Chapter 15 Functional Programming

THE OCP EXAM TOPICS COVERED IN THIS CHAPTER INCLUDE THE FOLLOWING:

- **Java Stream API**
  - Describe the Stream interface and pipelines
  - Use lambda expressions and method references
- **Built-in Functional Interfaces**
  - Use interfaces from the java.util.function package
  - Use core functional interfaces including Predicate, Consumer, Function and Supplier
  - Use primitive and binary variations of base interfaces of java.util.function package
- **Lambda Operations on Streams**
  - Extract stream data using map, peek and flatMap methods
  - Search stream data using search findFirst, findAny, anyMatch, allMatch and noneMatch methods
  - Use the Optional class
  - Perform calculations using count, max, min, average and sum stream operations
  - Sort a collection using lambda expressions
  - Use Collectors with streams, including the groupingBy and partitioningBy operations

---

By now, you should be comfortable with the lambda and method reference syntax. Both are used when implementing functional interfaces. If you need more practice, you may want to go back and review Chapter 12, "Java Fundamentals," and Chapter 14, "Generics and Collections." You even used methods like `forEach( )` and `merge()` in Chapter 14. In this chapter, we'll add actual functional programming to that, focusing on the Streams API.

Note that the Streams API in this chapter is used for functional programming. By contrast, there are also `java.io` streams, which we will talk about in Chapter 19, "I/O." Despite both using the word *stream*, they are nothing alike.

In this chapter, we will introduce many more functional interfaces and `Optional` classes. Then we will introduce the `Stream` pipeline and tie it all together. You might have noticed that this chapter covers a long list of objectives. Don't worry if you find the list daunting. By the time you finish the chapter, you'll see that many of the objectives cover similar topics. You might even want to read this chapter twice before doing the review questions so that you really get the hang of it. Functional programming tends to have a steep learning curve but can be really exciting once you get the hang of it.

Working with Built-in Functional Interfaces

In Table 14.1, we introduced some basic functional interfaces that we used with the Collections Framework. Now, we will learn them in more detail and more thoroughly. As discussed in Chapter 12, a functional interface has exactly one abstract method. We will focus on that method here.

All of the functional interfaces in Table 15.1 are provided in the `java.util.function` package. The convention here is to use the generic type `T` for the type parameter. If a second type parameter is needed, the next letter, `U`, is used. If a distinct return type is needed, `R` for *return* is used for the generic type.

TABLE 15.1 Common functional interfaces

Functional interface	**Return type**	**Method name**	**# of parameters**
`Supplier<T>`	`T`	`get()`	0
`Consumer<T>`	`void`	`accept(T)`	1 (`T`)
`BiConsumer<T, U>`	`void`	`accept(T,U)`	2 (`T, U`)
`Predicate<T>`	`boolean`	`test(T)`	1 (`T`)
`BiPredicate<T, U>`	`boolean`	`test(T,U)`	2 (`T, U`)
`Function<T, R>`	`R`	`apply(T)`	1 (`T`)
`BiFunction<T, U, R>`	`R`	`apply(T,U)`	2 (`T, U`)
`UnaryOperator<T>`	`T`	`apply(T)`	1 (`T`)
`BinaryOperator<T>`	`T`	`apply(T,T)`	2 (`T, T`)

There is one functional interface here that was not in Table 14.1 (`BinaryOperator` .) We introduced only what you needed in Chapter 14 at that point. Even Table 15.1 is a subset of what you need to know. Many functional interfaces are defined in the `java.util.function` package. There are even functional interfaces for handling primitives, which you'll see later in the chapter.

While you need to know a lot of functional interfaces for the exam, luckily many share names with the ones in Table 15.1. With that in mind, you need to memorize Table 15.1. We will give you lots of practice in this section to help make this memorable. Before you ask, most of the time we don't actually assign the implementation of the interface to a variable. The interface name is implied, and it gets passed directly to the method that needs it. We are introducing the names so that you can better understand and remember what is going on. Once we get to the streams part of the chapter, we will assume that you have this down and stop creating the intermediate variable.

As you saw in Chapter 12, you can name a functional interface anything you want. The only requirements are that it must be a valid interface name and contain a single abstract method. Table 15.1 is significant because these interfaces are often used in streams and other classes that come with Java, which is why you need to memorize them for the exam.

NOTE

As you'll learn in Chapter 18, "Concurrency," there are two more functional interfaces called `Runnable` and `Callable`, which you need to know for the exam. They are used for concurrency the majority of the time. However, they may show up on the exam when you are asked to recognize which functional interface to use. All you need to know is that `Runnable` and `Callable` don't take any parameters, with `Runnable` returning `void` and `Callable` returning a generic type.

Let's look at how to implement each of these interfaces. Since both lambdas and method references show up all over the exam, we show an implementation using both where possible. After introducing the interfaces, we will also cover some convenience methods available on these interfaces.

Implementing *Supplier*

A `Supplier` is used when you want to generate or supply values without taking any input. The `Supplier` interface is defined as follows:

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

You can create a `LocalDate` object using the factory method `now()`. This example shows how to use a `Supplier` to call this factory:

```
Supplier<LocalDate> s1 = LocalDate::now;
Supplier<LocalDate> s2 = () -> LocalDate.now();

LocalDate d1 = s1.get();
LocalDate d2 = s2.get();

System.out.println(d1);
System.out.println(d2);
```

This example prints a date such as `2020-02-20` twice. It's also a good opportunity to review `static` method references. The `LocalDate::now` method reference is used to create a `Supplier` to assign to an intermediate variable `s1`. A `Supplier` is often used when constructing new objects. For example, we can print two empty `StringBuilder` objects.

```
Supplier<StringBuilder> s1 = StringBuilder::new;
Supplier<StringBuilder> s2 = () -> new StringBuilder();

System.out.println(s1.get());
System.out.println(s2.get());
```

This time, we used a constructor reference to create the object. We've been using generics to declare what type of `Supplier` we are using. This can get a little long to read. Can you figure out what the following does? Just take it one step at a time.

```
Supplier<ArrayList<String>> s3 = ArrayList<String>::new;
ArrayList<String> a1 = s3.get();
System.out.println(a1);
```

We have a `Supplier` of a certain type. That type happens to be `ArrayList<String>`. Then calling `get()` creates a new instance of `ArrayList<String>`, which is the generic type of the `Supplier`—in other words, a generic that contains another generic. It's not hard to understand, so just look at the code carefully when this type of thing comes up.

Notice how we called `get()` on the functional interface. What would happen if we tried to print out `s3` itself?

```
System.out.println(s3);
```

The code prints something like this:

```
functionalinterface.BuiltIns$$Lambda$1/0x0000000800066840@4909b8da
```

That's the result of calling `toString()` on a lambda. Yuck. This actually does mean something. Our test class is named `BuiltIns`, and it is in a package that we created named `functionalinterface`. Then comes `$$`, which means that the class doesn't exist in a class file on the file system. It exists only in memory. You don't need to worry about the rest.

Implementing Consumer and BiConsumer

You use a `Consumer` when you want to do something with a parameter but not return anything. `BiConsumer` does the same thing except that it takes two parameters. The interfaces are defined as follows:

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
   // omitted default method
}

@FunctionalInterface
public interface BiConsumer<T, U> {
   void accept(T t, U u);
   // omitted default method
}
```

NOTE

You'll notice this pattern. *Bi* means two. It comes from Latin, but you can remember it from English words like *binary* (0 or 1) or *bicycle* (two wheels). Always add another parameter when you see *Bi* show up.

You used a `Consumer` in Chapter 14 with `forEach()`. Here's that example actually being assigned to the `Consumer` interface:

```
Consumer<String> c1 = System.out::println;
Consumer<String> c2 = x -> System.out.println(x);

c1.accept("Annie");
c2.accept("Annie");
```

This example prints `Annie` twice. `BiConsumer` is called with two parameters. They don't have to be the same type. For example, we can put a key and a value in a map using this interface:

```
var map = new HashMap<String, Integer>();
BiConsumer<String, Integer> b1 = map::put;
BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);

b1.accept("chicken", 7);
b2.accept("chick", 1);
```

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

The output is `{chicken=7, chick=1}`, which shows that both `BiConsumer` implementations did get called. When declaring `b1`, we used an instance method reference on an object since we want to call a method on the local variable `map`. The code to instantiate `b1` is a good bit shorter than the code for `b2`. This is probably why the exam is so fond of method references.

As another example, we use the same type for both generic parameters.

```
var map = new HashMap<String, String>();
BiConsumer<String, String> b1 = map::put;
BiConsumer<String, String> b2 = (k, v) -> map.put(k, v);

b1.accept("chicken", "Cluck");
b2.accept("chick", "Tweep");

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

The output is `{chicken=Cluck, chick=Tweep}`, which shows that a `BiConsumer` can use the same type for both the `T` and `U` generic parameters.

Implementing *Predicate* and *BiPredicate*

You saw `Predicate` with `removeIf()` in Chapter 14. `Predicate` is often used when filtering or matching. Both are common operations. A `BiPredicate` is just like a `Predicate` except that it takes two parameters instead of one. The interfaces are defined as follows:

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
   // omitted default and static methods
}

@FunctionalInterface
public interface BiPredicate<T, U> {
   boolean test(T t, U u);
   // omitted default methods
}
```

It should be old news by now that you can use a `Predicate` to test a condition.

```
Predicate<String> p1 = String::isEmpty;
Predicate<String> p2 = x -> x.isEmpty();

System.out.println(p1.test(""));  // true
System.out.println(p2.test(""));  // true
```

This prints `true` twice. More interesting is a `BiPredicate`. This example also prints `true` twice:

```
BiPredicate<String, String> b1 = String::startsWith;
BiPredicate<String, String> b2 =
   (string, prefix) -> string.startsWith(prefix);

System.out.println(b1.test("chicken", "chick"));  // true
System.out.println(b2.test("chicken", "chick"));  // true
```

The method reference includes both the instance variable and parameter for `startsWith()`. This is a good example of how method references save a good bit of typing. The downside is that they are less explicit, and you really have to understand what is going on!

Implementing *Function* and *BiFunction*

In Chapter 14, we used `Function` with the `merge()` method. A `Function` is responsible for turning one parameter into a value of a potentially different type and returning it. Similarly, a `BiFunction` is responsible for turning two parameters into a value and returning it. The interfaces are defined as follows:

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);
   // omitted default and static methods
}

@FunctionalInterface
public interface BiFunction<T, U, R> {
   R apply(T t, U u);
   // omitted default method
}
```

For example, this function converts a `String` to the length of the `String`:

```
Function<String, Integer> f1 = String::length;
Function<String, Integer> f2 = x -> x.length();

System.out.println(f1.apply("cluck")); // 5
System.out.println(f2.apply("cluck")); // 5
```

This function turns a `String` into an `Integer`. Well, technically it turns the `String` into an `int`, which is autoboxed into an `Integer`. The types don't have to be different. The following combines two `String` objects and produces another `String`:

```
BiFunction<String, String, String> b1 = String::concat;
BiFunction<String, String, String> b2 =
   (string, toAdd) -> string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick")); // baby chick
System.out.println(b2.apply("baby ", "chick")); // baby chick
```

The first two types in the `BiFunction` are the input types. The third is the result type. For the method reference, the first parameter is the instance that `concat()` is called on, and the second is passed to `concat()`.

CREATING YOUR OWN FUNCTIONAL INTERFACES

Java provides a built-in interface for functions with one or two parameters. What if you need more? No problem. Suppose that you want to create a functional interface for the wheel speed of each wheel on a tricycle. You could create a functional interface such as this:

```
@FunctionalInterface
interface TriFunction<T,U,V,R> {
   R apply(T t, U u, V v);
}
```

There are four type parameters. The first three supply the types of the three wheel speeds. The fourth is the return type. Now suppose that you want to create a function to determine how fast your quad-copter is going given the power of the four motors. You could create a functional interface such as the following:

```
@FunctionalInterface
interface QuadFunction<T,U,V,W,R> {
   R apply(T t, U u, V v, W w);
}
```

There are five type parameters here. The first four supply the types of the four motors. Ideally these would be the same type, but you never know. The fifth is the return type in this example.

Java's built-in interfaces are meant to facilitate the most common functional interfaces that you'll need. It is by no means an exhaustive list. Remember that you can add any functional interfaces you'd like, and Java matches them when you use lambdas or method references.

Implementing *UnaryOperator* and *BinaryOperator*

`UnaryOperator` and `BinaryOperator` are a special case of a `Function`. They require all type parameters to be the same type. A `UnaryOperator` transforms its value into one of the same type. For example, incrementing by one is a unary operation. In fact, `UnaryOperator` extends `Function`. A `BinaryOperator` merges two values into one of the same type. Adding two numbers is a binary operation. Similarly, `BinaryOperator` extends `BiFunction`. The interfaces are defined as follows:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> { }

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
   // omitted static methods
}
```

This means that method signatures look like this:

```
T apply(T t);         // UnaryOperator

T apply(T t1, T t2);  // BinaryOperator
```

In the Javadoc, you'll notice that these methods are actually inherited from the `Function` / `BiFunction` superclass. The generic declarations on the subclass are what force the type to be the same. For the unary example, notice how the return type is the same type as the parameter.

```
UnaryOperator<String> u1 = String::toUpperCase;
UnaryOperator<String> u2 = x -> x.toUpperCase();

System.out.println(u1.apply("chirp"));  // CHIRP
System.out.println(u2.apply("chirp"));  // CHIRP
```

This prints `CHIRP` twice. We don't need to specify the return type in the generics because `UnaryOperator` requires it to be the same as the parameter. And now here's the binary example:

```
BinaryOperator<String> b1 = String::concat;
BinaryOperator<String> b2 = (string, toAdd) -> string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick")); // baby chick
System.out.println(b2.apply("baby ", "chick")); // baby chick
```

Notice that this does the same thing as the `BiFunction` example. The code is more succinct, which shows the importance of using the correct functional interface. It's nice to have one generic type specified instead of three.

Checking Functional Interfaces

It's really important to know the number of parameters, types, return value, and method name for each of the functional interfaces. Now would be a good time to memorize Table 15.1 if you haven't done so already. Let's do some examples to practice.

What functional interface would you use in these three situations?

- Returns a `String` without taking any parameters
- Returns a `Boolean` and takes a `String`
- Returns an `Integer` and takes two `Integer` s

Ready? Think about what your answer is before continuing. Really. You have to know this cold. OK. The first one is a `Supplier<String>` because it generates an object and takes zero parameters. The second one is a `Function<String,Boolean>` because it takes one parameter and returns another type. It's a little tricky. You might think it is a `Predicate<String>`. Note that a `Predicate` returns a `boolean` primitive and not a `Boolean` object. Finally, the third one is either a `BinaryOperator<Integer>` or a `BiFunction<Integer,Integer,Integer>`. Since `BinaryOperator` is a special case of `BiFunction`, either is a correct answer. `BinaryOperator<Integer>` is the better answer of the two since it is more specific.

Let's try this exercise again but with code. It's harder with code. With code, the first thing you do is look at how many parameters the lambda takes and whether there is a return value. What functional interface would you use to fill in the blank for these?

```
6: _______<List> ex1 = x -> "".equals(x.get(0));
7: _______<Long> ex2 = (Long l) -> System.out.println(l);
8: _______<String, String> ex3 = (s1, s2) -> false;
```

Again, think about the answers before continuing. Ready? Line 6 passes one `List` parameter to the lambda and returns a `boolean`. This tells us that it is a `Predicate` or `Function`. Since the generic declaration has only one parameter, it is a `Predicate`.

Line 7 passes one `Long` parameter to the lambda and doesn't return anything. This tells us that it is a `Consumer`. Line 8 takes two parameters and returns a `boolean`. When you see a `boolean` returned, think `Predicate` unless the generics specify a `Boolean` return type. In this case, there are two parameters, so it is a `BiPredicate`.

Are you finding these easy? If not, review Table 15.1 again. We aren't kidding. You need to know the table really well. Now that you are fresh from studying the table, we are going to play “identify the error.” These are meant to be tricky:

```
6: Function<List<String>> ex1 = x -> x.get(0); // DOES NOT COMPILE
7: UnaryOperator<Long> ex2 = (Long l) -> 3.14; // DOES NOT COMIPLE
8: Predicate ex4 = String::isEmpty;            // DOES NOT COMPILE
```

Line 6 claims to be a `Function`. A `Function` needs to specify two generics—the input parameter type and the return value type. The return value type is missing from line 6, causing the code not to compile. Line 7 is a `UnaryOperator`, which returns the same type as it is passed in. The example returns a `double` rather than a `Long`, causing the code not to compile.

Line 8 is missing the generic for `Predicate`. This makes the parameter that was passed an `Object` rather than a `String`. The lambda expects a `String` because it calls a method that exists on `String` rather than `Object`. Therefore, it doesn't compile.

Convenience Methods on Functional Interfaces

By definition, all functional interfaces have a single abstract method. This doesn't mean they can have only one method, though. Several of the common functional interfaces provide a number of helpful `default` methods.

Table 15.2 shows the convenience methods on the built-in functional interfaces that you need to know for the exam. All of these facilitate modifying or combining functional interfaces of the same type. Note that Table 15.2 shows only the main interfaces. The `BiConsumer`, `BiFunction`, and `BiPredicate` interfaces have similar methods available.

Let's start with these two `Predicate` variables.

```
Predicate<String> egg = s -> s.contains("egg");
Predicate<String> brown = s -> s.contains("brown");
```

TABLE 15.2 Convenience methods

Interface instance	Method return type	Method name	Method parameters
`Consumer`	`Consumer`	`andThen()`	`Consumer`
`Function`	`Function`	`andThen()`	`Function`
`Function`	`Function`	`compose()`	`Function`
`Predicate`	`Predicate`	`and()`	`Predicate`
`Predicate`	`Predicate`	`negate()`	—
`Predicate`	`Predicate`	`or()`	`Predicate`

Now we want a `Predicate` for brown eggs and another for all other colors of eggs. We could write this by hand, as shown here:

```
Predicate<String> brownEggs =
   s -> s.contains("egg") && s.contains("brown");
Predicate<String> otherEggs =
   s -> s.contains("egg") && ! s.contains("brown");
```

This works, but it's not great. It's a bit long to read, and it contains duplication. What if we decide the letter *e* should be capitalized in *egg*? We'd have to change it in three variables: `egg`, `brownEggs`, and `otherEggs`. A better way to deal with this situation is to use two of the `default` methods on `Predicate`.

```
Predicate<String> brownEggs = egg.and(brown);
Predicate<String> otherEggs = egg.and(brown.negate());
```

Neat! Now we are reusing the logic in the original `Predicate` variables to build two new ones. It's shorter and clearer what the relationship is between variables. We can also change the spelling of *egg* in one place, and the other two objects will have new logic because they reference it.

Moving on to `Consumer`, let's take a look at the `andThen()` method, which runs two functional interfaces in sequence.

```
Consumer<String> c1 = x -> System.out.print("1: " + x);
Consumer<String> c2 = x -> System.out.print(",2: " + x);

Consumer<String> combined = c1.andThen(c2);
combined.accept("Annie");              // 1: Annie,2: Annie
```

Notice how the same parameter gets passed to both `c1` and `c2`. This shows that the `Consumer` instances are run in sequence and are independent of each other. By contrast, the `compose()` method on `Function` chains functional interfaces. However, it passes along the output of one to the input of another.

```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;

Function<Integer, Integer> combined = after.compose(before);
System.out.println(combined.apply(3));   // 8
```

This time the `before` runs first, turning the `3` into a `4`. Then the `after` runs, doubling the `4` to `8`. All of the methods in this section are helpful in simplifying your code as you work with functional interfaces.

Returning an *Optional*

Suppose that you are taking an introductory Java class and receive scores of 90 and 100 on the first two exams. Now, we ask you what your average is. An average is calculated by adding the scores and dividing by the number of scores, so you have (90+100)/2. This gives 190/2, so you answer with 95. Great!

Now suppose that you are taking your second class on Java, and it is the first day of class. We ask you what your average is in this class that just started. You haven't taken any exams yet, so you don't have anything to average. It wouldn't be accurate to say that your average is zero. That sounds bad, and it isn't true. There simply isn't any data, so you don't have an average yet.

How do we express this “we don't know” or “not applicable” answer in Java? We use the `Optional` type. An `Optional` is created using a factory. You can either request an empty `Optional` or pass a value for the `Optional` to wrap. Think of an `Optional` as a box that might have something in it or might instead be empty. Figure 15.1 shows both options.

FIGURE 15.1 `Optional`

Creating an *Optional*

Here's how to code our average method:

```
10: public static Optional<Double> average(int… scores) {
11:    if (scores.length == 0) return Optional.empty();
12:    int sum = 0;
13:    for (int score: scores) sum += score;
14:    return Optional.of((double) sum / scores.length);
15: }
```

Line 11 returns an empty `Optional` when we can't calculate an average. Lines 12 and 13 add up the scores. There is a functional programming way of doing this math, but we will get to that later in the chapter. In fact, the entire method could be written in one line, but that wouldn't teach you how `Optional` works! Line 14 creates an `Optional` to wrap the average.

Calling the method shows what is in our two boxes.

```
System.out.println(average(90, 100)); // Optional[95.0]
System.out.println(average());        // Optional.empty
```

You can see that one `Optional` contains a value and the other is empty. Normally, we want to check whether a value is there and/or get it out of the box. Here's one way to do that:

```
20: Optional<Double> opt = average(90, 100);
21: if (opt.isPresent())
22:    System.out.println(opt.get()); // 95.0
```

Line 21 checks whether the `Optional` actually contains a value. Line 22 prints it out. What if we didn't do the check and the `Optional` was empty?

```
Optional<Double> opt = average();
System.out.println(opt.get()); // NoSuchElementException
```

We'd get an exception since there is no value inside the `Optional`.

```
java.util.NoSuchElementException: No value present
```

When creating an `Optional`, it is common to want to use `empty()` when the value is `null`. You can do this with an `if` statement or ternary operator. We use the ternary operator (`? :`) to simplify the code, which you saw Chapter 3, “Operators”.

```
Optional o = (value == null) ? Optional.empty() : Optional.of(value);
```

If `value` is `null`, `o` is assigned the empty `Optional`. Otherwise, we wrap the value. Since this is such a common pattern, Java provides a factory method to do the same thing.

```
Optional o = Optional.ofNullable(value);
```

That covers the `static` methods you need to know about `Optional`. Table 15.3 summarizes most of the instance methods on `Optional` that you need to know for the exam. There are a few others that involve chaining. We will cover those later in the chapter.

TABLE 15.3 `Optional` instance methods

Method	When `Optional` is empty	When `Optional` contains a value
`get()`	Throws an exception	Returns value
`ifPresent(Consumer c)`	Does nothing	Calls `Consumer` with value
`isPresent()`	Returns `false`	Returns `true`
`orElse(T other)`	Returns `other` parameter	Returns value
`orElseGet(Supplier s)`	Returns result of calling `Supplier`	Returns value
`orElseThrow()`	Throws `NoSuchElementException`	Returns value
`orElseThrow(Supplier s)`	Throws exception created by calling `Supplier`	Returns value

You've already seen `get()` and `isPresent()`. The other methods allow you to write code that uses an `Optional` in one line without having to use the ternary operator. This makes the code easier to read. Instead of using an `if` statement, which we used when checking the average earlier, we can specify a `Consumer` to be run when there is a value inside the `Optional`. When there isn't, the method simply skips running the `Consumer`.

```
Optional<Double> opt = average(90, 100);
opt.ifPresent(System.out::println);
```

Using `ifPresent()` better expresses our intent. We want something done if a value is present. You can think of it as an `if` statement with no `else`.

Dealing with an Empty *Optional*

The remaining methods allow you to specify what to do if a value isn't present. There are a few choices. The first two allow you to specify a return value either directly or using a `Supplier`.

```
30: Optional<Double> opt = average();
31: System.out.println(opt.orElse(Double.NaN));
32: System.out.println(opt.orElseGet(() -> Math.random()));
```

This prints something like the following:

```
NaN
0.49775932295380165
```

Line 31 shows that you can return a specific value or variable. In our case, we print the “not a number” value. Line 32 shows using a `Supplier` to generate a value at runtime to return instead. I'm glad our professors didn't give us a random average, though!

Alternatively, we can have the code throw an exception if the `Optional` is empty.

```
30: Optional<Double> opt = average();
31: System.out.println(opt.orElseThrow());
```

This prints something like the following:

```
Exception in thread "main" java.util.NoSuchElementException:
   No value present
   at java.base/java.util.Optional.orElseThrow(Optional.java:382)
```

Without specifying a `Supplier` for the exception, Java will throw a `NoSuchElementException`. This method was added in Java 10. Remember that the stack trace looks weird because the lambdas are generated rather than named classes. Alternatively, we can have the code throw a custom exception if the `Optional` is empty.

```
30: Optional<Double> opt = average();
31: System.out.println(opt.orElseThrow(
32:    () -> new IllegalStateException()));
```

This prints something like the following:

```
Exception in thread "main" java.lang.IllegalStateException
   at optionals.Methods.lambda$orElse$1(Methods.java:30)
   at java.base/java.util.Optional.orElseThrow(Optional.java:408)
```

Line 32 shows using a `Supplier` to create an exception that should be thrown. Notice that we do not write `throw new IllegalStateException()`. The `orElseThrow()` method takes care of actually throwing the exception when we run it.

The two methods that take a `Supplier` have different names. Do you see why this code does not compile?

```
System.out.println(opt.orElseGet(
   () -> new IllegalStateException())); // DOES NOT COMPILE
```

The `opt` variable is an `Optional<Double>`. This means the `Supplier` must return a `Double`. Since this supplier returns an exception, the type does not match.

The last example with `Optional` is really easy. What do you think this does?

```
Optional<Double> opt = average(90, 100);
System.out.println(opt.orElse(Double.NaN));
System.out.println(opt.orElseGet(() -> Math.random()));
System.out.println(opt.orElseThrow());
```

It prints out `95.0` three times. Since the value does exist, there is no need to use the "or else" logic.

IS *OPTIONAL* THE SAME AS *NULL*?

Before Java 8, programmers would return `null` instead of `Optional`. There were a few shortcomings with this approach. One was that there wasn't a clear way to express that `null` might be a special value. By contrast, returning an `Optional` is a clear statement in the API that there might not be a value in there.

Another advantage of `Optional` is that you can use a functional programming style with `ifPresent()` and the other methods rather than needing an `if` statement. Finally, you'll see toward the end of the chapter that you can chain `Optional` calls.

Using Streams

A *stream* in Java is a sequence of data. A *stream pipeline* consists of the operations that run on a stream to produce a result. First we will look at the flow of pipelines conceptually. After that, we will actually get into code.

Understanding the Pipeline Flow

Think of a stream pipeline as an assembly line in a factory. Suppose that we were running an assembly line to make signs for the animal exhibits at the zoo. We have a number of jobs. It is one person's job to take signs out of a box. It is a second person's job to paint the sign. It is a third person's job to stencil the name of the animal on the sign. It's the last person's job to put the completed sign in a box to be carried to the proper exhibit.

Notice that the second person can't do anything until one sign has been taken out of the box by the first person. Similarly, the third person can't do anything until one sign has been painted, and the last person can't do anything until it is stenciled.

The assembly line for making signs is finite. Once we process the contents of our box of signs, we are finished. *Finite* streams have a limit. Other assembly lines essentially run forever, like one for food production. Of course, they do stop at some point when the factory closes down, but pretend that doesn't happen. Or think of a sunrise/sunset cycle as *infinite*, since it doesn't end for an inordinately large period of time.

Another important feature of an assembly line is that each person touches each element to do their operation and then that piece of data is gone. It doesn't come back. The next person deals with it at that point. This is different than the lists and queues that you saw in the previous chapter. With a list, you can access any element at any time. With a queue, you are limited in which elements you can access, but all of the elements are there. With streams, the data isn't generated up front—it is created when needed. This is an example of *lazy evaluation*, which delays execution until necessary.

Many things can happen in the assembly line stations along the way. In functional programming, these are called *stream operations*. Just like with the assembly line, operations occur in a pipeline. Someone has to start and end the work, and there can be any number of stations in between. After all, a job with one person isn't an assembly line! There are three parts to a stream pipeline, as shown in Figure 15.2.

- **Source:** Where the stream comes from
- **Intermediate operations:** Transforms the stream into another one. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.
- **Terminal operation:** Actually produces a result. Since streams can be used only once, the stream is no longer valid after a terminal operation completes.

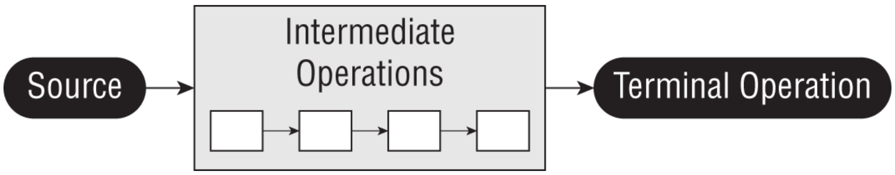

FIGURE 15.2 Stream pipeline

Notice that the operations are unknown to us. When viewing the assembly line from the outside, you care only about what comes in and goes out. What happens in between is an implementation detail.

You will need to know the differences between intermediate and terminal operations well. Make sure you can fill in Table 15.4.

TABLE 15.4 Intermediate vs. terminal operations

Scenario	Intermediate operation	Terminal operation
Required part of a useful pipeline?	No	Yes
Can exist multiple times in a pipeline?	Yes	No
Return type is a stream type?	Yes	No
Executed upon method call?	No	Yes
Stream valid after call?	Yes	No

A factory typically has a foreman who oversees the work. Java serves as the foreman when working with stream pipelines. This is a really important role, especially when dealing with lazy evaluation and infinite streams. Think of declaring the stream as giving instructions to the foreman. As the foreman finds out what needs to be done, he sets up the stations and tells the workers what their duties will be. However, the workers do not start until the foreman tells them to begin. The foreman waits until he sees the terminal operation to actually kick off the work. He also watches the work and stops the line as soon as work is complete.

Let's look at a few examples of this. We aren't using code in these examples because it is really important to understand the stream pipeline concept before starting to write the code. Figure 15.3 shows a stream pipeline with one intermediate operation.

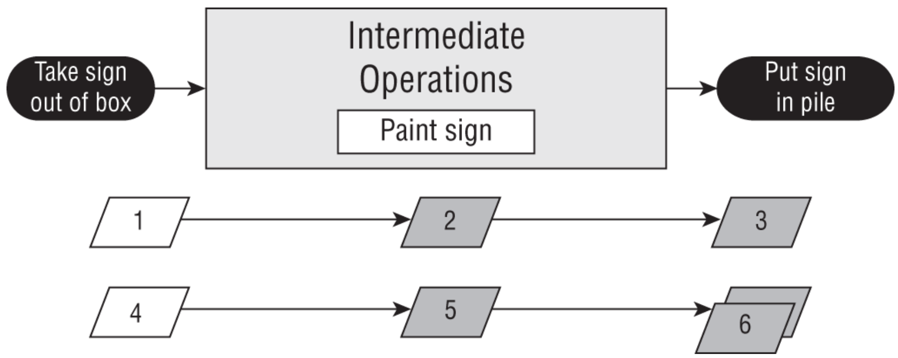

FIGURE 15.3 Steps in running a stream pipeline

Let's take a look at what happens from the point of the view of the foreman. First, he sees that the source is taking signs out of the box. The foreman sets up a worker at the table to unpack the box and says to await a signal to start. Then the foreman sees the intermediate operation to paint the sign. He sets up a worker with paint and says to await a signal to start. Finally, the foreman sees the terminal operation to put the signs into a pile. He sets up a worker to do this and yells out that all three workers should start.

Suppose that there are two signs in the box. Step 1 is the first worker taking one sign out of the box and handing it to the second worker. Step 2 is the second worker painting it and handing it to the third worker. Step 3 is the third worker putting it in the pile. Steps 4–6 are this same process for the other sign. Then the foreman sees that there are no more signs left and shuts down the entire enterprise.

The foreman is smart. He can make decisions about how to best do the work based on what is needed. As an example, let's explore the stream pipeline in Figure 15.4.

The foreman still sees a source of taking signs out of the box and assigns a worker to do that on command. He still sees an intermediate operation to paint and sets up another worker with instructions to wait and then paint. Then he sees an intermediate step that we need only two signs. He sets up a worker to count the signs that go by and notify him when the worker has seen two. Finally, he sets up a worker for the terminal operation to put the signs in a pile.

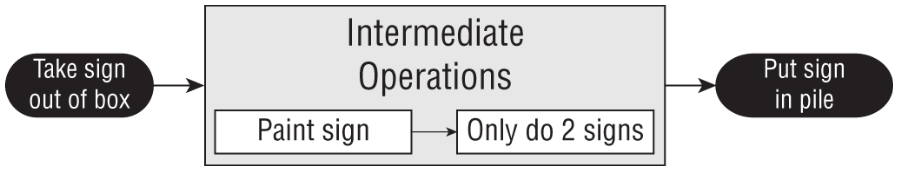

FIGURE 15.4 A stream pipeline with a limit

This time, suppose that there are 10 signs in the box. We start out like last time. The first sign makes its way down the pipeline. The second sign also makes its way down the pipeline. When the worker in charge of counting sees the second sign, she tells the foreman. The foreman lets the terminal operation worker finish her task and then yells out "stop the line." It doesn't matter that there are eight more signs in the box. We don't need them, so it would be unnecessary work to paint them. And we all want to avoid unnecessary work!

Similarly, the foreman would have stopped the line after the first sign if the terminal operation was to find the first sign that gets created.

In the following sections, we will cover the three parts of the pipeline. We will also discuss special types of streams for primitives and how to print a stream.

Creating Stream Sources

In Java, the streams we have been talking about are represented by the `Stream<T>` interface, defined in the `java.util.stream` package.

Creating Finite Streams

For simplicity, we'll start with finite streams. There are a few ways to create them.

```
11: Stream<String> empty = Stream.empty();          // count = 0
12: Stream<Integer> singleElement = Stream.of(1);   // count = 1
13: Stream<Integer> fromArray = Stream.of(1, 2, 3); // count = 3
```

Line 11 shows how to create an empty stream. Line 12 shows how to create a stream with a single element. Line 13 shows how to create a stream from a varargs. You've undoubtedly noticed that there isn't an array on line 13. The method signature uses varargs, which let you specify an array or individual elements.

Java also provides a convenient way of converting a `Collection` to a stream.

```
14: var list = List.of("a", "b", "c");
15: Stream<String> fromList = list.stream();
```

Line 15 shows that it is a simple method call to create a stream from a list. This is helpful since such conversions are common.

CREATING A PARALLEL STREAM

It is just as easy to create a parallel stream from a list.

```
var list = List.of("a", "b", "c");
Stream<String> fromListParallel = list.parallelStream();
```

This is a great feature because you can write code that uses concurrency before even learning what a thread is. Using parallel streams is like setting up multiple tables of workers who are able to do the same task. Painting would be a lot faster if we could have five painters painting signs instead of just one. Just keep in mind some tasks cannot be done in parallel, such as putting the signs away in the order that they were created in the stream. Also be aware that there is a cost in coordinating the work, so for smaller streams, it might be faster to do it sequentially. You'll learn much more about running tasks concurrently in Chapter 18.

Creating Infinite Streams

So far, this isn't particularly impressive. We could do all this with lists. We can't create an infinite list, though, which makes streams more powerful.

```
17: Stream<Double> randoms = Stream.generate(Math::random);
18: Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

Line 17 generates a stream of random numbers. How many random numbers? However many you need. If you call `randoms.forEach(System.out::println)`, the program will print random numbers until you kill it. Later in the chapter, you'll learn about operations like `limit()` to turn the infinite stream into a finite stream.

Line 18 gives you more control. The `iterate()` method takes a seed or starting value as the first parameter. This is the first element that will be part of the stream. The other parameter is a lambda expression that gets passed the previous value and generates the next value. As with the random numbers example, it will keep on producing odd numbers as long as you need them.

NOTE

If you try to call `System.out.print(stream)`, you'll get something like the following:

```
java.util.stream.ReferencePipeline$3@4517d9a3
```

This is different from a `Collection` where you see the contents. You don't need to know this for the exam. We mention it so that you aren't caught by surprise when writing code for practice.

What if you wanted just odd numbers less than 100? Java 9 introduced an overloaded version of `iterate()` that helps with just that.

```
19: Stream<Integer> oddNumberUnder100 = Stream.iterate(
20:    1,                 // seed
21:    n -> n < 100,      // Predicate to specify when done
22:    n -> n + 2);       // UnaryOperator to get next value
```

This method takes three parameters. Notice how they are separated by commas (`,`) just like all other methods. The exam may try to trick you by using semicolons since it is similar to a `for` loop. Similar to a `for` loop, you have to take care that you aren't accidentally creating an infinite stream.

Reviewing Stream Creation Methods

To review, make sure you know all the methods in Table 15.5. These are the ways of creating a source for streams, given a `Collection` instance `coll`.

TABLE 15.5 Creating a source

Method	Finite or infinite?	Notes
`Stream.empty()`	Finite	Creates `Stream` with zero elements
`Stream.of(varargs)`	Finite	Creates `Stream` with elements listed
`coll.stream()`	Finite	Creates `Stream` from a `Collection`
`coll.parallelStream()`	Finite	Creates `Stream` from a `Collection` where the stream can run in parallel
`Stream.generate(supplier)`	Infinite	Creates `Stream` by calling the `Supplier` for each element upon request
`Stream.iterate(seed, unaryOperator)`	Infinite	Creates `Stream` by using the seed for the first element and then calling the `UnaryOperator` for each subsequent element upon request
`Stream.iterate(seed, predicate, unaryOperator)`	Finite or infinite	Creates `Stream` by using the seed for the first element and then calling the `UnaryOperator` for each subsequent element upon request. Stops if the `Predicate` returns false

Using Common Terminal Operations

You can perform a terminal operation without any intermediate operations but not the other way around. This is why we will talk about terminal operations first. *Reductions* are a special type of terminal operation where all of the contents of the stream are combined into a single primitive or `Object`. For example, you might have an `int` or a `Collection`.

Table 15.6 summarizes this section. Feel free to use it as a guide to remember the most important points as we go through each one individually. We explain them from simplest to most complex rather than alphabetically.

TABLE 15.6 Terminal stream operations

Method	What happens for infinite streams	Return value	Reduction
count()	Does not terminate	long	Yes
min() max()	Does not terminate	Optional<T>	Yes
findAny() findFirst()	Terminates	Optional<T>	No
allMatch() anyMatch() noneMatch()	Sometimes terminates	boolean	No
forEach()	Does not terminate	void	No
reduce()	Does not terminate	Varies	Yes
collect()	Does not terminate	Varies	Yes

***count()*s**

The `count()` method determines the number of elements in a finite stream. For an infinite stream, it never terminates. Why? Count from 1 to infinity and let us know when you are finished. Or rather, don't do that because we'd rather you study for the exam than spend the rest of your life counting. The `count()` method is a reduction because it looks at each element in the stream and returns a single value. The method signature is as follows:

```
long count()
```

This example shows calling `count()` on a finite stream:

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
System.out.println(s.count());   // 3
```

min()* and *max()

The `min()` and `max()` methods allow you to pass a custom comparator and find the smallest or largest value in a finite stream according to that sort order. Like the `count()` method, `min()` and `max()` hang on an infinite stream because they cannot be sure that a smaller or larger value isn't coming later in the stream. Both methods are reductions because they return a single value after looking at the entire stream. The method signatures are as follows:

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

This example finds the animal with the fewest letters in its name:

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
Optional<String> min = s.min((s1, s2) -> s1.length()-s2.length());
min.ifPresent(System.out::println); // ape
```

Notice that the code returns an `Optional` rather than the value. This allows the method to specify that no minimum or maximum was found. We use the `Optional` method `ifPresent()` and a method reference to print out the minimum only if one is found. As an example of where there isn't a minimum, let's look at an empty stream.

```
Optional<?> minEmpty = Stream.empty().min((s1, s2) -> 0);
System.out.println(minEmpty.isPresent()); // false
```

Since the stream is empty, the comparator is never called, and no value is present in the `Optional`.

NOTE

What if you need both the `min()` and `max()` values of the same stream? For now, you can't have both, at least not using these methods. Remember, a stream can have only one terminal operation. Once a terminal operation has been run, the stream cannot be used again. As we'll see later in this chapter, there are built-in summary methods for some numeric streams that will calculate a set of values for you.

---

findAny() and *findFirst()*

The `findAny()` and `findFirst()` methods return an element of the stream unless the stream is empty. If the stream is empty, they return an empty `Optional`. This is the first method you've seen that can terminate with an infinite stream. Since Java generates only the amount of stream you need, the infinite stream needs to generate only one element.

As its name implies, the `findAny()` method can return any element of the stream. When called on the streams you've seen up until now, it commonly returns the first element, although this behavior is not guaranteed. As you'll see in Chapter 18, the `findAny()` method is more likely to return a random element when working with parallel streams.

These methods are terminal operations but not reductions. The reason is that they sometimes return without processing all of the elements. This means that they return a value based on the stream but do not reduce the entire stream into one value.

The method signatures are as follows:

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

This example finds an animal:

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
Stream<String> infinite = Stream.generate(() -> "chimp");

s.findAny().ifPresent(System.out::println);        // monkey (usually)
infinite.findAny().ifPresent(System.out::println); // chimp
```

Finding any one match is more useful than it sounds. Sometimes we just want to sample the results and get a representative element, but we don't need to waste the processing generating them all. After all, if we plan to work with only one element, why bother looking at more?

allMatch(), *anyMatch()*, and *noneMatch()*

The `allMatch()` , `anyMatch()` , and `noneMatch()` methods search a stream and return information about how the stream pertains to the predicate. These may or may not terminate for infinite streams. It depends on the data. Like the find methods, they are not reductions because they do not necessarily look at all of the elements.

The method signatures are as follows:

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

This example checks whether animal names begin with letters:

```
var list = List.of("monkey", "2", "chimp");
Stream<String> infinite = Stream.generate(() -> "chimp");
Predicate<String> pred = x -> Character.isLetter(x.charAt(0));

System.out.println(list.stream().anyMatch(pred));  // true
System.out.println(list.stream().allMatch(pred));  // false
System.out.println(list.stream().noneMatch(pred)); // false
System.out.println(infinite.anyMatch(pred));       // true
```

This shows that we can reuse the same predicate, but we need a different stream each time. The `anyMatch()` method returns `true` because two of the three elements match. The `allMatch()` method returns `false` because one doesn't match. The `noneMatch()` method also returns `false` because one matches. On the infinite stream, one match is found, so the call terminates. If we called `allMatch()` , it would run until we killed the program.

NOTE

Remember that `allMatch()` , `anyMatch()` , and `noneMatch()` return a `boolean` . By contrast, the find methods return an `Optional` because they return an element of the stream.

---

forEach()

Like in the Java Collections Framework, it is common to iterate over the elements of a stream. As expected, calling `forEach()` on an infinite stream does not terminate. Since there is no return value, it is not a reduction.

Before you use it, consider if another approach would be better. Developers who learned to write loops first tend to use them for everything. For example, a loop with an `if` statement could be written with a filter. You will learn about filters in the intermediate operations section.

The method signature is as follows:

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

Notice that this is the only terminal operation with a return type of `void`. If you want something to happen, you have to make it happen in the `Consumer`. Here's one way to print the elements in the stream (there are other ways, which we cover later in the chapter):

```
Stream<String> s = Stream.of("Monkey", "Gorilla", "Bonobo");
s.forEach(System.out::print); // MonkeyGorillaBonobo
```

NOTE

Remember that you can call `forEach()` directly on a `Collection` or on a `Stream`. Don't get confused on the exam when you see both approaches.

Notice that you can't use a traditional `for` loop on a stream.

```
Stream<Integer> s = Stream.of(1);
for (Integer i  : s) {} // DOES NOT COMPILE
```

While `forEach()` sounds like a loop, it is really a terminal operator for streams. Streams cannot be used as the source in a `for-each` loop to run because they don't implement the `Iterable` interface.

reduce()

The `reduce()` method combines a stream into a single object. It is a reduction, which means it processes all elements. The three method signatures are these:

```
T reduce(T identity, BinaryOperator<T> accumulator)

Optional<T> reduce(BinaryOperator<T> accumulator)

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

Let's take them one at a time. The most common way of doing a reduction is to start with an initial value and keep merging it with the next value. Think about how you would concatenate an array of `String` objects into a single `String` without functional programming. It might look something like this:

```
var array = new String[] { "w", "o", "l", "f" };
var result = "";
for (var s: array) result = result + s;
System.out.println(result); // wolf
```

The *identity* is the initial value of the reduction, in this case an empty `String`. The *accumulator* combines the current result with the current value in the stream. With lambdas, we can do the same thing with a stream and reduction:

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
String word = stream.reduce("", (s, c) -> s + c);
System.out.println(word); // wolf
```

Notice how we still have the empty `String` as the identity. We also still concatenate the `String` objects to get the next value. We can even rewrite this with a method reference.

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
String word = stream.reduce("", String::concat);
System.out.println(word); // wolf
```

Let's try another one. Can you write a reduction to multiply all of the `Integer` objects in a stream? Try it. Our solution is shown here:

```
Stream<Integer> stream = Stream.of(3, 5, 6);
System.out.println(stream.reduce(1, (a, b) -> a*b));  // 90
```

We set the identity to `1` and the accumulator to multiplication. In many cases, the identity isn't really necessary, so Java lets us omit it. When you don't specify an identity, an `Optional` is returned because there might not be any data. There are three choices for what is in the `Optional`.

- If the stream is empty, an empty `Optional` is returned.
- If the stream has one element, it is returned.
- If the stream has multiple elements, the accumulator is applied to combine them.

The following illustrates each of these scenarios:

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> empty = Stream.empty();
Stream<Integer> oneElement = Stream.of(3);
Stream<Integer> threeElements = Stream.of(3, 5, 6);

empty.reduce(op).ifPresent(System.out::println);         // no output
oneElement.reduce(op).ifPresent(System.out::println);    // 3
threeElements.reduce(op).ifPresent(System.out::println); // 90
```

Why are there two similar methods? Why not just always require the identity? Java could have done that. However, sometimes it is nice to differentiate the case where the stream is empty rather than the case where there is a value that happens to match the identity being returned from calculation. The signature returning an `Optional` lets us differentiate these cases. For example, we might return `Optional.empty()` when the stream is empty and `Optional.of(3)` when there is a value.

The third method signature is used when we are dealing with different types. It allows Java to create intermediate reductions and then combine them at the end. Let's take a look at an example that counts the number of characters in each `String`:

```
Stream<String> stream = Stream.of("w", "o", "l", "f!");
int length = stream.reduce(0, (i, s) -> i+s.length(), (a, b) -> a+b);
System.out.println(length); // 5
```

The first parameter (`0`) is the value for the initializer. If we had an empty stream, this would be the answer. The second parameter is the *accumulator*. Unlike the accumulators you saw previously, this one handles mixed data types. In this example, the first argument, `i`, is an `Integer`, while the second argument, `s`, is a `String`. It adds the length of the current `String` to our running total. The third parameter is called the *combiner*, which combines any intermediate totals. In this case, `a` and `b` are both `Integer` values.

The three-argument `reduce()` operation is useful when working with parallel streams because it allows the stream to be decomposed and reassembled by separate threads. For example, if we needed to count the length of four 100-character strings, the first two values and the last two values could be computed independently. The intermediate result (200 + 200) would then be combined into the final value.

collect()

The `collect()` method is a special type of reduction called a *mutable reduction*. It is more efficient than a regular reduction because we use the same mutable object while accumulating. Common mutable objects include `StringBuilder` and `ArrayList`. This is a really useful method, because it lets us get data out of streams and into another form. The method signatures are as follows:

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

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

Let's start with the first signature, which is used when we want to code specifically how collecting should work. Our wolf example from `reduce` can be converted to use `collect()`.

```
Stream<String> stream = Stream.of("w", "o", "l", "f");

StringBuilder word = stream.collect(
   StringBuilder::new,
   StringBuilder::append,
   StringBuilder::append)

System.out.println(word); // wolf
```

The first parameter is the *supplier,* which creates the object that will store the results as we collect data. Remember that a `Supplier` doesn't take any parameters and returns a value. In this case, it constructs a new `StringBuilder`.

The second parameter is the *accumulator*, which is a `BiConsumer` that takes two parameters and doesn't return anything. It is responsible for adding one more element to the data collection. In this example, it appends the next `String` to the `StringBuilder`.

The final parameter is the *combiner*, which is another `BiConsumer`. It is responsible for taking two data collections and merging them. This is useful when we are processing in parallel. Two smaller collections are formed and then merged into one. This would work with `StringBuilder` only if we didn't care about the order of the letters. In this case, the accumulator and combiner have similar logic.

Now let's look at an example where the logic is different in the accumulator and combiner.

```
Stream<String> stream = Stream.of("w", "o", "l", "f");

TreeSet<String> set = stream.collect(
   TreeSet::new,
   TreeSet::add,
   TreeSet::addAll);

System.out.println(set); // [f, l, o, w]
```

The collector has three parts as before. The supplier creates an empty `TreeSet`. The accumulator adds a single `String` from the `Stream` to the `TreeSet`. The combiner adds all of the elements of one `TreeSet` to another in case the operations were done in parallel and need to be merged.

We started with the long signature because that's how you implement your own collector. It is important to know how to do this for the exam and to understand how collectors work. In practice, there are many common collectors that come up over and over. Rather than making developers keep reimplementing the same ones, Java provides a class with common collectors cleverly named `Collectors`. This approach also makes the code easier to read because it is more expressive. For example, we could rewrite the previous example as follows:

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> set =
   stream.collect(Collectors.toCollection(TreeSet::new));
System.out.println(set); // [f, l, o, w]
```

If we didn't need the set to be sorted, we could make the code even shorter:

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.collect(Collectors.toSet());
System.out.println(set); // [f, w, l, o]
```

You might get different output for this last one since `toSet()` makes no guarantees as to which implementation of `Set` you'll get. It is likely to be a `HashSet`, but you shouldn't expect or rely on that.

NOTE

The exam expects you to know about common predefined collectors in addition to being able to write your own by passing a supplier, accumulator, and combiner.

Later in this chapter, we will show many `Collectors` that are used for grouping data. It's a big topic, so it's best to master how streams work before adding too many `Collectors` into the mix.

Using Common Intermediate Operations

Unlike a terminal operation, an intermediate operation produces a stream as its result. An intermediate operation can also deal with an infinite stream simply by returning another infinite stream. Since elements are produced only as needed, this works fine. The assembly line worker doesn't need to worry about how many more elements are coming through and instead can focus on the current element.

filter()

The `filter()` method returns a `Stream` with elements that match a given expression. Here is the method signature:

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

This operation is easy to remember and powerful because we can pass any `Predicate` to it. For example, this filters all elements that begin with the letter *m*:

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.filter(x -> x.startsWith("m"))
   .forEach(System.out::print); // monkey
```

distinct()

The `distinct()` method returns a stream with duplicate values removed. The duplicates do not need to be adjacent to be removed. As you might imagine, Java calls `equals()` to determine whether the objects are the same. The method signature is as follows:

```
Stream<T> distinct()
```

Here's an example:

```
Stream<String> s = Stream.of("duck", "duck", "duck", "goose");
s.distinct()
   .forEach(System.out::print); // duckgoose
```

limit() and skip()

The `limit()` and `skip()` methods can make a `Stream` smaller, or they could make a finite stream out of an infinite stream. The method signatures are shown here:

```
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

The following code creates an infinite stream of numbers counting from 1. The `skip()` operation returns an infinite stream starting with the numbers counting from 6, since it skips the first five elements. The `limit()` call takes the first two of those. Now we have a finite stream with two elements, which we can then print with the `forEach()` method.

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
s.skip(5)
   .limit(2)
   .forEach(System.out::print); // 67
```

map()

The `map()` method creates a one-to-one mapping from the elements in the stream to the elements of the next step in the stream. The method signature is as follows:

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

This one looks more complicated than the others you have seen. It uses the lambda expression to figure out the type passed to that function and the one returned. The return type is the stream that gets returned.

NOTE

The `map()` method on streams is for transforming data. Don't confuse it with the `Map` interface, which maps keys to values.

As an example, this code converts a list of `String` objects to a list of `Integer` objects representing their lengths.

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.map(String::length)
   .forEach(System.out::print); // 676
```

Remember that `String::length` is shorthand for the lambda `x -> x.length()`, which clearly shows it is a function that turns a `String` into an `Integer`.

flatMap()

The `flatMap()` method takes each element in the stream and makes any elements it contains top-level elements in a single stream. This is helpful when you want to remove empty elements from a stream or you want to combine a stream of lists. We are showing you the method signature for consistency with the other methods, just so you don't think we are hiding anything. You aren't expected to be able to read this:

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

This gibberish basically says that it returns a `Stream` of the type that the function contains at a lower level. Don't worry about the signature. It's a headache.

What you should understand is the example. This gets all of the animals into the same level along with getting rid of the empty list.

```
List<String> zero = List.of();
var one = List.of("Bonobo");
var two = List.of("Mama Gorilla", "Baby Gorilla");
```

```
Stream<List<String>> animals = Stream.of(zero, one, two);

animals.flatMap(m -> m.stream())
   .forEach(System.out::println);
```

Here's the output:

```
Bonobo
Mama Gorilla
Baby Gorilla
```

As you can see, it removed the empty list completely and changed all elements of each list to be at the top level of the stream.

sorted()

The `sorted()` method returns a stream with the elements sorted. Just like sorting arrays, Java uses natural ordering unless we specify a comparator. The method signatures are these:

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

Calling the first signature uses the default sort order.

```
Stream<String> s = Stream.of("brown-", "bear-");
s.sorted()
   .forEach(System.out::print); // bear-brown-
```

We can optionally use a `Comparator` implementation via a method or a lambda. In this example, we are using a method:

```
Stream<String> s = Stream.of("brown bear-", "grizzly-");
s.sorted(Comparator.reverseOrder())
   .forEach(System.out::print); // grizzly-brown bear-
```

Here we passed a `Comparator` to specify that we want to sort in the reverse of natural sort order. Ready for a tricky one? Do you see why this doesn't compile?

```
s.sorted(Comparator::reverseOrder); // DOES NOT COMPILE
```

Take a look at the method signatures again. `Comparator` is a functional interface. This means that we can use method references or lambdas to implement it. The `Comparator` interface implements one method that takes two `String` parameters and returns an `int` . However, `Comparator::reverseOrder` doesn't do that. It is a reference to a function that takes zero parameters and returns a `Comparator` . This is not compatible with the interface. This means that we have to use a method and not a method reference. We bring this up to remind you that you really do need to know method references well.

peek()

The `peek()` method is our final intermediate operation. It is useful for debugging because it allows us to perform a stream operation without actually changing the stream. The method signature is as follows:

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

You might notice the intermediate `peek()` operation takes the same argument as the terminal `forEach()` operation Think of `peek()` as an intermediate version of `forEach()` that returns the original stream back to you.

The most common use for `peek()` is to output the contents of the stream as it goes by. Suppose that we made a typo and counted bears beginning with the letter *g* instead of *b*. We are puzzled why the count is 1 instead of 2. We can add a `peek()` method to find out why.

```
var stream = Stream.of("black bear", "brown bear", "grizzly");
long count = stream.filter(s -> s.startsWith("g"))
   .peek(System.out::println).count();             // grizzly
System.out.println(count);                         // 1
```

In Chapter 14, you saw that `peek()` looks only at the first element when working with a `Queue` . In a stream, `peek()` looks at each element that goes through that part of the stream pipeline. It's like having a worker take notes on how a particular step of the process is doing.

DANGER: CHANGING STATE WITH *PEEK()*

Remember that `peek()` is intended to perform an operation without changing the result. Here's a straightforward stream pipeline that doesn't use `peek()`:

```
var numbers = new ArrayList<>();
var letters = new ArrayList<>();
numbers.add(1);
letters.add('a');

Stream<List<?>> stream = Stream.of(numbers, letters);
stream.map(List::size).forEach(System.out::print); // 11
```

Now we add a `peek()` call and note that Java doesn't prevent us from writing bad peek code.

```
Stream<List<?>> bad = Stream.of(numbers, letters);
bad.peek(x -> x.remove(0))
   .map(List::size)
   .forEach(System.out::print); // 00
```

This example is bad because `peek()` is modifying the data structure that is used in the stream, which causes the result of the stream pipeline to be different than if the peek wasn't present.

Putting Together the Pipeline

Streams allow you to use chaining and express what you want to accomplish rather than how to do so. Let's say that we wanted to get the first two names of our friends alphabetically that are four characters long. Without streams, we'd have to write something like the following:

```
var list = List.of("Toby", "Anna", "Leroy", "Alex");
List<String> filtered = new ArrayList<>();
for (String name: list)
   if (name.length() == 4) filtered.add(name);

Collections.sort(filtered);
var iter = filtered.iterator();
if (iter.hasNext()) System.out.println(iter.next());
if (iter.hasNext()) System.out.println(iter.next());
```

This works. It takes some reading and thinking to figure out what is going on. The problem we are trying to solve gets lost in the implementation. It is also very focused on the how rather than on the what. With streams, the equivalent code is as follows:

```
var list = List.of("Toby", "Anna", "Leroy", "Alex");
list.stream().filter(n -> n.length() == 4).sorted()
   .limit(2).forEach(System.out::println);
```

Before you say that it is harder to read, we can format it.

```
var list = List.of("Toby", "Anna", "Leroy", "Alex");
list.stream()
   .filter(n -> n.length() == 4)
   .sorted()
   .limit(2)
   .forEach(System.out::println);
```

The difference is that we express what is going on. We care about `String` objects of length 4. Then we want them sorted. Then we want the first two. Then we want to print them out. It maps better to the problem that we are trying to solve, and it is simpler.

Once you start using streams in your code, you may find yourself using them in many places. Having shorter, briefer, and clearer code is definitely a good thing!

In this example, you see all three parts of the pipeline. Figure 15.5 shows how each intermediate operation in the pipeline feeds into the next.

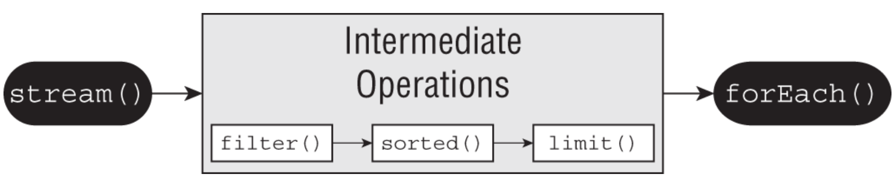

FIGURE 15.5 Stream pipeline with multiple intermediate operations

Remember that the assembly line foreman is figuring out how to best implement the stream pipeline. He sets up all of the tables with instructions to wait before starting. He tells the `limit()` worker to inform him when two elements go by. He tells the `sorted()` worker that she should just collect all of the elements as they come in and sort them all at once. After sorting, she should start passing them to the `limit()` worker one at a time. The data flow looks like this:

1. The `stream()` method sends Toby to `filter()`. The `filter()` method sees that the length is good and sends Toby to `sorted()`. The `sorted()` method can't sort yet because it needs all of the data, so it holds Toby.
2. The `stream()` method sends Anna to `filter()`. The `filter()` method sees that the length is good and sends Anna to `sorted()`. The `sorted()` method can't sort yet because it needs all of the data, so it holds Anna.
3. The `stream()` method sends Leroy to `filter()`. The `filter()` method sees that the length is not a match, and it takes Leroy out of the assembly line processing.
4. The `stream()` method sends Alex to `filter()`. The `filter()` method sees that the length is good and sends Alex to `sorted()`. The `sorted()` method can't sort yet because it needs all of the data, so it holds Alex. It turns out `sorted()` does have all of the required data, but it doesn't know it yet.
5. The foreman lets `sorted()` know that it is time to sort and the sort occurs.
6. The `sorted()` method sends Alex to `limit()`. The `limit()` method remembers that it has seen one element and sends Alex to `forEach()`, printing `Alex`.
7. The `sorted()` method sends Anna to `limit()`. The `limit()` method remembers that it has seen two elements and sends Anna to `forEach()`, printing `Anna`.
8. The `limit()` method has now seen all of the elements that are needed and tells the foreman. The foreman stops the line, and no more processing occurs in the pipeline.

Make sense? Let's try a few more examples to make sure that you understand this well. What do you think the following does?

```
Stream.generate(() -> "Elsa")
   .filter(n -> n.length() == 4)
   .sorted()
   .limit(2)
   .forEach(System.out::println);
```

It actually hangs until you kill the program or it throws an exception after running out of memory. The foreman has instructed `sorted()` to wait until everything to sort is present. That never happens because there is an infinite stream. What about this example?

```
Stream.generate(() -> "Elsa")
   .filter(n -> n.length() == 4)
```

```
      .limit(2)
      .sorted()
      .forEach(System.out::println);
```

This one prints `Elsa` twice. The filter lets elements through, and `limit()` stops the earlier operations after two elements. Now `sorted()` can sort because we have a finite list. Finally, what do you think this does?

```
Stream.generate(() -> "Olaf Lazisson")
   .filter(n -> n.length() == 4)
   .limit(2)
   .sorted()
   .forEach(System.out::println);
```

This one hangs as well until we kill the program. The filter doesn't allow anything through, so `limit()` never sees two elements. This means we have to keep waiting and hope that they show up.

You can even chain two pipelines together. See if you can identify the two sources and two terminal operations in this code.

```
30: long count =  Stream.of("goldfish", "finch")
31:    .filter(s -> s.length()> 5)
32:    .collect(Collectors.toList())
33:    .stream()
34:    .count();
35: System.out.println(count);   // 1
```

Lines 30–32 are one pipeline, and lines 33 and 34 are another. For the first pipeline, line 30 is the source, and line 32 is the terminal operation. For the second pipeline, line 33 is the source, and line 34 is the terminal operation. Now that's a complicated way of outputting the number 1!

NOTE

On the exam, you might see long or complex pipelines as answer choices. If this happens, focus on the differences between the answers. Those will be your clues to the correct answer. This approach will also save you time from not having to study the whole pipeline on each option.

When you see chained pipelines, note where the source and terminal operations are. This will help you keep track of what is going on. You can even rewrite the code in your head to have a variable in between so it isn't as long and complicated. Our prior example can be written as follows:

```
List<String> helper =  Stream.of("goldfish", "finch")
   .filter(s -> s.length()> 5)
   .collect(Collectors.toList());
long count = helper.stream()
   .count();
System.out.println(count);
```

Which style you use is up to you. However, you need to be able to read both styles before you take the exam.

PEEKING BEHIND THE SCENES

The `peek()` method is useful for seeing how a stream pipeline works behind the scenes. Remember that the methods run against each element one at a time until processing is done. Suppose that we have this code:

```
var infinite = Stream.iterate(1, x -> x + 1);
infinite.limit(5)
   .filter(x -> x % 2 == 1)
   .forEach(System.out::print); // 135
```

The source is an infinite stream of numbers. Only the first five elements are allowed through before the foreman instructs work to stop. The `filter()` operation is limited to seeing whether these five numbers from 1 to 5 are odd. Only three are, and those are the ones that get printed, giving `135`.

Now what do you think this prints?

```
var infinite = Stream.iterate(1, x -> x + 1);
infinite.limit(5)
   .peek(System.out::print)
   .filter(x -> x % 2 == 1)
   .forEach(System.out::print);
```

The correct answer is `11233455`. As the first element passes through, 1 shows up in the `peek()` and `print()`. The second element makes it past `limit()` and `peek()`, but it gets caught in `filter()`. The third and fifth elements behave like the first element. The fourth behaves like the second.

Reversing the order of the intermediate operations changes the result.

```
var infinite = Stream.iterate(1, x -> x + 1);
infinite.filter(x -> x % 2 == 1)
   .limit(5)
   .forEach(System.out::print); // 13579
```

The source is still an infinite stream of numbers. The first element still flows through the entire pipeline, and `limit()` remembers that it allows one element through. The second element doesn't make it past `filter()`. The third element flows through the entire pipeline, and `limit()` allows its second element. This proceeds until the ninth element flows through and `limit()` has allowed its fifth element through.

Finally, what do you think this prints?

```
var infinite = Stream.iterate(1, x -> x + 1);
infinite.filter(x -> x % 2 == 1)
   .peek(System.out::print)
   .limit(5)
   .forEach(System.out::print);
```

The answer is `1133557799`. Since `filter()` is before `peek()`, we see only the odd numbers.

Working with Primitive Streams

Up until now, all of the streams we've created used the `Stream` class with a generic type, like `Stream<String>`, `Stream<Integer>`, etc. For numeric values, we have been using the wrapper classes you learned about in Chapter 14. We did this with the `Collections` API so it would feel natural.

Java actually includes other stream classes besides `Stream` that you can use to work with select primitives: `int`, `double`, and `long`. Let's take a look at why this is needed. Suppose that we want to calculate the sum of numbers in a finite stream.

```
Stream<Integer> stream = Stream.of(1, 2, 3);
System.out.println(stream.reduce(0, (s, n) -> s + n));  // 6
```

Not bad. It wasn't hard to write a reduction. We started the accumulator with zero. We then added each number to that running total as it came up in the stream. There is another way of doing that, shown here:

```
Stream<Integer> stream = Stream.of(1, 2, 3);
System.out.println(stream.mapToInt(x -> x).sum());  // 6
```

This time, we converted our `Stream<Integer>` to an `IntStream` and asked the `IntStream` to calculate the sum for us. An `IntStream` has many of the same intermediate and terminal methods as a `Stream` but includes specialized methods for working with numeric data. The primitive streams know how to perform certain common operations automatically.

So far, this seems like a nice convenience but not terribly important. Now think about how you would compute an average. You need to divide the sum by the number of elements. The problem is that streams allow only one pass. Java recognizes that calculating an average is a common thing to do, and it provides a method to calculate the average on the stream classes for primitives.

```
IntStream intStream = IntStream.of(1, 2, 3);
OptionalDouble avg = intStream.average();
System.out.println(avg.getAsDouble());  // 2.0
```

Not only is it possible to calculate the average, but it is also easy to do so. Clearly primitive streams are important. We will look at creating and using such streams, including optionals and functional interfaces.

Creating Primitive Streams

Here are three types of primitive streams.

- **`IntStream`:** Used for the primitive types `int`, `short`, `byte`, and `char`
- **`LongStream`:** Used for the primitive type `long`
- **`DoubleStream`:** Used for the primitive types `double` and `float`

Why doesn't each primitive type have its own primitive stream? These three are the most common, so the API designers went with them.

NOTE

When you see the word *stream* on the exam, pay attention to the case. With a capital *S* or in code, `Stream` is the name of a class that contains an `Object` type. With a lowercase *s*, a stream is a concept that might be a `Stream`, `DoubleStream`, `IntStream`, or `LongStream`.

Table 15.7 shows some of the methods that are unique to primitive streams. Notice that we don't include methods in the table like `empty()` that you already know from the `Stream` interface.

TABLE 15.7 Common primitive stream methods

Method	Primitive stream	Description
`OptionalDouble average()`	`IntStream` `LongStream` `DoubleStream`	The arithmetic mean of the elements
`Stream<T> boxed()`	`IntStream` `LongStream` `DoubleStream`	A `Stream<T>` where `T` is the wrapper class associated with the primitive value
`OptionalInt max()`	`IntStream`	The maximum element of the stream
`OptionalLong max()`	`LongStream`	
`OptionalDouble max()`	`DoubleStream`	
`OptionalInt min()`	`IntStream`	The minimum element of the stream
`OptionalLong min()`	`LongStream`	
`OptionalDouble min()`	`DoubleStream`	
`IntStream range(int a, int b)`	`IntStream`	Returns a primitive stream from `a` (inclusive) to `b` (exclusive)
`LongStream range(long a, long b)`	`LongStream`	
`IntStream rangeClosed(int a, int b)`	`IntStream`	Returns a primitive stream from `a` (inclusive) to `b` (inclusive)
`LongStream rangeClosed(long a, long b)`	`LongStream`	
`int sum()`	`IntStream`	Returns the sum of the elements in the stream
`long sum()`	`LongStream`	

Method	Primitive stream	Description
`double sum()`	`DoubleStream`	
`IntSummaryStatistics summaryStatistics()`	`IntStream`	Returns an object containing numerous stream statistics such as the average, min, max, etc.
`LongSummaryStatistics summaryStatistics()`	`LongStream`	
`DoubleSummaryStatistics summaryStatistics()`	`DoubleStream`	

Some of the methods for creating a primitive stream are equivalent to how we created the source for a regular `Stream`. You can create an empty stream with this:

```
DoubleStream empty = DoubleStream.empty();
```

Another way is to use the `of()` factory method from a single value or by using the varargs overload.

```
DoubleStream oneValue = DoubleStream.of(3.14);
oneValue.forEach(System.out::println);

DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);
varargs.forEach(System.out::println);
```

This code outputs the following:

```
3.14
1.0
1.1
1.2
```

You can also use the two methods for creating infinite streams, just like we did with `Stream`.

```
var random = DoubleStream.generate(Math::random);
var fractions = DoubleStream.iterate(.5, d -> d / 2);
random.limit(3).forEach(System.out::println);
fractions.limit(3).forEach(System.out::println);
```

Since the streams are infinite, we added a limit intermediate operation so that the output doesn't print values forever. The first stream calls a `static` method on `Math` to get a random `double`. Since the numbers are random, your output will obviously be different. The second stream keeps creating smaller numbers, dividing the previous value by two each time. The output from when we ran this code was as follows:

```
0.07890654781186413
0.28564363465842346
0.6311403511266134
0.5
0.25
0.125
```

You don't need to know this for the exam, but the `Random` class provides a method to get primitives streams of random numbers directly. Fun fact! For example, `ints()` generates an infinite `IntStream` of primitives.

It works the same way for each type of primitive stream. When dealing with `int` or `long` primitives, it is common to count. Suppose that we wanted a stream with the numbers from 1 through 5. We could write this using what we've explained so far:

```
IntStream count = IntStream.iterate(1, n -> n+1).limit(5);
count.forEach(System.out::println);
```

This code does print out the numbers 1–5, one per line. However, it is a lot of code to do something so simple. Java provides a method that can generate a range of numbers.

```
IntStream range = IntStream.range(1, 6);
range.forEach(System.out::println);
```

This is better. If we wanted numbers 1–5, why did we pass 1–6? The first parameter to the `range()` method is *inclusive*, which means it includes the number. The second parameter to the `range()` method is *exclusive*, which means it stops right before that number. However, it still could be clearer. We want the numbers 1–5 inclusive. Luckily, there's another method, `rangeClosed()`, which is inclusive on both parameters.

```
IntStream rangeClosed = IntStream.rangeClosed(1, 5);
rangeClosed.forEach(System.out::println);
```

Even better. This time we expressed that we want a closed range, or an inclusive range. This method better matches how we express a range of numbers in plain English.

Mapping Streams

Another way to create a primitive stream is by mapping from another stream type. Table 15.8 shows that there is a method for mapping between any stream types.

TABLE 15.8 Mapping methods between types of streams

Source stream class	To create `Stream`	To create `DoubleStream`	To create `IntStream`	To create `LongStream`
`Stream<T>`	`map()`	`mapToDouble()`	`mapToInt()`	`mapToLong()`
`DoubleStream`	`mapToObj()`	`map()`	`mapToInt()`	`mapToLong()`
`IntStream`	`mapToObj()`	`mapToDouble()`	`map()`	`mapToLong()`
`LongStream`	`mapToObj()`	`mapToDouble()`	`mapToInt()`	`map()`

Obviously, they have to be compatible types for this to work. Java requires a mapping function to be provided as a parameter, for example:

```
Stream<String> objStream = Stream.of("penguin", "fish");
IntStream intStream = objStream.mapToInt(s -> s.length());
```

This function takes an `Object`, which is a `String` in this case. The function returns an `int`. The function mappings are intuitive here. They take the source type and return the target type. In this example, the actual function type is `ToIntFunction`. Table 15.9 shows the mapping function names. As you can see, they do what you might expect.

TABLE 15.9 Function parameters when mapping between types of streams

Source stream class	To create Stream	To create DoubleStream	To create IntStream	To create LongStrea
Stream<T>	Function<T,R>	ToDoubleFunction<T>	ToIntFunction<T>	ToLongFu
DoubleStream	Double Function<R>	DoubleUnary Operator	DoubleToInt Function	DoubleToL Function
IntStream	IntFunction<R>	IntToDouble Function	IntUnary Operator	IntToLong Function
LongStream	Long Function<R>	LongToDouble Function	LongToInt Function	LongUnary Operator

You do have to memorize Table 15.8 and Table 15.9. It's not as hard as it might seem. There are patterns in the names if you remember a few rules. For Table 15.8, mapping to the same type you started with is just called `map()`. When returning an object stream, the method is `mapToObj()`. Beyond that, it's the name of the primitive type in the map method name.

For Table 15.9, you can start by thinking about the source and target types. When the target type is an object, you drop the `To` from the name. When the mapping is to the same type you started with, you use a unary operator instead of a function for the primitive streams.

USING *FLATMAP()*

The `flatMap()` method exists on primitive streams as well. It works the same way as on a regular `Stream` except the method name is different. Here's an example:

```
var integerList = new ArrayList<Integer>();
IntStream ints = integerList.stream()
.flatMapToInt(x -> IntStream.of(x));
DoubleStream doubles = integerList.stream()
.flatMapToDouble(x -> DoubleStream.of(x));
LongStream longs = integerList.stream()
.flatMapToLong(x -> LongStream.of(x));
```

Additionally, you can create a `Stream` from a primitive stream. These methods show two ways of accomplishing this:

```
private static Stream<Integer> mapping(IntStream stream) {
   return stream.mapToObj(x -> x);
}

private static Stream<Integer> boxing(IntStream stream) {
  return stream.boxed();
}
```

The first one uses the `mapToObj()` method we saw earlier. The second one is more succinct. It does not require a mapping function because all it does is autobox each primitive to the corresponding wrapper object. The `boxed()` method exists on all three types of primitive streams.

Using *Optional* with Primitive Streams

Earlier in the chapter, we wrote a method to calculate the average of an `int[]` and promised a better way later. Now that you know about primitive streams, you can calculate the average in one line.

```
var stream = IntStream.rangeClosed(1,10);
OptionalDouble optional = stream.average();
```

The return type is not the `Optional` you have become accustomed to using. It is a new type called `OptionalDouble`. Why do we have a separate type, you might wonder? Why not just use `Optional<Double>`? The difference is that `OptionalDouble` is for a primitive and `Optional<Double>` is for the `Double` wrapper class. Working with the primitive optional class looks similar to working with the `Optional` class itself.

```
optional.ifPresent(System.out::println);                  // 5.5
System.out.println(optional.getAsDouble());               // 5.5
System.out.println(optional.orElseGet(() -> Double.NaN)); // 5.5
```

The only noticeable difference is that we called `getAsDouble()` rather than `get()`. This makes it clear that we are working with a primitive. Also, `orElseGet()` takes a `DoubleSupplier` instead of a `Supplier`.

As with the primitive streams, there are three type-specific classes for primitives. Table 15.10 shows the minor differences among the three. You probably won't be surprised that you have to memorize it as well. This is really easy to remember since the primitive name is the only change. As you should remember from the terminal operations section, a number of stream methods return an optional such as `min()` or `findAny()`. These each return the corresponding optional type. The primitive stream implementations also add two new methods that you need to know. The `sum()` method does not return an optional. If you try to add up an empty stream, you simply get zero. The `average()` method always returns an `OptionalDouble` since an average can potentially have fractional data for any type.

TABLE 15.10 Optional types for primitives

	`OptionalDouble`	`OptionalInt`	`OptionalLong`
Getting as a primitive	`getAsDouble()`	`getAsInt()`	`getAsLong()`
`orElseGet()` parameter type	`DoubleSupplier`	`IntSupplier`	`LongSupplier`
Return type of `max()` and `min()`	`OptionalDouble`	`OptionalInt`	`OptionalLong`
Return type of `sum()`	`double`	`int`	`long`
Return type of `average()`	`OptionalDouble`	`OptionalDouble`	`OptionalDouble`

Let's try an example to make sure that you understand this.

```
5: LongStream longs = LongStream.of(5, 10);
6: long sum = longs.sum();
7: System.out.println(sum);     // 15
8: DoubleStream doubles = DoubleStream.generate(() -> Math.PI);
9: OptionalDouble min = doubles.min(); // runs infinitely
```

Line 5 creates a stream of `long` primitives with two elements. Line 6 shows that we don't use an optional to calculate a sum. Line 8 creates an infinite stream of `double` primitives. Line 9 is there to remind you that a question about code that runs infinitely can appear with primitive streams as well.

Summarizing Statistics

You've learned enough to be able to get the maximum value from a stream of `int` primitives. If the stream is empty, we want to throw an exception.

```
private static int max(IntStream ints) {
   OptionalInt optional = ints.max();
   return optional.orElseThrow(RuntimeException::new);
}
```

This should be old hat by now. We got an `OptionalInt` because we have an `IntStream`. If the optional contains a value, we return it. Otherwise, we throw a new `RuntimeException`.

Now we want to change the method to take an `IntStream` and return a range. The range is the minimum value subtracted from the maximum value. Uh-oh. Both `min()` and `max()` are terminal operations, which means that they use up the stream when they are run. We can't run two terminal operations against the same stream. Luckily, this is a common problem and the primitive streams solve it for us with summary statistics. *Statistic* is just a big word for a number that was calculated from data.

```
private static int range(IntStream ints) {
   IntSummaryStatistics stats = ints.summaryStatistics();
   if (stats.getCount() == 0) throw new RuntimeException();
   return stats.getMax()-stats.getMin();
}
```

Here we asked Java to perform many calculations about the stream. Summary statistics include the following:

- **Smallest number (minimum):** `getMin()`
- **Largest number (maximum):** `getMax()`
- **Average:** `getAverage()`
- **Sum:** `getSum()`
- **Number of values:** `getCount()`

If the stream were empty, we'd have a count and sum of zero. The other methods would return an empty optional.

Learning the Functional Interfaces for Primitives

Remember when we told you to memorize Table 15.1, with the common functional interfaces, at the beginning of the chapter? Did you? If you didn't, go do it now. We are about to make it more involved. Just as there are special streams and optional classes for primitives, there are also special functional interfaces.

Luckily, most of them are for the `double`, `int`, and `long` types that you saw for streams and optionals. There is one exception, which is `BooleanSupplier`. We will cover that before introducing the ones for `double`, `int`, and `long`.

Functional Interfaces for *boolean*

`BooleanSupplier` is a separate type. It has one method to implement:

```
boolean getAsBoolean()
```

It works just as you've come to expect from functional interfaces. Here's an example:

```
12: BooleanSupplier b1 = () -> true;
13: BooleanSupplier b2 = () -> Math.random()> .5;
14: System.out.println(b1.getAsBoolean());  // true
15: System.out.println(b2.getAsBoolean());  // false
```

Lines 12 and 13 each create a `BooleanSupplier`, which is the only functional interface for `boolean`. Line 14 prints `true`, since it is the result of `b1`. Line 15 prints out `true or false`, depending on the random value generated.

Functional Interfaces for double, int, and long

Most of the functional interfaces are for `double`, `int`, and `long` to match the streams and optionals that we've been using for primitives. Table 15.11 shows the equivalent of Table 15.1 for these primitives. You probably won't be surprised that you have to memorize it. Luckily, you've memorized Table 15.1 by now and can apply what you've learned to Table 15.11.

TABLE 15.11 Common functional interfaces for primitives

Functional interfaces	# parameters	Return type	Single abstract method

`DoubleSupplier`
`IntSupplier`
`LongSupplier`
	0	`double`
		`int`
		`long`
			`getAsDouble`
			`getAsInt`
			`getAsLong`

`DoubleConsumer`
`IntConsumer`
`LongConsumer`
	1 (`double`)
	1 (`int`)
	1 (`long`)
		`void`	`accept`

`DoublePredicate`
`IntPredicate`
`LongPredicate`
	1 (`double`)
	1 (`int`)
	1 (`long`)
		`boolean`	`test`

`DoubleFunction<R>`
`IntFunction<R>`
`LongFunction<R>`
	1 (`double`)
	1 (`int`)
	1 (`long`)
		`R`	`apply`

`DoubleUnaryOperator`
`IntUnaryOperator`
`LongUnaryOperator`
	1 (`double`)
	1 (`int`)
	1 (`long`)
		`double`
		`int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

`DoubleBinaryOperator`
`IntBinaryOperator`
`LongBinaryOperator`
	2 (`double`, `double`)
	2 (`int`, `int`)
	2 (`long`, `long`)
		`double`
		`int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

There are a few things to notice that are different between Table 15.1 and Table 15.11.

- Generics are gone from some of the interfaces, and instead the type name tells us what primitive type is involved. In other cases, such as `IntFunction`, only the return type generic is needed because we're converting a primitive `int` into an object.
- The single abstract method is often renamed when a primitive type is returned.

In addition to Table 15.1 equivalents, some interfaces are specific to primitives. Table 15.12 lists these.

TABLE 15.12 Primitive-specific functional interfaces

Functional interfaces	# parameters	Return type	Single abstract method

`ToDoubleFunction<T>`
`ToIntFunction<T>`
`ToLongFunction<T>`
	1 (`T`)	`double`
		`int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

`ToDoubleBiFunction<T, U>`
`ToIntBiFunction<T, U>`
`ToLongBiFunction<T, U>`
	2 (`T` , `U`)	`double`
		`int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

`DoubleToIntFunction`
`DoubleToLongFunction`
`IntToDoubleFunction`
`IntToLongFunction`
`LongToDoubleFunction`
`LongToIntFunction`
	1 (`double`)
	1 (`double`)
	1 (`int`)
	1 (`int`)
	1 (`long`)
	1 (`long`)
		`int`
		`long`
		`double`
		`long`
		`double`
		`int`
			`applyAsInt`
			`applyAsLong`
			`applyAsDouble`
			`applyAsLong`
			`applyAsDouble`
			`applyAsInt`

`ObjDoubleConsumer<T>`
`ObjIntConsumer<T>`
`ObjLongConsumer<T>`
	2 (`T`, `double`)
	2 (`T`, `int`)
	2 (`T`, `long`)
		`void`	`accept`

We've been using functional interfaces all chapter long, so you should have a good grasp of how to read the table by now. Let's do one example just to be sure. Which functional interface would you use to fill in the blank to make the following code compile?

```
var d = 1.0;
_______________ f1 = x -> 1;
f1.applyAsInt(d);
```

When you see a question like this, look for clues. You can see that the functional interface in question takes a `double` parameter and returns an `int`. You can also see that it has a single abstract method named `applyAsInt`. The `DoubleToIntFunction` and `ToIntFunction` meet all three of those criteria.

Working with Advanced Stream Pipeline Concepts

You've almost reached the end of learning about streams. We have only a few more topics left. You'll see the relationship between streams and the underlying data, chaining `Optional` and grouping collectors.

Linking Streams to the Underlying Data

What do you think this outputs?

```
25: var cats = new ArrayList<String>();
26: cats.add("Annie");
27: cats.add("Ripley");
28: var stream = cats.stream();
29: cats.add("KC");
30: System.out.println(stream.count());
```

The correct answer is `3`. Lines 25–27 create a `List` with two elements. Line 28 requests that a stream be created from that `List`. Remember that streams are lazily evaluated. This means that the stream isn't actually created on line 28. An object is created that knows where to look for the data when it is needed. On line 29, the `List` gets a new element. On line 30, the stream pipeline actually runs. The stream pipeline runs first, looking at the source and seeing three elements.

Chaining Optionals

By now, you are familiar with the benefits of chaining operations in a stream pipeline. A few of the intermediate operations for streams are available for `Optional`.

Suppose that you are given an `Optional<Integer>` and asked to print the value, but only if it is a three-digit number. Without functional programming, you could write the following:

```
private static void threeDigit(Optional<Integer> optional) {
   if (optional.isPresent()) {  // outer if
      var num = optional.get();
      var string = "" + num;
      if (string.length() == 3) // inner if
         System.out.println(string);
```

```
    }
}
```

It works, but it contains nested `if` statements. That's extra complexity. Let's try this again with functional programming.

```
private static void threeDigit(Optional<Integer> optional) {
   optional.map(n -> "" + n)            // part 1
      .filter(s -> s.length() == 3)     // part 2
      .ifPresent(System.out::println);  // part 3
}
```

This is much shorter and more expressive. With lambdas, the exam is fond of carving up a single statement and identifying the pieces with a comment. We've done that here to show what happens with both the functional programming and nonfunctional programming approaches.

Suppose that we are given an empty `Optional`. The first approach returns `false` for the outer `if` statement. The second approach sees an empty `Optional` and has both `map()` and `filter()` pass it through. Then `ifPresent()` sees an empty `Optional` and doesn't call the `Consumer` parameter.

The next case is where we are given an `Optional.of(4)`. The first approach returns `false` for the inner `if` statement. The second approach maps the number `4` to `"4"`. The `filter()` then returns an empty `Optional` since the filter doesn't match, and `ifPresent()` doesn't call the `Consumer` parameter.

The final case is where we are given an `Optional.of(123)`. The first approach returns `true` for both `if` statements. The second approach maps the number `123` to `"123"`. The `filter()` then returns the same `Optional`, and `ifPresent()` now does call the `Consumer` parameter.

Now suppose that we wanted to get an `Optional<Integer>` representing the length of the `String` contained in another `Optional`. Easy enough.

```
Optional<Integer> result = optional.map(String::length);
```

What if we had a helper method that did the logic of calculating something for us that returns `Optional<Integer>`? Using `map` doesn't work.

```
Optional<Integer> result = optional
   .map(ChainingOptionals::calculator); // DOES NOT COMPILE
```

The problem is that calculator returns `Optional<Integer>`. The `map()` method adds another `Optional`, giving us `Optional<Optional<Integer>>`. Well, that's no good. The solution is to call `flatMap()` instead.

```
Optional<Integer> result = optional
    .flatMap(ChainingOptionals::calculator);
```

This one works because `flatMap` removes the unnecessary layer. In other words, it flattens the result. Chaining calls to `flatMap()` is useful when you want to transform one `Optional` type to another.

CHECKED EXCEPTIONS AND FUNCTIONAL INTERFACES

You might have noticed by now that most functional interfaces do not declare checked exceptions. This is normally OK. However, it is a problem when working with methods that declare checked exceptions. Suppose that we have a class with a method that throws a checked exception.

```
import java.io.*;
import java.util.*;
public class ExceptionCaseStudy {
   private static List<String> create() throws IOException {
      throw new IOException();
   }
}
```

Now we use it in a stream.

```
public void good() throws IOException {
   ExceptionCaseStudy.create().stream().count();
}
```

Nothing new here. The `create()` method throws a checked exception. The calling method handles or declares it. Now what about this one?

```
public void bad() throws IOException {
   Supplier<List<String>> s = ExceptionCaseStudy::create; // DOES NOT COMPILE
}
```

The actual compiler error is as follows:

```
unhandled exception type IOException
```

Say what now? The problem is that the lambda to which this method reference expands does not declare an exception. The `Supplier` interface does not allow checked exceptions. There are two approaches to get around this problem. One is to catch the exception and turn it into an unchecked exception.

```
public void ugly() {
   Supplier<List<String>> s = () -> {
      try {
         return ExceptionCaseStudy.create();
      } catch (IOException e) {
```

```
            throw new RuntimeException(e);
         }
      };
   }
```

This works. But the code is ugly. One of the benefits of functional programming is that the code is supposed to be easy to read and concise. Another alternative is to create a wrapper method with the `try/catch`.

```
   private static List<String> createSafe() {
      try {
         return ExceptionCaseStudy.create();
      } catch (IOException e) {
         throw new RuntimeException(e);
      } }
```

Now we can use the safe wrapper in our `Supplier` without issue.

```
   public void wrapped() {
      Supplier<List<String>> s2 = ExceptionCaseStudy::createSafe;
   }
```

Collecting Results

You're almost finished learning about streams. The last topic builds on what you've learned so far to group the results. Early in the chapter, you saw the `collect()` terminal operation. There are many predefined collectors, including those shown in Table 15.13. These collectors are available via `static` methods on the `Collectors` interface. We will look at the different types of collectors in the following sections.

TABLE 15.13 Examples of grouping/partitioning collectors

Collector	Description	Return value when passed to `collect`
`averagingDouble(ToDoubleFunction f)` `averagingInt(ToIntFunction f)` `averagingLong(ToLongFunction f)`	Calculates the average for our three core primitive types	`Double`
`counting()`	Counts the number of elements	`Long`
`groupingBy(Function f)` `groupingBy(Function f, Collector dc)` `groupingBy(Function f, Supplier s, Collector dc)`	Creates a map grouping by the specified function with the optional map type supplier and optional downstream collector	`Map<K, List<T>>`
`joining(CharSequence cs)`	Creates a single `String` using `cs` as a delimiter between elements if one is specified	`String`
`maxBy(Comparator c)` `minBy(Comparator c)`	Finds the largest/smallest elements	`Optional<T>`
`mapping(Function f, Collector dc)`	Adds another level of collectors	`Collector`

Collector	**Description**	**Return value when passed to** `collect`
`partitioningBy(Predicate p)` `partitioningBy(Predicate p, Collector dc)`	Creates a map grouping by the specified predicate with the optional further downstream collector	`Map<Boolean, List<T>>`

`summarizingDouble(ToDoubleFunction f)`
`summarizingInt(ToIntFunction f)`
`summarizingLong(ToLongFunction f)`
	Calculates average, min, max, and so on	`DoubleSummaryStatistics`
		`IntSummaryStatistics`
		`LongSummaryStatistics`

`summingDouble(ToDoubleFunction f)`
`summingInt(ToIntFunction f)`
`summingLong(ToLongFunction f)`
	Calculates the sum for our three core primitive types	`Double`
		`Integer`
		`Long`

`toList()` `toSet()`	Creates an arbitrary type of list or set	`List` `Set`
`toCollection(Supplier s)`	Creates a `Collection` of the specified type	`Collection`

`toMap(Function k, Function v)`
`toMap(Function k, Function v, BinaryOperator m)`
`toMap(Function k, Function v, BinaryOperator m, Supplier s)`
	Creates a map using functions to map the keys, values, an optional merge function, and an optional map type supplier	`Map`

Collecting Using Basic Collectors

Luckily, many of these collectors work in the same way. Let's look at an example.

```
var ohMy = Stream.of("lions", "tigers", "bears");
String result = ohMy.collect(Collectors.joining(", "));
System.out.println(result); // lions, tigers, bears
```

Notice how the predefined collectors are in the `Collectors` class rather than the `Collector` interface. This is a common theme, which you saw with `Collection` versus `Collections`. In fact, you'll see this pattern again in Chapter 20, “NIO.2,” when working with `Paths` and `Path`, and other related types.

We pass the predefined `joining()` collector to the `collect()` method. All elements of the stream are then merged into a `String` with the specified delimiter between each element. It is important to pass the `Collector` to the `collect` method. It exists to help collect elements. A `Collector` doesn't do anything on its own.

Let's try another one. What is the average length of the three animal names?

```
var ohMy = Stream.of("lions", "tigers", "bears");
Double result = ohMy.collect(Collectors.averagingInt(String::length));
System.out.println(result); // 5.333333333333333
```

The pattern is the same. We pass a collector to `collect()`, and it performs the average for us. This time, we needed to pass a function to tell the collector what to average. We used a method reference, which returns an `int` upon execution. With primitive streams, the result of an average was always a `double`, regardless of what type is being averaged. For collectors, it is a `Double` since those need an `Object`.

Often, you'll find yourself interacting with code that was written without streams. This means that it will expect a `Collection` type rather than a `Stream` type. No problem. You can still express yourself using a `Stream` and then convert to a `Collection` at the end, for example:

```
var ohMy = Stream.of("lions", "tigers", "bears");
TreeSet<String> result = ohMy
   .filter(s -> s.startsWith("t"))
   .collect(Collectors.toCollection(TreeSet::new));
System.out.println(result); // [tigers]
```

This time we have all three parts of the stream pipeline. `Stream.of()` is the source for the stream. The intermediate operation is `filter()`. Finally, the terminal operation is `collect()`, which creates a `TreeSet`. If we didn't care which implementation of `Set` we got, we could have written `Collectors.toSet()` instead.

At this point, you should be able to use all of the `Collectors` in Table 15.13 except `groupingBy()`, `mapping()`, `partitioningBy()`, and `toMap()`.

Collecting into Maps

Code using `Collectors` involving maps can get quite long. We will build it up slowly. Make sure that you understand each example before going on to the next one. Let's start with a straightforward example to create a map from a stream.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<String, Integer> map = ohMy.collect(
   Collectors.toMap(s -> s, String::length));
System.out.println(map); // {lions=5, bears=5, tigers=6}
```

When creating a map, you need to specify two functions. The first function tells the collector how to create the key. In our example, we use the provided `String` as the key. The second function tells the collector how to create the value. In our example, we use the length of the `String` as the value.

NOTE

Returning the same value passed into a lambda is a common operation, so Java provides a method for it. You can rewrite `s -> s` as `Function.identity()`. It is not shorter and may or may not be clearer, so use your judgment on whether to use it.

Now we want to do the reverse and map the length of the animal name to the name itself. Our first incorrect attempt is shown here:

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, String> map = ohMy.collect(Collectors.toMap(
   String::length,
   k -> k)); // BAD
```

Running this gives an exception similar to the following:

```
Exception in thread "main"
   java.lang.IllegalStateException: Duplicate key 5
```

What's wrong? Two of the animal names are the same length. We didn't tell Java what to do. Should the collector choose the first one it encounters? The last one it encounters? Concatenate the two? Since the collector has no idea what to do, it “solves” the problem by throwing an exception and making it our problem. How thoughtful. Let's suppose that our requirement is to create a comma-separated `String` with the animal names. We could write this:

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, String> map = ohMy.collect(Collectors.toMap(
   String::length,
   k -> k,
  (s1, s2) -> s1 + "," + s2));
System.out.println(map);            // {5=lions,bears, 6=tigers}
System.out.println(map.getClass()); // class java.util.HashMap
```

It so happens that the `Map` returned is a `HashMap`. This behavior is not guaranteed. Suppose that we want to mandate that the code return a `TreeMap` instead. No problem. We would just add a constructor reference as a parameter.

```
var ohMy = Stream.of("lions", "tigers", "bears");
TreeMap<Integer, String> map = ohMy.collect(Collectors.toMap(
   String::length,
   k -> k,
   (s1, s2) -> s1 + "," + s2,
   TreeMap::new));
System.out.println(map); //         // {5=lions,bears, 6=tigers}
System.out.println(map.getClass()); // class java.util.TreeMap
```

This time we got the type that we specified. With us so far? This code is long but not particularly complicated. We did promise you that the code would be long!

Collecting Using Grouping, Partitioning, and Mapping

Great job getting this far. The exam creators like asking about `groupingBy()` and `partitioningBy()`, so make sure you understand these sections very well. Now suppose that we want to get groups of names by their length. We can do that by saying that we want to group by length.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, List<String>> map = ohMy.collect(
   Collectors.groupingBy(String::length));
System.out.println(map);    // {5=[lions, bears], 6=[tigers]}
```

The `groupingBy()` collector tells `collect()` that it should group all of the elements of the stream into a `Map`. The function determines the keys in the `Map`. Each value in the `Map` is a `List` of all entries that match that key.

NOTE

Note that the function you call in `groupingBy()` cannot return `null`. It does not allow `null` keys.

Suppose that we don't want a `List` as the value in the map and prefer a `Set` instead. No problem. There's another method signature that lets us pass a *downstream collector*. This is a second collector that does something special with the values.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Set<String>> map = ohMy.collect(
   Collectors.groupingBy(
      String::length,
      Collectors.toSet()));
System.out.println(map);    // {5=[lions, bears], 6=[tigers]}
```

We can even change the type of `Map` returned through yet another parameter.

```
var ohMy = Stream.of("lions", "tigers", "bears");
TreeMap<Integer, Set<String>> map = ohMy.collect(
   Collectors.groupingBy(
      String::length,
      TreeMap::new,
      Collectors.toSet()));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

This is very flexible. What if we want to change the type of `Map` returned but leave the type of values alone as a `List`? There isn't a method for this specifically because it is easy enough to write with the existing ones.

```
var ohMy = Stream.of("lions", "tigers", "bears");
TreeMap<Integer, List<String>> map = ohMy.collect(
   Collectors.groupingBy(
       String::length,
       TreeMap::new,
       Collectors.toList()));
System.out.println(map);
```

Partitioning is a special case of grouping. With partitioning, there are only two possible groups—true and false. *Partitioning* is like splitting a list into two parts.

Suppose that we are making a sign to put outside each animal's exhibit. We have two sizes of signs. One can accommodate names with five or fewer characters. The other is needed for longer names. We can partition the list according to which sign we need.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Boolean, List<String>> map = ohMy.collect(
   Collectors.partitioningBy(s -> s.length() <= 5));
System.out.println(map);    // {false=[tigers], true=[lions, bears]}
```

Here we passed a `Predicate` with the logic for which group each animal name belongs in. Now suppose that we've figured out how to use a different font, and seven characters can now fit on the smaller sign. No worries. We just change the `Predicate`.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Boolean, List<String>> map = ohMy.collect(
   Collectors.partitioningBy(s -> s.length() <= 7));
System.out.println(map);    // {false=[], true=[lions, tigers, bears]}
```

Notice that there are still two keys in the map—one for each `boolean` value. It so happens that one of the values is an empty list, but it is still there. As with `groupingBy()`, we can change the type of `List` to something else.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Boolean, Set<String>> map = ohMy.collect(
   Collectors.partitioningBy(
      s -> s.length() <= 7,
      Collectors.toSet()));
System.out.println(map);    // {false=[], true=[lions, tigers, bears]}
```

Unlike `groupingBy()`, we cannot change the type of `Map` that gets returned. However, there are only two keys in the map, so does it really matter which `Map` type we use?

Instead of using the downstream collector to specify the type, we can use any of the collectors that we've already shown. For example, we can group by the length of the animal name to see how many of each length we have.

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Long> map = ohMy.collect(
   Collectors.groupingBy(
      String::length,
      Collectors.counting()));
System.out.println(map);    // {5=2, 6=1}
```

DEBUGGING COMPLICATED GENERICS

When working with `collect()`, there are often many levels of generics, making compiler errors unreadable. Here are three useful techniques for dealing with this situation:

- Start over with a simple statement and keep adding to it. By making one tiny change at a time, you will know which code introduced the error.
- Extract parts of the statement into separate statements. For example, try writing `Collectors.groupingBy(String::length, Collectors.counting());`. If it compiles, you know that the problem lies elsewhere. If it doesn't compile, you have a much shorter statement to troubleshoot.
- Use generic wildcards for the return type of the final statement; for example, `Map<?, ?>`. If that change alone allows the code to compile, you'll know that the problem lies with the return type not being what you expect.

Finally, there is a `mapping()` collector that lets us go down a level and add another collector. Suppose that we wanted to get the first letter of the first animal alphabetically of each length. Why? Perhaps for random sampling. The examples on this part of the exam are fairly contrived as well. We'd write the following:

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Optional<Character>> map = ohMy.collect(
```

```
    Collectors.groupingBy(
       String::length,
       Collectors.mapping(
          s -> s.charAt(0),
          Collectors.minBy((a, b) -> a -b))));
System.out.println(map);    // {5=Optional[b], 6=Optional[t]}
```

We aren't going to tell you that this code is easy to read. We will tell you that it is the most complicated thing you need to understand for the exam. Comparing it to the previous example, you can see that we replaced `counting()` with `mapping()`. It so happens that `mapping()` takes two parameters: the function for the value and how to group it further.

You might see collectors used with a `static` import to make the code shorter. The exam might even use `var` for the return value and less indentation than we used. This means that you might see something like this:

```
var ohMy = Stream.of("lions", "tigers", "bears");
var map = ohMy.collect(groupingBy(String::length,
   mapping(s -> s.charAt(0), minBy((a, b) -> a -b))));
System.out.println(map);    // {5=Optional[b], 6=Optional[t]}
```

The code does the same thing as in the previous example. This means that it is important to recognize the collector names because you might not have the `Collectors` class name to call your attention to it.

NOTE

There is one more collector called `reducing()`. You don't need to know it for the exam. It is a general reduction in case all of the previous collectors don't meet your needs.

Summary

A functional interface has a single abstract method. You must know the functional interfaces.

- `Supplier<T>` with method: `T get()`
- `Consumer<T>` with method: `void accept(T t)`
- `BiConsumer<T, U>` with method: `void accept(T t, U u)`
- `Predicate<T>` with method: `boolean test(T t)`

- `BiPredicate<T, U>` with method: `boolean test(T t, U u)`
- `Function<T, R>` with method: `R apply(T t)`
- `BiFunction<T, U, R>` with method: `R apply(T t, U u)`
- `UnaryOperator<T>` with method: `T apply(T t)`
- `BinaryOperator<T>` with method: `T apply(T t1, T t2)`

An `Optional<T>` can be empty or store a value. You can check whether it contains a value with `isPresent()` and `get()` the value inside. You can return a different value with `orElse(T t)` or throw an exception with `orElseThrow()`. There are even three methods that take functional interfaces as parameters: `ifPresent(Consumer c)`, `orElseGet(Supplier s)`, and `orElseThrow(Supplier s)`. There are three optional types for primitives: `OptionalDouble`, `OptionalInt`, and `OptionalLong`. These have the methods `getAsDouble()`, `getAsInt()`, and `getAsLong()`, respectively.

A stream pipeline has three parts. The source is required, and it creates the data in the stream. There can be zero or more intermediate operations, which aren't executed until the terminal operation runs. The first stream class we covered was `Stream<T>`, which takes a generic argument `T`. The `Stream<T>` class includes many useful intermediate operations including `filter()`, `map()`, `flatMap()`, and `sorted()`. Examples of terminal operations include `allMatch()`, `count()`, and `forEach()`.

Besides the `Stream<T>` class, there are three primitive streams: `DoubleStream`, `IntStream`, and `LongStream`. In addition to the usual `Stream<T>` methods, `IntStream` and `LongStream` have `range()` and `rangeClosed()`. The call `range(1, 10)` on `IntStream` and `LongStream` creates a stream of the primitives from 1 to 9. By contrast, `rangeClosed(1, 10)` creates a stream of the primitives from 1 to 10. The primitive streams have math operations including `average()`, `max()`, and `sum()`. They also have `summaryStatistics()` to get many statistics in one call. There are also functional interfaces specific to streams. Except for `BooleanSupplier`, they are all for `double`, `int`, and `long` primitives as well.

You can use a `Collector` to transform a stream into a traditional collection. You can even group fields to create a complex map in one line. Partitioning works the same way as grouping, except that the keys are always `true` and `false`. A partitioned map always has two keys even if the value is empty for the key.

You should review the tables in the chapter. While there's a lot of tables, many share common patterns, making it easier to remember them. You absolutely must memorize Table 15.1. You should memorize Table 15.8

and Table 15.9 but be able to spot incompatibilities, such as type differences, if you can't memorize these two. Finally, remember that streams are lazily evaluated. They take lambdas or method references as parameters, which execute later when the method is run.

Exam Essentials

- **Identify the correct functional interface given the number of parameters, return type, and method name—and vice versa.** The most common functional interfaces are `Supplier`, `Consumer`, `Function`, and `Predicate`. There are also binary versions and primitive versions of many of these methods.
- **Write code that uses *Optional*.** Creating an `Optional` uses `Optional.empty()` or `Optional.of()`. Retrieval frequently uses `isPresent()` and `get()`. Alternatively, there are the functional `ifPresent()` and `orElseGet()` methods.
- **Recognize which operations cause a stream pipeline to execute.** Intermediate operations do not run until the terminal operation is encountered. If no terminal operation is in the pipeline, a `Stream` is returned but not executed. Examples of terminal operations include `collect()`, `forEach()`, `min()`, and `reduce()`.
- **Determine which terminal operations are reductions.** Reductions use all elements of the stream in determining the result. The reductions that you need to know are `collect()`, `count()`, `max()`, `min()`, and `reduce()`. A mutable reduction collects into the same object as it goes. The `collect()` method is a mutable reduction.
- **Write code for common intermediate operations.** The `filter()` method returns a `Stream<T>` filtering on a `Predicate<T>`. The `map()` method returns a `Stream` transforming each element of type `T` to another type `R` through a `Function <T,R>`. The `flatMap()` method flattens nested streams into a single level and removes empty streams.
- **Compare primitive streams to *Stream*<*T*>.** Primitive streams are useful for performing common operations on numeric types including statistics like `average()`, `sum()`, etc. There are three primitive stream classes: `DoubleStream`, `IntStream`, and `LongStream`. There are also three primitive `Optional` classes: `OptionalDouble`, `OptionalInt`, and `OptionalLong`. Aside from `BooleanSupplier`, they all involve the `double`, `int`, or `long` primitives.
- **Convert primitive stream types to other primitive stream types.** Normally when mapping, you just call the *map()* method. When changing the class used for the stream, a different method is needed. To convert to `Stream`, you use `mapToObj()`. To convert to `DoubleStream`, you use `mapToDouble()`. To convert to `IntStream`, you use `mapToInt()`. To convert to `LongStream`, you use `mapToLong()`.

- **Use *peek()* to inspect the stream.** The `peek()` method is an intermediate operation often used for debugging purposes. It executes a lambda or method reference on the input and passes that same input through the pipeline to the next operator. It is useful for printing out what passes through a certain point in a stream.
- **Search a stream. The `findFirst()` and `findAny()` methods return a single element from a stream in an `Optional`.** The `anyMatch()`, `allMatch()`, and `noneMatch()` methods return a `boolean`. Be careful, because these three can hang if called on an infinite stream with some data. All of these methods are terminal operations.
- **Sort a stream.** The `sorted()` method is an intermediate operation that sorts a stream. There are two versions: the signature with zero parameters that sorts using the natural sort order, and the signature with one parameter that sorts using that `Comparator` as the sort order.
- **Compare *groupingBy()* and *partitioningBy()*.** The `groupingBy()` method is a terminal operation that creates a `Map`. The keys and return types are determined by the parameters you pass. The values in the `Map` are a `Collection` for all the entries that map to that key. The `partitioningBy()` method also returns a `Map`. This time, the keys are `true` and `false`. The values are again a `Collection` of matches. If there are no matches for that `boolean`, the `Collection` is empty.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. What could be the output of the following?

```
var stream = Stream.iterate("", (s) -> s + "1");
System.out.println(stream.limit(2).map(x -> x + "2"));
```

   A. `12112`
   B. `212`
   C. `212112`
   D. `java.util.stream.ReferencePipeline$3@4517d9a3`
   E. The code does not compile.
   F. An exception is thrown.
   G. The code hangs.

2. What could be the output of the following?

```
Predicate<String> predicate = s -> s.startsWith("g");
var stream1 = Stream.generate(() -> "growl!");
```

```
var stream2 = Stream.generate(() -> "growl!");
var b1 = stream1.anyMatch(predicate);
var b2 = stream2.allMatch(predicate);
System.out.println(b1 + " " + b2);
```

A. `true false`
B. `true true`
C. `java.util.stream.ReferencePipeline$3@4517d9a3`
D. The code does not compile.
E. An exception is thrown.
F. The code hangs.

3. What could be the output of the following?

```
Predicate<String> predicate = s -> s.length()> 3;
var stream = Stream.iterate("-",
   s -> ! s.isEmpty(), (s) -> s + s);
var b1 = stream.noneMatch(predicate);
var b2 = stream.anyMatch(predicate);
System.out.println(b1 + " " + b2);
```

A. `false false`
B. `false true`
C. `java.util.stream.ReferencePipeline$3@4517d9a3`
D. The code does not compile.
E. An exception is thrown.
F. The code hangs.

4. Which are true statements about terminal operations in a stream that runs successfully? (Choose all that apply.)
A. At most, one terminal operation can exist in a stream pipeline.
B. Terminal operations are a required part of the stream pipeline in order to get a result.
C. Terminal operations have `Stream` as the return type.
D. The `peek()` method is an example of a terminal operation.
E. The referenced `Stream` may be used after calling a terminal operation.

5. Which of the following sets `result` to `8.0`? (Choose all that apply.)

A.
```
double result = LongStream.of(6L, 8L, 10L)
   .mapToInt(x -> (int) x)
   .collect(Collectors.groupingBy(x -> x))
   .keySet()
   .stream()
   .collect(Collectors.averagingInt(x -> x));
```

B.
```
double result = LongStream.of(6L, 8L, 10L)
   .mapToInt(x -> x)
   .boxed()
   .collect(Collectors.groupingBy(x -> x))
   .keySet()
   .stream()
   .collect(Collectors.averagingInt(x -> x));
```

C.
```
double result = LongStream.of(6L, 8L, 10L)
   .mapToInt(x -> (int) x)
   .boxed()
   .collect(Collectors.groupingBy(x -> x))
   .keySet()
   .stream()
   .collect(Collectors.averagingInt(x -> x));
```

D.
```
double result = LongStream.of(6L, 8L, 10L)
   .mapToInt(x -> (int) x)
   .collect(Collectors.groupingBy(x -> x, Collectors.toSet()))
   .keySet()
   .stream()
   .collect(Collectors.averagingInt(x -> x));
```

E.
```
double result = LongStream.of(6L, 8L, 10L)
   .mapToInt(x -> x)
   .boxed()
   .collect(Collectors.groupingBy(x -> x, Collectors.toSet()))
   .keySet()
   .stream()
   .collect(Collectors.averagingInt(x -> x));
```

F.
```
double result = LongStream.of(6L, 8L, 10L)
   .mapToInt(x -> (int) x)
   .boxed()
   .collect(Collectors.groupingBy(x -> x, Collectors.toSet()))
   .keySet()
   .stream()
   .collect(Collectors.averagingInt(x -> x));
```

6. Which of the following can fill in the blank so that the code prints out `false` ? (Choose all that apply.)

```
var s = Stream.generate(() -> "meow");
var match = s.________________(String::isEmpty);
System.out.println(match);
```

A. `allMatch`

B. `anyMatch`

C. `findAny`

D. `findFirst`

E. `noneMatch`

F. None of the above

7. We have a method that returns a sorted list without changing the original. Which of the following can replace the method implementation to do the same with streams?

```
private static List<String> sort(List<String> list) {
   var copy = new ArrayList<String>(list);
   Collections.sort(copy, (a, b) -> b.compareTo(a));
   return copy;
}
```

A.
```
return list.stream()
   .compare((a, b) -> b.compareTo(a))
   .collect(Collectors.toList());
```

B.
```
return list.stream()
   .compare((a, b) -> b.compareTo(a))
   .sort();
```

C.
```
return list.stream()
   .compareTo((a, b) -> b.compareTo(a))
   .collect(Collectors.toList());
```

D.
```
return list.stream()
   .compareTo((a, b) -> b.compareTo(a))
   .sort();
```

E.
```
return list.stream()
   .sorted((a, b) -> b.compareTo(a))
   .collect();
```

F.
```
return list.stream()
   .sorted((a, b) -> b.compareTo(a))
   .collect(Collectors.toList());
```

8. Which of the following are true given this declaration? (Choose all that apply.)

```
var is = IntStream.empty();
```

   A. `is.average()` returns the type `int`.
   B. `is.average()` returns the type `OptionalInt`.
   C. `is.findAny()` returns the type `int`.
   D. `is.findAny()` returns the type `OptionalInt`.
   E. `is.sum()` returns the type `int`.
   F. `is.sum()` returns the type `OptionalInt`.

9. Which of the following can we add after line 6 for the code to run without error and not produce any output? (Choose all that apply.)

```
4: var stream = LongStream.of(1, 2, 3);
5: var opt = stream.map(n -> n * 10)
6:    .filter(n -> n < 5).findFirst();
```

   A.
```
if (opt.isPresent())
   System.out.println(opt.get());
```

   B.
```
if (opt.isPresent())
   System.out.println(opt.getAsLong());
```

   C.
```
opt.ifPresent(System.out.println);
```

   D.
```
opt.ifPresent(System.out::println);
```

   E. None of these; the code does not compile.
   F. None of these; line 5 throws an exception at runtime.

10. Given the four statements (L, M, N, O), select and order the ones that would complete the expression and cause the code to output 10 lines. (Choose all that apply.)

```
Stream.generate(() -> "1")
   L: .filter(x -> x.length()> 1)
   M: .forEach(System.out::println)
   N: .limit(10)
   O: .peek(System.out::println)
;
```

A. L, N

B. L, N, O

C. L, N, M

D. L, N, M, O

E. L, O, M

F. N, M

G. N, O

11. What changes need to be made together for this code to print the string `12345`? (Choose all that apply.)

```
Stream.iterate(1, x -> x++)
   .limit(5).map(x -> x)
   .collect(Collectors.joining());
```

A. Change `Collectors.joining()` to `Collectors.joining(",")`.

B. Change `map(x -> x)` to `map(x -> "" + x)`.

C. Change `x -> x++` to `x -> ++x`.

D. Add `forEach(System.out::print)` after the call to `collect()`.

E. Wrap the entire line in a `System.out.print` statement.

F. None of the above. The code already prints `12345`.

12. Which functional interfaces complete the following code? For line 7, assume `m` and `n` are instances of functional interfaces that exist and have the same type as `y`. (Choose three.)

```
6: _______________ x = String::new;
7: _______________ y = m.andThen(n);
8: _______________ z = a -> a + a;
```

A. `BinaryConsumer<String, String>`

B. `BiConsumer<String, String>`

C. `BinaryFunction<String, String>`

D. `BiFunction<String, String>`

E. `Predicate<String>`

F. `Supplier<String>`

G. `UnaryOperator<String>`

H. `UnaryOperator<String, String>`

13. Which of the following is true?

```
List<Integer> x1 = List.of(1, 2, 3);
List<Integer> x2 = List.of(4, 5, 6);
List<Integer> x3 = List.of();
Stream.of(x1, x2, x3).map(x -> x + 1)
   .flatMap(x -> x.stream())
   .forEach(System.out::print);
```

A. The code compiles and prints `123456`.

B. The code compiles and prints `234567`.

C. The code compiles but does not print anything.

D. The code compiles but prints stream references.

E. The code runs infinitely.

F. The code does not compile.

G. The code throws an exception.

14. Which of the following is true? (Choose all that apply.)

```
4: Stream<Integer> s = Stream.of(1);
5: IntStream is = s.boxed();
6: DoubleStream ds = s.mapToDouble(x -> x);
7: Stream<Integer> s2 = ds.mapToInt(x -> x);
8: s2.forEach(System.out::print);
```

A. Line 4 causes a compiler error.

B. Line 5 causes a compiler error.

C. Line 6 causes a compiler error.

D. Line 7 causes a compiler error.

E. Line 8 causes a compiler error.

F. The code compiles but throws an exception at runtime.

G. The code compiles and prints `1`.

15. Given the generic type `String`, the `partitioningBy()` collector creates a `Map<Boolean, List<String>>` when passed to `collect()` by default. When a downstream collector is passed to `partitioningBy()`, which return types can be created? (Choose all that apply.)

A. `Map<boolean, List<String>>`

B. `Map<Boolean, List<String>>`

C. `Map<Boolean, Map<String>>`

D. `Map<Boolean, Set<String>>`

E. `Map<Long, TreeSet<String>>`

F. None of the above

16. Which of the following statements are true about this code? (Choose all that apply.)

```
Predicate<String> empty = String::isEmpty;
Predicate<String> notEmpty = empty.negate();

var result = Stream.generate(() -> "")
    .limit(10)
    .filter(notEmpty)
    .collect(Collectors.groupingBy(k -> k))
    .entrySet()
    .stream()
```

```
29:     .map(Entry::getValue)
30:     .flatMap(Collection::stream)
31:     .collect(Collectors.partitioningBy(notEmpty));
32: System.out.println(result);
```

A. It outputs: `{}`

B. It outputs: `{false=[], true=[]}`

C. If we changed line 31 from `partitioningBy(notEmpty)` to `groupingBy(n -> n)`, it would output: `{}`

D. If we changed line 31 from `partitioningBy(notEmpty)` to `groupingBy(n -> n)`, it would output: `{false=[], true=[]}`

E. The code does not compile.

F. The code compiles but does not terminate at runtime.

17. Which of the following is equivalent to this code? (Choose all that apply.)

```
UnaryOperator<Integer> u = x -> x * x;
```

A. `BiFunction<Integer> f = x -> x*x;`

B. `BiFunction<Integer, Integer> f = x -> x*x;`

C. `BinaryOperator<Integer, Integer> f = x -> x*x;`

D. `Function<Integer> f = x -> x*x;`

E. `Function<Integer, Integer> f = x -> x*x;`

F. None of the above

18. What is the result of the following?

```
var s = DoubleStream.of(1.2, 2.4);
s.peek(System.out::println).filter(x -> x> 2).count();
```

A. `1`

B. `2`

C. `2.4`

D. `1.2` and `2.4`

E. There is no output.

F. The code does not compile.

G. An exception is thrown.

19. What does the following code output?

```
Function<Integer, Integer> s = a -> a + 4;
Function<Integer, Integer> t = a -> a * 3;
Function<Integer, Integer> c = s.compose(t);
System.out.println(c.apply(1));
```

A. `7`
B. `15`
C. The code does not compile because of the data types in the lambda expressions.
D. The code does not compile because of the `compose()` call.
E. The code does not compile for another reason.

20. Which of the following functional interfaces contain an abstract method that returns a primitive value? (Choose all that apply.)
    A. `BooleanSupplier`
    B. `CharSupplier`
    C. `DoubleSupplier`
    D. `FloatSupplier`
    E. `IntSupplier`
    F. `StringSupplier`
21. What is the simplest way of rewriting this code?

```
List<Integer> x = IntStream.range(1, 6)
   .mapToObj(i -> i)
   .collect(Collectors.toList());
x.forEach(System.out::println);
```

A.
```
IntStream.range(1, 6);
```

B.
```
IntStream.range(1, 6)
   .forEach(System.out::println);
```

C.
```
IntStream.range(1, 6)
   .mapToObj(i -> i)
   .forEach(System.out::println);
```

D. None of the above is equivalent.
E. The provided code does not compile.

22. Which of the following throw an exception when an `Optional` is empty? (Choose all that apply.)
    A. `opt.orElse("");`
    B. `opt.orElseGet(() -> "");`
    C. `opt.orElseThrow();`
    D. `opt.orElseThrow(() -> throw new Exception());`
    E. `opt.orElseThrow(RuntimeException::new);`
    F. `opt.get();`
    G. `opt.get("");`

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE PRIVACY POLICY