Chapter 15 - Functional Programming

Working with Built-in 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)`

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();
}
```

```
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);
```

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
}
```

```
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);
```

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
}
```

```
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
```

Note that a `Predicate` returns a `boolean` primitive and not a `Boolean` object.

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
}
```

```
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
```

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
}
```

```
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
```

Convenience Methods on Functional Interfaces

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`

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

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

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*

```
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?

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

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

```
Optional<String> emptyOptional = Optional.empty();
Optional<String> optional = Optional.of("testingOptional");

@Testvoid testOptionalGet() {
    assertThrows(NoSuchElementException.class, () -> emptyOptional.get());
    assertEquals(optional.get(), "testingOptional");}

@Testvoid testOptionalIsPresent() {
    assertFalse(emptyOptional.isPresent());
    assertTrue(optional.isPresent());}

@Testvoid testOptionalIfPresent() {
    final var emptyList = new ArrayList<String>();
    final var list = new ArrayList<String>();
    emptyOptional.ifPresent(emptyList::add);
    optional.ifPresent(list::add);
```

```
    assertTrue(emptyList.isEmpty());
    assertFalse(list.isEmpty());}

@Testvoid testOptionalOrElse() {
    assertEquals(emptyOptional.orElse("another"), "another");
    assertEquals(optional.orElse("another"), "testingOptional");}

@Testvoid testOptionalOrElseGet() {
    assertEquals(emptyOptional.orElseGet(() -> "another"), "another");
    assertEquals(optional.orElseGet(() -> "another"), "testingOptional");}

@Testvoid testOptionalOrElseThrow() {
    assertThrows(NoSuchElementException.class, () -> emptyOptional.orElseThrow());
    assertEquals(optional.orElseThrow(), "testingOptional");

}
```

Using Streams

Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.

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

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
```

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);
```

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
```

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`

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**

```
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()

```
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
```

findAny() and *findFirst()*

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
```

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

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
```

forEach()

```
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
```

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)
```

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

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 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
```

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.

Using Common Intermediate Operations

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)
```

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

distinct()

```
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()

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

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

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

map()

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

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

flatMap()

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

```
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
```

sorted()

```
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-
```

```
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()

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

```
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);
```

Working with Primitive Streams

```
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
```

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
```

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`

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`	
`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);
```

Java provides a method that can generate a range of numbers.

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

Luckily, there's another method, `rangeClosed()`, which is inclusive on both parameters.

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

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()`

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 `LongStream`
`Stream<T>`	`Function<T,R>`	`ToDoubleFunction<T>`	`ToIntFunction<T>`	`ToLongFunction<T>`
`DoubleStream`	`Double Function<R>`	`DoubleUnary Operator`	`DoubleToInt Function`	`DoubleToLong Function`
`IntStream`	`IntFunction<R>`	`IntToDouble Function`	`IntUnary Operator`	`IntToLong Function`
`LongStream`	`Long Function<R>`	`LongToDouble Function`	`LongToInt Function`	`LongUnary Operator`

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();
}
```

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()`

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`

Summarizing Statistics

```
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()`

Learning the Functional Interfaces for Primitives

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
```

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`

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`

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.

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`
`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.

```
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`.

```
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.

```
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.

```
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
```

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.

```
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 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]}
```

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]}
```

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]}
```

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.