.range(5, 10);
      show("is2", is2);
      IntStream is3 = IntStream.rangeClosed(5, 10);
      show("is3", is3);

      Path path = Paths.get("../gutenberg/alice30.txt");
      String contents = new String(Files.readAllBytes(path), StandardCharsets.UTF_8);

      Stream<String> words = Stream.of(contents.split("\\PL+"));
      IntStream is4 = words.mapToInt(String::length);
      show("is4", is4);
      String sentence = "\uD835\uDD46 is the set of octonions.";
      System.out.println(sentence);
      IntStream codes = sentence.codePoints();
      System.out.println(codes.mapToObj(c -> String.format("%X ", c)).collect(
            Collectors.joining()));

      Stream<Integer> integers = IntStream.range(0, 100).boxed();
      IntStream is5 = integers.mapToInt(Integer::intValue);
      show("is5", is5);
   }
}
```

java.util.stream.IntStream 8

- `static IntStream range(int startInclusive, int endExclusive)`
- `static IntStream rangeClosed(int startInclusive, int endInclusive)`

  yields an `IntStream` with the integers in the given range.
- `static IntStream of(int... values)`

  yields an `IntStream` with the given elements.
- `int[] toArray()`

  yields an array with the elements of this stream.

(Continues)

***java.util.stream.IntStream* 8** *(Continued)*

- `int sum()`
- `OptionalDouble average()`
- `OptionalInt max()`
- `OptionalInt min()`
- `IntSummaryStatistics summaryStatistics()`

  yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.
- `Stream<Integer> boxed()`

  yields a stream of wrapper objects for the elements in this stream.

***java.util.stream.LongStream* 8**

- `static LongStream range(long startInclusive, long endExclusive)`
- `static LongStream rangeClosed(long startInclusive, long endInclusive)`

  yields a `LongStream` with the integers in the given range.
- `static LongStream of(long... values)`

  yields a `LongStream` with the given elements.
- `long[] toArray()`

  yields an array with the elements of this stream.
- `long sum()`
- `OptionalDouble average()`
- `OptionalLong max()`
- `OptionalLong min()`
- `LongSummaryStatistics summaryStatistics()`

  yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.
- `Stream<Long> boxed()`

  yields a stream of wrapper objects for the elements in this stream.

***java.util.stream.DoubleStream* 8**

- `static DoubleStream of(double... values)`

  yields a `DoubleStream` with the given elements.

(Continues)

java.util.stream.DoubleStream* 8 *(Continued)

- `double[] toArray()`

  yields an array with the elements of this stream.

- `double sum()`
- `OptionalDouble average()`
- `OptionalDouble max()`
- `OptionalDouble min()`
- `DoubleSummaryStatistics summaryStatistics()`

  yields the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these results can be obtained.

- `Stream<Double> boxed()`

  yields a stream of wrapper objects for the elements in this stream.

***java.lang.CharSequence* 1.0**

- `IntStream codePoints()` 8

  yields a stream of all Unicode code points of this string.

java.util.Random 1.0

- `IntStream ints()`
- `IntStream ints(int randomNumberOrigin, int randomNumberBound)` 8
- `IntStream ints(long streamSize)` 8
- `IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound)` 8
- `LongStream longs()` 8
- `LongStream longs(long randomNumberOrigin, long randomNumberBound)` 8
- `LongStream longs(long streamSize)` 8
- `LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound)` 8
- `DoubleStream doubles()` 8
- `DoubleStream doubles(double randomNumberOrigin, double randomNumberBound)` 8
- `DoubleStream doubles(long streamSize)` 8
- `DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound)` 8

  yields streams of random numbers. If `streamSize` is provided, the stream is finite with the given number of elements. When bounds are provided, the elements are between `randomNumberOrigin` (inclusive) and `randomNumberBound` (exclusive).

java.util.Optional(Int|Long|Double) 8

- `static Optional(Int|Long|Double) of((int|long|double) value)`

  yields an optional object with the supplied primitive type value.
- `(int|long|double) getAs(Int|Long|Double)()`

  yields the value of this optional object, or throws a `NoSuchElementException` if it is empty.
- `(int|long|double) orElse((int|long|double) other)`
- `(int|long|double) orElseGet((Int|Long|Double)Supplier other)`

  yields the value of this optional object, or the alternative value if this object is empty.
- `void ifPresent((Int|Long|Double)Consumer consumer)`

  If this optional object is not empty, passes its value to `consumer`.

java.util.(Int|Long|Double)SummaryStatistics 8

- `long getCount()`
- `(int|long|double) getSum()`
- `double getAverage()`
- `(int|long|double) getMax()`
- `(int|long|double) getMin()`

  yields the count, sum, average, maximum, and minimum of the collected elements.

1.14 Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. You can get a parallel stream from any collection with the `Collection.parallelStream()` method:

```
Stream<String> parallelWords = words.parallelStream();
```

Moreover, the `parallel` method converts any sequential stream into a parallel one.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

As long as the stream is in parallel mode when the terminal method executes, all intermediate stream operations will be parallelized.

When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations can be executed in an arbitrary order.

Here is an example of something you cannot do. Suppose you want to count all short words in a stream of strings:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
   s -> { if (s.length() < 12) shortWords[s.length()]++; });
      // Error-race condition!
System.out.println(Arrays.toString(shortWords));
```

This is very, very bad code. The function passed to `forEach` runs concurrently in multiple threads, each updating a shared array. As we explained in Chapter 14 of Volume 1, that's a classic *race condition*. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run—each of them wrong.

It is your responsibility to ensure that any functions you pass to parallel stream operations are safe to execute in parallel. The best way to do that is to stay away from mutable state. In this example, you can safely parallelize the computation if you group strings by length and count them.

```
Map<Integer, Long> shortWordCounts =
   words.parallelStream()
      .filter(s -> s.length() < 10)
      .collect(groupingBy(
         String::length,
         counting()));
```

CAUTION: The functions that you pass to parallel stream operations should not block. Parallel streams use a fork-join pool for operating on segments of the stream. If multiple stream operations block, the pool may not be able to do any work.

By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling `Stream.sorted`, are *ordered*. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results.

Ordering does not preclude efficient parallelization. For example, when computing `stream.map(fun)`, the stream can be partitioned into n segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the `unordered` method on a stream, you indicate that you are not interested in ordering. One operation that can benefit from this is `Stream.distinct`. On an ordered stream, `distinct` retains the first of all equal elements. That impedes parallelization—the thread processing a segment can't know which elements to discard until the preceding segment has been processed. If it is acceptable to retain *any* of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates).

You can also speed up the `limit` method by dropping ordering. If you just want any `n` elements from a stream and you don't care which ones you get, call

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

As discussed in Section 1.9, "Collecting into Maps," on p. 24, merging maps is expensive. For that reason, the `Collectors.groupingByConcurrent` method uses a shared concurrent map. To benefit from parallelism, the order of the map values will not be the same as the stream order.

```
Map<Integer, List<String>> result = words.parallelStream().collect(
   Collectors.groupingByConcurrent(String::length));
   // Values aren't collected in stream order
```

Of course, you won't care if you use a downstream collector that is independent of the ordering, such as

```
Map<Integer, Long> wordCounts =
   words.parallelStream()
      .collect(
         groupingByConcurrent(
            String::length,
            counting()));
```

CAUTION: It is very important that you don't modify the collection that is backing a stream while carrying out a stream operation (even if the modification is threadsafe). Remember that streams don't collect their data—that data is always in a separate collection. If you were to modify that collection, the outcome of the stream operations would be undefined. The JDK documentation refers to this requirement as *noninterference*. It applies both to sequential and parallel streams.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point when the terminal operation executes. For example, the following, while certainly not recommended, will work:

```
List<String> wordList = . . .;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

But this code is wrong:

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
   // Error—interference
```

For parallel streams to work well, a number of conditions need to be fulfilled:

- The data should be in memory. It would be inefficient to have to wait for the data to arrive.
- The stream should be efficiently splittable into subregions. A stream backed by an array or a balanced binary tree works well, but the result of `Stream.iterate` does not.
- The stream operations should do a substantial amount of work. If the total work load is not large, it does not make sense to pay for the cost of setting up the parallel computation.
- The stream operations should not block.

In other words, don't turn all your streams into parallel streams. Use parallel streams only when you do a substantial amount of sustained computational work on data that is already in memory.

The example program in Listing 1.8 demonstrates how to work with parallel streams.

Listing 1.8 `parallel/ParallelStreams.java`

```
package parallel;

import static java.util.stream.Collectors.*;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class ParallelStreams
{
   public static void main(String[] args) throws IOException
   {
      String contents = new String(Files.readAllBytes(
            Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
      List<String> wordList = Arrays.asList(contents.split("\\PL+"));

      // Very bad code ahead
      int[] shortWords = new int[10];
      wordList.parallelStream().forEach(s ->
         {
            if (s.length() < 10) shortWords[s.length()]++;
         });
      System.out.println(Arrays.toString(shortWords));

      // Try again--the result will likely be different (and also wrong)
      Arrays.fill(shortWords, 0);
      wordList.parallelStream().forEach(s ->
         {
            if (s.length() < 10) shortWords[s.length()]++;
         });
      System.out.println(Arrays.toString(shortWords));

      // Remedy: Group and count
      Map<Integer, Long> shortWordCounts = wordList.parallelStream()
         .filter(s -> s.length() < 10)
         .collect(groupingBy(String::length, counting()));

      System.out.println(shortWordCounts);

      // Downstream order not deterministic
      Map<Integer, List<String>> result = wordList.parallelStream().collect(
         Collectors.groupingByConcurrent(String::length));

      System.out.println(result.get(14));

      result = wordList.parallelStream().collect(
         Collectors.groupingByConcurrent(String::length));

      System.out.println(result.get(14));

      Map<Integer, Long> wordCounts = wordList.parallelStream().collect(
         groupingByConcurrent(String::length, counting()));

      System.out.println(wordCounts);
   }
}
```

java.util.stream.BaseStream<T,S extends BaseStream<T,S>> **8**

- `S parallel()`

  yields a parallel stream with the same elements as this stream.
- `S unordered()`

  yields an unordered stream with the same elements as this stream.

java.util.Collection<E> 1.2

- `Stream<E> parallelStream()` 8

  yields a parallel stream with the elements of this collection.

In this chapter, you have learned how to put the stream library of Java 8 to use. The next chapter covers another important topic: processing input and output.