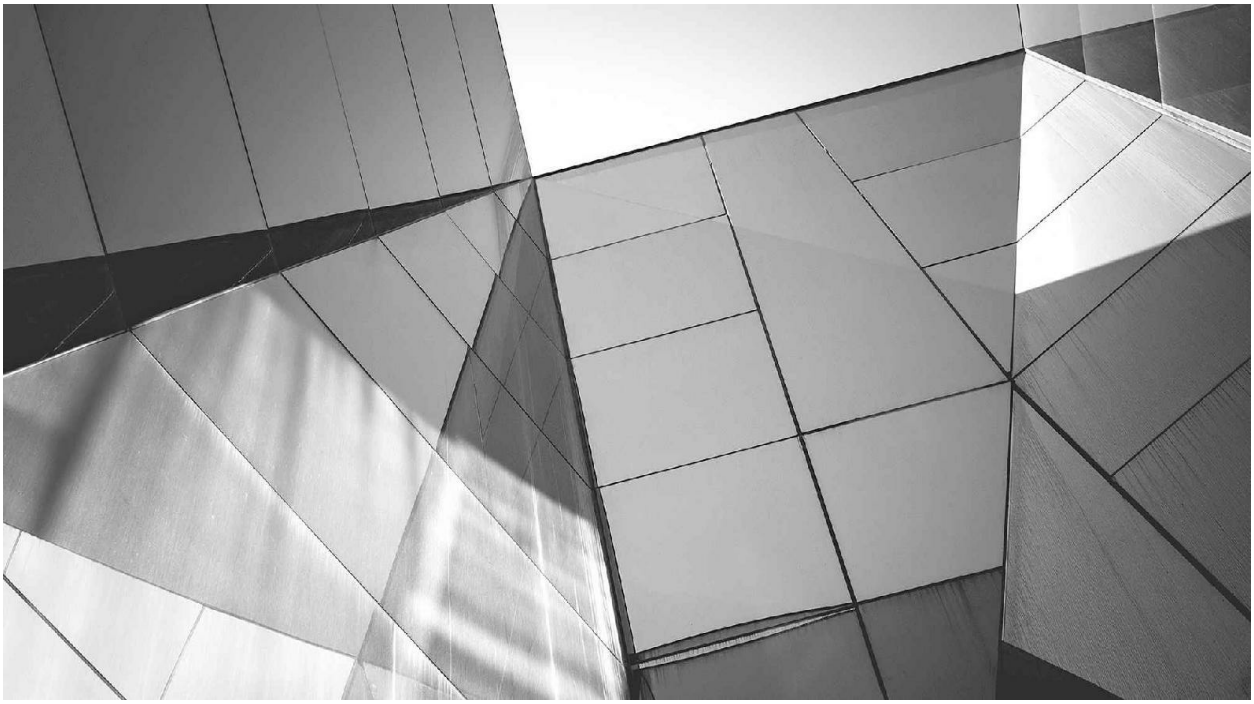


9

Streams

CERTIFICATION OBJECTIVES

- Collections Streams and Filters
- Iterate Using forEach Methods of Streams and List
- Describe Stream Interface and Stream Pipeline
- Filter a Collection by Using Lambda Expressions
- Use Method References with Streams
- Develop Code to Extract Data from an Object Using peek() and map() Methods Including Primitive Versions of the map() Method
- Search for Data by Using Search Methods of the Stream Classes Including findFirst, findAny, anyMatch, allMatch, noneMatch
- Develop Code That Uses the Optional Class
- Develop Code That Uses Stream Data Methods and Calculation Methods
- Sort a Collection Using the Stream API
- Save Results to a Collection Using the Collect Method and Group/Partition Data Using the Collectors Class
- Use flatMap() Methods in the Stream API
- Use the Stream API with NIO.2
- Use parallel Streams Including Reduction, Decomposition, Merging Processes, Pipelines and Performance

✓ Two-Minute Drill

Q&A Self Test

In the previous chapter we looked at how we can use lambda expressions to represent instances of classes that implement functional interfaces and explored a few places in the JDK where functional interfaces are used, like with the Logger's `log()` method and with the new `Iterable.forEach()` method.

Well, get ready for more. In this chapter about streams, we'll use the functional interfaces you just learned about. The `java.util.stream` package's Stream interface has over 30 methods, and about three-quarters of those work with functional interfaces in some way.

Like lambdas and functional interfaces, streams are another new addition in Java 8. The syntax of streams will be new to you and will take some getting used to. As you go through the chapter, take time to practice writing examples and testing the code to get the hang of it. The concept of streams is something quite new, too; at first, you might think streams are just another way of organizing data, like a collection, but they are actually about processing data efficiently, sometimes in ways you might initially find unintuitive given how you're used to programming in Java.

Because streams are related to collections (you can make a stream from a `Collection` type to process the data in the collection) and to lambdas (we'll be using lambdas frequently to operate on streams); on the exam, you'll often see streams questions mixed in with questions that use collections and lambdas. The certification objectives related to streams are also mixed in with collections, functional interfaces, and lambdas, as you'll see throughout this chapter.

CERTIFICATION OBJECTIVE

What Is a Stream? (OCP Objective 3.4)

3.4 Collections Streams and Filters.

A *stream* is a sequence of elements (you can think of these elements as data) that can be processed with operations. That's plenty vague as a definition, so to get a better handle on what a stream is, let's make one:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

Here, `myNums` is an array of three Integers. We're using the `Arrays.stream()` method to create a stream from that array. The resulting stream, `myStream`, is a stream of Integers. What does the stream look like?

```
System.out.println(myStream);
```

This results in the output:

```
java.util.stream.ReferencePipeline$Head@14ae5a5
```

Okay, so far this is clear as mud, right?!

Initially, you might think that a stream is a bit like a data structure, like a `List` or an array. After all, we're making a stream out an array of three integers:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

But streams are not a data structure to organize data like a `List` is or an array is; rather, they are a way to process data that you can think of as flowing through the stream, much like water flows through a stream in the real world. An *array* is a way of describing how data is organized and gives you flexible ways to access that data. Now, think of a stream as a way to *operate* on data that's flowing from an array through the stream. We say that the array is the stream's source (like a spring is the source of water for a real stream).

So far, `myStream` is just a description of where we're sourcing data. In this example:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

we're saying "the source of the data for `myStream` is `myNums`, an array of three integers, 1, 2, 3." That's why when we try to display the stream, we don't see

any data; we just see a cryptic description of the object that's describing how to get at the data.

Okay, so let's add an operation and do something with the stream of elements:

```
Integer[] myNums = { 1, 2, 3 };   // create an array
Stream<Integer> myStream = Arrays.stream(myNums);   // stream the array
long numElements = myStream.count(); // get the number of elements in the stream
System.out.println("Number of elements in the stream: " + numElements);
```

The result is

```
Number of elements in the stream: 3
```

Here, we've added an operation `count()` to `myStream`. The `count()` method simply returns the count of elements in the stream as a long value. Getting a number from the stream, the count of elements, means that stream is done. That is, we can't perform any more operations on the stream because the stream's been turned into one number by the `count()` operation. We say that the `count()` operation is a "terminal operation" because the stream ends there: no more data flows through the stream after the `count()` is done.

The real power of streams comes from the "intermediate operations" you can perform between the source and the end of the stream. For instance, you could filter for even numbers (intermediate operation one), multiply each of those even numbers by 2 (intermediate operation two), and then display the results in the console (terminal operation). By specifying multiple operations on a stream, you are essentially defining a set of things you want to do in a particular order to the data in the source. You could write it all out using a `for` loop and multiple lines of code (possibly even using one or more temporary or new data structures), but with streams, you can be more concise.

You can string together as many of these intermediate operations as you like, but for now, we'll begin with just one:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
long numElements =
  myStream
    .filter((i) -> i > 1) // add an intermediate operation to filter the stream
    .count();             // terminal operation, counts the elements in a
stream
System.out.println("Number of elements > 1: " + numElements);
```

The result we get with this code is

```
Number of elements > 1: 2
```

Now, rather than just counting the elements of the stream, we're first filtering the stream, looking for elements whose value is greater than 1. Notice that the `filter()` method takes a `Predicate`, and recall from the previous chapter that a `Predicate` is an interface with one abstract method, `test()`, that takes a value and returns a boolean. Here, we're representing the `Predicate` with a lambda expression; the value we take as an argument is one element from the stream of data; and we return true if the value of the element is greater than 1. You can read the code

```
myStream.filter((i) -> i > 1).count()
```

like this: "As the elements of `myStream` flow by, keep only those greater than 1, and count how many elements there are." The `filter()` method is calling the `test()` method of the `Predicate` you pass to `filter()` behind the scenes, and if the value passes the test, that value gets passed on to `count()`.

Notice that the `filter()` method of `myStream` produces a stream. It's a slightly modified stream consisting only of elements whose values are greater than 1. We're now calling the `count()` method on that filtered stream, rather than on the original `myStream`. Intermediate operations always produce another stream—that's how we can chain multiple operations together to manipulate the data as it flows by in the stream. As long as we keep doing intermediate operations, we keep the stream of data flowing; it ends only

when we perform a terminal operation like `count()`. That turns the stream into one thing—say, a number—and ends the stream. We'll look at streams with multiple intermediate operations shortly.

Now, let's try filtering for elements > 2 and count the results:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
long numElements = myStream.filter((i) -> i > 1).count();
System.out.println("Number of elements > 1: " + numElements);
numElements = myStream.filter((i) -> i > 2).count(); // filter by > 2 instead
System.out.println("Number of elements > 2: " + numElements);
```

Run that code and you'll get an exception:

```
Exception in thread "main" java.lang.IllegalStateException: stream has
already been operated upon or closed
```

Hmm. What does it mean the "stream has already been operated upon or closed"?

Streams can be used only once. To turn again to our analogy: Imagine you're standing on the bank of the stream. Once the water in the stream has flowed by you, you can't see it again. That water is gone, and you can't get it back.

No problem, we can just create the stream again. In Java, streams are lightweight objects, so you can create multiple streams if you need to:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
long numElements = myStream.filter((i) -> i > 1).count();
System.out.println("Number of elements > 1: " + numElements);
numElements =
  Arrays.stream(myNums)
    .filter((i) -> i > 2) // filter by > 2 on a whole new stream
    .count();
System.out.println("Number of elements > 2: " + numElements);
```

We can't reuse `myStream`, so here, we've created a new stream from the `myNums` array. We're filtering that stream using a `Predicate` that tests for an element greater than 2 and then counting the elements that pass that test. We do all three things: create the stream, filter the stream, and count the elements in one line of code.

In our example, we have only 1 element greater than 2, so we see this as the result:

```
Number of elements > 1: 2   // result of the 1st filter by > 1
Number of elements > 2: 1   // result of the 2nd filter by > 2
```

In Java, a stream is an object that gets its data from a source, but it doesn't store any data itself. The data flowing through the stream can be operated on, multiple times if we want, with intermediate operations, like `filter()`. The stream ends when we use a terminal operation, like `count()`, and once we've used a stream, we can't reuse it.

CERTIFICATION OBJECTIVE

How to Create a Stream (OCP Objectives 3.5 and 9.3)

3.5 Iterate using forEach methods of Streams and List.

9.3 Use Stream API with NIO.2.

There are a variety of ways you can create a stream. Given that streams are for data processing, you'll probably find you most often create streams from collections, arrays, and files. The collection, array, or file you use to create the stream is the "source" of the stream and provides the data that will flow through the stream. Remember that the stream itself does not contain any data; it operates on the data that is contained in the source, as that data flows through the stream operations.

Create a Stream from a Collection

Let's step through some examples of how to create a stream. First, here's how you can create a stream from a basic `List`, one of the collection types:

```
List<Double> tempsInPhoenix = Arrays.asList(123.6, 118.0, 113.0, 112.5,
     115.8, 117.0, 110.2, 110.1, 106.0, 106.4);
System.out.println("Number of days over 110 in 10 day period: " +
tempsInPhoenix
    .stream()                  // stream the List of Doubles
    .filter(t -> t > 110.0)    // filter the stream
    .count());                 // count the Doubles that pass the filter test
```

This code produces the output:

```
Number of days over 110 in 10 day period: 8
```

What we're doing in this code is first creating a `List` of `Doubles`; then we're using that `List` as a source for a stream that filters values, in this case temperatures greater than 110.0, and counts them. Notice that to create the stream, we're calling the `stream()` method of the `tempsInPhoenix List`. This method is a default method of the `Collection` interface and so is inherited by all classes that implement `Collection`. The `stream()` method's signature is

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

You can see that `stream()` returns an object of type `Stream`, with a generic object type parameter, meaning the stream is a stream of any object type. If we were to split the line where we create the stream in the code above into two and store the stream in a variable, we'd write the code like this:

```
Stream<Double> tempStream = tempsInPhoenix.stream();
System.out.println("Number of days over 110 in 10 day period: " +
tempStream.filter(t -> t > 110.0).count());
```

using `Double` as the type parameter for the stream whose source is a `List` of `Doubles`.

Don't forget that a `Map` (`HashMap`, `TreeMap`, etc.) is not a collection inheriting from `Collection`. If you want to stream a `Map`, you must first use the `entrySet()` method to turn the `Map` into a `Set`, which *is* a `Collection` type:

```
Map<String, Integer> myMap = new HashMap<String, Integer>();
myMap.put("Boi", 6); myMap.put("Zooey", 3); myMap.put("Charis", 8);
System.out.println("Number of items in the map with value > 4: " +
myMap
    .entrySet()      // get a Set of Map.Entry objects
    .stream()        // stream the Set
    .filter(d -> d.getValue() > 4)      // filter the Map.Entry objects
    .count());       // count the objects
```

Here, we create a `HashMap` and add three items to the `Map`, each with a `String` and `Integer` value—a dog's name and age. Then we stream the dog data from the `Map` and count the number of dogs older than 4.

If we try to stream the `Map` directly, we'll get a compile-time error. Instead, we first call `entrySet()` on the `Map` and then call `stream()` on the resulting `Set`. We then call the `filter()` operation on this stream and get a `Map.Entry` object as the argument to the `Predicate` lambda we pass to the filter. We filter to get only dogs older than 4 and count the results (there are two).

Build a Stream with Stream.of()

`Stream.of()` is quite flexible; it works with any object values, so you can create a `Stream` of `Strings`, `Integers`, `Doubles`, etc. The method signature is

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

meaning you can supply the `of()` method with any number of arguments, and you get back an ordered stream of those values.

We can use `Stream.of()` to create a stream from our array of Integers, `myNums`, like this:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Stream.of(myNums);
```

Then we use `myStream` just like we did before to filter and count the items.

We can make that code even shorter by skipping declaring the array altogether and supply Integer values directly, like this:

```
Stream<Integer> myStream = Stream.of(1, 2, 3);
```

The source of the `Stream` here is a little fuzzy; you aren't actually storing the data values in a data structure first, like you are if you're streaming the `myNums` array. The source is there; it's just hidden behind the scenes.

Create a Stream from an Array

You've already seen how to use `Arrays.stream()` to stream an array; earlier we created an array of Integers and streamed it. Here's another example of streaming an array, this time, an array of Strings:

```
String[] dogs = { "Boi", "Zooey", "Charis" };        // make an array
Stream<String> dogStream = Arrays.stream(dogs);      // stream it
System.out.println("Number of dogs in array: " + dogStream.count()); // count it
```

And we see 3 as the result for the number of dogs in the array. (Of course, this code is completely contrived as you'd never stream an array to count the

number of items, but you get the point, we hope).

Another way to create a stream from an array is to use the `Stream.of()` method. For our example with the Strings of dog names, we could rewrite the line to create the stream like this:

```
Stream<String> dogStream = Stream.of(dogs);
```

Create a Stream from a File

Using a stream to process data in a file is easy. You know you can use `Files` to read data from a file; the static `lines()` method of `Files` returns a `Stream`, so we can stream the data using the file as the source. Here is the signature of the `Files.lines()` method:

```
public static Stream<String> lines(Path path) throws IOException
```

So to create a stream from a file you write:

```
Stream<String> stream = Files.lines(Paths.get(filename));
```

This sets up the stream, but how do you process the data from the stream? You can use the `Stream`'s `forEach()` method:

```
stream.forEach(line -> …do something with the line of data from the file…)
```

The `forEach()` method on a stream works much like the `forEach()` method on a collection. It takes a `Consumer`, which we can represent with a lambda expression, and processes each line from the file in the body of the lambda. The `File.lines()` method provides one line at a time from the file as each data element in the stream, which makes processing data from the file easy.

Here's an example to bring this together. We have a file, "dvdinfo.txt," containing the name, genre, and star of a movie, one per line:

```
Donnie Darko/sci-fi/Gyllenhall, Jake
Raiders of the Lost Ark/action/Ford, Harrison
2001/sci-fi/??
Caddyshack/comedy/Murray, Bill
Star Wars/sci-fi/Ford, Harrison
Lost in Translation/comedy/Murray, Bill
Patriot Games/action/Ford, Harrison
```

We want to read the lines of the file, creating a new `DVDInfo` object for each movie entry:

```
class DVDInfo {
  String title;
  String genre;
  String leadActor;

  DVDInfo(String t, String g, String a) {
     title = t; genre = g; leadActor = a;
  }
  public String toString() {
     return title + " / " + genre + " / " + leadActor;
  }
  // getters and setters here
}
```

Here's how we can do that using the file as a source for a stream, and using the `forEach()` method to process each line from the file:

```
public class DVDs {
public static void main(String[] args) {
  List<DVDInfo> dvds = loadDVDs("dvdinfo.txt"); // load the DVDs from a file
  dvds.forEach(System.out::println);           // just print the DVDs
}
public static List<DVDInfo> loadDVDs(String filename) {
  List<DVDInfo> dvds = new ArrayList<DVDInfo>();
  // stream a file, line by line
    try (Stream<String> stream = Files.lines(Paths.get(filename))) {
      stream.forEach(line -> {      // use forEach to display each line
        String[] dvdItems = line.split("/");
        DVDInfo dvd = new DVDInfo(dvdItems[0], dvdItems[1], dvdItems[2]);
        dvds.add(dvd);              // for now; there's a better way
      });
    } catch (IOException e) {
      System.out.println("Error reading DVDs");
      e.printStackTrace();
    }
    return dvds;
 }
}
```

We see the output from printing the DVDs:

```
Donnie Darko / sci-fi / Gyllenhall, Jake
Raiders of the Lost Ark / action / Ford, Harrison
2001 / sci-fi / ??
Caddyshack / comedy / Murray, Bill
Star Wars / sci-fi / Ford, Harrison
Lost in Translation / comedy / Murray, Bill
Patriot Games / action / Ford, Harrison
```

When processing data from a file using a stream with `forEach()` like we do here, remember that within the lambda expression we pass to `forEach()`, all variables must be final or effectively final, so we can't modify a variable directly (e.g., by changing its value to a value from the file). However, also remember that we *can* add to or modify the fields of an object from within the lambda expression, so that's how we can add each DVD to the `List` of `dvds`. The `dvds List` is effectively final (we don't try to create a new `List` object and assign it to the `dvds` property within the lambda), but the contents of the `List` can still be modified by adding new DVDs to it. As a result, once the stream terminates (i.e., the last line from the file is read), we have all the DVDs from the file stored in the `dvds List`. We must store the DVDs somewhere if we want to use them after the stream is complete because the stream itself doesn't hold any data! However, there's a better way to store the DVDs in the `dvds List`, which we'll see later on in the chapter, so put a bookmark here and we'll return to this example later. For more `Files` methods that create streams, check the Online Appendix.

The `forEach()` method of `Stream` is another example of a terminal operation (along with `count()`, which we saw earlier). It does not produce another stream; rather it takes each item flowing by in the stream and consumes it (with a `Consumer`). It doesn't produce anything (i.e., `forEach()` is void), so whatever you want to do with the data you're processing with `forEach()` must happen in the `Consumer` you pass to it.

Primitive Value Streams

As you might expect, there are also primitive streams designed to avoid

autoboxing, for `doubles`, `ints`, and `longs`. These are `DoubleStream`, `IntStream`, and `LongStream`, respectively. So, you can create a `DoubleStream` like this:

```
DoubleStream s3 = DoubleStream.of(406.13, 406.42, 407.18, 409.01);
```

Notice there's no type parameter on the stream because this is a stream of `double` values and that's specified by the type itself, `DoubleStream`.

Keep in mind the difference between a `Stream<Double>` and `DoubleStream`. The first is a stream of `Double` objects; the second is a stream of `double` values. If you create a `List<Double>` and then stream it:

```
List<Double> co2Monthly = Arrays.asList(406.13, 406.42, 407.18, 409.01);
Stream<Double> s1 = co2Monthly.stream();
```

the stream is type `Stream<Double>`. Don't get fooled on the exam with this small distinction!

Summary of Methods to Create Streams

As you've just seen, there are a variety of ways to make streams—from collections, arrays, files, and values. Table 9-1 summarizes the methods you should be familiar with for the exam. And keep a close eye on those types; it can be easy to slip up and think you're creating a `DoubleStream` when you're creating a `Stream<Double>`! Note: This is not an exhaustive list of all the ways to create streams, so see each interface/class for more details and options.

TABLE 9-1 Methods to Create Streams

Interface/Class	Creates a...	With method...
`Collection`	`Stream<E>`	`stream()`
`Arrays`	`Stream<T>`	`stream(T[] array)` `stream(T[] array, int startInclusive, int endExclusive)`
`Arrays`	`IntStream`, `DoubleStream`, `LongStream`	`stream(int[] array)`, `stream(double[] array)`, `stream(long[] array)` (and versions with start/end like `Stream`)
`Files`	`Stream<String>`	`lines(Path path)`, `lines(Path path, Charset cs)`
`Stream`	`Stream<T>`	`of(T... values)`, `of(T t)`
`DoubleStream`	`DoubleStream`	`of(double... values)`, `of(double t)`
`IntStream`	`IntStream`	`of(int... values)`, `of(int t)`
`LongStream`	`LongStream`	`of(long... values)`, `of(long t)`

Why Streams?

You might be wondering why streams were added to Java. So far the examples you've seen are relatively simple, and you could easily accomplish the same thing using iteration over a `Collection`. The code might be a bit more concise, but it doesn't seem to really do anything fantastically different.

The main reason to use streams is when you start doing multiple intermediate operations. So far, we've been performing only one intermediate operation: a filter, using a variety of different `Predicates` to filter the data we get from the stream before we count it. However, when we use multiple intermediate operations, we start seeing the benefits of streams.

First, here's a quick example:

```
List<String> names =
    Arrays.asList("Boi", "Charis", "Zooey", "Bokeh", "Clover", "Aiko");
names.stream()                                  // Create the stream
    .filter(   s -> s.startsWith("B")
            || s.startsWith("C"))               // Filter by first letter
.filter(s -> s.length() > 3)                    // Filter by length
.forEach(System.out::println);                  // print
```

Here, we've got a list of names. Let's say we want to see the names that begin with "B" or "C" and have a length > 3.

First, we stream the names from the source `List`, `names`. Then, we filter on the starting letter "B" or "C". The result is that only names that start with "B" or "C" get passed through to the next filter. The next filter checks the length of the string and only passes through strings whose length is greater than 3. Both of these filter operations are "intermediate operations" because the stream continues; the stream is slightly modified (some data elements are discarded if they don't pass the test in the filter), but any data elements left in the stream continue flowing through the stream.

Finally we have a "terminal operation" that ends the stream: the `forEach()` method, which just prints the names. And we see the output:

```
Charis
Bokeh
Clover
```

You're probably saying to yourself, what's the big deal? We could do the same thing using iteration, right?

The big deal is that streams use something called a "pipeline," which can, in some circumstances, dramatically improve the efficiency of data processing. To understand how the stream pipeline works, let's break the above code down a bit more and see what happens in each step.

CERTIFICATION OBJECTIVE

The Stream Pipeline (OCP Objective 3.6)

3.6 Describe Stream interface and Stream pipeline.

A stream pipeline consists of three parts: a source, zero or more intermediate operations, and a terminal operation. The *source* describes where the data is coming from; the *intermediate operations* operate on the stream and produce another, perhaps modified, stream; and the *terminal operation* ends the stream and typically produces a value or some output.

So the stream pipeline in our code above defines the `names List` as the source for the stream, contains two intermediate operations that filter the data, and then terminates with `forEach()`, which prints the data:

```
List<String> names =
     Arrays.asList("Boi", "Charis", "Zooey", "Bokeh", "Clover", "Aiko");
names.stream()                          // the source
    .filter(   s -> s.startsWith("B")   // intermediate op
           || s.startsWith("C"))
    .filter(s -> s.length() > 3)        // intermediate op
    .forEach(System.out::println);      // terminal op
```

One analogy often used at this point is an assembly line. When you stream the names `List`, the first data element of the list is streamed to the first filter operation. At that point the second filter and the `forEach()` are just waiting for a data element. Once the first filter is complete, that data element is possibly discarded, if it doesn't begin with a "B" or "C", in which case the second filter and the `forEach()` never see it. This makes the subsequent stream more efficient.

If the data element is not discarded, then it's passed along to the second filter, which starts working on that element. In the meantime, the second data element is streamed to the first filter. Just like the assembly line, we now have the first filter working on the second data element, while the second filter is still working on the first one.

If the second filter's `Predicate` test is passed on the first data element, that element is passed along to the `forEach()`, which starts working on it (by printing it out). In the meantime, the second data element is passed to the second filter, assuming it passes the `Predicate` test in the first filter. Then the third data element is streamed to the first filter.

As you can see, the assembly-line analogy works pretty well for the pipeline. Like an assembly line, we can get some efficiencies working with streams for two reasons: First, we can do multiple operations on data in one pass, and Java is optimized so it keeps minimal intermediate state during the operations part of the pipeline. Second, in some circumstances, we can parallelize streams to take advantage of the underlying architecture of the system and do parallel computations very easily. Not all streams can be parallelized, but some can. We'll talk more about parallel streams later in "A Taste of Parallel Streams."

There's one other thing you should know about streams: they are lazy! And, despite what you might think, lazy streams can be more efficient.

Streams Are Lazy

Look again at the code:

```
names.stream()                                  // the source
    .filter(   s -> s.startsWith("B")           // intermediate op
           || s.startsWith("C"))
    .filter(s -> s.length() > 3)                // intermediate op
    .forEach(System.out::println);              // terminal op
```

What do you think happens if we write the following instead:

```
names.stream()                                  // the source
    .filter(   s -> s.startsWith("B")           // intermediate op
           || s.startsWith("C"))
    .filter(s -> s.length() > 3);               // intermediate op
```

That is, we've written the pipeline without the `forEach()` terminal operation.

You know that what you get back from the second `filter()` is another stream. But has anything actually happened yet? Is any data flowing through the stream?

The answer is no. Nothing has happened. Going back to the assembly-line analogy, the worker at station 1 (filter one) is still sitting idle, as is the worker at station 2 (filter two). That's because no terminal operation has been executed yet. We've got everything set up, but nothing to kick-start the data processing.

Not until the `forEach()` is executed does anything happen. As soon as the terminal operation is executed, then the assembly line kicks into gear, and the data starts flowing from the source through the stream and into the operations.

It's important to note here that our analogies of the water stream and the assembly line break down when you remember that streams don't hold any data. Even though we talk about data elements as "flowing through the stream," the data is never "in" the stream (unlike water that really is in a real-life stream and items that really are in an assembly line). The operations you define on a stream are how you specify ways to manipulate the data that's in the source in a particular order. In that sense, of course, streams are lazy

because no data is flowing, even though it helps us to think about streams that way.

This laziness makes streams more efficient because the JDK can perform optimizations to combine the operations efficiently, to operate on the data in a single pass, and to reduce operations on data whenever possible. If it's not necessary to run an operation on a piece of data (e.g., because we've already found the data element we're looking for, or because we've eliminated a data element in a prior intermediate operation, or because we've limited the number of data elements we want to operate on), then we can avoid even getting the data element from the source.

For most, if not all, of the simple examples we'll be working with in this book in preparation for the exam, the stream pipeline will not offer any tremendous advantage beyond thinking about data processing in a new way, and perhaps more concise code. However, once you get in the real world and start working with large amounts of data, you may realize the benefits of streams from an efficiency standpoint, especially if you can work with streams that can be parallelized (which we'll get to later, as we said before).

CERTIFICATION OBJECTIVE

Operating on Streams (OCP Objectives 3.7 and 5.1)

3.7 Filter a collection by using lambda expressions.

5.1 Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method.

You may have heard the expression "map-filter-reduce": a general abstraction to describe functions that operate on a sequence of elements. It perfectly describes what we are often doing with streams: we might "map" an input to a slightly different output, then "filter" that output by some criteria, and finally "reduce" to a single value or to a printed output. When you map-filter-reduce, you are simply specifying a sequence of operations to be performed on the data in the stream's source that leads to a result. As we said earlier, instead of using a stream, you could create a `for` loop over the source data structure yourself, apply each of the operations in turn, accumulate the results into a new data structure, and you'd accomplish the same thing as a stream. The advantage of a stream is twofold: you can (often) write a

sequence of operations to perform on the stream more concisely, and you can (sometimes) take advantage of some optimizations the JDK does under the covers to perform those operations more efficiently.

With streams, we take the map-filter-reduce abstraction and translate it into operations using the `map()`, `filter()`, and `reduce()` methods (and their variations) on a stream.

You've already seen the Java `Stream`'s `filter()` method. As we said earlier, `filter()` takes a `Predicate` and tests each element in the stream pipeline, producing a stream of elements that pass the test.

The `Stream`'s `map()` method is another intermediate operation; however, unlike `filter()`, `map()` doesn't winnow down the elements of a stream; rather, `map()` transforms elements of a stream. For instance, `map()` might compute the square of a number, mapping a stream of numbers to a stream of their squares. Or `map()` might get the age of a `Person` so that the next step (say, a filter) can find ages greater than 21.

The `map()` method takes a `Function`. Remember from the previous chapter that the purpose of a `Function` is to transform a value. The `Function`'s `apply()` method takes one value and produces another value (not necessarily of the same type), so the `Function` you pass to `map()` will "map" one value to another. The `Function`'s functional method `apply()` that you pass into `map()` gets called by `map()` behind the scenes, and the value returned from `apply()` gets passed on to the next operation in the stream pipeline.

Reduce is both a general method, `reduce()`, as well as a specific method like `count()` and others you'll see shortly. All reductions are also terminal operations. Reduction operations are designed to combine multiple inputs into one summary result, which could be a single number, like the reduction operation `count()` produces, or a collection of values, which we'll see examples of later.

The `Stream`'s `reduce()` method is a general method for reducing a stream to a value; the basic version of `reduce()` takes a `BiFunction` (a `Function` whose `apply()` method takes two arguments and produces one value) and applies it to pairs of elements from the stream, returning a value. You can think of `reduce()` as a way to "accumulate" a result, such as summing the values of a stream. Reductions like `count()`, `sum()`, and `average()` are defined as methods of streams; if you want a custom reduce function you can define it yourself using `reduce()`.

(Notice how all those functional interfaces from the previous chapter are showing up here?)

As we work with map-filter-reduce, keep in mind that the stream pipeline specifies a sequence of operations to perform on the data in a source data structure, like an array. There is one stream of data that gets modified and winnowed (the *map-filter* part) as it passes through the pipeline, and eventually, the data elements in the stream are reduced (the *reduce* part) to a value (when the stream is no longer a stream again, but rather a single value or another data structure).

Let's take a look at a map-filter-reduce set of operations on a stream. We'll define a `List` of `Integers`, take the square of each `Integer`, test to see whether the square is greater than 20, and if it is, we'll add 1 to a count, `result`. Before we do, we'll look at how we might do this with Java 7, so we can compare it with how we compute this using streams.

Here's the Java 7 way:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6);
long result = 0;
for (Integer n : nums) {
   int square = n * n;
   if (square > 20) {
      result = result + 1;
      System.out.println("Square of " + n + " is: " + square);
   }
}
System.out.println("Result: " + result);
```

With the output:

```
Square of 5 is: 25
Square of 6 is: 36
Result: 2
```

We're simply creating a `List`, and using a `for` loop to iterate over the list, compute the square, print out the square if it is greater than 20, and print the number of squares > 20.

And here's the Java 8 way, with streams:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6);
long result = nums.stream()
    .map(n -> n * n)          // map values in stream to squares
                              //    (map intermediate op)
    .filter(n -> n > 20)      // keep only squares > 20
                              //    (filter intermediate op)
    .count()                  // count the squares > 20 (reduction op)

System.out.println("Result (stream): " + result);
```

With the output:

```
Result (stream): 2
```

Here, we're streaming the `List` of `Integers`, first mapping the number from the stream to its square, then filtering the squares so that we keep only those squares greater than 20, and then reducing using the `count()` operation to count the squares > 20.

Which code do you like better? You're very familiar with the `for` loop iteration; the stream pipeline with a map-filter-reduce sequence is likely new to you. They both accomplish the same thing. Using streams like this has the potential to be a bit more efficient (not so much with this particular example because it's so simple) and is more concise.

Notice that we can't print the number with its square using the streams way. Why? Because by the time we get to the `count()` reduction operation, the only elements we have access to in the stream are the squares of the numbers. The original numbers are stored only in the original `List`. And don't be tempted to try to somehow keep track of the index of the square; that

won’t work and doesn’t fit the patterns of how we use stream pipelines.

We can create a “sort of” solution to the problem like this, however:

```
long result = nums.stream()
    .peek(n -> System.out.print("Number is: " + n + ", ")) // print the number
    .map(n -> n * n)
    .filter(n -> n > 20)
    .peek(n -> System.out.print("Square is: " + n + ", ")) // print the square
    .count();
```

And see the output:

```
Number is: 1, Number is: 2, Number is: 3, Number is: 4, Number is: 5,
Square is: 25, Number is: 6, Square is: 36, Result (stream): 2
```

The method `peek()` is an intermediate operation that allows you to “peek” into the stream as the elements flow by. It takes a `Consumer` and produces the same exact stream as it’s called on, so it doesn’t change the values or filter them in any way. Here we’re using it to peek at the numbers in the original stream before we map those numbers to squares and then filter them. We see all the numbers in the original stream, rather than just the numbers that have squares greater than 20.

You might feel a bit limited by this, but remember that streams are designed for data processing, so typically we’re looking for the result of a sequence of operations applied to a big set of data, and, other than when we’re debugging, we’re not going to be looking at the data as it flows by.

A more typical example of a map-filter-reduce operation might be to stream a data set of readings, map the stream to get a particular value from a reading, filter that stream to eliminate any outliers, and then find the average reading of that stream. We’ll tackle this example next, and in the process, you’ll see a new way to map values and learn about the concept of “optional” values.

CERTIFICATION OBJECTIVE

Map-Filter-Reduce with average() and Optionals (OCP Objectives 5.3 and 5.4)

5.3 Develop code that uses the Optional class.
5.4 Develop code that uses Stream data methods and calculation methods.

Imagine you have a piece of equipment that's taking a reading once per week, and you want to find the average of the readings you've measured so far this year. You know that the readings should probably be between 406 and 407, so you decide to throw away readings that are less than 406 or greater than 407 because those may be errors.

You start by creating a class for the readings:

```
class Reading {
   int year;
   int month;
   int day;
   double value;

   Reading(int year, int month, int day, double value) {
      this.year = year; this.month = month;
      this.day = day; this.value = value;
   }
}
```

Each reading gets stored with its value, along with the year, month, and day, in a `Reading` object, and the `Reading` objects are stored in a `List`:

```
List<Reading> readings = Arrays.asList(
    new Reading(2017, 1, 1, 405.91),
    new Reading(2017, 1, 8, 405.98),
    new Reading(2017, 1, 15, 406.14),
    new Reading(2017, 1, 22, 406.48),
    new Reading(2017, 1, 29, 406.20),
    new Reading(2017, 2, 5, 407.12),
    new Reading(2017, 2, 12, 406.03));
```

Now let's use a stream to find the average of the readings that are between 406 and 407. The first thing we'll do is stream the `readings`:

```
readings.stream()
```

For this computation, we're interested only in the value portion of the readings because we're trying to get the average. We can map each reading to its value, essentially converting the reading to a single `double` value. But we can't use `map()` because `map()` takes a `Function` whose `apply()` method takes an object and returns an object. Here's the method signature of `map()`:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

As we said earlier, `map()` takes a `Function` (called `mapper` in the signature above). `map()` applies that `Function` (by calling the `apply()` method) on each value in the stream. Recall that the `Function`'s `apply()` method takes an object and returns an object. So if we call `map()` on a `Stream` of objects, like we want to do here with our stream of `Reading` objects, what we get back is also a stream of objects. But `Reading.value` is a `double`, not a `Double`, so what we really want to get back is a stream of `doubles`.

Is there a solution? Yes, of course! The `Stream.mapToDouble()` method takes a `ToDoubleFunction`, whose `applyAsDouble()` method takes an object and returns a `double`:

```
double applyAsDouble(T value)
```

This is exactly what we need, so the next step in the stream pipeline is `mapToDouble()`:

```
readings.stream().mapToDouble(r -> r.value)
```

We use a lambda expression for the `ToDoubleFunction`, pass in a `Reading` object, `r`, to the `applyAsDouble()` method (which gets called by `mapToDouble()` behind the scenes), and get back the `double` value, `r.value`.

Keeping track of the type in these situations is tricky! Notice: what we get back from the `mapToDouble()` method of the `Stream` that we created with `readings.stream()` is a `DoubleStream`:

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

So by calling `mapToDouble()`, we've converted the initial `Stream` of `Reading` objects (`Stream<Reading>`) into a stream of `doubles`, and that stream has the type `DoubleStream`.

The capitalization on these types can be very confusing. Don't mix up a `DoubleStream` with a `Stream<Double>` here, or forget that `ToDoubleFunction`'s `applyAsDouble()` method returns a `double`. It's easy to get mixed up on this issue and think you're working with `Doubles` when you're actually working with `doubles` (or vice versa), and in this case, it matters a lot, as you'll see shortly.

After all that, where are we?

```
readings.stream().mapToDouble(r -> r.value)
```

We've mapped the stream of `Reading` objects into a stream of `doubles`, and next we want to filter the stream so we keep only the values greater than or equal to 406.0 and less than 407.0:

```
readings.stream()
        .mapToDouble(r -> r.value)
        .filter(v -> v >= 406.0 && v < 407.00)
```

We've changed the variable we're using in the lambda expression in `filter()` from `r` (for `Reading`) to `v` (for `value`) as a reminder that the stream

we're filtering is a `DoubleStream`: that is, a stream of the `double` values from the `List` of `Readings`.

Note again that at this point in the computation, nothing has actually happened. The stream is lazy! No computation is going on until we get to the reduce part of the map-filter-reduce: the terminal operation. So far we have a source (the `List` of `Readings`), a stream, and two intermediate operations.

The reduction operation we are going to use is the `average()` method. But if you look for `average()` in the `Stream` interface, you will not find it. That's because `average()` is a method of the `DoubleStream` interface (and its primitive cousins, `IntStream` and `LongStream`). That makes sense because `average()` is designed to work with numbers, not objects. And it's convenient that we mapped the stream of `Readings` into a stream of `doubles` (we *do* plan ahead for these examples).

Here is the type signature for the `DoubleStream`'s `average()` method:

```
OptionalDouble average()
```

You probably expected `average()` to return a `double`, right? And what on earth is this `OptionalDouble` thing?

An `OptionalDouble` is one of several types of optionals that are new in Java 8, which represent values that may or may not be there. Strange! Why would the average of our readings not be there?

Well, think about it this way. What if the source is empty? That is, what if our `List` of `Readings` is empty? Or what if, when we filter the stream, the filter eliminates all the values from the stream, so the result of the filter is an empty stream? We don't know until we get to `average()` whether there's anything actually *in* the stream (remember, the stream is lazy). And, as you know, the average value of a set of values is computed by adding up all the values and then dividing by the number of values. If the number of values is 0, then you'd be dividing by 0, which, as we all know, is a *very bad thing to do*.

Whenever we compute the average of some collection of values, we always have to check to make sure there's at least one of them so we don't get ourselves into this situation. But if we forget to check to see whether our source is empty before we stream it, or, perhaps more likely, we're in the middle of a stream pipeline and we've created an empty stream by filtering out all the values, presto: we're in a situation where we're trying to compute

the average of an empty stream and we're going to get ourselves into trouble. Optionals help prevent that trouble.

It doesn't make sense to return `null` in this situation because we're expecting a primitive `double`. And it doesn't make sense to return 0.0 because that could be a valid result for the average. Instead, we get back an `OptionalDouble`, which is a `double` value that might or might not be there. To finish off this line of code, we terminate the stream pipeline with a call to `average()` and store the result in an `OptionalDouble` variable, `avg`:

```
OptionalDouble avg =
    readings.stream()
        .mapToDouble(r -> r.value)
        .filter(v -> v >= 406.0 && v < 407.00)
        .average();
```

If you try to print the value of `avg`, like this:

```
System.out.println("Average of 406 readings: " + avg);
```

what you see is

```
Average of 406 readings: OptionalDouble[406.2125]
```

The correct result—that is, the average of the readings—is 406.2125, but we're seeing that result wrapped an `OptionalDouble`. How do you get a `double` value out of an `OptionalDouble`? You use the `getAsDouble()` method:

```
System.out.println("Average of 406 readings: " + avg.getAsDouble());
```

And now you'll see the result:

```
Average of 406 readings: 406.2125
```

The problem with this code, however, is if the `OptionalDouble` is empty, then you'll get a `NoSuchElementException`. Just because you get an

`OptionalDouble` back from `average()` doesn't mean you completely abdicate responsibility for checking to make sure there's a value there before you try to use it!

If you do try to call `getAsDouble()` on an empty `OptionalDouble`, you'll see

```
Exception in thread "main" java.util.NoSuchElementException: No value present
```

meaning you're trying to get a `double` that isn't there. A better way to write this code is to first check to make sure there is a value *present* in the `OptionalDouble` with the `isPresent()` method:

```
if (avg.isPresent()) {
   System.out.println("Average of 406 readings: " + avg.getAsDouble());
} else {
   System.out.println("Empty optional!");
}
```

If there's a value, we print it; otherwise, we print a message saying the optional must have been empty. Now if you run the code on an empty `List`, or a `List` with no values in the 406–407 range, you won't get a runtime error and you'll see the message:

```
Empty optional!
```

Whew! We finally have our average value, 406.2125, for our `List` of `Readings` using a stream pipeline. That was a lot to take in. We introduced map-filter-reduce, talked about how map works in more detail and how map and filter differ, and introduced optionals, which we'll be returning to in more detail.

There's ample room for confusion with `DoubleStream`,

`Stream<Double>`, `getAsDouble()`, etc. Adding to the confusion is that sometimes a reduction will return an `OptionalDouble` and sometimes an `Optional<Double>` (and, likewise, for `long/Long` and `int/Integer`).

This is a good time to pay close attention to types, type parameters, and method signatures. There will be times on the exam when you'll have to select the correct type as part of your answer, and knowing the difference between `DoubleStream` and `Stream<Double>` or `OptionalDouble` and `Optional<Double>` will be the key to choosing the correct answer.

Reduce

What we did with this example code is take a source, a `List` of `Readings`, stream it, map the readings (mapping a `Reading` to a `double`), filter the values (keeping only values between 406 and 407), and then reduce (by taking the average). Let's return to the reduce part of map-filter-reduce.

You've seen a couple of reductions so far: `count()` and `average()`. The `count()` method is defined for both `Stream` and the primitive versions of `Stream`: `IntStream`, `LongStream`, and `DoubleStream`, while `average()` is defined only for the primitive streams. Other handy methods defined on the primitive streams include `min()`, `max()`, and `sum()`, which result in the minimum value in a stream, the maximum value in a stream, and the sum of the values in a stream.

Like `average()`, `min()` and `max()` also return optional values. If you are looking for the `max()` reading in our `List` of `Readings`, you'd write:

```
OptionalDouble max =
    readings.stream()
        .mapToDouble(r -> r.value)
        .max();
if (max.isPresent()) {
    System.out.println("Max of all readings: " + max.getAsDouble());
} else {
    System.out.println("Empty optional!");
}
```

Looking at the type signature of `sum()`, however, you will probably notice that `sum()` does not return an optional value. Rather, `DoubleStream.sum()` returns a `double`. Why do `average()`, `max()`, and `min()` return optionals, but `sum()` does not?

As with `average()`, it makes no sense to take the `max()` or `min()` of an empty stream. Therefore, an optional value is returned, indicating that a value may not exist.

Taking the `sum()` of an empty list makes a little more sense if you assume the sum of an empty stream is 0.0. And, in fact, if you try to `sum()` an empty `DoubleStream`, that's exactly what you'll get:

```
List<Reading> readings2 = Arrays.asList();  // empty list for testing sum
double sum = readings2.stream().mapToDouble(r -> r.value).sum();
System.out.println("Sum of all readings: " + sum);
```

produces the output:

```
Sum of all readings: 0.0
```

By default, the sum of an empty stream is 0. Let's take a look at how you can write your own reduction methods with `reduce()` and have an opportunity to provide a default value in case you have an empty stream.

Using reduce()

The methods `count()`, `average()`, `min()`, `max()`, and `sum()` are all reduction methods already defined on streams. You can also define your own reductions using the `reduce()` method. We'll rewrite the code to sum the values in the (original, nonempty) `readings` list using `reduce()` rather than `sum()`.

What is `sum()` doing to "reduce" a `DoubleStream` of `double` values into one value? It's taking values as they flow from the stream and adding them up, so that's what we need to do in `reduce()` if we want to create our own sum reduction method.

If we look at the type signature of `DoubleStream.reduce()`, we see

```
OptionalDouble reduce(DoubleBinaryOperator op)
```

In the previous chapter, we looked only at `UnaryOperators`, but hopefully, if you studied the Java 8 Functional APIs a bit further, you ran across `BinaryOperators`, which are like `UnaryOperators` except their functional methods take two values as arguments, rather than one. And recall that operators are a special case of `Function` in that the arguments and the return value of the functional method must be the same type.

A `DoubleBinaryOperator` is an operator whose functional method, `applyAsDouble()`, takes two `double` values and returns a `double`.

To write our own method to sum all the values in the stream using `reduce()`, we pass a `DoubleBinaryOperator` to `reduce()`, which adds the two arguments together and returns the sum:

```
OptionalDouble sum =
    readings.stream()
        .mapToDouble(r -> r.value)
        .reduce((v1, v2) -> v1 + v2);
if (sum.isPresent()) {
    System.out.println("Sum of all readings: " + sum.getAsDouble());
}
```

And we see the output:

```
Sum of all readings: 2843.8600000000006
```

The `reduce()` method sums all the values in the stream, two at a time, to get a total sum. In other words, it computes the final result by "accumulating" the values coming in from the stream.

You might wonder, well, how does this work when we are streaming elements one at a time? If you're thinking of the assembly-line analogy, you might be thinking of `reduce()` as the last station on the assembly line and imagining `reduce()` getting one element at a time.

Again, think of reductions as *accumulators:* they accumulate values from the stream so they can compute one value. It's as if the final station in the assembly line is putting all the values from the stream into one big box in order to do the terminal operation on them.

Don't forget: this analogy of the assembly line isn't quite correct. We think of elements flowing through a stream one at a time, like items on an assembly line, but in reality, it doesn't quite work that way. There is no data in a stream; we're simply defining operations on the stream's source (a data structure) that happen in a particular sequence and then get a result by terminating with a reduction. Although it's helpful to think of it like a stream of water or an assembly line as a way to understand streams, how the JDK handles that computation internally is probably quite different: with optimizations and even parallelization, the reality is not like the analogy.

Take another look at the code above and notice that when we switched from `sum()` to `reduce()`, we had to go back to using `OptionalDouble` for the result. That's because `reduce()` is a general reduction function that takes any `DoubleBinaryOperator`. We know that the `applyAsDouble()` method must produce a number, but in the situation where the stream is empty, that `applyAsDouble()` method will never get called, so we need a way to say

there might not be a result at all. Again, that's what the `OptionalDouble` says.

There's another `reduce()` method in `DoubleStream` that takes an *identity* along with an accumulator function and returns a `double`:

```
double reduce(double identity, DoubleBinaryOperator op)
```

The identity argument serves two roles: it provides an initial value and a value to return if the stream is empty.

If you provide the identity value, you're providing an initial value for the result of applying the accumulator function, `op`. The sum is computed by adding values from the stream to this initial value, and when the stream is empty, the identity provides a default result for the sum. For a method that produces a sum, it makes sense for that identity value to be 0.0. Because we have a value to return when the stream is empty, we no longer have to use an `OptionalDouble` because we'll always get a value back:

```
double sum = readings.stream()
   .mapToDouble(r -> r.value)
   .reduce(0.0, (v1, v2) -> v1 + v2);              // provide an identity
value
System.out.println("Sum of all readings: " + sum); // print 0.0 if stream
                                                   // is empty
```

Associative Accumulations

We just showed you how to use `reduce()` to build your own function that sums up the values in a stream. So you might be thinking, hey I could try the same thing for average. You might even try writing the following code:

```
OptionalDouble avgWithReduce =
    readings.stream()                                  // stream the readings
        .mapToDouble(r -> r.value)                     // map to double values
        .filter(v -> v >= 406.0 && v < 407.00)         // filter 406 values
        .reduce((v1, v2) -> (v1 + v2) / 2);            // take the average
if (avgWithReduce.isPresent()) {
    System.out.println("Average of 406 readings: " +
    avgWithReduce.getAsDouble());
} else {
   System.out.println("Empty optional!");
}
```

What you might see is

```
Average of 406 readings: 406.31125
```

Not only is that answer different from the one we got before (which was 406.2125), but you also might get a different answer from ours. What is going on here?

Let's take a look at how we replaced the call to `average()` with a call to `reduce()`:

```
reduce((v1, v2) -> (v1 + v2) / 2)
```

As we did with sum, we're passing a `DoubleBinaryOperator` to `reduce`, whose functional method takes two values. We're summing those two values and dividing by two. The idea is that `reduce()` operates on two values from the stream at a time, so we'll get an average for two values and then average that result with the next value that comes in from the stream. It sounds like it should work, but it clearly doesn't. Something's up.

The problem is that `average()`—unlike `sum()`, `max()`, and `min()`—is not *associative*. If you stretch your mind back to high-school algebra, you might

(or might not) recall that an operator is associative if the following is true:

```
(A operator B) operator C = A operator (B operator C)
```

Addition is associative, thus,

```
(1 + 2) + 3 = 1 + (2 + 3)
```

The parentheses indicate which operators to apply first. So, on the left, we compute 1 + 2, get 3, and then add 3, to get 6. On the right, we compute 2 + 3, get 5, and then add 1, to get 6 again. We get the same result; therefore, addition (i.e., `sum()`) is associative.

Our problem with the code above that replaces the call to `average()` with a `reduce()`:

```
reduce((v1, v2) -> (v1 + v2) / 2)
```

is that average is *not* an associative operator. That might seem odd; it feels like it should be associative, but it's really not.

The average of 1, 2, and 3 is 2, right?

1 + 2 + 3 = 6. 6 / 3 = 2

Yep.

Okay, now try this:

The average of 1 and 2 = 1.5

The average of 1.5 and 3 = 2.25

And this:

The average of 2 and 3 = 2.5

The average of 2.5 and 1 = 1.75

Average is clearly *not* associative!

Reduction operations *must* be associative in order to work correctly with streams. That's why you can't define your own average function with `reduce()`; average is not associative. So to take the average of a stream, you

must use the `average()` method.

What about `min()` and `max()`? They are both associative reductions. Try writing your own versions using `reduce()`. Convince yourself that these operators really are associative and then try writing the code to find the min and max of a stream of values.

map-filter-reduce Methods

With map, as with all things streams, there are a variety of options depending on the types you are working with. Review Table 9-2 to get a sense of the patterns of names used with streams and primitive streams. (Note: This is not an exhaustive list of all the ways to map, filter, and reduce, so see each interface for more details and options.)

TABLE 9-2 Methods to Map, Filter, and Reduce

Interface	Method	Returns...
Stream	map(Function<? super T, ? extends R> mapper	Stream<R>
Stream	mapToDouble(ToDoubleFunction<? Super T> mapper),	DoubleStream,
	mapToInt(ToIntFunction<? Super T> mapper),	IntStream,
	mapToLong(ToLongFunction<? Super T> mapper)	LongStream
DoubleStream	map(DoubleUnaryOperator mapper),	DoubleStream,
	mapToInt(DoubleToIntFunction mapper),	IntStream,
	mapToLong(DoubleToLongFunction mapper),	LongStream,
	mapToObj(DoubleFunction<? Extends U> mapper)	Stream<U>

Interface	Method	Returns...
IntStream	Like DoubleStream, but for ints	
LongStream	Like DoubleStream, but for longs	
Stream	filter(Predicate<? super T> predicate)	Stream<T>
DoubleStream	filter(DoublePredicate predicate)	DoubleStream
IntStream	filter(IntPredicate predicate)	IntStream
LongStream	filter(LongPredicate predicate)	LongStream
Stream	reduce(BinaryOperator<T> accumulator), reduce(T identity, BinaryOperator<T> Accumulator)	Optional<T>, T
DoubleStream	reduce(DoubleBinaryOperator op), reduce(double identity, DoubleBinaryOperator op)	OptionalDouble, double
IntStream	reduce(IntBinaryOperator op), reduce(int identity, IntBinaryOperator op)	OptionalInt, int
LongStream	reduce(LongBinaryOperator op), reduce(long identity, LongBinaryOperator op)	OptionalLong, long
Stream	count()	long
DoubleStream, IntStream, LongStream	count(), sum()	long, double, int, long
DoubleStream, IntStream, LongStream	average()	OptionalDouble
DoubleStream, IntStream, LongStream	max(), min()	OptionalDouble, OptionalInt, OptionalLong

The key idea to remember with map-filter-reduce is the purpose of each: `map()` maps values to modify the type or create a new value from the existing value, but without changing the number of elements in the stream you're working with; `filter()` potentially winnows the values you're working with, depending on the result of the `Predicate` test; and `reduce()` (and its equivalents) changes the stream of values to one value (or a collection of values) as a terminal operation.

CERTIFICATION OBJECTIVE

Optionals (OCP Objective 5.3)

5.3 Develop code that uses the Optional class.

We've explored optionals a bit, and now it's time to dig in a bit more as optionals are an important part of working with streams and you'll encounter optionals on the exam. You'll need to make sure you know which stream operations return optionals and the correct type for those optionals. Table 9-3 summarizes the methods that result in optionals that we cover in this chapter.

TABLE 9-3 Stream Methods That Return Optionals

Interface	Method	Returns...
Stream	findAny()	Optional<T>
Stream	findFirst()	Optional<T>
Stream	max(Comparator<? super T> comparator)	Optional<T>
Stream	min(Comparator<? super T> comparator)	Optional<T>
Stream	reduce(BinaryOperator<T> accumulator)	Optional<T>
DoubleStream	average()	OptionalDouble
DoubleStream	findAny(), findFirst()	OptionalDouble
DoubleStream	max(), min()	OptionalDouble
DoubleStream	reduce(DoubleBinaryOperator op)	OptionalDouble
IntStream, LongStream	Similar methods to DoubleStream	OptionalInt, OptionalLong

What is an optional? It's a container that may or may not contain a value. You can think of an optional as a wrapper around a value and that wrapper provides various methods to determine whether a value is there and, if it is, to get that value.

Optionals show up a lot with streams, because, as you've seen, if a stream is empty at any part of the pipeline, then there's a chance that no value will

be returned by the terminal operation of a stream pipeline. Optionals provide a way for the JDK to handle that situation without having to throw a runtime exception.

The types of optionals include `Optional<T>` (for objects) and three primitive optionals, `OptionalDouble`, `OptionalInt`, and `OptionalLong`.

You've seen how we used `OptionalDouble` with the `average()` operation; now let's take a look at how we might use `Optional<Double>` for comparison:

```
Stream<Double> doublesStream =
      Stream.of(1.0, 2.0, 3.0, 4.0);              // stream of doubles
Optional<Double> aNum = doublesStream.findFirst(); // first the first double
if (aNum.isPresent()) {                           // check to see if aNum
                                                  // has a value
   System.out.println("First number from the doubles stream: " + aNum.get());
} else {
   System.out.println("Doubles stream is empty");
}
```

You should see the output:

```
First number from the doubles stream: 1.0
```

We're creating a stream of `Double` values, using `Stream.of()` (notice that the `doubles` are being autoboxed into `Doubles` to create the `Stream<Double>`), and then we're using the stream terminal operation method, `findFirst()`, to find the first element of the stream.

We'll get into `findFirst()` in more detail later, but what it does, as you might guess, is find the first element of the stream. Typically, you'll use `findFirst()` after filtering the stream, but here, we just want to return the first element of the original stream.

That element is a `Double` object, so the type we use for the return value from `findFirst()` is `Optional<Double>`. Note that `OptionalDouble` will not

work here! Why? Because `findFirst()` is operating on a stream of `Doubles`, not a stream of `doubles`. Again, pay very close attention to the types here.

Just like before, now that we have an optional value, we need to first test to see whether it's present, with `isPresent()`, before trying to get that `Double` value from the `Optional`. Because we're working with a `Double` object rather than a primitive type, the method we use to get the value is `get()`.

Recall that the method for `OptionalDouble` is `getAsDouble()`, and note the patterns for the method names for optionals and primitive optionals:

- ***Get a value from an `Optional<T>`: `get()`***
- ***Get a value from an `OptionalDouble`: `getAsDouble()`***
- ***Get a value from an `OptionalInt`: `getAsInt()`***
- ***Get a value from an `OptionalLong`: `getAsLong()`***

Another way to test to see whether an optional contains a value and, if it does, to then get that value, is the `ifPresent()` method. This method takes a `Consumer` and tests to see whether the optional value is present; if it is, then the unwrapped value is passed to the `Consumer`:

```
Stream<Double> doublesStream = Stream.of(1.0, 2.0, 3.0, 4.0);
Optional<Double> aNum = doublesStream.findFirst();
aNum.ifPresent(n ->
          System.out.println("First number from the doubles stream: " + n));
```

Note here that the type of `n` is `Double`, not `Optional<Double>`, so to print `n`, we just write `n`, not `n.get()`. You'll see the output:

```
First number from the doubles stream: 1.0
```

If your `doublesStream` is empty, then you will not see any output at all.

You can create your own optionals, too. For instance, here's how to create an `Optional<Dog>` (using our previous class for `Dog`):

```
Dog boi = new Dog("boi", 30, 6); // create a dog named "boi" with weight 30,
                                 // age 6
Optional<Dog> optionalBoi = Optional.of(boi);
optionalBoi.ifPresent(System.out::println);
```

This will give you the output:

```
boi is 6 years old and weighs 30 pounds
```

What happens if `boi` happens to be `null`? Try this:

```
boi = null;                                    // boi is null!
Optional<Dog> optionalBoi = Optional.of(boi);  // potential problem here
optionalBoi.ifPresent(System.out::println);
```

Run it and you'll see the error:

```
Exception in thread "main" java.lang.NullPointerException
```

If you are in a situation in which you might be creating an optional from a `null` object, then you have to take an extra precaution and use the `Optional.ofNullable()` method rather than `Optional.of()`. The `ofNullable()` method creates the optional if the object you pass in is not `null`; otherwise, it creates an empty optional:

```
boi = null;
Optional<Dog> optionalBoi = Optional.ofNullable(boi); // check for null
optionalBoi.ifPresent(System.out::println);
if (!optionalBoi.isPresent()) System.out.println("Boi must be null");
```

You should see the output:

```
Boi must be null
```

You can also create empty optionals directly. Assume we have a `List` of `Dog` objects. We can create an empty `Optional<Dog>` and then assign it a value later by streaming the `List` of `Dogs`, using `findFirst()` to find the first `Dog`:

```
Optional<Dog> emptyDog = Optional.empty(); // make an empty Dog optional
if (!emptyDog.isPresent()) {
    System.out.println("Empty dog must be empty");
}
emptyDog = dogs.stream().findFirst();      // find the first dog in
                                           // the list, assign it to emptyDog
emptyDog.ifPresent(d -> System.out.println("Empty dog is no longer empty"));
```

Run this code and you'll see

```
Empty dog must be empty
Empty dog is no longer empty
```

Another way to handle an empty optional is the `orElse()` method. With this method you get the value in the optional, or, if that optional is empty, you get the value you specify with the `orElse()` method:

```
Optional<Dog> emptyDog = Optional.empty();
Dog aDog = emptyDog.orElse(new Dog("Default Dog", 50, 10));
System.out.println("A Dog: " + aDog);
```

You will get the output:

```
A Dog: Default Dog is 10 years old and weighs 50 pounds
```

Try changing `emptyDog` to `optionalBoi`:

```
Dog boi = new Dog("boi", 30, 6);                 // create a dog named "boi"
                                                 // with weight 30, age 6
Optional<Dog> optionalBoi = Optional.of(boi);    // not an empty optional
Dog aDog =                                       // get boi, or if optionalBoi
                                                 // is empty, get Default Dog.
   optionalBoi.orElse(new Dog("Default Dog", 50, 10));
System.out.println("A Dog: " + aDog);
```

And now you should see

```
A Dog: boi is 6 years old and weighs 30 pounds
```

because `optionalBoi` does, indeed, contain a `Dog`, `boi`.

Table 9-4 summarizes the `Optional` methods.

TABLE 9-4 Methods of the Optional Class

Class	Method	Returns...
Optional	empty()	Optional<T>
Optional	get()	T
Optional	ifPresent(Consumer<? super T> consumer)	void
Optional	isPresent()	boolean
Optional	of()	Optional<T>
Optional	ofNullable(T value)	Optional<T>
Optional	orElse(T other)	T
OptionalDouble, OptionalInt, OptionalLong	Each has similar methods to the above methods that return primitive optionals or primitives for each type	double, int, long

CERTIFICATION OBJECTIVE

Searching and Sorting with Streams (OCP Objectives 5.2 and 5.5)

5.2 Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch.
5.5 Sort a collection using Stream API.

The `Stream` interface provides several methods for searching for elements in streams: `allMatch`, `anyMatch`, `noneMatch`, `findFirst`, and `findAny`. All of

these operations are terminal operations; that is, they all return a single value, not a stream.

In addition, all of these operations are *short-circuiting* operations: that is, as soon as the result is determined, then the operation stops. So, for instance, if you are using `allMatch()` to determine whether each element passes a matching test, as soon as an element that doesn't pass the test is found, the operation stops, and the boolean result, `false`, is returned. All these operations can also be parallelized, so all can operate on streams efficiently.

Searching to See Whether an Element Exists

The methods `allMatch()`, `anyMatch()`, and `noneMatch()` all take a `Predicate` to do a matching test and return a boolean. Let's go back to our trusty dogs, and try these methods on a stream of dogs. Here's the `Dog` class:

```
class Dog {
   private String name;
   private int age;
   private int weight;

   public Dog(String name, int weight, int age) {
      this.name = name; this.weight = weight; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return this.name + " is " + this.age + " years old and weighs "
                     + this.weight + " pounds";
   }
}
```

And here's a `List` of dogs:

```
List<Dog> dogs = new ArrayList<>();
Dog boi = new Dog("boi", 30, 6);
Dog clover = new Dog("clover", 35, 12);
Dog aiko = new Dog("aiko", 50, 10);
Dog zooey = new Dog("zooey", 45, 8);
Dog charis = new Dog("charis", 120, 7);
dogs.add(boi); dogs.add(clover); dogs.add(aiko);
dogs.add(zooey); dogs.add(charis);
```

Now that we've got a `List` of dogs, we can stream it and perform stream pipeline operations on the dogs.

First, we'll look for dogs whose weight is greater than 50 pounds and whose names start with `"c"` using `anyMatch()`. The `anyMatch()` method will stop searching as soon as it's found at least one dog whose name starts with "c":

```
boolean cNames =
   dogs.stream()                          // stream the dogs
                                          // keep dogs whose weight > 50
       .filter(d -> d.getWeight() > 50)
                                          // do any dog names start with c?
       .anyMatch(d -> d.getName().startsWith("c"));
System.out.println(
   "Are there any dogs > 50 pounds whose name starts with 'c'? " + cNames);
```

You should see the output:

```
Are there any dogs > 50 pounds whose name starts with 'c'? true
```

The dog that matched must be "charis" because he weighs over 50 pounds.

Now let's use `allMatch()` to find whether all the dogs in the list have an age greater than 5. Here, we map the stream of dogs to a stream of their ages first and then use `allMatch()` to check each dog to make sure each has an age greater than 5795. `allMatch()` will stop and return false as soon as it finds any dog that fails this test.

```
boolean isOlder =
   dogs.stream()
    .mapToInt(d -> d.getAge())     // map from Dog to the Dog's age (integer)
    .allMatch(a -> a > 5);         // do all dogs have an age > 5?
System.out.println("Are all the dogs age older than 5? " + isOlder);
```

All our dogs are older than 5, so you should see

```
Are all the dogs age older than 5? true
```

Finally, we'll use `noneMatch()` to make sure that none of the dogs in the stream are named `"red"`. First, we map the stream of dogs to a stream of the dog names, and then we use `noneMatch()` to check the name:

```
boolean notRed =
   dogs.stream()
    .map(d -> d.getName())             // map from Dog to Dog's name (String)
    .noneMatch(n -> n.equals("red"));  // are any of the dogs named "red"?
System.out.println("None of the dogs are red: " + notRed);
```

Since none of our dogs have the name "red," you should see

```
None of the dogs are red: true
```

Searching to Find and Return an Object

All of these matching methods determine whether a match exists or not. If you want to actually get back a result of a match, then you can use

`findFirst()` or `findAny()`. Neither of these methods takes an argument, so you need to filter first to narrow down the elements you might be searching for. Because `filter()` could potentially filter out all the elements of the stream, leaving you with an empty stream, you can guess that `findFirst()` and `findAny()` both return optionals in case they are called on empty streams; in which case, there is no valid result and the optional will be empty.

Let's use `findAny()` to find any `Dog` in our list of dogs that weighs more than 50 pounds and whose name begins with `"c"`. You already know we have one of these dogs (we had a true result when we did the same test earlier with `anyMatch()`), but this time we want to actually get a `Dog` object back and use it.

```
Optional<Dog> c50 =
   dogs.stream()                                  // stream the dogs
      .filter(d -> d.getWeight() > 50)            // keep dogs with weight > 50
      .filter(d -> d.getName().startsWith("c"))   // keep dogs with name "c"
      .findAny();                                 // pick any dog from the
                                                  // stream
                                                  // and return it
c50.ifPresent(System.out::println);
```

The output is

```
charis is 7 years old and weighs 120 pounds
```

In this example, only one dog passed the tests in both filters; only "charis" weighs more than 50 pounds and has a name beginning with `"c"`. So `findAny()` returns the one dog left in the stream when we call that operation.

But what if there's more than one dog in the stream when we call `findAny()`? For instance, if we look for any dog whose age is older than 5, all the dogs will pass that filter test, so all the dogs will still be in the stream when we call `findAny()`:

```
Optional<Dog> d5 =
    dogs.stream()
        .filter(d -> d.getAge() > 5)
        .findAny();
d5.ifPresent(System.out::println);
```

You'll probably see the output:

```
boi is 6 years old and weighs 30 pounds
```

Of course, boi happens to be the first dog in the stream, but technically `findAny()` could return *any* of the dogs in the stream. There's no guarantee it will be the first one (particularly when you parallelize the stream, which we'll get to later on).

Don't forget that all these matching and finding methods are short-circuiting. That means that as soon as `findAny()` finds a dog that matches, everything stops.

Let's add a `peek()` at just the right spot to see what's happening in the stream when we run this code. Give this a try:

```
Optional<Dog> d5 =
   dogs.stream()
      .filter(d -> d.getAge() > 5).peek(System.out::println)
      .findAny();
d5.ifPresent(System.out::println);
```

Remember that `peek()` allows you to "peek" into the stream. It takes a `Consumer` and returns exactly the same stream as you call it on, so it makes no changes to anything. It's simply a window to what's flowing

by on the stream.

Run this and you'll probably see

```
boi is 6 years old and weighs 30 pounds   // from the peek
boi is 6 years old and weighs 30 pounds   // from the final print
```

So if you're asked on the exam what output you'll see with a `peek()`, make sure you understand how this works. You might be tempted to choose an answer that shows all the dogs printed with the peek and then `boi` for the final print, but that wouldn't be correct.

Sorting

We often want to sort things stored in data structures, so it makes sense we might want to sort elements flowing through a stream pipeline, too. The `Stream` interface includes a `sorted()` method that you can use to sort a stream of elements by natural order or by providing a `Comparator` to determine the order.

Sorting by natural order is easy: just add the `sorted()` operator into the stream pipeline, like this:

```
Stream.of("Jerry", "George", "Kramer", "Elaine")
    .sorted()
    .forEach(System.out::println);
```

If you run this code, you'll see the output:

```
Elaine
George
Jerry
Kramer
```

What if you want to sort more complex objects, like `Duck` objects? Here's a `Duck`:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here…
   public String toString() {
      return (getName() + " is " + getColor() + " and is "
            + getAge() + " years old.");
   }
   @Override                    // describe how to sort Ducks
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}
```

We've gone ahead and made the `Duck` a `Comparable` and implemented the `compareTo()` method that sorts by name so we can sort these `Ducks`:

```
List<Duck> ducks = Arrays.asList(     // create a List of Ducks
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2)
);
ducks.stream()
    .sorted()                         // sort ducks by name
    .forEach(System.out::println);    // print them
```

Run this code and you should see

```
Elaine is white and is 2 years old.
George is brown and is 4 years old.
Jerry is yellow and is 3 years old.
Kramer is mottled and is 6 years old.
```

The `compareTo()` method we provided in the `Duck` class sorts the ducks by their names.

What if you want to change the sort when you stream the ducks? You can use `sorted()` with a `Comparator`:

```
ducks.stream()
    .sorted((d1, d2) -> d1.getAge() - d2.getAge())    // sort ducks by age
    .forEach(System.out::println);
```

Here, you're providing a `Comparator` to `sorted()` that it will use to sort the `Ducks` (instead of the `compareTo()` method in the `Duck` class). The result is

```
Elaine is white and is 2 years old.
Jerry is yellow and is 3 years old.
George is brown and is 4 years old.
Kramer is mottled and is 6 years old.
```

Now the ducks are sorted by age.

Comparators are handy because they can be defined separately from the class, like we did to sort the ducks by age rather than by name (which is the default sort order, defined by the `Comparable`).

Above, we passed the `Comparator` directly to the `sorted()` method using a lambda expression. If we wanted to, we could write out the `Comparator` separately, assign it to a variable, and then pass that variable to `sorted()`, like this:

```
Comparator<Duck> byAgeLambda = (d1, d2) -> d1.getAge() - d2.getAge();
ducks.stream()
    .sorted(byAgeLambda)          // pass the Comparator to sorted
    .forEach(System.out::println);
```

You've seen us use comparators like this before (in the previous chapter). `Comparator` also has some handy static methods that you're likely to see when defining comparators for use with streams: `comparing()`, `reversed()`, and `thenComparing()`.

Let's write three `Comparators` to sort ducks by their `color`, by their `name`, and by their `age`, using the `Comparator.comparing()` method. The `comparing()` method takes a `Function` whose functional method expects a property to sort by, like `Duck.age`, as an argument and returns a `Comparator` that compares objects by that property.

```
Comparator<Duck> byColor = Comparator.comparing(Duck::getColor);
Comparator<Duck> byName = Comparator.comparing(Duck::getName);
Comparator<Duck> byAge = Comparator.comparing(Duck::getAge);
```

Here, we're using a method reference to specify each `Function` that `comparing()` will use to get the properties of the `Duck` we want to compare by for each `Comparator`. Now that we have these comparators, we can use them to sort ducks in various combinations.

We can sort by age:

```
ducks.stream().sorted(byAge).forEach(System.out::println);
```

Or sort by age reversed:

```
ducks.stream().sorted(byAge.reversed()).forEach(System.out::println);
```

Note that `reversed()` is a method of the `Comparator`, that returns a new `Comparator` that has the reverse ordering.

We can also sort by one property and then by another property, using the `thenComparing()` method. `thenComparing()` returns a `Comparator` that compares first by the `Comparator` you call it on and then by the `Comparator` you pass in. To sort by name, and then by age, we'll write:

```
byName.thenComparing(byAge)
```

Before we do that, let's add a few more `Ducks`:

```
List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Jerry", "mottled", 10),
    new Duck("George", "white", 12),
    new Duck("Kramer", "brown", 11),
    new Duck("Elaine", "brown", 13)
);
```

We have our original four ducks, as well as four new ducks with the same names but different colors and all much older. (Think of these as the "bizarro" ducks.)

Now let's sort the ducks, first by name and then by age:

```
ducks.stream()
     .sorted(byName.thenComparing(byAge))
     .forEach(System.out::println);
```

The result is:

```
Elaine is white and is 2 years old.
Elaine is brown and is 13 years old.
George is brown and is 4 years old.
George is white and is 12 years old.
Jerry is yellow and is 3 years old.
Jerry is mottled and is 10 years old.
Kramer is mottled and is 6 years old.
Kramer is brown and is 11 years old.
```

And, of course, we could sort by color and then by age:

```
ducks.stream()
     .sorted(byColor.thenComparing(byAge))
     .forEach(System.out::println);
```

Or a variety of other combinations.

Finally, another method of `Stream` that comes in handy when sorting streams is `distinct()`. This method returns a stream with distinct elements, so if an element is repeated in the stream, you'll end up with only one of them.

Let's say, for instance, that you want to see how many different colors

your ducks are. You only need to see the color once, even if you have multiple ducks with the same color.

Here's a revised List of Ducks:

```
List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

You can see that we have 1 yellow, 2 brown, 2 mottled, and 2 white ducks.

To get the list of distinct colors of ducks, we'll first map each `Duck` to its color, then use `distinct()` to make sure we get a stream of distinct color `String`s, and then print those:

```
ducks.stream()
    .map(d -> d.getColor())          // get the duck colors
    .distinct()                      // make sure there are no repeats!
    .forEach(System.out::println);   // print the colors
```

And we see

```
yellow
brown
mottled
white
```

exam Watch

We slipped some method references into this section: `Duck::getColor, Duck::getName, Duck::getAge,` and `System.out::println`. Recall from the previous chapter when we described how `System.out::println` is* a method reference: a *shorthand for a lambda expression. For example, we replaced a lambda expression like this:

```
Stream.of("Elain","Jerry", "George", "Kramer")
    .forEach(n -> System.out.println(n));      // uses a lambda expression
```

with a method reference like this:

```
Stream.of("Elain","Jerry", "George", "Kramer")
    .forEach(System.out::println);             // uses a method reference
```

Method references simply replace lambda expressions that do nothing except call another function. For this instance method reference, System.out is the class and `println` is the instance method. We use the `"::"` syntax to indicate the method reference.

Take another look at `Duck::getColor, Duck::getName,` and `Duck::getAge`. These, too, are method references to instance methods. They are a type of method reference that refers to a nonstatic method of an instance of a type. If we wrote those method references out as lambda expressions, they'd look like this:

```
Comparator<Duck> byColor = Comparator.comparing(d -> d.getColor());
Comparator<Duck> byName = Comparator.comparing(d -> d.getName());
Comparator<Duck> byAge = Comparator.comparing(d -> d.getAge());
```

So, for instance, the lambda `d -> d.getColor()` is a `Function` whose functional method takes an object of type `Duck` and returns an object of type `String`, mapping a `Duck` to his or her color.

The code

```
Comparator<Duck> byColor = Comparator.comparing(Duck::getColor);
```

uses an instance method reference to replace the lambda, where the instance we're using is an instance of `Duck` and the method is `getColor()`. Remember, method references are shorthand for lambdas that just turn around and call another method.

Methods to Search and Sort Streams

The searching and sorting methods on streams are straightforward; the main thing is to get the types correct! Table 9-5 details these methods.

TABLE 9-5 Searching and Sorting on Streams

Interface	Method	Description
`Stream`	`allMatch(Predicate<? super T> predicate)`	Returns a `boolean`; true if all elements in the stream pass the `Predicate` test. Stops when any element fails the test.
`Stream`	`anyMatch(Predicate<? super T> predicate)`	Returns a `boolean`; true if any of the elements in the stream pass the `Predicate` test. Stops when any element passes the test.
`Stream`	`noneMatch(Predicate<? super T> predicate)`	Returns a `boolean`; true if none of the elements in the stream pass the `Predicate` test. Stops when any element passes the test.
`Stream`	`findFirst()`	Returns an `Optional<T>` with the first element from the stream or an empty optional if the stream is empty.
`Stream`	`findAny()`	Returns an `Optional<T>` with an element from the stream (no guarantees about which one!) or an empty optional if the stream is empty.

`DoubleStream`, `IntStream`, `LongStream`	`allMatch(DoublePredicate predicate),`
	`allMatch(IntPredicate predicate),`
	`allMatch(LongPredicate predicate)`
		Returns a `boolean`; true if all elements in the stream pass the `Predicate` test. Stops when any element fails the test.

`DoubleStream`, `IntStream`, `LongStream`	`anyMatch(DoublePredicate predicate),`
	`anyMatch(IntPredicate predicate),`
	`anyMatch(LongPredicate predicate)`
		Returns a `boolean`; true if any of the elements in the stream pass the `Predicate` test. Stops when any element passes the test.

Interface	Method	Description
`DoubleStream`, `IntStream`, `LongStream`	`noneMatch(DoublePredicate predicate)`, `noneMatch(IntPredicate predicate)`, `noneMatch(LongPredicate predicate)`	Returns a `boolean`; true if none of the elements in the stream pass the `Predicate` test. Stops when any element passes the test.
`DoubleStream`	`findFirst()`	Returns an `OptionalDouble`: the first element in the stream or an empty optional if the stream is empty.
`IntStream`	`findFirst()`	Returns an `OptionalInt`: the first element in the stream or an empty optional if the stream is empty.
`LongStream`	`findFirst()`	Returns an `OptionalLong`: the first element in the stream or an empty optional if the stream is empty.
`DoubleStream`	`findAny()`	Returns an `OptionalDouble`: an element in the stream or an empty optional if the stream is empty.
`IntStream`	`findAny()`	Returns an `OptionalInt`: an element in the stream or an empty optional if the stream is empty.
`LongStream`	`findAny()`	Returns an `OptionalDouble`: an element in the stream or an empty optional if the stream is empty.
`Stream`, `DoubleStream`, `IntStream`, `LongStream`	`sorted()`	Returns a `Stream<T>` (or primitive stream type). Sort elements of a stream by natural order
`Stream`	`sorted(Comparator<? super T> comparator)`	Returns a `Stream<T>`. Sort elements of a stream by the provided `Comparator`.
`Stream`, `DoubleStream`, `IntStream`, `LongStream`	`distinct()`	Returns a `Stream<T>` (or primitive stream type) with distinct elements. For `Stream<T>`, the `Object.equals(Object)` test is used.

Don’t Modify the Source of a Stream

You might have heard that when you’re using Java streams, you are programming in a “functional” style. Aside from being a more declarative way of writing code (thus the stream pipeline versus several statements to specify operations), the key tenet of functional programming is to avoid *side effects*—that is, changing state stored in variables and fields—during computation. Instead, you process data using operations that produce new values (rather than changing old ones). Functional programming style makes it easier to prove the correctness of your program and to replicate results, and avoiding state changes makes it easier to implement optimizations in the compiled code. Although Java is far from being a functional programming language, streams are a step toward a functional style of programming that encourages this idea of avoiding state changes on the original data structure being processed via a stream.

At this point in the chapter, we’re going to share something really important about streams. That is, you should never, ever try to modify the source of a stream from within the stream pipeline.

You might be tempted at times, but don’t. It just isn’t done. Java won’t give you a compile-time error if you try and may not even give you a runtime error, although your results will not be guaranteed. But just don’t do it. Ever.

One reason you might be tempted, say, is if you’re looking for elements that pass a test and you want to modify that element in the source, or perhaps even delete that element from the source. For example, you might want to remove all dogs who weigh less than 50 pounds from the list of dogs:

```
dogs.stream().filter(d -> {
    if (d.getWeight() < 50) {
        dogs.remove(d);
        return false;
    }
    return true;
}).forEach(System.out::println);
```

You might *think* that you'll see a list of the dogs whose weights are greater than or equal to 50, and you might *think* the `dogs` list will now contain only dogs whose weights are greater than or equal to 50, but what you'll probably get instead is something like this:

```
aiko is 10 years old and weighs 50 pounds
Exception in thread "main" java.lang.NullPointerException
```

So what do you do if you want to get a list of all dogs whose weight is greater than or equal to 50 after you've done the stream computation?

The best way to do this is to collect the dogs at the end of the stream pipeline into a new collection. And, of course, Java provides an easy way for you to do just that with the `Stream.collect()` method.

Let's collect all dogs who weigh 50 pounds or more. Remember, here are our dogs:

```
List<Dog> dogs = new ArrayList<>();
Dog boi = new Dog("boi", 30, 6);
Dog clover = new Dog("clover", 35, 12);
Dog aiko = new Dog("aiko", 50, 10);
Dog zooey = new Dog("zooey", 45, 8);
Dog charis = new Dog("charis", 120, 7);
dogs.add(boi); dogs.add(clover); dogs.add(aiko);
dogs.add(zooey); dogs.add(charis);
```

Looking at that list, we'd expect to end up with two dogs: Aiko who weighs 50 pounds and Charis who weighs 120 pounds (that's a big dog!).

Here's how we can use a stream pipeline to filter all dogs who weigh 50 pounds or more and collect them into a new `List`, `heavyDogs`:

```
List<Dog> heavyDogs =
    dogs.stream()                          // stream the dogs
        .filter(d -> d.getWeight() >= 50) // filter only dogs >= 50 pounds
        .collect(Collectors.toList());    // collect the dogs into a new List
```

We can then print this list:

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

And see the output:

```
aiko is 10 years old and weighs 50 pounds
charis is 7 years old and weighs 120 pounds
```

CERTIFICATION OBJECTIVE

Collecting Values from Streams (OCP Objectives 3.8, 5.6, and 9.3)

3.8 Use method references with Streams.
5.6 ave results to a collection using the collect method and group/partition data using the Collectors class.
9.3 Use Stream API with NIO.2.

The `collect()` method is a reduction operation: it reduces a stream into a collection of objects or a value. The `Collector` you pass to the method specifies how to reduce the stream, say, into a `List` or a `Set`. The `Collectors` class provides implementations of `Collector` that each determine how you collect, everything from methods to make a simple list, like `toList()`, to methods that allow you to group items in your `Collection`, resulting in a map so values are organized by a key value.

As you collect, you can

- Group values together based on a `Function` (returns a `Map`)
- Partition values into true/false partitions based on a `Predicate`

(returns a `Map`)

- Map values into other values using a `Function` (returns a `Collector`)
- Join values into a `String`
- Count values as you collect

Note that `Collectors` (with an s) is a helper class that contains implementations of the `Collector` interface. Don't get the two mixed up! This is similar to `Collections`/`Collection`.

Let's make a `Person` class and a nice collection of people with names and ages, so we have some data to work with. Then we'll stream the `Collection` of people and use the various `Collectors` to process the data in a few different ways.

Here's the `Person` class:

```
class Person {
   public String name;
   public Integer age;

   public Person(String name, Integer age) {
      this.name = name; this.age = age;
   }
   public String getName() { return this.name; }
   public Integer getAge() { return this.age; }
   public String toString() {
      return this.name + " is " + this.age + " years old";
   }
}
```

Now, we'll make a bunch of `Persons` and add them to a `List`:

```
Person beth = new Person("Beth", 30);
Person eric = new Person("Eric", 31);
Person deb = new Person("Deb", 31);
```

```
Person liz = new Person("Liz", 30);
Person wendi = new Person("Wendi", 34);
Person kathy = new Person("Kathy", 35);
Person bert = new Person("Bert", 32);
Person bill = new Person("Bill", 34);
Person robert = new Person("Robert", 38);

List<Person> people = new ArrayList<Person>();
people.add(beth); people.add(eric); people.add(deb);
people.add(liz); people.add(wendi); people.add(kathy);
people.add(bert); people.add(bill); people.add(robert);
```

Now that we have a great collection of people with names and ages, we can stream people and use `Collectors` to organize and process our collection. Let's begin with the simplest way to collect people: we'll collect everyone whose age is 34 into a `List`:

```
List<Person> peopleAge34 =
    people.stream()                          // stream the people
        .filter(p -> p.getAge() == 34)       // find people age 34
        .collect(Collectors.toList());       // collect 34s into a new List
System.out.println("People aged 34: "        // print 34s
                + peopleAge34);
```

Here, we're streaming the `List` of `Persons`, using `filter()` with a `Predicate` to test to see whether the `Person`'s age is 34, and collecting those `Persons` who pass that test into a new `List`.

When we print that `List`, we see

```
People aged 34: [Wendi is 34 years old, Bill is 34 years old]
```

There's no guarantee what kind of `List` you get using `Collectors.toList()`. If you specifically want an `ArrayList`, you can use the `toCollection()` method instead, like this:

```
List<Person> peopleAge34 =
   people.stream()                          // stream the people
      .filter(p -> p.getAge() == 34)        // find people age 34
      .collect(Collectors.toCollection(ArrayList::new));   // make an ArrayList
```

And get the same output.

Notice here that we're passing a method reference to the `Collectors. toCollection()` method. Looking at the documentation for `Collectors.toCollection()`, we see that this method takes a `Supplier` whose functional method must return a new empty `Collection` of the "appropriate type." We could write the code like this:

```
ArrayList<Dog> heavyDogs =
    dogs.stream()
      .filter(d -> d.getWeight() >= 50)
      .collect(Collectors.toCollection(() -> new ArrayList<Dog>()));
```

providing the `Supplier () -> new ArrayList<Dog>()` as the argument to `Collectors.toCollection()`. Recall that a method reference is shorthand for a lambda expression that just turns around and calls another method. So by writing `ArrayList::new` instead of the lambda, we are saying the same thing as the lambda expression above: that is, just call the constructor (with new) of an `ArrayList<Dog>` to create a new `ArrayList`. This is a slightly different kind of method reference than you saw earlier in this chapter, when we used instance method references to refer to instance methods, like a `Duck`'s `getColor()`

method with `Duck::getColor`. This form of method reference is the* constructor method reference*, and it's just a shorthand for a lambda that creates a new instance of a class.

Using collect() with Files.lines()

Think back to how we handled getting DVDs from a file using `Files.lines()` earlier (in the section "Create a Stream from a File"). In that example, we streamed `String` data about DVDs from the file and added DVDs one at a time to a `List` in a `forEach()` consumer:

```
List<DVDInfo> dvds = new ArrayList<DVDInfo>();
try (Stream<String> stream = Files.lines(Paths.get(filename))) {
   stream.forEach(line -> {
      String[] dvdItems = line.split("/");
      DVDInfo dvd = new DVDInfo(dvdItems[0], dvdItems[1], dvdItems[2]);
      dvds.add(dvd);  // need a better way to do this!
   });
}
```

Now that you know how to use `collect()`, you can probably think of a better way to do this, right? While the solution above works, it's not the recommended way to collect items from a stream into a `List` because you are modifying an object that's defined outside the stream pipeline—in other words, your stream pipeline has a *side effect*. Using `collect()` is the preferred way to do this, and, as you'll see, along with avoiding an unnecessary side effect, it also makes your code clearer.

Here's another quick example of using `Files.lines()` to stream data from a file, and this time we'll add that data to a `List` using `collect()`. Imagine we have a file, "names.txt," with the following names in it:

```
Jerry
George
Kramer
Elaine
Huey
Louie
Dewey
```

Using `collect()`, we can easily add these names to a `List`, like this:

```
String filename = "names.txt";
try (Stream<String> stream = Files.lines(Paths.get(filename))) {
   List<String> data = stream.collect(Collectors.toList()); // collect names
   data.forEach(System.out::println);                       // print names
} catch(IOException e) {
   System.out.println(e);
}
```

EXERCISE 9-1

Collecting Items in a List

Try rewriting the earlier DVDs example to use `collect()` instead of adding DVDs manually to the `dvds List` in the `forEach()` loop. Just as with the simple names example, you can use `Collectors.toList()` to add DVDs to the `dvds List`. Here's a hint: you'll need to map the lines (`Strings`) you're streaming from the "dvdinfo.txt" file to `DVDInfo` objects before you collect them.

Using `collect()` is a much better, and preferred way, to collect streamed items into a `List`.

Grouping and Partitioning

Now what if we want to group people by age? We can use the `Collectors.groupingBy()` method. We specify *how* to group `Person` objects by passing a `Function` to the `groupingBy()` method, which returns a `Collector` that will collect data elements from the stream and group them in a `Map` by a key, according to that `Function`.

You can think of the `Function` as a "classification function." If we want to group people by age, then we need to pass a `Function` to `groupingBy()` that maps a `Person` to their age:

```
Map<Integer, List<Person>> peopleByAge =
    people.stream()
        .collect(Collectors.groupingBy(Person::getAge));
System.out.println("People by age: " + peopleByAge);
```

When we print the resulting `Map`, we see

```
People by age: {32=[Bert is 32 years old], 34=[Wendi is 34 years old, Bill
is 34 years old], 35=[Kathy is 35 years old], 38=[Robert is 38 years old],
30=[Beth is 30 years old, Liz is 30 years old], 31=[Eric is 31 years old, Deb
is 31 years old]}
```

Notice that the `Map` you get back uses an age (`Integer`) as a key and the value is a `List` of the same type of object in the stream; in this case, that's `Person`. Also notice that the way `System.out.println()` shows us a `Person` in the output is by using the `toString()` method, but the values in the `List` associated with each are not `String`s; they are `Person`s (you can see from the type parameter on `List<Person>` that that is the case).

So now we have a map with `Integer`s (ages) for keys and `List<Person>`s for values, and we can see that, for instance, Bert is 32 and Wendi and Bill are 34, and so on.

The `groupingBy()` method is heavily overloaded with a variety of options

for grouping values in a `Collection`. For instance, you can use a version of `groupingBy()` that takes a classification `Function` as a first argument and a `Collector` as a second argument. That `Collector` argument allows you to reduce the values of the `Map` you get as the result of the `groupingBy()` method, thus reducing each value `List` into another value. Yeah, we know, that's tricky to understand, so let's look at an example.

Here, we're going to group people by age, but rather than create a `List` of the people associated with a certain age in a `Map`, as we did above, now we're going to count the number of people in the `List` associated with a given age and use that value in the resulting `Map` instead. So we have two reductions going on here: we have a `groupingBy()` reduction to group people by age and then we have a `counting()` reduction to count the people in the `List` associated with a particular age:

```
Map<Integer, Long> numPeopleWithAge =
    people.stream()
        .collect(Collectors.groupingBy(   // we're going to group by...
            Person::getAge,               // … age
            Collectors.counting()));      // and count rather than List
System.out.println("People by age: " + numPeopleWithAge);
```

The result of this code is

```
People by age: {32=1, 34=2, 35=1, 38=1, 30=2, 31=2}
```

So now we can see we have 2 people who are 30, 2 people who are 34, 2 people who are 31, and 1 of each other age. To count the people in each list, we use the `Collectors.counting()` method, which returns a `Collector` that simply counts the number of input elements. The result of calling `collect()` on this `Collector` is a `Map` from ages to the number of people of a given age.

What if we want to group people by age, but list only their name rather than the entire `Person` object in the `Map`? That is, we want a map of ages and names (`Map<Integer, List<String>>`) rather than a map of ages and `Person` objects (`Map<Integer, List<Person>>`)?

We can do that by passing a `Collectors.mapping() Collector` as that

second argument to `groupingBy()`:

```
Map<Integer, List<String>> namesByAge =
   people.stream()
      .collect(
         Collectors.groupingBy(
            Person::getAge,          // group by age
            Collectors.mapping(      // map from Person to...
               Person::getName,      // .. name
               Collectors.toList()   // collect names in a list
            )
         )
      );
System.out.println("People by age: " + namesByAge);
```

Here, we're streaming the people and calling the `collect()` method on the stream, passing in the `groupingBy` collector. This `groupingBy()` collector takes two arguments, the `Function` that determines how we're going to group (by `age`) and another `Collector` that tells us how to reduce the values in the `Map` associated with each age. Remember that for the simplest version of `groupingBy()`, each value is just a `List` of `Persons` who are that age. But now we're using the `mapping()` collector to further reduce or modify that `List` of `Persons`.

The `mapping()` method maps each `Person` to another value. What we'd like to do is map a `Person` to their name. Taking a look at the `mapping()` method, we see that its first argument is a `Function` whose functional method takes an object of the type we're mapping from (a `Person`) and returns another object, in this case, the `Person`'s name, a `String`. The second argument to `mapping()` is a collector that tells us what to do with the potentially multiple values we're mapping. Remember that we can have multiple people of the same age, so we're mapping a `List` of `Persons` to a `List` of their names. That second argument specifies how we'd like to collect

those names. We'll use `toList()` to keep it easy, but you could also choose `toCollection()` to create an `ArrayList` or some other collection for the names.

The output we see is

```
People by age: {32=[Bert], 34=[Wendi, Bill], 35=[Kathy], 38=[Robert],
30=[Beth, Liz], 31=[Eric, Deb]}
```

So now our `Map` maps age keys to `Lists` of `String` names.

Partitioning as you collect is essentially a more specialized kind of `groupingBy()`. The `partitioningBy()` method organizes the results into a `Map` like `groupingBy()` does, but `partitioningBy()` takes a `Predicate` rather than a `Function`, so the results are split into two groups (partitions) based on whether the items in the stream pass the test in the `Predicate`. Let's partition our results by the test: is the person older than 34?

```
Map<Boolean, List<Person>> peopleOlderThan34 =
   people.stream()
      .collect(
         Collectors.partitioningBy(p -> p.getAge() > 34));
System.out.println("People > 34: " + peopleOlderThan34);
```

As you might expect, the result is a `Map` that maps `booleans` to `Persons`, so all people 34 or younger will be mapped to the key `false`, and all people older than 34 will be mapped to the key `true`:

```
People > 34: {false=[Beth is 30 years old, Eric is 31 years old, Deb is 31
years old, Liz is 30 years old, Wendi is 34 years old, Bert is 32 years old,
Bill is 34 years old], true=[Kathy is 35 years old, Robert is 38 years old]}
```

Perhaps you can see how powerful streams and collectors are together? Although this code may not feel intuitive to you, it certainly is more concise than writing the equivalent code without using streams to create these `Maps`, and you avoid having to create some intermediate data structures yourself.

The whole idea is that Java can optimize the code used to create the resulting `Maps` behind the scenes so these types of operations become a lot more efficient.

We slipped another example of an instance method reference by you in this section: `Person::getAge`. We did this to group people by age. `groupingBy()` takes a `Function` that maps a Person object to another value to use that value as the mapping key—in this case, telling `groupingBy()` to map people by age.

The method reference `Person::getAge` is an instance method reference that refers to a nonstatic method of an instance of a `Person`. If we wrote that method reference out as a lambda expression, it would look like this:

```
(p) -> p.getAge()
```

So this lambda is a `Function` whose functional method takes an object of type `Person` and returns an object of type `Integer`, mapping a Person to their age.

Summing and Averaging

A couple of other useful collectors are `summingInt()` and `averagingInt()` (and their long and double counterparts). As you might guess, these need to work on numbers, so both take a `ToIntFunction` (that is a `Function` that maps an object to an `int`). To experiment with these, we'll need to add a few more people to our list of people. We're going to add people with the same names so we can group by name and then find the sum of ages and the average of ages. You'll see what we mean in just a moment. Let's first add three more people to our list:

```
Person bill2 = new Person("Bill", 40);
Person beth2 = new Person("Beth", 45);
Person bert2 = new Person("Bert", 38);
people.add(bill2); people.add(beth2); people.add(bert2);
```

Now we have two people named “Bert” in our list; the first Bert (from way back) is age 32, and the second Bert (bert2) is age 38. Likewise, for “Beth” and “Bill,” we now have two people with the name “Beth” and two people with the name “Bill,” each of different ages.

First, we want to sum the ages of the two Berts, the two Beths, and the two Bills, and group by name. Perhaps we’re computing person-years of life experience of all people whose names begin with “B.” (This scenario is completely contrived, of course, but if you were counting occurrences of Scrabble words, then this would make a lot more sense.)

Here’s how we get the sum of the ages of people whose names begin with “B” and group by name:

```
Map<String, Integer> sumOfBAges =
    people.stream()                                  // stream people
        .filter(p -> p.getName().startsWith("B"))    // filter "B" names
        .collect(                                    // collect
            Collectors.groupingBy(                   // groupBy
            Person::getName,                         // … name
            Collectors.summingInt(Person::getAge)    // and sum of ages
    )
  );
System.out.println("People by sum of age: " + sumOfBAges);
```

The output of this code is

```
People by sum of age: {Bill=74, Beth=75, Bert=70}
```

The code took the ages of the two Bills and added them together. It does the same thing with the two Beths and the two Berts, and groups the results by name in a `Map`. So the total age of all the Bills is 74, all the Beths is 75, and all the Berts is 70.

What about the average age of the people whose names begin with "B"? Yep, that's what `averagingInt()` is for:

```
Map<String, Double> avgOfBAges =   // note we need Double not Integer
                                   // for the values!
    people.stream()
          .filter(p -> p.getName().startsWith("B"))
          .collect(
                Collectors.groupingBy(
                      Person::getName,
                      // now average ages instead of sum of ages
                      Collectors.averagingInt(Person::getAge)
                )
        );
System.out.println("People by avg of age: " + avgOfBAges);
```

This code produces the output:

```
People by avg of age: {Bill=37.0, Beth=37.5, Bert=35.0}
```

Now, we've computed the average age of all the Bills, the average age of all the Beths, and the average age of all the Berts, grouped them by name, and collected the results in a `Map`.

It's important to note here that the `averagingInt() Collector` reduces the ages to a `Double`, so the type signature on this `Map` is `Map<String, Double>` (compare with the type signature on the result of the `summingInt() Collector` example, which is `Map<String, Integer>`).

In both these examples, we are first collecting `Person` objects, grouping

people by name, and then doing a second reduction on the `Person` objects that are the values of the first (implicit) `Map` created by the grouping. In the case of `summingInt()`, we are reducing with a `Collector` that sums the ages of all the `Person` objects associated with a given name; in the case of `averagingInt()`, we are reducing with a `Collector` that averages the ages of all the `Person` objects associated with a given name. Sometimes these second reductions on the value portions of the first reduction are called "downstream processing" because they work on the results of the first reduction on a stream.

Counting, joining, maxBy, and minBy

So far we've looked at collectors that, when used with the `collect()` method, reduce a stream to a `Collection` type, like a `List` or a `Map`. There are a few other collectors that work with `collect()` to reduce a stream to a single value, such as a `String`, instead.

We used the `Collectors.counting()` method above to create a `Collector` that counts elements being collected. A simple example of using `counting()` is

```
Long n = people.stream().collect(Collectors.counting());
System.out.println("Count: " + n);
```

This simply counts the items in the people stream. You already know how to do this with the `Stream` method `count()`, and there's no reason to create a `Collector` here to count elements in a stream, so we provide this example just to show how it works at the most basic level. More typically, you'll find `Collectors.counting()` used like we did above, as part of a `groupingBy()` operation.

The `Collectors.joining()` method returns a `Collector` that takes stream elements and concatenates them into a `String` by order in which they are encountered (which may or may not be the order of the original `Collection` you're streaming!).

For instance, we can get the name of every `Person` who's older than 34 and join those names together into one `String` like this:

```
String older34 =
    people.stream()                              // stream people
        .filter(p -> p.getAge() > 34)            // filter for older than 34
        .map(Person::getName)                    // map Person to name
        .collect(Collectors.joining(","));       // join names into one string
System.out.println("Names of people older than 34: " + older34);
```

Here, we're streaming the people `List` (the original list, without the duplicate names), filtering to get only people older than 34, mapping those people to their names (so at that point in the pipeline, we have a stream of `String` names), and then collecting those names into one `String`, with each name separated by a comma:

```
Names of people older than 34: Kathy,Robert
```

The `joining()` method requires as input an object that implements the `CharSequence` interface and includes `String`, which is what we're using here.

The methods `maxBy()` and `minBy()` do what you expect: they collect (reduce) to the max and min of the input elements, respectively. Let's use `maxBy()` to find the oldest person in the `people` stream:

```
Optional<Person> oldest =
  people.stream()
    .collect(Collectors.maxBy((p1, p2) -> p1.getAge() - p2.getAge()));
oldest.ifPresent(p -> System.out.println("Oldest person: " + p));
```

The `maxBy()` method takes a `Comparator` (as does `minBy()`) to compare the elements from the stream as they are being collected and returns a `Collector` that, when used with the `Stream.collect()` method, will reduce the stream to the "max" of the elements in the stream as measured by that `Comparator`.

Notice that `maxBy()` returns an `Optional<Person>`, not a `Person`, because when finding the max as you are collecting a stream, if that stream is empty,

there will be no value. So we use the `ifPresent()` method of the `Optional` to make sure there is a value there before we try to print it out.

The advantage to finding the max of our `Person` stream using `collect()` with the `maxBy()` collector, rather than just using the `max()` terminal operation on the stream, like this:

```
people.stream().mapToInt(p -> p.getAge()).max();
```

is that our result is a `Person` object. Remember, `IntStream.max()` works only on numbers, so we when we mapped our `Person` stream to a stream of `Integers` in order to use `max()` to find the oldest person, we got a number back. By using `maxBy()` with `collect()`, we can find the `Person` object who has the highest age and get the whole `Person` back, not just their age. (We could also use the `Stream.max()` method and provide a comparator to do essentially the same thing.)

Stream Methods to Collect and Their Collectors

Using streams with collectors can get fairly complex, especially when you have multiple downstream operations. On the exam, you'll likely see only one or two levels of nesting when using collectors (e.g., a `groupingBy()` used with a `mapping()`) but be sure to get lots of practice using the `Stream collect()` method with collectors of various kinds.

TABLE 9-6 Using Streams with Collectors

Interface	Method	Description
Stream	`collect(Collector<? super T, A, R> collector)`	Reduces a stream to an instance of a mutable type, like a `Collection`. The `Collector` stands in for a constructor, an accumulator, and a combiner (see below).
Stream	`collect(Supplier<R> supplier, BiConsumer(<R,? super T> accumulator, BiConsumer<R, R> combiner)`	Reduces a stream to an instance of a mutable type. `Supplier` provides a new instance of the mutable type such as a `Collection`; `accumulator` adds each element of the stream to the collection; `combiner` specifies how to merge elements from one collection to another. A `Collector` can be used to capture all three components.

DoubleStream, IntStream, LongStream	`collect(Supplier<R> supplier, ObjDoubleConsumer(<R> accumulator, BiConsumer<R, R> combiner)`
	
	`collect(Supplier<R> supplier, ObjIntConsumer(<R> accumulator, BiConsumer<R, R> combiner)`
	
	`collect(Supplier<R> supplier, ObjLongConsumer(<R> accumulator, BiConsumer<R, R> combiner)`
		Reduces a stream to an instance of a mutable type.

Table 9-6 shows the variations on the `collect()` method. In this section we used only the first variation—the simplest method that takes a `Collector` created with one of the `Collectors` methods [e.g., `toList()`]—but be aware of the other variation where you need to supply supplier, accumulator, and combiner arguments to specify how the `collect()` method reduces elements in case it shows up on the exam. Most of the time, you'll typically use a `Collectors` method to create a `Collector` (which you can think of as an abstraction hiding the details of a supplier, accumulator, and combiner that actually determine how the collect reduction is done) and use that with `collect()`, as we did in this section.

Table 9-7 shows some of the `Collectors` methods you can use to create `Collectors` to collect elements; for more options and variations (which will likely not be on the exam), check out the `Collectors` class in the documentation. Remember that all the methods in the `Collectors` class are static methods that each produce a `Collector`—and don't get those two types mixed up!

TABLE 9-7 Collectors Methods

Method	Description
`toList()`	Returns a `Collector` that accumulates elements into a new `List`.
`toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper)`	Returns a `Collector` that accumulates elements into a new `Map`.
`toSet()`	Returns a `Collector` that accumulates elements into a new `Set`.
`toCollection( Supplier<C> collectionFactory)`	Returns a `Collector` that accumulates elements into a new `Collection`.
`groupingBy(Function<? super T, ? extends K> classifier)`	Returns a `Collector` that implements the group by operation, which groups elements by a classification `Function`, and returns the results in a `Map`. We used this to map `Person` objects by age.
`groupingBy(Function<? super T, ? extends K> classifier, Collector<? super T, A, D> downstream)`	Returns a `Collector` that groups elements according to a classification `Function` and then performs an additional reduction on the values associated with a given key in the resulting `Map`. We used this to count the number of people of a certain age and map the count to age.
`partitioningBy(Predicate<? super T> predicate)`	Returns a `Collector` that partitions elements according to the `Predicate` (into true and false partitions) and returns a `Map`. This is a more specific version of `groupingBy`.
`mapping(Function<? super T, ? extends U> mapper, Collector<? super U, A, R> downstream)`	Modifies a `Collector` that accepts elements of type `U` into a `Collector` that accepts elements of type `T` by applying the mapper `Function` to each element before it's accumulated in the `Collector's` reduction. We used this to map a `Person` object to a `String` (the person's name) before grouping by the person's age.

Method	Description
`summingInt(ToIntFunction<? super T> mapper)`	Returns a `Collector` that computes the sum of input elements. We used this to sum the ages (type `Integer`) of `Person` objects with the same name; we used the mapper to map a `Person` to their age. Similar methods are `summingDouble` and `summingLong` (producing a `Double` sum and a `Long` sum, respectively). If no elements are present, the result is 0.
`averagingInt(ToIntFunction<? super T> mapper)`	Returns a `Collector` that computes the average of input elements. We used this to create a `Map` of `Person` names and their average age (type `Double`). Similar methods are `averagingDouble` and `averagingLong` (both of which produce `Double` values for the average). The `mapper` we used maps a `Person` to their age.
`counting()`	Returns a `Collector` that counts the number of elements and returns a `long`. If no elements are present, the result is 0.
`joining()`	Returns a `Collector` that concatenates elements into a `String`. We used this to concatenate `Person` names to associate a `String` of names with an age.
`maxBy(Comparator<? super T> comparator)`	Returns a `Collector` that returns the maximum element determined by the `Comparator`.
`minBy(Comparator<? super T> comparator)`	Returns a `Collector` that returns the minimum element determined by the `Comparator`.

CERTIFICATION OBJECTIVE

Streams of Streams (OCP Objective 5.7)

5.7 Use flatMap() methods in the Stream API.

Imagine you have a file containing space-separated words in multiple lines. Something like this:

```
rabbits dogs giraffes lions Java tigers
penguins Java deer birds Java Java monkeys
horses whales Java antelope bears Java
Java insects Java raccoons rats zebras
koalas snakes spiders cats hippopotamuses
```

You want to read the file and determine how many times the word "Java" appears in the file.

You already know you can create a stream from a file, so you do that to get started:

```
Stream<String> input = Files.lines(Paths.get("java.txt"));
```

So far so good.

Now, you know the input stream is going to stream one line at a time from the file "java.txt," and so to get words, rather than lines, you can split each line, like this:

```
input.map(line -> line.split(" "))
```

What does this produce? You know the `split()` method on `String` creates an array of `Strings`, and you know `map()` produces a stream, so what this line of code does is take the stream of lines coming from the file, splits each line into a `String[]`, and generates a `Stream` of `String` arrays, which we write as `Stream<String[]>`:

```
Stream<String[]> inputStream = input.map(line -> line.split(" "));
```

So what do we do with a stream of `String` arrays? What we want is the individual strings in each array, so we can filter the words that are equal to "Java" and count them to see how many times "Java" appears in the file. So how do we turn the `Stream<String[]>` into a stream of `Strings`?

What if we stream each of the arrays in the stream? We know we can create a stream from an array with `Arrays.stream()` and that `inputStream` is a stream of `String` arrays. What about mapping each array to a stream, so each array becomes a stream of strings, and then process that? Let's try:

```
inputStream.map(array -> Arrays.stream(array)).forEach(System.out::println);
```

This is what you'll see:

```
java.util.stream.ReferencePipeline$Head@72ea2f77
java.util.stream.ReferencePipeline$Head@33c7353a
java.util.stream.ReferencePipeline$Head@681a9515
java.util.stream.ReferencePipeline$Head@3af49f1c
java.util.stream.ReferencePipeline$Head@19469ea2
```

Hmm, something's not quite right… We expected each word from the file to print, but instead we see what looks like streams. Why?

Well, what we're creating with

```
inputStream.map(array -> Arrays.stream(array))
```

is actually a stream of streams:

```
Stream<Stream<String>> ss =
    inputStream.map(array -> Arrays.stream(array));
```

So when we try to print this out:

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

each item we see displayed is a `Stream<String>`, rather than a `String`. When we streamed each array of `String`s created by the split, each of those streams becomes an element in the main stream that is streaming the arrays. Thus, a stream of streams.

What can we do with a stream of streams? Well, it turns out there's a method just for this situation in the Stream API: `flatMap()`.

The `flatMap()` method is similar to `map()` in that it maps a stream of one type into a stream of another type, but it does something extra; `flatMap()` "flattens" out the streams, essentially concatenating them into one stream. It replaces each stream with its contents, creating one stream from many.

So instead of mapping each array to a stream, we're going to *flat map* each array to a stream. The stream that results from the `flatMap()` is one big flat stream, rather than a stream of streams. That's really hard to see without an example, so let's give this a try:

```
Stream<String> ss =
    inputStream.flatMap(array -> Arrays.stream(array));
ss.forEach(System.out::println);
```

Think of taking a two-dimensional array and flattening it to a one-dimensional array, like this:

```
[ [ 1, 2 ], [ 3, 4 ] ] -> [ 1, 2, 3, 4 ]
```

That's what `flatMap()` does with streams. The result of using `flatMap()` on the stream of arrays is a flat stream of all the contents of each array, which is the words in the file.

If you run that code, you'll see all the words in the file:

```
rabbits
dogs
giraffes
lions
Java
tigers
penguins
Java
deer
...
```

Great! We've made progress. Our goal is count the number of words equal to "Java." Hopefully that task is easier now; you just need to `filter()` and `count()`.

Here's the whole code so you can run it yourself:

```
try {
  long n = Files.lines(Paths.get("java.txt")) // stream lines from the file
     .map(line -> line.split(" "))            // split each line into String[]
     .flatMap(array -> Arrays.stream(array))  // stream arrays, and flatten
     .filter(w -> w.equals("Java"))           // filter the words = "Java"
     .count();                                // count "Java"
   System.out.println("Number of times 'Java' appears: " + n);
} catch (IOException e) {
   System.out.println("Oops, error! " + e.getMessage());
}
```

Run that and you should see that "Java" appears in the file eight times.

The `Stream` interface also includes `flatMapToDouble()`, `flatMapToInt()`, and `flatMapToLong()`, to flat map to a `DoubleStream`, an `IntStream`, and a `LongStream`, respectively, in case you need that. Each of the primitive stream types just mentioned also include their own `flatMap()` methods that take a `DoubleFunction`, an `IntFunction`, and a `LongFunction`, respectively.

What do you think? The final code is actually pretty concise, and it's fairly easy to process the data in the file. It's certainly a different way of thinking about processing data. Do you find the code easier or more difficult to read? It definitely takes some getting used to.

Keeping track of your streams—e.g., What is the type of your stream? Is it a stream of `String`s or a stream of streams?—can get a bit tricky at times. Paying attention to exactly what each method produces so you know what type you're dealing with is important in all aspects of working with streams, as we hope you've discovered in this chapter.

CERTIFICATION OBJECTIVE

Generating Streams (OCP Objective 3.4)

3.4 Collections, Streams, and Filters.

What do you think will happen if you run the following code:

```
Stream.iterate(0, s -> s + 1);
```

Looking at the documentation for `Stream.iterate()`, it says `iterate()` returns an "infinite sequential order Stream." Oh my, did you get yourself an infinite loop here?

Actually no, you didn't. Remember that streams are lazy. `Stream.iterate()` returns a stream, and you know that nothing starts to happen until you tack a terminal operation, like `count()` or `forEach()` or `collect()`, onto the stream pipeline.

The `iterate()` operation creates an infinite sequential `Stream`, starting with the first argument (known as the "seed") followed by elements created by the `UnaryOperator` that you supply as the second argument. In our example above, the stream will generate whole numbers starting at 0, adding 1 to the seed first and then each subsequent number, forever.

Okay, so here's code you probably shouldn't try (okay, *definitely* shouldn't try):

```
Stream
    .iterate(0, s -> s + 1)         // create an infinite stream
    .forEach(System.out::println);  // don't try this at home!
```

Here, the `forEach()` terminal operation will never end because the stream is infinite. Because `iterate()` creates an infinite stream, we need some way to limit how much we get from the stream so we don't get ourselves in trouble. We can do that with the `limit()` operation:

```
Stream
    .iterate(0, s -> s + 1)         // create an infinite stream…
    .limit(4)                       // but limit it to 4 things
    .forEach(System.out::println);  // print the 4 things
```

Now it's safe to run this code, because we are limiting the stream to 4 numbers, starting with 0, so we see the output:

```
0
1
2
3
```

To work safely with infinite streams, you need a short-circuiting operation. That could be `limit()`, like we used above, or it can also be an operation like `findFirst()`, `findAny()`, or `anyMatch()`.

Let's say you've got a sensor that generates an infinite stream of data. We won't actually build one of those, but we'll simulate one, like this:

```
class Sensor {
    String value = "up";
    int i = 0;
    public Sensor() { }
    public String next() {
        i = i + 1;
        return i > 10 ? "down" : "up";
    }
}
```

This sensor has a `next()` method that returns the status of the sensor. We return “up” values for the status until `i` is 10 and return “down” when `i` > 10. The value `i` is incremented each time we call `next()`, so the first 10 results will be “up” and all subsequent results will be “down.” In a real sensor, we’d get data from the sensor and keep returning the latest status (use your imagination).

Now let’s use this sensor with an infinite stream. For this example, we’ll use the `Stream` method `generate()`, which takes a `Supplier`. The `generate()` method generates an infinite stream from elements supplied by the `Supplier`. Our `Supplier` is going to get values from the sensor to stream:

```
Sensor s = new Sensor();
Stream<String> sensorStream = Stream.generate(() -> s.next());
```

So `sensorStream` is an infinite stream of values we get by calling the sensor’s `next()` method. That infinite stream contains 10 “up” elements and a potentially infinite number of “down” elements.

Now we can write code to look for the first “down” value in the infinite stream, like this:

```
Optional<String> result =
  sensorStream
    .filter(v -> v.equals("down"))    // filter to get all down values
    .findFirst();                     // find the first down value and stop
result.ifPresent(System.out::println);
```

The `findFirst()` method is a short-circuiting method, so as soon as we find a “down” value in the stream, everything stops. Whew! We averted another infinity problem. Of course, that assumes there is a “down” value somewhere in the stream; if there’s not, then the stream will keep generating values and `findFirst()` will never complete.

Infinite streams are handy when you’re dealing with a source of potentially infinite data, like a sensor. In practice, because we’d actually like to do something with the data we get from a sensor, we need to process the data into chunks, defined perhaps by a timestamp or by the number of results so far or by a particular value that indicates a change. Use caution when dealing with infinite streams because you need a way to specify a stopping point with a short-circuiting operation in order to perform the reduction and get a result.

Although `Stream.iterate()` is a good way to generate numbers up to a certain point, another way you can generate numbers is with `range()`. In practice, this might actually be more useful. This is a method on the primitive streams `IntStream` and `LongStream`:

```
IntStream numStream = IntStream.range(10, 15); // generate numbers 10...14
numStream.forEach(System.out::println);
```

This produces the output:

```
10
11
12
13
14
```

If you want a stream of numbers inclusive of the second argument, use `rangeClosed()`:

```
IntStream numStream =
    IntStream.rangeClosed(10, 15);   // generate numbers 10...15
numStream.forEach(System.out::println);
```

You'll get the output:

```
10
11
12
13
14
15
```

As you saw, the `limit()` method allows you to limit the number of elements in a stream, so you can, for instance, limit a stream to the first five even numbers in a stream:

```
IntStream evensBefore10 =
    IntStream                                   // rangeClosed is static
        .rangeClosed(0, 20)                     // generate numbers 0...20
        .filter(i -> i % 2 == 0)                // filter evens
        .limit(5);                              // limit to 5 results
evensBefore10.forEach(System.out::println);
```

And get the output:

```
0
2
4
6
8
```

What if you want to skip the first five items instead and print only the even numbers between 10 and 20? You can use `skip()`:

```
IntStream evensAfter10 =
    IntStream
        .rangeClosed(0, 20)             // generate numbers 0...20
        .filter(i -> i % 2 == 0)        // filter evens
        .skip(5);                       // skip first 5 results
evensAfter10.forEach(System.out::println);
```

And you see

```
10
12
14
16
18
20
```

Methods to Generate Streams

All the methods in Table 9-8 are static stream methods to generate streams, except `limit()` and `skip()`, which you'll use to control how many items and which items go into the stream. Be extra careful with `iterate()` and `generate()`; both methods create infinite streams so you need to use these with `limit()` or one of the short-circuiting methods discussed earlier.

TABLE 9-8 Stream Methods to Generate Streams

Method	Description
`iterate(T seed, UnaryOperator<T> function)`	Applies a `function` to a `seed` and then to each subsequent value, and returns an infinite stream consisting of the resulting elements. For example, we created an infinite stream by starting with a `seed` of 0 and using a `function` that adds 1 to its input, resulting in a stream of `Integers`: 0, 1, 2, 3.... Primitive streams `DoubleStream`, `IntStream`, and `LongStream` each have their own versions that use the primitive unary operators for the `function` (`DoubleUnaryOperator`, `IntUnaryOperator`, and `LongUnaryOperator`) and whose seeds are of type `double`, `int`, and `long`, respectively.
`generate(Supplier<T> s)`	Returns an infinite stream consisting of elements generated by the provided `Supplier`. We used this to generate a stream of "up" and "down" values. Primitive streams `DoubleStream`, `IntStream`, and `LongStream` have their own versions that use the primitive suppliers (`DoubleSupplier`, `IntSupplier`, and `LongSupplier`) and generate primitive streams, respectively.

`range(int startInclusive, int endExclusive),`
 	
`range(long startInclusive, long endExclusive)`
	Primitive streams `IntStream` and `LongStream` only: this method produces a sequential stream of elements, incrementing by 1 (e.g., 0, 1, 2, 3...), not including the `endExclusive` value.

`rangeClosed(int startInclusive, int endInclusive),`
 	
`rangeClosed(long startInclusive, long endInclusive)`
	Primitive streams `IntStream` and `LongStream` only: this method produces a sequential stream of elements, incrementing by 1 (e.g., 0, 1, 2, 3...), including the `endInclusive` value.

`limit(long maxSize)`	Returns a stream consisting of the elements of the stream on which it's called, limited to `maxSize` number of elements (for all stream types).
`skip(long n)`	Returns a stream consisting of the elements of the stream on which it's called, after `n` elements from the stream have been skipped (for all stream types).

Caveat Time Again

Streams are fun to play with, and they are cool because they are still kind of new. But just because it's new doesn't mean it's always better. We sold you pretty hard on streams being concise and potentially more efficient for data processing, but there are times when you may not want to use streams. For example, just doing a simple iteration over a `List` might be *slower* using a stream than a `for` loop. It's worth testing to find out for your particular use case. In addition, functional code with streams and lambdas isn't always easy to read, particularly if you're used to reading code written in a more imperative style. So our caveat is this: just because you *can* use streams doesn't mean they are always appropriate. Keep this in mind as you go beyond the exam and into the real world again.

Where streams can really shine is when you can parallelize them. We're going to briefly talk about parallel streams now and then talk about them more later in the book when we talk about concurrency in depth.

CERTIFICATION OBJECTIVE

A Taste of Parallel Streams

10.6 Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

We've mentioned a couple of times in this chapter that it's possible to process streams in parallel. So far, all the streams we've worked with have been serial streams: streams that process one data element at a time. *Parallel streams* can process elements in a stream concurrently. The way Java does this is to split a stream into substreams and then execute the operations defined in the stream pipeline on each of these substreams concurrently, meaning each substream is processed in a thread. If you have multiple cores, then Java will use multiple threads to process the stream and you might get some performance benefits. (We say "might" here because whether you get those benefits depends on your system as well as the data you're processing and how you're processing it. We won't go into an in-depth discussion on the performance tradeoffs of parallel streams, but if you're interested, it's well

worth a deep dive into the literature about what makes for good use cases for parallel streams).

You're going to learn a whole lot more about threads and parallel streams in the upcoming chapter on threads, so for now, we're just going to give you a taste.

Let's say you have some numbers and you want to sum them.

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()                          // stream the numbers
    .mapToInt(n -> n)                            // map Integer to int for sum
    .sum();                                      // sum the ints
System.out.println("Sum is: " + sum);            // result is 55
```

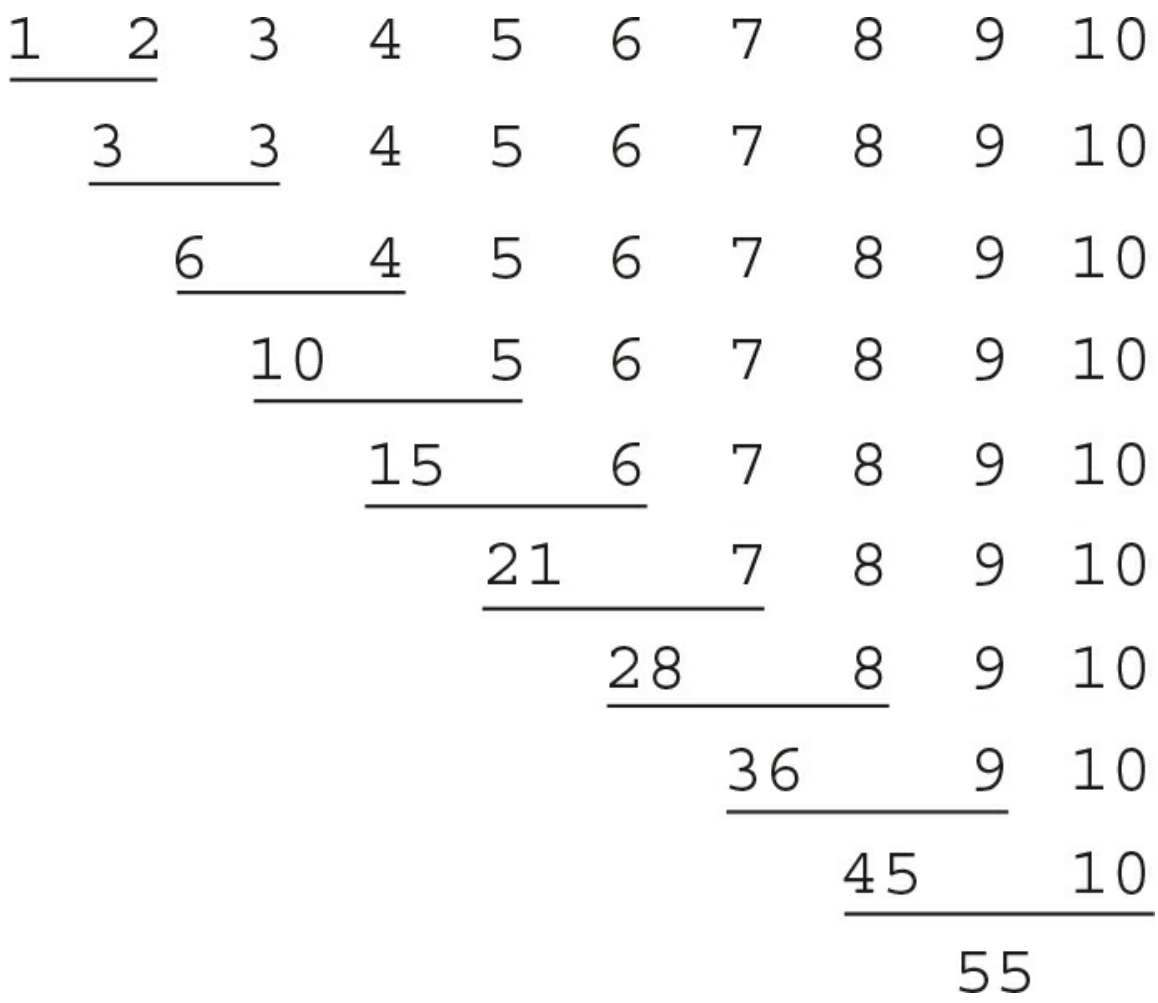
This stream is a serial stream, so each number is added to the sum one at a time.

To make this a parallel stream, we simply call the method `parallel()` on the stream before we sum:

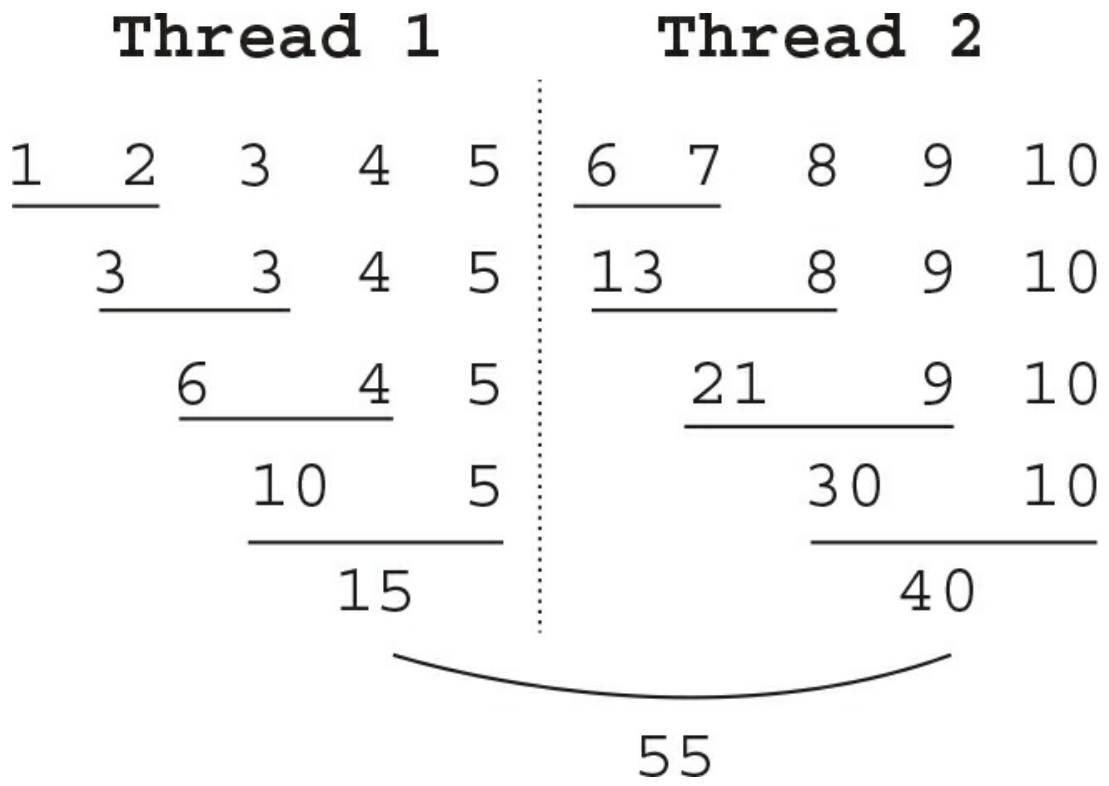
```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()
         .parallel()                        // make the stream parallel
         .mapToInt(n -> n)
         .sum();
System.out.println("Sum is: " + sum);     // result is still 55 (whew!)
```

Now, the stream is processed concurrently, meaning the stream is split into substreams, and (if you have enough cores) each substream is processed on a separate thread. Let's visualize how that works with some diagrams.

For a serial stream, we have one thread handling the entire sum operation:



For a parallel stream, we have multiple threads handling the sum operation concurrently:



Because each thread can compute the sum simultaneously with the other threads, the whole operation should take less time to compute than if you're using a serial stream.

One thing to be careful of with parallel streams is performing an operation that relies on a specific ordering. If you use a parallel stream, you may get unexpected results. Let's compare using `forEach()` to display the items in a stream when the stream is serial and when the stream is parallel.

Here's the code using a serial stream:

```
Arrays.asList("boi", "charis", "zooey", "aiko")
    .stream()                          // stream the names
    .forEach(System.out::println);     // display them
```

The output you see is

```
boi
charis
zooey
aiko
```

In other words, you see the data in the same order as it appears in the source of the stream, the original `List` of names. That is because, by default, the stream is serial, and the elements in the stream are processed in order.

If you run the same code with a parallel stream instead:

```
Arrays.asList("boi", "charis", "zooey", "aiko")
    .stream()                                   // stream the names
    .parallel()                                 // ... in parallel
    .forEach(System.out::println);              // display them
```

You'll potentially get a different ordering in the output every time you run the code. For instance, when we ran this code, we saw

```
zooey
charis
boi
aiko
```

And if we run it again, we'll likely see a different ordering again. That's because the final result depends on the order in which the threads complete, not the order of the original collection. The ordering of the stream didn't matter when we were summing numbers (the sum is the same, independent of order), but the order matters a lot if you are expecting to see the content of the stream in the same order as the content in the original `Collection`.

That is just a small taste of parallel streams, and you'll learn a lot more about them a bit later on in Chapter 11, where we cover concurrency.

CERTIFICATION SUMMARY

A stream is a fairly abstract concept: it is a sequence of elements supporting operations. Knowing the difference between a stream and a data structure is key to understanding streams, and the analogies we used in this chapter—real-life streams and assembly lines—are meant to remind you that streams are for specifying a sequence of operations—the stream pipeline—to perform on a source. Streams never actually hold data like a data structure does. Once you wrap your head around that, then you'll have a much easier time getting the hang of the pipeline operations, of which there are many that you can perform on a stream.

You create a stream from a source, like a `List` or a `File`; specify the sequence of intermediate operations to perform; and finally, provide the terminal operation that terminates the stream and perhaps reduces the stream into a single value or displays the values from the stream in the console. But streams are lazy, so none of the intermediate operations in the pipeline actually do anything until that terminal operation is executed.

The main way you operate on streams is with mapping functions, filter functions, and reduction functions. Mapping functions map values to other values, perhaps changing the type along the way. Filter functions filter out some of the values (or none or all, depending on the filter), so you can eliminate values that don't pass a test. Finally, reduction functions provide the terminal operations on streams to reduce a stream to a single value or to a collection. This map-filter-reduce perspective on stream operations is a handy way to organize the many operations you'll encounter as you work with streams.

Streams can be sorted and searched, like data structures can—but remember you can't modify the source of a stream. When you, say, sort a stream, you need to collect the sorted values into a new data structure if you want to keep them around. Searching produces a value, but because a stream might be empty, you need a way to represent the concept of a value that may or may not be there. This is when you first encountered the optionals: values that are wrapped in the `Optional` type so you can work with empty streams without creating any problems. Of course, there are a variety of ways to get values back out of the `Optional` wrapper so you can work with the result as you normally do.

If you want to keep multiple values from a stream rather than reducing the stream to a single value, then you need a way to collect those values into a new data structure. This is what the `collect()` reduction method does. This method takes a `Collector`, which is an operation that accumulates the elements into a data structure like a `List` or a `Map`. As you're collecting elements, you can group them, partition them, count them, map them, reduce them, and more. Collectors are versatile, providing many different ways of collecting values from the stream into a result. We touched on some of the collection strategies in this chapter for the operations you'll apply on the exam, but you can take this topic a lot further if you want to explore on your own.

Generating streams is a way of pushing a (potentially) infinite number of values through the stream pipeline. However, in practice, you can't actually work with an infinite stream (at least not on today's computers!), so you need to limit it in some way, either by limiting how many items to use or by short-circuiting the stream pipeline with a searching method like `findAny()`.

At the end of the chapter, we took a sneak peek at parallel streams, which are streams that can be operated on concurrently. Parallel streams are where you are likely to see the main performance benefits of using streams in your code, and parallel streams are certainly one of the easiest ways to take advantage of concurrency in your code, as you'll see when we get to Chapter 11 (although, as with all things related to concurrency, there are some potential landmines you'll need to watch out for when you get there).

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

What Is a Stream? (OCP Objective 3.4)

- A stream is a sequence of elements that can be processed with operations.
- A stream is not a data structure; it does not store any data.
- A stream pipeline consists of a source, an optional sequence of intermediate operations, and a terminal operation.

- The source of a stream can be a collection, an array, a file, or one or more values.
- You can create a stream of objects or a stream of primitive values.
- The type of a stream of objects is `Stream<T>`, and the primitive stream types are `DoubleStream`, `IntStream`, and `LongStream`.
- Pay close attention to the difference between `Stream<Double>` and `DoubleStream` and the operations allowed on each (and the same with other primitive type streams).
- Intermediate operations operate on a stream and return a stream.
- Terminal operations terminate the stream, returning a value other than a stream or void.
- `filter()` is an example of an intermediate operation. Pass `filter()` a `Predicate` and use `filter()` to filter values out of the stream that don't pass the `Predicate` test. For example, you can filter out elements that are less than 5 from a stream like this:

```
Stream.of(0, 1, 1, 2, 3, 5, 8, 13).filter(i -> i >= 5);
```

  The stream returned by this filter operation is a stream of numbers: 5, 8, 13.
- `forEach()` is a terminal operation on a stream. `forEach()` takes a `Consumer` and consumes each element of the stream on which it's called. We often use `forEach()` to display the values in the stream at the end of the stream pipeline, like this:

```
Stream.of(0, 1, 1, 2, 3, 5, 8, 13).filter(i -> i >= 5);
```

- `count()` is a terminal operation on a stream that counts the elements in the stream.
- Streams can only be used once. Streams are lightweight objects, so you can easily create another stream.

How to Create a Stream (OCP Objectives 3.5 and 9.3)

- Create a stream from a collection, like a `List`, using the `stream()` method.
- Create a stream from one or more values using `Stream.of()`.
- Create an empty stream using `Stream.empty()`.

- Create a stream from an array using `Arrays.stream()` or `Stream.of()` (and `IntStream.of()`, `LongStream.of()`, and `DoubleStream.of()`).
- Create a stream (`Stream<String>`) from a `File` using `Files.lines()`.
- Streams have several benefits: you can define multiple intermediate operations on streams. The JDK can optimize the operations, so you may see a performance enhancement, especially when you can parallelize the stream.

The Stream Pipeline (OCP Objectives 3.6)

- A stream pipeline consists of a source, a sequence of intermediate operations, and a terminal operation.
- A common analogy to use for the stream pipeline is the assembly line. Each assembly-line station is analogous to an intermediate operation, so that a stream element visits each operation, where it is (usually) modified and passed on until it reaches the terminal operation or is discarded (via a filter).
- Streams are lazy, meaning until you define and execute a terminal operation on the stream, no processing happens.

Operating on Streams (OCP Objectives 3.7 and 5.1)

- Map-filter-reduce is a general abstraction to describe functions that operate on a sequence of elements.
- Stream operations `map()`, `filter()`, and `reduce()` implement mapping operations, filtering operations, and reduction operations (terminal operations), respectively.
- Mapping operations typically modify elements from the stream, transforming an element from one value to another or from one type to another.
- Filter operations typically winnow elements from the stream so that any element that doesn’t pass a test is discarded from the stream.
- Reduction operations reduce the stream to a single value, or a collection.

- `map()` and `filter()`, and their variations, are intermediate operations.
- `reduce()` and its variations are terminal operations.
- `map()` takes a `Function`, which takes an input value and produces an output value.
- `filter()` takes a `Predicate`, which tests the input value and passes the stream element on to the next intermediate operation if the result of the test is true.
- The methods `average()`, `count()`, `sum()`, `max()`, and `min()` are predefined reductions on streams.
- `sum()` and `average()` are defined only on the primitive stream types.
- `max()` and `min()` are defined on all streams. On `Stream<T>`, `max()` and `min()` take a `Comparator` to determine which stream element is the maximum or minimum, respectively.
- `reduce()`takes an (optional) identity value and a `BinaryOperator` to reduce the stream to one value. If the identity value is provided, this is used as the result if the stream is empty; otherwise, the identity is used as the basis for the `BinaryOperator` accumulator. If no identity value is provided, `reduce()` returns an `Optional`.
- The `peek()` method is a way to "peek" into the values currently in the stream. The method makes no changes to the values in the stream and is often used with a `Consumer` that displays the values to the console for debugging purposes. `peek()` should not be used in production code.

Map-Filter-Reduce with average() and Optionals (OCP Objectives 5.3 and 5.4)

- Some stream methods produce `Optional` values.
- `Optional` is a wrapper around a value that may or may not be there.
- `average()` is an example of a reduction operation that produces an optional value, an `OptionalDouble`.
- The `reduce()` method used without an identity argument produces an optional because there may be no value if the stream is empty.
- If you supply an identity argument to `reduce()`, the method will

produce a value (not an optional), and use the identity value if the stream is empty.

- The `sum()` method does not produce an optional because, by default, it uses 0 as the identity value.
- The `average()` method cannot be replaced with your own implementation using `reduce()` because average is not an associative operation. All `reduce()` operations must be associative operations, meaning you can accumulate the elements for the reduction in any order and get the same result. Sum is an example of an associative operation, so you could implement `sum()` yourself using `reduce()`.

Optionals (OCP Objective 5.3)

- An optional is a container, or wrapper, that may or may not contain a value.
- Operations that produce no value if a stream is empty produce optionals. For example, `findFirst()`, `findAny()`, `max()`, `min()`, and `average()` all produce optionals.
- Before getting a value from an optional, first test to see if the optional contains a value with `isPresent()`.
- The types of optionals are `Optional<T>` (for objects) and primitive optionals, `OptionalDouble`, `OptionalInt`, and `OptionalLong`.
- To get a value from a nonempty optional, use `get()`, `getAsDouble()`, `getAsInt()`, or `getAsLong()` for the different types of optionals.
- The `ifPresent()` method tests to see whether an optional is present and then passes the unwrapped value to the `Consumer` argument to `ifPresent()`.
- Most of the time optionals are created by stream operations; however, you can create your own optional using `Optional.of()`.
- The `ofNullable()` method creates an `Optional` value from an object, first testing to see whether the object is `null`. If the object is `null`, then an empty `Optional` is created.
- You can create your own empty `Optional` with `Optional.empty()`.
- Use the `orElse()` method to get a value from an `Optional` and

provide a default value to use if the `Optional` is empty.

Searching and Sorting with Streams (OCP Objectives 5.2 and 5.5)

- Searching operations on streams are short-circuiting, meaning the stream processing stops once the `Predicate` test passed to the search method is passed.
- The search methods on streams include `allMatch()`, `anyMatch()`, `noneMatch()`, `findFirst()`, and `findAny()`.
- The methods `allMatch()`, `anyMatch()`, and `noneMatch()` are terminal operations on a stream that return a boolean indicating if a match was found (or not).
- The methods `findFirst()`and `findAny()`are terminal operations that return an `Optional` value whose parameterized type depends on the type of the stream. In case no item is found, the value returned is an empty `Optional`.
- You can sort elements of a stream using the `sorted()` method. By default, the `sorted()` method uses natural ordering. If the objects in the source of the stream are `Comparable`, the sort will use that order. If the elements of a stream are primitive, the natural order for sorting is used for `double`, `int`, and `long`.
- You can pass a `Comparator` to the `sorted()` method to override the natural sort order and define your own sort order.
- `Comparator` is a functional interface with one functional method, `compare()`, so you can use a lambda expression to pass a `Comparator` to the `sorted()` method.
- You can use the `Comparator.comparing()` method with a `Function` to create a `Comparator` to use with `sorted()`.
- Use `thenComparing()` to combine comparators for sorting by one value then another.
- Instance method references refer to methods of an instance. Use the `::` syntax to create a method reference as a shorthand for a lambda that simply calls another method. For example, you can replace a lambda expression that calls the `getColor()` method of a `Duck` instance:

```
d -> d.getColor()
```

with an instance method reference:

```
Duck::getColor
```

- Streams are a functional style of programming in which you must be careful to avoid side effects. Stream patterns encourage you to avoid modifying the source of the stream in any stream pipeline operations, and in fact, modifying the source of a stream can lead to unexpected results.

Collecting Values from Streams (OCP Objectives 3.8 and 5.6)

- The source of a stream is often a `Collection`. Many reduction methods on streams, such as `sum()` and `average()`, turn stream elements into a single value. You can collect the elements in a stream into a new `Collection` using the stream method `collect()`.
- The `collect()`stream method is a reduction (terminal) operation.
- A `Collector` is an operation that accumulates elements into a result. You can specify a `Collector` with four functions: a supplier, an accumulator, a combiner, and a finisher (optional). Or, you can use one of the methods in the `Collectors` class to create a `Collector`.
- The `Collectors` class has ready-made `Collectors`, such as a `Collector` to accumulate elements into a `List`.
- Use the `Collectors.toList()` method to create a `Collector` that accumulates stream elements into a `List`. `Collectors.toMap()` and `Collectors.toSet()` accumulate stream elements into a `Map` and `Set`, respectively.
- If you want a more specific `Collection` type, use the `Collectors.toCollection()` method, passing a `Supplier` that provides a new instance of the `Collection` type you want (such as `ArrayList`).
  - The method reference `ArrayList::new` is a constructor method reference that is shorthand for a lambda expression `() -> new ArrayList<T>()` (where `T` is a valid type parameter, such as `String`, and depends on the type of the elements in the stream).

- You can use constructor method references as `Suppliers` for the `Collectors.toCollection()` method.

- Use the `Collectors.groupingBy()` method to group elements as you collect them. The `Collectors.groupingBy()` method returns a `Collector` that produces a `Map`.
- Use the `Collectors.partitioningBy()` method to group elements into two partitions as you collect them. The `Collectors.partitioningBy()` method returns a `Collector` that produces a `Map` with two keys, `true` and `false`.
- The `Collectors.groupingBy()` method can accept downstream collectors to further reduce results. For instance, instead of creating a `Map` that maps age keys to a `List` of `Person` objects that have that age, you can use the `Collectors.counting()` method to reduce the `List` of `Person` objects to their count, resulting in a `Map` from age keys to number of `Persons` with that age values.
- `Collectors.counting()` returns a `Collector` that counts the number of input elements and returns a `Long`.
- You can use `Collectors.mapping()` as a downstream `Collector` that maps values from one type to another before they are accumulated. For instance, we used `Collectors.mapping()` to create a `Collector` that maps `Person` objects to names and to collect them in a `List`.
- `Collectors.summingInt()` and `Collectors.averagingInt()`, and their `Double` and `Long` counterparts, produce `Collectors` that sum values and average values, respectively. These can be used as downstream collectors.
- `Collectors.joining()` produces a `Collector` that concatenates `String` elements (and any `CharSequence`-based types) to one `String`.
- `Collectors.counting()` produces a `Collector` that counts elements and returns a `Long`.
- `Collectors.maxBy()` and `Collectors.minBy()` produce a `Collector` that finds the maximum or minimum element in a stream when supplied with a `Comparator`.

Streams of Streams (OCP Objective 5.7)

- Sometimes the type of `Stream` in a pipeline is a `Stream` of `Streams`. For instance: `Stream<Stream<String>>` is a stream of string streams.
- You can use the `Stream` method `flatMap()` to flatten streams, essentially concatenating the values from each stream within the stream. For instance, to flatten a `Stream<Stream<String>> sss` that you created with code like this:

```
Stream<Stream<String>> sss =
    Stream.of(Stream.of("one", "two"),
              Stream.of("three", "four"),
              Stream.of("five", "six"));
```

  you could write:

```
sss.flatMap(s -> s).forEach(System.out::println);
```

Generating Streams (OCP Objective 3.4)

- You can create infinite streams with the `Stream` methods `iterate()` and `generate()`.
- The `Stream` method `iterate()` method takes a seed (e.g., 0) and a `UnaryOperator` to generate a sequential ordered stream of values. The primitive streams have corresponding `iterate()` methods that generate streams of primitive values.
- The `Stream` method `generate()` method takes a `Supplier` that generates values for a `Stream`.
- Be careful when working with infinite streams. To do something useful with them, you need to limit the size of the stream with `limit()` or by using a short-circuiting method like `anyMatch()` or `findAny()`.
- You can generate `IntStreams` and `LongStreams` streams using the `range()` and `rangeClosed()` methods, which produce a sequence of numbers from a start value to an end value.

A Taste of Parallel Streams (OCP Objective 10.6)

- By default, streams are sequential, meaning the operations in the stream pipeline are processed sequentially on the source data.

- You can parallelize a stream using the `parallel()` method. But be careful how you use this method; not all stream pipelines can be parallelized!
- We'll take a more in-depth look at parallel streams in Chapter 11, which covers concurrency.

SELF TEST

The following questions will help you practice using stream operations, and build and measure your understanding of the material in this chapter.

1. Given the code fragment:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = Arrays.stream(nums)
   // L1
```

Which code fragment inserted at line `// L1` will compile correctly and sum the numbers in the array `nums` and store the value in the variable sum?

A.
```
.reduce((n1, n2) -> n1 + n2);
```

B.
```
.reduce(0, (n1, n2) -> n1 + n2);
```

C.
```
.reduce(nums);
```

D.
```
.reduce(0, n1 + n2);
```

2. Given the following code fragment:

```
try (Stream<String> stream = Files.lines(Paths.get("names.txt"))) {
   // L1
   System.out.println("Sorted names in the file: " + names);
} catch(IOException e) {
   System.out.println("Error reading names.txt");
   e.printStackTrace();
}
```

and a file, “names.txt,” with one name per line, which of the following code fragments inserted at line `// L1` will produce a sorted `List` of names in the variable `names`?

A.

```
List<String> names = stream.sorted().toList();
```

B.

```
List<String> names = stream
   .comparing((n1, n2) -> n1.compareTo(n2))
   .collect(Collectors.toList());
```

C.

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

D.

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

3. What are the correct types for variables `s1` and `s2` in the code fragment below?

```
TYPE s1 = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
TYPE s2 = s1.mapToInt(i -> i);
```

A. Stream<Integer> and Stream<Integer>

B. IntStream and IntStream

C. IntStream and Stream<Integer>

D. Stream<Integer> and IntStream

E. Stream<Integer> and List<Integer>

4. Given the code fragments:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return getName() + " is " + getColor() + " and is "
                       + getAge() + " years old.";
   }
   @Override
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}

List<Duck> ducks = Arrays.asList(
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2),
   new Duck("Huey", "mottled", 2),
   new Duck("Louie", "white", 4),
   new Duck("Dewey", "brown", 6)
);
```

Which code fragment could you use for CODE in the call to `map()` in the following code fragment:

```
System.out.print("Duck names: ");
ducks.stream().map(CODE).forEach(d -> System.out.print(d + " "));
System.out.println();
```

to correctly print the string (choose all that apply):

```
Duck names: Jerry George Kramer Elaine Huey Louie Dewey
```

A. `Duck::getName`

B. `d::d.getName()`

C. `d -> d.getName()`

D. `d.getName()`

5. Given the `Duck` class and `ducks List`:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return getName() + " is " + getColor() + " and is "
                       + getAge() + " years old.";
   }
   @Override
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}

List<Duck> ducks = Arrays.asList(
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2),
   new Duck("Huey", "mottled", 2),
   new Duck("Louie", "white", 4),
   new Duck("Dewey", "brown", 6)
);
```

What output does the following code fragment produce?

```
ducks.stream()
      .filter(d -> d.getColor().equals("mottled"))
      .map(d -> d.getName())
      .forEach(d -> System.out.print(d + " "));
```

A. `Kramer Huey`

B. `Jerry George Kramer Elaine Huey Louie Dewey`

C. No output, the code does not compile

D. `Kramer is mottled and is 6 years old. Huey is mottled and is 2 years old.`

6. Given the `Duck` class and `ducks List`:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return getName() + " is " + getColor() + " and is "
                       + getAge() + " years old.";
   }
   @Override
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}

List<Duck> ducks = Arrays.asList(
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2),
   new Duck("Huey", "mottled", 2),
   new Duck("Louie", "white", 4),
   new Duck("Dewey", "brown", 6)
);
```

Which code fragment would you use to compute the average age of the ducks as a whole number?

A.

```
double avgAge =
   ducks.stream().mapToInt(d -> d.getAge()).average();
```

B.

```
OptionalDouble avgAge =
   ducks.stream().mapToInt(d -> d.getAge()).average();
```

C.

```
Double avgAge =
   ducks.stream().mapToDouble(d -> d.getAge()).average();
```

D.

```
double avgAge =
   ducks.stream().map(d -> d.getAge()).average();
```

7. Given the `Duck` class and `ducks List`:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return getName() + " is " + getColor() + " and is "
                       + getAge() + " years old.";
   }
   @Override
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}

List<Duck> ducks = Arrays.asList(
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2),
   new Duck("Huey", "mottled", 2),
   new Duck("Louie", "white", 4),
   new Duck("Dewey", "brown", 6)
);
```

Which code fragment would you use to count how many mottled ducks there are?

A.

```
long count = ducks.stream().filter(d -> d.getColor().equals("mottled")).count();
```

B.

```
int count = ducks.stream().filter(d -> d.getColor().equals("mottled")).count();
```

C.

```
long count = ducks.stream().filter(d -> d.equals("mottled")).count();
```

D.

```
long count = ducks.stream().filter(Duck::getColor().equals("mottled")).count();
```

8. Given the `Duck` class and `ducks List`:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return getName() + " is " + getColor() + " and is "
                       + getAge() + " years old.";
   }
   @Override
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}

List<Duck> ducks = Arrays.asList(
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2),
   new Duck("Huey", "mottled", 2),
   new Duck("Louie", "white", 4),
   new Duck("Dewey", "brown", 6)
);
```

And the following code fragment:

```
ducks.stream()
    .collect(Collectors.groupingBy(d -> d.getColor()))
    .forEach((c, dl) -> {
       System.out.print("Ducks who are " + c + ": ");
       dl.forEach(d -> System.out.print(d.getName() + " "));
       System.out.println();
    });
```

What is the result?

A. Compilation fails

B. An exception is thrown at runtime

C.

```
{Ducks who are color:[Kramer Huey ], Ducks who are color:[Elaine Louie ], Ducks
who are color:[Jerry ], Ducks who are color:[George Dewey ]}
```

D.

```
Ducks who are mottled: Kramer Huey
Ducks who are white: Elaine Louie
Ducks who are yellow: Jerry
Ducks who are brown: George Dewey
```

9. Given the `Duck` class and `ducks List`:

```
class Duck implements Comparable<Duck> {
   String name;
   String color;
   int age;

   public Duck(String name, String color, int age) {
      this.name = name; this.color = color; this.age = age;
   }
   // getters and setters here
   public String toString() {
      return getName() + " is " + getColor() + " and is "
                       + getAge() + " years old.";
   }
   @Override
   public int compareTo(Duck duck) {
      return this.getName().compareTo(duck.getName());
   }
}

List<Duck> ducks = Arrays.asList(
   new Duck("Jerry", "yellow", 3),
   new Duck("George", "brown", 4),
   new Duck("Kramer", "mottled", 6),
   new Duck("Elaine", "white", 2),
   new Duck("Huey", "mottled", 2),
   new Duck("Louie", "white", 4),
   new Duck("Dewey", "brown", 6)
);
```

and the following code fragment:

```
TYPE duckMap =
  ducks.stream()
  .collect(Collectors.groupingBy(d -> d.getColor(), Collectors.toList()));
```

What is the correct type for `duckMap` in `TYPE`?

A. `Map<String, Duck>`

B. `List<Duck>`

C. `Map<String, List<Duck>>`

D. `Map<List<Duck>, String>`

10. Given the code fragment:

```
List<Integer> myInts = Arrays.asList(5, 10, 7, 2, 8);
TYPE minInteger = myInts.stream().mapToInt(i -> i).min();
```

What is the correct `TYPE` for `minInteger`?

A. `int`

B. `Integer`

C. `OptionalInt`

D. `Optional<Integer>`

E. `Stream<Integer>`

F. `IntStream`

11. Given the following code:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
   public Double getTemp() { return this.temp; }
   public String toString() {
      return "July average temp at " + location + " was " + temp;
   }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
Comparator<Temperature>
  tCompare = ((t1, t2) -> t1.getTemp().compareTo(t2.getTemp()));
TYPE max = julyAvgs.stream().max(tCompare);
```

What is the correct TYPE for max?

A. `Optional<Double>`

B. `Optional<Temperature>`

C. `OptionalDouble`

D. `Temperature`

E. `Double`

F. `double`

12. Given the `Temperature` class and `julyAvgs List`:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
   public Double getTemp() { return this.temp; }
   public String toString() {
      return "July average temp at " + location + " was " + temp;
   }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
TYPE maxT = julyAvgs.stream().mapToDouble(t -> t.getTemp()).max();
```

What is the correct `TYPE` for `maxT`?

A. `Optional<Double>`

B. `Optional<Temperature>`

C. `OptionalDouble`

D. `Temperature`

E. `Double`

F. `double`

13. Given the `Temperature` class and `julyAvgs List`:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
   public Double getTemp() { return this.temp; }
   public String toString() {
      return "July average temp at " + location + " was " + temp;
   }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
Comparator<Temperature> tCompare =
   ((t1, t2) -> t1.getTemp().compareTo(t2.getTemp()));
Optional<Temperature> min = julyAvgs.stream().min(tCompare);
min.ifPresent(m -> System.out.println("Min: " + m));
Optional<Temperature> coolerSpot =
   julyAvgs.stream().filter(t -> t.getTemp() < 100.0).findAny();
if (coolerSpot.isPresent()) {
   System.out.println("A cooler spot for July: " + coolerSpot.get());
} else {
   System.out.println("No place cool in July!");
}
```

Will the minimum temperature, `min`, be the same as `coolerSpot`?

A. Yes

B. No

C. Maybe

14. Given the `Temperature` class and `julyAvgs List`:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
```

```
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
Comparator<Temperature> tCompare =
    ((t1, t2) -> t1.getTemp().compareTo(t2.getTemp()));
List<Temperature> sortedTemps =
    julyAvgs.stream()
            // L1
sortedTemps.forEach(System.out::println);
```

Which fragment(s), inserted independently at `// L1`, cause the code to print the `Temperatures` from high temp to low temp?

A.

```
.sorted().reversed().collect(Collectors.toList());
```

B.

```
.sorted(Comparator.comparing(t -> t.getTemp())).collect(Collectors.toList());
```

C.

```
.sorted(tCompare).collect(Collectors.toList());
```

D.

```
.sorted(tCompare.reversed()).collect(Collectors.toList());
```

15. Given the `Temperature` class and `julyAvgs List`:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
   public Double getTemp() { return this.temp; }
```

```
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
julyAvgs.stream()
    .mapToLong(t -> Math.round(t.getTemp()))
    .sorted()
    .distinct()
    .forEach(t -> System.out.print(t + " "));
```

What does this code display?

A. 80.5 80.8 85.3 85.7 106.0 107.4

B. 81 81 85 86 106 107

C. 81 85 86 106 107

D. 107 85 81 81 106 86

16. Given the `Temperature` class and `julyAvgs List`:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
   public Double getTemp() { return this.temp; }
   public String toString() {
      return "July average temp at " + location + " was " + temp;
   }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

What does the following code fragment display?

```
Map<Boolean, List<String>> temp100 =
   julyAvgs.stream().collect(
      Collectors.partitioningBy(t -> t.getTemp() >= 100.0,
                                Collectors.mapping(t -> t.getLocation(),
                                                   Collectors.toList())));
System.out.println(temp100);
```

A.

```
Salt Lake City, UT, Reno, NV, Bishop, CA, Miami, FL, Death Valley, CA, Phoenix, AZ
```

B.

```
{false=[Salt Lake City, UT, Reno, NV, Bishop, CA, Miami, FL], true=[Death Valley,
CA, Phoenix, AZ]}
```

C.

```
106 107
```

D.

```
Death Valley, CA, Phoenix, AZ
```

17. Given the `Temperature` class and `julyAvgs List`:

```
class Temperature {
   String location;
   Double temp;
   public Temperature(String location, Double temp) {
      this.location = location; this.temp = temp;
   }
   public String getLocation() { return this.location; }
   public Double getTemp() { return this.temp; }
   public String toString() {
      return "July average temp at " + location + " was " + temp;
   }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

Which of the following fragments will print `Temperature` objects sorted by location?

A.

```
julyAvgs.stream()
   .map(t -> t.getLocation())
   .sorted()
   .forEach(System.out::println);
```

B.

```
julyAvgs.stream()
   .sorted(Temperature::getLocation)
   .forEach(System.out::println);
```

C.

```
julyAvgs.stream()
   .sorted((t1, t2) -> t1.getLocation().compareTo(t2.getLocation()))
   .forEach(System.out::println);
```

D.

```
julyAvgs.stream()
   .map(t -> t.getLocation())
   .sorted((l1, l2) -> l1.compareTo(l2))
   .forEach(System.out::println);
```

A SELF TEST ANSWERS

1. ☑ **B** is correct. We provide an identity for `reduce()` so we get back an `int`, rather than an optional.

   ☒ **A** is incorrect because `reduce()` needs an identity or it creates an `Optional`. **C** and **D** are incorrect syntax for `reduce()`, and both result in compile errors. (OCP Objective 3.4)

2. ☑ **D** is correct. We call `sorted()` to sort the stream and `collect()` to collect the results in a `List`.

   ☒ **A, B,** and **C** are incorrect. For **A,** `toList()` is a method of `Collectors`, not `Stream`. For **B,** `comparing()` is a method of `Comparator`, not `Stream` [you pass a `Comparator` to `sorted()`]. For **C,** `sorted()` is an intermediate stream operation that should be called before the `collect()` reduction, a terminal operation. (OCP Objectives 3.4, 3.6, 5.5, 5.6, and 9.3)

3. ☑ **D** is correct. The `ints` in the call to `Stream.of()` get autoboxed to `Integers`. (If we'd used `IntStream.of()` instead, then we could use `IntStream` as the type for `s1`.) Given that `s1` is a `Stream<Integer>`, when we call `mapToInt()` on elements of the stream (which maps objects to `ints`), we are converting the `Integer` objects to `ints`, so the type of `s2` must be `IntStream`.

   ☒ **A, B, C,** and **E** are incorrect based on the above information. (OCP Objectives 3.4 and 5.1)

4. ☑ **A** and **C** are correct. **A** is a method reference shorthand for **C.**

   ☒ **B** and **D** are incorrect. **B** has incorrect syntax for a lambda expression and method reference. **D** is not a lambda expression and should be. (OCP Objective 4.4)

5. ☑ **A** is correct. Filter all ducks who have "mottled" color and map ducks to their name.

   ☒ **B, C,** and **D** are incorrect. **B** shows all the ducks instead of just "mottled" ducks. **D** shows the result if you print the full `Duck` [via `toString()`] but you are mapping a duck to its name before printing.

(OCP Objectives 3.4, 3.5, 3.7, and 5.1)

6. ☑ **B** is correct. We stream the ducks and map each duck to its age (an `int`) and then reduce to the average of the ages. Note that we are using `mapToInt()` to map a duck to its age. `average()` returns an optional.

   ☒ **A, C, D,** and **E** are incorrect. **A, C,** and **D** don't compile. `average()` returns an `OptionalDouble`. In **D,** we have an additional problem with `map()` because the lambda maps a duck to its age (a primitive), but `map()` is used to map an object to an object. (OCP Objectives 3.4, 5.1, 5.3, and 5.4)

7. ☑ **A** is correct. We stream the ducks and filter for ducks whose color is "mottled." We call `count()` to reduce to the number of mottled ducks.

   ☒ **B, C,** and **D** are incorrect. **B** doesn't compile [`count()` returns a `long`]. In **C,** `d` in filter is a `Duck`, not a `String`, so it will not equal "mottled" and the `count` will be 0. **D** doesn't compile because of invalid syntax for the expected `Predicate` in `filter()`. (OCP Objectives 3.4, 3.7, and 5.4)

8. ☑ **D** is correct. We stream the ducks and group them by their color, creating a `Map`. We then use `forEach()` to take each map entry (ducks by their color) and display the `Map` key—the color—and the `Map` value, which is a list of `Ducks`. We further iterate through each list of ducks to display just the names.

   ☒ **A, B,** and **C** are incorrect. **A** and **B** are incorrect based on the above information. **C** is incorrect because it shows the `Duck` name printed as a `List` rather than as `Strings` and shows the word "color" rather than the `color` value of each duck (the `Map` key for the `Map` produced by `Collectors.groupingBy()`). (OCP Objectives 5.1 and 5.6)

9. ☑ **C** is correct. `Collectors.groupingBy()` creates a `Map` with `color Strings` as keys and `Lists` of `Ducks` as values.

   ☒ **A, B,** and **D** are incorrect based on the above information. (OCP Objectives 3.4 and 5.6)

10. ☑ **C** is correct. `min()` is a reduction method that produces an optional type. We start with a `Stream<Integer>`; `mapToInt()` creates an `IntStream`, so `min()` produces an `OptionalInt`.

    ☒ **A, B,** and **D** are incorrect based on the above information. (OCP

Objectives 3.4, 5.1, and 5.4)

11. ☑ **B** is correct. `max()` takes a `Comparator`, which takes two `Temperature` objects and compares them using the `temp` field, producing the maximum `Temperature` object in the `List`.

☒ **A, C, D, E,** and **F** are incorrect based on the above information. (OCP Objectives 3.4, 5.3, and 5.4)

12. ☑ **C** is correct. We map each `Temperature` object to its (unboxed) `double temp` and take the `max()` of the stream of `doubles`, producing an `OptionalDouble`.

☒ **A, B, D, E,** and **F** are incorrect. (OCP Objectives 3.4, 5.1, 5.3, and 5.4)

13. ☑ **C** is correct. The minimum is 80.5 in Reno, Nevada. `findAny()` finds any `Temperature` from the filtered stream of `Temperatures` < 100; for a nonparallel stream, it's usually the first one, 85.3 in Salt Lake City, but there is no guarantee that `findAny()` will return the first `Temperature` in the filtered stream.

☒ **A** and **B** are incorrect based on the above information. (OCP Objectives 3.4, 3.7, 5.1, 5.2, 5.3, and 5.4)

14. ☑ **D** is correct. We stream the `Temperatures` using the `tCompare Comparator`, which sorts the `Temperatures` from low to high based on the `temp` value, so we then reverse that ordering with `reversed()`. Then we collect the results in a `List`.

☒ **A, B,** and **C** are incorrect. **A** has two problems: `Temperature` isn't `Comparable`, and `reversed()` is a method on `Comparators`, so this code will not compile. **B** and **C** work but print the temperatures from low to high instead of high to low. (OCP Objectives 3.4, 5.5, and 5.6)

15. ☑ **C** is correct. We stream the `Temperatures` and map each temperature to a long value by rounding the `Temperature`'s `temp` value. We then remove any duplicates by calling `distinct()` (eliminating the repeated 81 value) and then print them out.

☒ **A, B,** and **D** are incorrect. For **A,** notice in the code fragment that we are using `Math.round()` to cut off the decimal points (so we process `longs`, not `doubles`). We are using `distinct()` to eliminate duplicates,

so we shouldn't see 81 twice as we do in **B**. **D** is sorted in the wrong order; natural order is ascending for `long` values. (OCP Objectives 3.4, 3.5, 3.6, 5.1, and 5.5)

16. ☑ **B** is correct. We are creating a partition, so we will get (and display) a `Map` with two keys, `true` and `false`, partitioned by whether the `temp` value of a `Temperature` is >= 100.0. Each `Temperature` is mapped (downstream) to its `location` and the locations for each partition are collected into a `List`. The locations are `Strings`, so we see a `List` of `String` locations for each partition.

☒ **A, C,** and **D** are incorrect based on the above information. (OCP Objectives 3.4 and 5.6)

17. ☑ **C** is correct. We sort the `Temperature` objects by location and print. `sorted()` takes a `Comparator`.

☒ **A, B,** and **D** are incorrect. **A** and **D** display only locations, not the whole `Temperature` object. **B** produces a compile error because `sorted()` requires a `Comparator`, and we've provided a `Function`. (OCP Objectives 3.4, 3.8, 5.1, and 5.5)

EXERCISE ANSWER

Exercise 9-1: Collecting Items in a List

Using `collect()` to collect DVDs into a `List` of `DVDInfo` objects as we read the dvds from a file, we can rewrite our `loadDVDs()` method as shown on the next page.

```
public static List<DVDInfo> loadDVDs(String filename) {
   List<DVDInfo> dvds = new ArrayList<DVDInfo>();
   try (Stream<String> stream = Files.lines(Paths.get(filename))) {
      dvds = stream.map(line -> {
                String[] dvdItems = line.split("/");
                DVDInfo dvd =
                   new DVDInfo(dvdItems[0], dvdItems[1], dvdItems[2]);
                return dvd;
             }).collect(Collectors.toList());
   } catch (IOException e) {
      System.out.println("Error reading DVDs");
      e.printStackTrace();
   }
   return dvds;
}
```

This solution has the benefit of being more clear and avoids the side effect of modifying an object that’s defined outside the stream pipeline.