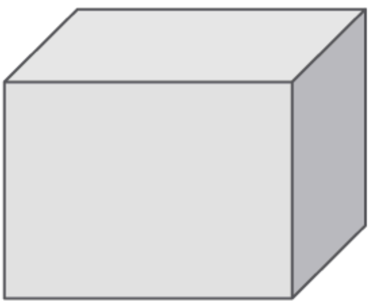
# Streams

# Optional

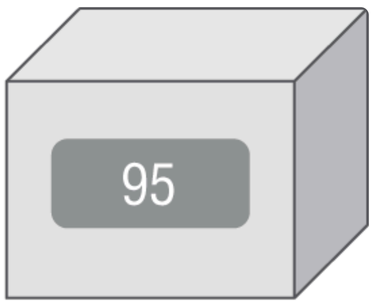
> ✎ **Optional**
>
> Is created using a factory. You can either request an empty `Optional` or pass a value for the `Optional` to wrap. Think of an `Optional` as a box that might have something in it or might instead be empty.

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

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



Optional.empty()




Optional.of(95)

An `Optional` can take a generic type, making it easier to retrieve values from it. You can see that one `Optional<Double>` 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.

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

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

------using ternary-------------
```

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

| Method | When `Optional` is empty | When `Optional` contains value |
|---|---|---|
| `get()` | Throws 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 |

## Dealing with an Empty `Optional`

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

-----

NaN
0.49775932295380165
```

Alternatively, we can have the code throw an exception.

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

or

Optional<Double> opt = average();
System.out.println(opt.orElseThrow(
   () -> new IllegalStateException())); // Your defined error
```

# Stream

Stream

Is a sequence of data.
A *Stream pipeline* consists of the operation that run on a stream to produce a result.

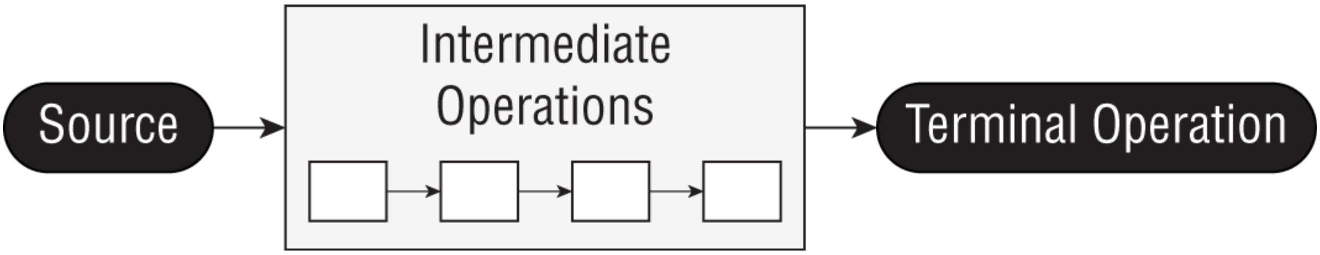



**TABLE 10.2** Intermediate vs. terminal operations

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

## Creating Stream sources

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

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

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

Also. you can create infinite streams:

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

```
// odd numbers less than 100

Stream<Integer> oddNumberUnder100 = Stream.iterate(
    1,                  // seed
    n -> n < 100,       // Predicate to specify when done
    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 `Collection`. |
| `coll.parallelStream()` | Finite | Creates `Stream` from `Collection` where the stream can run in parallel. |
| `Stream.generate(supplier)` | Infinite | Creates `Stream` by calling `Supplier` for each element upon request. |
| `Stream.iterate(seed, unaryOperator)` | Infinite | Creates `Stream` by using seed for first element and then calling `UnaryOperator` for each subsequent element upon request. |
| `Stream.iterate(seed, predicate, unaryOperator)` | Finite or infinite | Creates `Stream` by using seed for first element and then calling `UnaryOperator` for each subsequent element upon request. Stops if `Predicate` returns false. |

# Using Common terminal operation

TABLE 10.4 Terminal stream operations

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

example for min and max:

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

//Example
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
```

Example for findAny():

```
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
```

Example for matching methods:

```
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
```

Syntax for reduce():

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

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

public <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
```

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

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


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.

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.

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.

```
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]
```

# Intermediate operations

> 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.

## Filtering

> The `filter()` method returns a `Stream` with elements that match a given expression.

```
public 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
```

## Removing duplicates

> The `distinct()` method returns a stream with duplicate values removed.

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

## Restricting by Position

> The `limit()` and `skip()` methods can make a `Stream` smaller. The `limit()` method could also make a finite stream out of an infinite stream.

```
public Stream<T> limit(long maxSize)
public Stream<T> skip(long n)
----
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
s.skip(5)
   .limit(2)
   .forEach(System.out::print); // 67 // sare 5 pozitii si se opreste dupa a 2 a
```

## Mapping

> 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.

```
public <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
```

## Using *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 combine a stream of lists.

```
public <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);
```

## Sorting

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

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

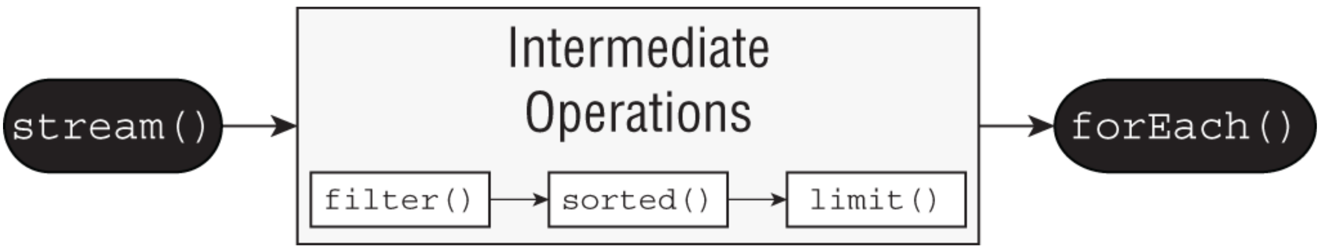
## Taking a Peek

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

```
public 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);                          // 1
```

Streams allow you to use chaining and express what you want to accomplish rather than how to do so.

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




# Working with primitive streams

# Creating Primitive Streams

Here are the 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 10.5 Common primitive stream methods

| Method | Primitive stream | Description |
|---|---|---|
| `OptionalDouble average()` | `IntStream` `LongStream` `DoubleStream` | Arithmetic mean of elements |
| `Stream<T> boxed()` | `IntStream` `LongStream` `DoubleStream` | `Stream<T>` where `T` is wrapper class associated with primitive value |
| `OptionalInt max()` | `IntStream` | Maximum element of stream |
| `OptionalLong max()` | `LongStream` | |
| `OptionalDouble max()` | `DoubleStream` | |
| `OptionalInt min()` | `IntStream` | Minimum element of stream |
| `OptionalLong min()` | `LongStream` | |
| `OptionalDouble min()` | `DoubleStream` | |
| `IntStream range(int a, int b)` | `IntStream` | Returns primitive stream from `a` (inclusive) to `b` (exclusive) |
| `LongStream range(long a, long b)` | `LongStream` | |
| `IntStream rangeClosed(int a, int b)` | `IntStream` | Returns primitive stream from `a` (inclusive) to `b` (inclusive) |
| `LongStream rangeClosed(long a, long b)` | `LongStream` | |
| `int sum()` | `IntStream` | Returns sum of elements in stream |
| `long sum()` | `LongStream` | |
| `double sum()` | `DoubleStream` | |
| `IntSummaryStatistics summaryStatistics()` | `IntStream` | Returns object containing numerous stream statistics such as average, min, max, etc. |

| | |
|---|---|
| LongSummaryStatistics summaryStatistics() | LongStream |
| DoubleSummaryStatistics summaryStatistics() | DoubleStream |

# Mapping streams

TABLE 10.6 Mapping methods between types of streams

| Source stream | 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 10.7 Function parameters when mapping between types of streams

| Source stream | 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 Optional with Primitive Streams

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

//this compute average
```

The return type is not the `Optional` you have become accustomed to using. It is a new type called `OptionalDouble`. 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
```

**TABLE 10.8** Optional types for primitives

| | `OptionalDouble` | `OptionalInt` | `OptionalLong` |
|---|---|---|---|
| Getting as 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` |

```
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
```

## Summarizing Statistics

```
private static int max(IntStream ints) {
   OptionalInt optional = ints.max();
   return optional.orElseThrow(RuntimeException::new);
}// We got an `OptionalInt` because we have an `IntStream`. If the optional
contains a value, we return it. Otherwise, we throw a new
`RuntimeException`.


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

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.

Summary statistics include the following:

- `getCount()`: Returns a `long` representing the number of values.
- `getAverage()`: Returns a `double` representing the average. If the stream is empty, returns `0.0`.
- `getSum()`: Returns the sum as a `double` for `DoubleSummaryStatistics`, and `long` for `IntSummaryStatistics` and `LongSummaryStastistics`.
- `getMin()`: Returns the smallest number (minimum) as a `double`, `int`, or `long`, depending on the type of the stream. If the stream is empty, returns the largest numeric value based on the type.
- `getMax()`: Returns the largest number (maximum) as a `double`, `int`, or `long` depending on the type of the stream. If the stream is empty, returns the smallest numeric value based on the type.

# Working with Advanced Stream Pipeline Concepts

## Linking Streams to the Underlying Data

```
var cats = new ArrayList<String>();
cats.add("Annie");
cats.add("Ripley");
var stream = cats.stream();
cats.add("KC");
System.out.println(stream.count()) // display 3
```

## Chaining *Optionals*

TABLE 10.9 Advanced `Optional` instance methods

| Method | When `Optional` is empty | When `Optional` contains value |
|---|---|---|
| `filter(Predicate p)` | Returns empty `Optional` | Returns `Optional` containing the element if it matches the `Predicate`, otherwise empty `Optional` |
| `flatMap(Function f)` | Returns empty `Optional` | Returns `Optional` with `Function` applied to the element. Return type of `Function` must inherit `Optional`. |
| `map(Function f)` | Returns empty `Optional` | Returns `Optional` with `Function` applied to the element |

```java
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);
   }
}

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
}
```

## Collecting results

TABLE 10.10 Examples of grouping/partitioning collectors

| Collector | Description | Return value when passed to `collect` |
|---|---|---|
| `averagingDouble(ToDoubleFunction f)`<br>`averagingInt(ToIntFunction f)`<br>`averagingLong(ToLongFunction f)` | Calculates average for three core primitive types | `Double` |
| `counting()` | Counts number of elements | `Long` |
| `filtering(Predicate p, Collector c)` | Applies filter before calling downstream collector | `R` |
| `groupingBy(Function f)` | Creates map grouping by specified function with optional map type supplier and optional downstream collector of type `D` | `Map<K, List<T>>` |
| `groupingBy(Function f, Collector dc)` | | `Map<K, List<D>>` |
| `groupingBy(Function f, Supplier s, Collector dc)` | | `Map<K, List<D>>` |
| `joining(CharSequence cs)` | Creates single `String` using `cs` as delimiter between elements if one is specified | `String` |
| `maxBy(Comparator c)`<br>`minBy(Comparator c)` | Finds largest/smallest elements | `Optional<T>` |
| `mapping(Function f, Collector dc)` | Adds another level of collectors | `Collector` |
| `partitioningBy(Predicate p)`<br>`partitioningBy(Predicate p, Collector dc)` | Creates map grouping by specified predicate with optional further downstream collector | `Map<Boolean, List<T>>` |
| `summarizingDouble(ToDoubleFunction f)`<br>`summarizingInt(ToIntFunction f)`<br>`summarizingLong(ToLongFunction f)` | Calculates average, min, max, etc. | `DoubleSummaryStatistics`<br>`IntSummaryStatistics`<br>`LongSummaryStatistics` |

| | | |
|---|---|---|
| summingDouble( ToDoubleFunction f)<br>summingInt( ToIntFunction f)<br>summingLong( ToLongFunction f) | Calculates sum for our three core primitive types | Double<br>Integer<br>Long |
| teeing(Collector c1, Collector c2, BiFunction f) | Works with results of two collectors to create new type | R |
| toList()<br>toSet() | Creates arbitrary type of list or set | List<br>Set |
| toCollection(Supplier s) | Creates Collection of specified type | Collection |
| toMap(Function k, Function v)<br>toMap(Function k, Function v, BinaryOperator m)<br>toMap(Function k, Function v, BinaryOperator m, Supplier s) | Creates map using functions to map keys, values, optional merge function, and optional map type supplier | Map |

## Using Basic Collectors

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

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.

```
Stream<String> ohMy1 = Stream.of("lions", "tigers", "bears");
List<String> mutableList = ohMy1.collect(Collectors.toList());

Stream<String> ohMy2 = Stream.of("lions", "tigers", "bears");
List<String> immutableList = ohMy2.toList();

mutableList.add("zebras");   // No issues
immutableList.add("zebras"); // UnsupportedOperationException
```

Almost? While both return a `List<String>`, the contract is different. The `Collectors.toList()` gives you a mutable list that you can edit later. The shorter `toList()` does not allow changes. We can see the difference in the following additional lines of code:

```
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)); // BAD

   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
```

## Grouping, Partitioning, and Mapping

```
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.
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]}
```

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.

```
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]}
```

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.

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.

```
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]}
```

## Teeing Collectors

```
record Separations(String spaceSeparated, String commaSeparated) {}
```

```
var list = List.of("x", "y", "z");

Separations result = list.stream()
   .collect(Collectors.teeing(
               Collectors.joining(" "),
               Collectors.joining(","),
               (s, c) -> new Separations(s, c)));
System.out.println(result);
----------------------------
Separations[spaceSeparated=x y z, commaSeparated=x,y,z]
```

There are three `Collector`s in this code. Two of them are for `joining()` and produce the values we want to return. The third is `teeing()`, which combines the results into the single object we want to return. This way, Java is happy because only one object is returned, and we are happy because we don't have to go through the stream twice.

## Using a *Spliterator*

The characteristics of a `Spliterator` depend on the underlying data source. A `Collection` data source is a basic `Spliterator`. By contrast, when using a `Stream` data source, the `Spliterator` can be parallel or even infinite. The `Stream` itself is executed lazily rather than when the `Spliterator` is created.

TABLE 10.11 `Spliterator` methods

| Method | Description |
| --- | --- |
| `Spliterator<T> trySplit()` | Returns `Spliterator` containing ideally half of the data, which is removed from the current `Spliterator`. This method can be called multiple times and will eventually return `null` when data is no longer splittable. |
| `void forEachRemaining(Consumer<T> c)` | Processes remaining elements in `Spliterator`. |
| `boolean tryAdvance(Consumer<T> c)` | Processes a single element from `Spliterator` if any remain. Returns whether element was processed. |

```
var stream = List.of("bird-", "bunny-", "cat-", "dog-", "fish-", "lamb-",
   "mouse-");
Spliterator<String> originalBagOfFood = stream.spliterator();
Spliterator<String> emmasBag = originalBagOfFood.trySplit();
emmasBag.forEachRemaining(System.out::print);  // bird-bunny-cat-

Spliterator<String> jillsBag = originalBagOfFood.trySplit();
```

```
jillsBag.tryAdvance(System.out::print);        // dog-
jillsBag.forEachRemaining(System.out::print);  // fish-

originalBagOfFood.forEachRemaining(System.out::print); // lamb-mouse-
```