Chapter 8. Streams

Topics 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.

The key points of this chapter are:

1. Iterators imply a specific traversal strategy and prohibit efficient concurrent execution.
2. You can create streams from collections, arrays, generators, or iterators.
3. Use `filter` to select elements and `map` to transform elements.
4. Other operations for transforming streams include `limit`, `distinct`, and

`sorted`.

5. To obtain a result from a stream, use a reduction operator such as `count`, `max`, `min`, `findFirst`, or `findAny`. Some of these methods return an `Optional` value.
6. The `Optional` type is intended as a safe alternative to working with `null` values. To use it safely, take advantage of the `ifPresent` and `orElse` methods.
7. You can collect stream results in collections, arrays, strings, or maps.
8. The `groupingBy` and `partitioningBy` methods of the `Collectors` class allow you to split the contents of a stream into groups, and to obtain a result for each group.
9. There are specialized streams for the primitive types `int`, `long`, and `double`.
10. Parallel streams automatically parallelize stream operations.

8.1 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:

Click here to view code image

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

Now we are ready to iterate:

Click here to view code image

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

With streams, the same operation looks like this:

Click here to view code image

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

Now you don't have to scan the loop for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, where 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.

Click here to view code image

```
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 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 our example, the stream was created with the `stream` or `parallelStream` method. The `filter` method transformed it, and `count` was 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.

8.2 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.

Click here to view code image

```
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:

Click here to view code image

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

Use `Arrays.stream(array, from, to)` to make a stream from a part of an array.

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

Click here to view code image

```
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—see Section 3.6.2, "Choosing a Functional Interface," page 120). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as Click here to view code image

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

or a stream of random numbers as

Click here to view code image

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

To produce 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, Click here to view code image

```
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.

To produce a finite stream instead, add a predicate that specifies when the iteration should finish:

Click here to view code image

```
BigInteger limit = new BigInteger("10000000");
Stream<BigInteger> integers
= Stream.iterate(BigInteger.ZERO,
n -> n.compareTo(limit) < 0,
n -> n.add(BigInteger.ONE));
```

As soon as the predicate rejects an iteratively generated value, the stream ends.

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: Click here to view code image

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

The `Scanner.tokens` method yields a stream of tokens of a scanner. Another way to get a stream of words from a string is Click here to view code image

```
Stream<String> words = new Scanner(contents).tokens();
```

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

Click here to view code image

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

8.3 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 new stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

Click here to view code image

```
List<String> words = ...;
Stream<String> longWords = words.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: Click here to view code image

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

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

Click here to view code image

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

The resulting stream contains the first letter of each word.

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. Here is an example—a method that turns a string into a stream of strings, namely the individual code points: Click here to view code image

```
public static Stream<String> codePoints(String s) {
List<String> result = new ArrayList<>();
int i = 0;
```

```
while (i < s.length()) {
int j = s.offsetByCodePoints(i, 1);
result.add(s.substring(i, j));
i = j;
}
return result.stream();
}
```

This method correctly handles Unicode characters that require two `char` values because that's the right thing to do. But you don't have to dwell on that.

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

Now let's map the `codePoints` method on a stream of strings:

Click here to view code image

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

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

```
Stream<String> flatResult = words.stream().flatMap(w -> codePoints(w))
// Calls codePoints 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.

8.4 Extracting Substreams and Combining 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,

Click here to view code image

```
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 in our book reading example where, 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`: Click here to view code image

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

The *stream*`.takeWhile(`*predicate*`)` call takes all elements from the stream while the predicate is true, and then stops.

For example, suppose we use the `codePoints` method of the preceding section to split a string into characters, and we want to collect all initial digits. The `takeWhile` method can do this: Click here to view code image

```
Stream<String> initialDigits = codePoints(str).takeWhile(
s -> "0123456789".contains(s));
```

The `dropWhile` method does the opposite, dropping elements while a condition is true and yielding a stream of all elements starting with the first one for which the condition was false. For example, Click here to view code image

```
Stream<String> withoutInitialWhiteSpace = codePoints(str).dropWhile(
s -> s.trim().length() == 0);
```

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

Click here to view code image

```
Stream<String> combined = Stream.concat(
codePoints("Hello"), codePoints("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 wouldn't ever get a chance.

8.5 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 duplicates need not be adjacent.

Click here to view code image

```
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: Click here to view code image

```
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: Click here to view code image

```
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.

Tip

When you use a debugger to debug a stream computation, you can set a breakpoint in a method that is called from one of the transformations. With most IDEs, you can also set breakpoints in lambda expressions. If you just want to know what happens at a particular point in the stream pipeline, add .peek(x -> {
return; })

and set a breakpoint on the second line.

8.6 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 nonstream value that can be used in your program.

You have already seen a simple reduction: the `count` method that 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: Click here to view code image

```
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: Click here to view code image

```
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 that it finds instead of being constrained to the first one.

Click here to view code image

```
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`.

Click here to view code image

```
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.

8.7 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`. But it is only safer if you use it right. The next section shows you how.

8.7.1 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: Click here to view code image

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

You can also invoke code to compute the default:

Click here to view code image

```
String result = optionalString.orElseGet(() ->
System.getProperty("myapp.default"));
// The function is only called when needed
```

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

Click here to view code image

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

The `orElseGet` method assumes that the alternative computation always succeeds. If that computation can fail, use the `or` method: Click here to view code image

```
Optional<String> result = optionalString.or(() ->
Optional.ofNullable(System.getProperty("myapp.default")));
```

If `optionalString` has a value, then `result` is `optionalString`. If not, and `System.getProperty("myapp.default")` returns a non-`null` value, then that value, wrapped in an `Optional`, becomes the result. Otherwise, the result is empty.

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.

Click here to view code image

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

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

Click here to view code image

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

or simply

Click here to view code image

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

If you want to take one action if the `Optional` has a value and another action if it doesn't, use `ifPresentOrElse`: Click here to view code image

```
optionalValue.ifPresentOrElse(
v -> Process v,
() -> Do something else);
```

When using `ifPresent` to pass an optional value to a function, the function return value is lost. If you want to process the function result, use `map` instead: Click here to view code image

```
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 8.3, "The `filter`, `map`, and `flatMap` Methods" (page 263). Simply imagine an optional value as a stream of size zero or one. The result again has size zero or one, and in the latter case, the function has been applied.

8.7.2 How Not to Work with Optional Values

If you don't use `Optional` values correctly, you have 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, Click here to view code image

```
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

Click here to view code image

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

is no easier than

Click here to view code image

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

8.7.3 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,

Click here to view code image

```
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.

8.7.4 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

Click here to view code image

```
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 parts do.

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

Click here to view code image

```
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

Click here to view code image

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

or, if you prefer,

Click here to view code image

```
Optional<Double> result
= Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::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 8.3, "The `filter`, `map`, and `flatMap` Methods," page 263). 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 having zero or one elements.

8.7.5 Turning an Optional Into a Stream

The `stream` method turns an `Optional<T>` into a `Stream<T>` with zero or one elements. Sure, why not, but why would you ever want that?

This becomes useful with methods that return an `Optional` result. Suppose you have a stream of user IDs and a method

Click here to view code image

```
Optional<User> lookup(String id)
```

How do you get a stream of users, skipping those IDs that are invalid?

Of course, you can filter out the invalid IDs and then apply `get` to the remaining ones:

Click here to view code image

```
Stream<String> ids = ...;
Stream<User> users = ids.map(Users::lookup)
.filter(Optional::isPresent)
.map(Optional::get);
```

But that uses the `isPresent` and `get` methods that we warned about. It is more elegant to call Click here to view code image

```
Stream<User> users = ids.map(Users::lookup)
.flatMap(Optional::stream);
```

Each call to `stream` returns a stream with 0 or 1 elements. The `flatMap` method combines them all. That means the nonexistent users are simply dropped.

Note

In this section, we consider the happy scenario in which we have a method that returns an `Optional` value. These days, many methods return `null` when there is no valid result. Suppose `Users.classicLookup(id)` returns a `User` object or `null`, not an `Optional<User>`. Then you can of course filter out the `null` values: Click here to view code image

```
Stream<User> users = ids.map(Users::classicLookup)
.filter(Objects::nonNull);
```

But if you prefer the `flatMap` approach, you can use

Click here to view code image

```
Stream<User> users = ids.flatMap(
id -> Stream.ofNullable(Users.classicLookup(id)));
```

or

Click here to view code image

```
Stream<User> users = ids.map(Users::classicLookup)
.flatMap(Stream::ofNullable);
```

The call `Stream.ofNullable(obj)` yields an empty stream if `obj` is `null` or a stream just containing `obj` otherwise.

8.8 Collecting Results

When you are done with a stream, you will often want to look at the results. 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:

Click here to view code image

```
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: Click here to view code image

```
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 Click here to view code image

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

or

Click here to view code image

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

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

Click here to view code image

```
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

Click here to view code image

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

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

Click here to view code image

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

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

Click here to view code image

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

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

Click here to view code image

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

8.9 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 ID. The

`Collectors.toMap` method has two function arguments that produce the map's keys and values. For example,

Click here to view code image

```
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.

Click here to view code image

```
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, as key its name in your default locale (such as `"German"`), and as value its localized name (such as `"Deutsch"`).

Click here to view code image

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
Collectors.toMap(
Locale::getDisplayLanguage,
loc -> loc.getDisplayLanguage(loc),
(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, I use the `Locale` class as a source of an interesting data set. See Chapter 13 for more information about working with 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.

Click here to view code image

```
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`: Click here to view code image

```
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.

8.10 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:

Click here to view code image

```
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 Click here to view code image

```
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: Click here to view code image

```
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.

8.11 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 preceding section:

Click here to view code image

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

Note

In this example, as well as the remaining examples of this section, I 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,

Click here to view code image

```
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, Click here to view code image

```
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, Click here to view code image

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

  produces the largest city per state.

The `mapping` collector applies a function to downstream results, and it requires yet another collector for processing its results. For example, Click here to view code image

```
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.

Click here to view code image

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

There is a `flatMapping` method as well, for use with functions that return streams (see Exercise 8).

In the preceding section, I 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 8.8, "Collecting Results" (page 271). For example, Click here to view code image

```
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.

The `filtering` collector applies a filter to each group, for example:

Click here to view code image

```
Map<String, Set<City>> largeCitiesByState
= cities.collect(
groupingBy(City::getState,
filtering(c -> c.getPopulation() > 500000,
toSet())); // States without large cities have empty sets
```

Note

There are also three versions of a `reducing` method that apply general reductions, as described in the next section.

Composing collectors is powerful, but it can also lead to very convoluted expressions. The 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.

8.12 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:

Click here to view code image

```
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)`.

More generally, you can use any operation that combines a partial result `x` with the next value `y` to yield a new partial result.

Here is another way of looking at reductions. Given a reduction operation *op*, the reduction yields v_0 *op* v_1 *op* v_2 *op* ..., where v_i *op* v_{i+1} denotes the function call $op(v_i, v_{i+1})$. There are many operations that might be useful in practice, such as sum, product, string concatenation, maximum and minimum, set union and intersection.

If you want to use reduction with parallel streams, the operation must 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). 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* 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 for addition. Then call the second form of `reduce`: Click here to view code image

```
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 Click here to view code image

```
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 8.13, "Primitive Type Streams," page 279.) 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* to make new instances of the target object, for example a constructor for a hash set

2. An *accumulator* that adds an element to the target, such as an `add` method
3. A *combiner* that merges two objects into one, such as `addAll`

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

Click here to view code image

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

8.13 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:

Click here to view code image

```
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:
Click here to view code image

```
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 1 for the sordid details.) Click here to view code image

```
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`: Click here to view code image

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

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

Click here to view code image

```
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 they 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, count, 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.

8.14 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:

Click here to view code image

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

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

Click here to view code image

```
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 are *stateless* and 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:

Click here to view code image

```
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 you will see in Chapter 10, 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.

Click here to view code image

```
Map<Integer, Long> shortWordCounts
= words.parallelStream()
.filter(s -> s.length() < 12)
.collect(groupingBy(
String::length,
counting()));
```

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 `Stream.unordered` method, 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 Click here to view code image

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

As discussed in Section 8.9, “Collecting into Maps” (page 273), 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.

Click here to view code image

```
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

Click here to view code image

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

Note

Don't turn all your streams into parallel streams with the hope of speeding up their operations. There is a substantial overhead to parallelization that will only pay off for very large data sets. Moreover, the thread pool that is used by parallel streams may perform poorly for blocking operations such as file I/O or network operations. Parallel streams work best with huge in-memory collections of data and computationally intensive processing.

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: Click here to view code image

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

But this code is wrong: