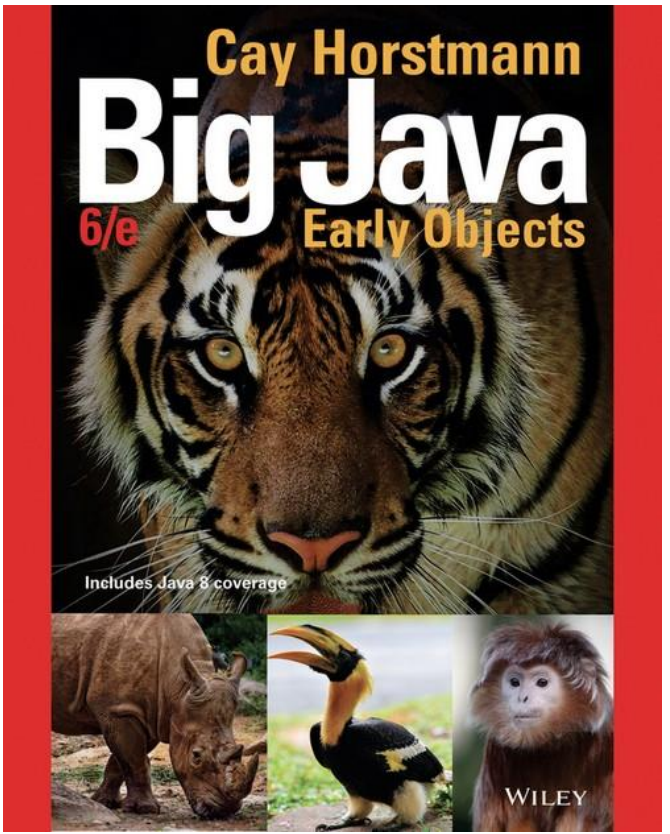
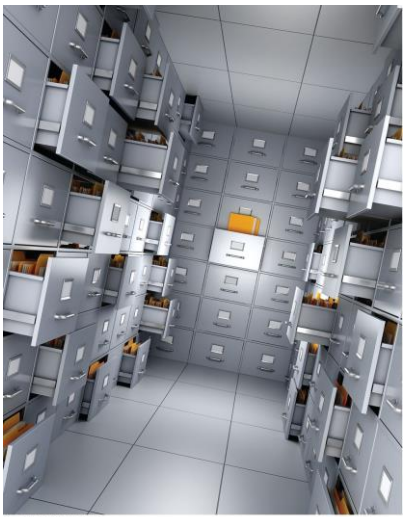
Chapter 19 – Stream Processing



Chapter Goals



- To be able to develop filter/map/reduce strategies for solving data processing problems
- To convert between collections and streams

- To use function objects for transformations and predicates
- To work with the `Optional` type
- To be able to express common algorithms as stream operations

Overview

- Streams process data by specifying actions.
- Library implements the actions using lazy execution.
- Can skip steps that aren't needed.
- Can split work over multiple processors.

The Stream Concept

- Algorithm for counting matches:

```
List<String> wordList = . . .; long count = 0;
for (String w : wordList)
{
   if (w.length() > 10) { count++; }
}
```

- With the Java 8 stream library:

```
Stream<String> words = . . .;
long count = words
   .filter(w -> w.length() > 10)
   .count();
```

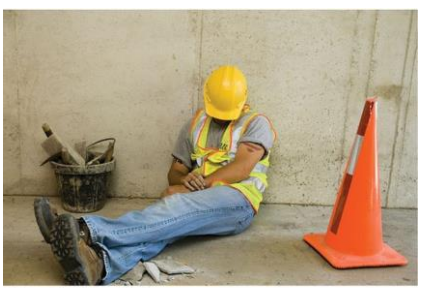
- You tell *what* you want to achieve (Keep the long strings, count them).
- You don't dwell on the *how* (visit each element in turn, if it is long, increment a variable).
- "What, not how" is powerful:
  - Operations can occur in parallel.
  - The data can be anywhere (e.g. distributed "big data").

Some Facts About Streams

- Streams are similar to collections, but...
- They don't store their own data.
  - The data comes from elsewhere.
  - From a collection, a file, a database, data from the internet, ...
- Streams were designed to work well with lambda expressions:

```
stream.filter(w -> w.length() > 10)
```

- Streams are immutable.
  - Methods such as `filter` produce new streams.
- Stream processing is *lazy*.



Lazy Processing

- Instead of counting the words, let's see some (but not all).
- Here is how to get the first five:

```
Stream<String> fiveLongWords = words
   .filter(w -> w.length() > 10)
   .limit(5);
```

- Bad approach: first generate all long words, and throw most of them away.
- Fortunately, stream processing is not bad but *lazy*.
  - Works “backwards” and only computes what is necessary.
- `limit(5)` needs an element...
- ... and `filter(....)` examines elements until it finds one.
- That repeats another four times.
- And then...nothing.
- The other stream elements never get examined.

section_1/StreamDemo.java

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Scanner;
6
7  public class StreamDemo
8  {
9     public static void main(String[] args) throws IOException
10    {
11       Scanner in = new Scanner(new File("../countries.txt"));
12       // This file contains one country name per line
13       List wordList = new ArrayList<>();
14       while (in.hasNextLine()) { wordList.add(in.nextLine()); }
15       // Now wordList is a list of country names
16
17       // Traditional loop for counting the long words
18       long count = 0;
19       for (String w : wordList)
20       {
21          if (w.length() > 10) { count++; }
22       }
23
24       System.out.println("Long words: " + count);
25
26       // The same computation with streams
27       count = wordList.stream()
28          .filter(w -> w.length() > 10)
29          .count();
30
31       System.out.println("Long words: " + count);
32    }
33
34 }
```

Program Run:

```
Long words: 63
Long words: 63
```

Self Check 19.1

Write a statement to count all overdrawn accounts in a `Stream<BankAccount>`.

Answer:

```
long count = stream
    .filter(b -> b.getBalance() < 0)
    .count();
```

Self Check 19.2

Given a stream of strings, how do you remove all empty strings?

Answer:

```
Stream<String> result = stream.filter(
    w -> w.length() > 0)
```

Self Check 19.3

How would you collect the first five strings of length greater than ten in a `List<String>` without using streams?

Answer:

```
List<String> result = new ArrayList<>();
int i = 0;
while (i < strings.size() && result.size() < 5)
{
    String s = strings.get(i);
    if (s.length() > 10) { result.add(s); }
}
```

Self Check 19.4

Given a stream of strings, how do you calculate how many have exactly ten characters?

Answer:

```
long result = stream.filter(
    w -> w.length() == 10).count();
```

Self Check 19.5

Given a stream of strings, how do you find the first one with length equal to ten?

Answer: As a stream, that is

```
Stream<String> result = stream.filter(
    w -> w.length() == 10).limit(1);
```

You will see in Section 19.3 how to get the answer as a string. And Section 19.6 will present an easier way to obtain this result.

Self Check 19.6

Given a stream of strings, how can you find out whether it has at least ten strings with three letters, *without counting them all* if there are more than ten?

Answer:

```
boolean atLeastTen = stream.filter(
    w -> w.length() == 3)
    .limit(10).count() == 10;
```

Because stream processing is lazy, the limit operation stops filtering as soon as ten matches have been found.

Producing Streams

- In order to process streams, you first need to have one.
- Simplest way: the `of` method:

```
Stream<String> words = Stream.of("Mary", "had", "a", "little", "lamb");
Stream<Integer> digits = Stream.of(3, 1, 4, 1, 5, 9);
```

- Also works for arrays:

```
Integer[] digitArray = { 3, 1, 4, 1, 5, 9 };
Stream<Integer> digitStream = Stream.of(digitArray);
```

  - This is a stream of `Integer` objects.
  - You'll see later how to make a stream of `int`.

- Any collection can be turned into a stream:

```
List<String> wordList = new ArrayList<>();
// Populate wordList
Stream<String> words = wordList.stream();
```

Producing Streams

- Several utility methods yield streams:

```
String filename = . . .;
try (Stream<String> lineStream = Files.lines(Paths.get(filename)))
{
   ...
} // File is closed here
```

- You can make infinite streams:

```
Stream<Integer> integers = Stream.iterate(0, n -> n + 1);
```

- You can turn any stream into a *parallel stream*.
  - Operations such as `filter` and `count` run in parallel, each processor working on chunks of the data.

```
Stream<String> parStream = lineStream.parallel();
```

Producing Streams

Table 1 Producing Streams

Example	Result
`Stream.of(1, 2, 3)`	A stream containing the given elements. You can also pass an array.
`Collection<String> coll = . . .; coll.stream()`	A stream containing the elements of a collection.
`Files.lines(path)`	A stream of the lines in the file with the given path. Use a `try`-with-resources statement to ensure that the underlying file is closed.
`Stream<String> stream = . . .; stream.parallel()`	Turns a stream into a parallel stream.
`Stream.generate(() -> 1)`	An infinite stream of ones (see Special Topic 19.1).
`Stream.iterate(0, n -> n + 1)`	An infinite stream of `Integer` values (see Special Topic 19.1).
`IntStream.range(0, 100)`	An `IntStream` of `int` values between 0 (inclusive) and 100 (exclusive)—see Section 19.8.
`Random generator = new Random(); generator.ints(0, 100)`	An infinite stream of random `int` values drawn from a random generator—see Section 19.8.
`"Hello".codePoints()`	An `IntStream` of code points of a string—see Section 19.8.

Self Check 19.7

Write a statement to create a stream of Color objects.

Answer: For example,

```
Stream<Color> colors = Stream.of(
    Color.RED, Color.WHITE, Color.BLUE);
```

Self Check 19.8

Given a list of `String` objects, use streams to count how many have length less than or equal to three.

Answer:

```
long count = list.stream()
    .filter(w -> w.length() <= 3).count();
```

Self Check 19.9

Repeat Self Check 8 with an array of strings.

Answer:

```
long count = Stream.of(array)
    .filter(w -> w.length() <= 3).count();
```

Self Check 19.10

Write a statement to count how many lines in the file input.txt have length greater than 80.

Answer:

```
try (Stream lines = Files.lines(
    Paths.get("input.txt")))
    {
        long count = lines.filter(
            l -> l.length() > 80).count();
        ...
    }
```

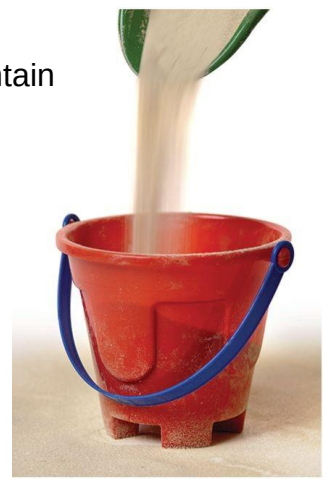
Collecting Results

- When you are done transforming a stream (e.g. with `filter`), want to harvest results.
- Some methods (e.g. `count`, `sum`) yield a single value.
- Other methods yield a collection.
- Here is how to collect into an array:

```
String[] result = stream.toArray(String[]::new);
```

Strange-looking expression `String[]::new` is a constructor reference (to the array constructor).

Replace `String` with another class if the stream doesn't contain strings.



Collecting Results

- To collect into a `List` or `Set`, use `collect`:

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

The argument to `collect` is a `Collector` object.

We'll always use one of the static method of `Collectors` to get one.

- A stream of string can be collected into a single string:

```
String result = words.collect(Collectors.joining(", "));
   // Stream elements separated with commas
```

Collecting Results

Table 2 Collecting Results from a `Stream<T>`

Example	Comments
`stream.toArray(T[]::new)`	Yields a `T[]` array.
`stream.collect(Collectors.toList())` `stream.collect(Collectors.toSet())`	Yields a `List<T>` or `Set<T>`.
`stream.collect(Collectors.joining(", ")`	Yields a string, joining the elements by the given separator. Only for `Stream<String>`.
`stream.collect(Collectors.groupingBy(` *keyFunction*, *collector*)	Yields a map that associates group keys with collected group values—see Section 19.9.

Self Check 19.11

Collect all strings of length greater than ten from a list of strings and store them in another list.

Answer:

```
List<String> result = list.stream()
    .filter(w -> w.length() > 10)
    .collect(Collectors.toList());
```

Self Check 19.12

Repeat Self Check 11, but collect the result in a set.

Answer:

```
Set<String> result = list.stream()
    .filter(w -> w.length() > 10)
    .collect(Collectors.toSet());
```

Self Check 19.13

Find the first string of length greater than ten in a list of strings. Use `filter` and `limit`, then convert the stream to a list and retrieve the result. Assume that there is at least one such string.

Answer:

```
String result = list.stream()
    .filter(w -> w.length() > 10)
    .limit(1)
    .collect(Collectors.toList())
    .get(0);
```

Self Check 19.14

Repeat Self Check 13, but use `toArray`.

Answer:

```
String result = list.stream()
    .filter(w -> w.length() > 10)
    .limit(1)
    .toArray(String[]::new)[0];
```

Self Check 19.15

The solutions to Self Check 13 and Self Check 14 would work even if you omitted the call to `limit`. Why would that not be a good idea?

Answer: If you omitted the call to `limit(1)`, all strings of length greater than 10 would be collected and converted to a list or array. With the call to `limit`, collecting stops as soon as the first match has been found.

Tip: One Stream Operation Per Line

- It's best to put one stream operation per line:

```
List<String> result = list.stream() // Create the stream
   .filter(w -> w.length() > 10) // Keep long strings
   .limit(50) // Keep only the first fifty.
   .collect(Collectors.toList()); // Turn into a list
```

- If you cram as much as possible into one line, it is tedious to figure out the steps:

```
List<String> result = list.stream().filter(w -> w.length() > 10).limit(50)
.collect(Collectors.toList()); // Don’t use this formatting style
```

Stream Tranformations

- Life cycle of a stream:

  Create stream.

  Transform stream (possibly multiple times).

  Collect results.

Stream Tranformations - map

- `map` transforms stream by applying function to each element.
- Turn all words into lowercase:

```
Stream<String> words = Stream.of("A", "Tale", "of", "Two", "Cities");
Stream<String> lowerCaseWords = words.map(w -> w.toLowerCase());
   // "a", "tale", "of", "two", "cities"
```

- Remove vowels from all words:

```
Stream<String> consonantsOnly = lowerCaseWords.map(
   w -> w.replaceAll("[aeiou]", ""));
   // "", "tl", "f", "tw", "cts"
```

- Get the length of each element:

```
Stream<Integer> consonantCount = consonantsOnly.map(w -> w.length());
    // 0, 2, 1, 2, 3
```

Stream Transformations -Table of Examples

Table 3 Stream Transformations

Example	Comments
`stream.filter(`*condition*`)`	A stream with the elements matching the condition.
`stream.map(`*function*`)`	A stream with the results of applying the function to each element.

`stream.mapToInt(`*function*`)`
`stream.mapToDouble(`*function*`)`
`stream.mapToLong(`*function*`)`
	A primitive-type stream with the results of applying a function with a return value of a primitive type—see Section 19.8.

`stream.limit(n)` `stream.skip(n)`	A stream consisting of the first `n`, or all but the first `n` elements.

`stream.distinct()`
`stream.sorted()`
`stream.sorted(`*comparator*`)`
	A stream of the distinct or sorted elements from the original stream.

More Stream Transformations

- Applying `map` yields a stream with the same number of elements.
- `filter` only retains matching elements:

```
Stream<String> aWords = words.filter(w -> w.substring(0, 1).equals("a"));
   // Only the words starting with "a"
```

- `limit` takes the first `n`:

```
Stream<String> first100aWords = aWords.limit(100);
```

- `skip` takes all but the first `n`:

```
Stream<String> allButFirst100aWords = aWords.skip(100);
```

- `distinct` yields a stream with duplicates removed:

```
Stream<String> words = Stream.of(
   "how much wood could a wood chuck chuck".split(" "));
Stream<String> distinctWords = words.distinct();
   // "how", "much", "wood", "could", "a", "chuck"
```

- `sorted` yields a new stream in which the elements are

sorted:

```
Stream<String> sortedWords = distinctWords.sorted();
   // "a", "chuck", "could", "how", "much", "wood"
```

Element type must be `Comparable`

Or supply a comparator: `distinctWords.sorted((s, t) -> s.length() - t.length())`

section_4/StreamDemo.java

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4  import java.util.List;
5  import java.util.stream.Collectors;
6  import java.util.stream.Stream;
7
8  public class StreamDemo
9  {
10    public static void main(String[] args) throws IOException
11    {
12       try (Stream lines = Files.lines(Paths.get("../countries.txt")))
13       { // Read the lines
14          List result = lines
15             .filter(w -> w.length() > 10) // Keep only long words
16             .map(w -> w.substring(0, 7)) // Truncate to seven characters
17             .map(w -> w + "...") // Add ellipses
18             .distinct() // Remove duplicates
19             .limit(20) // Keep only the first twenty
20             .collect(Collectors.toList()); // Collect into a list
21          System.out.println(result);
22       }
23    }
24 }
```

Program Run:

```
[Afghani..., America..., Antigua..., Bahamas..., Bosnia ...,
British..., Burkina..., Cayman ..., Central..., Christm...,
Cocos (..., Congo, ..., Cook Is..., Cote d'..., Czech R...,
Dominic..., El Salv..., Equator..., Falklan..., Faroe I...]
```

Self Check 19.16

Given a stream of words, get a stream of all that start with a or A, converted to lowercase. Provide two solutions, one applying filter before map and one after.

Answer:

```
Stream<String> result = words
    .filter(w -> w.substring(0, 1)
    .equalsIgnoreCase("a"))
    .map(w -> w.toLowerCase());
```

or

```
Stream<String> result = words
    .map(w -> w.toLowerCase())
    .filter(w -> w.substring(0, 1)
    .equals("a"));
```

Self Check 19.17

Given a stream of words, produce a stream of Integer values containing the lengths of the words.

Answer:

```
Stream<Integer> result = words
    .map(w -> w.length());
```

Self Check 19.18

Given a list of strings, get a list of the first ten in sorted order.

Answer:

```
List<String> result = list.stream()
    .sorted()
    .limit(10)
    .collect(Collectors.toList());
```

Self Check 19.19

Given a list of words, how do you find how many distinct words there are of length equal to three?

Answer:

```
int result = list.stream()
    .filter(w -> w.length() == 3)
    .distinct()
    .count();
```

Self Check 19.20

How can you solve Self Check 19 without streams?

Answer: Put all words of length 3 into a set and get its size:

```
Set<String> words = new HashSet<>();
for (String w : list)
{
    if (w.length() == 3)
    {
        words.add(w);
    }
}
int result = words.size();
```

Common Error: Don't Use a Terminated Stream

- Once you apply a terminal operation, a stream is “used up”.
- Can no longer apply any stream operations.
- Easy mistake if you have a reference to an intermediate stream:

```
Stream<String> stream = list.stream();
List<String> result1 =  stream.limit(50).collect(Collectors.toList());
   // Save the first fifty
stream = stream.skip(50);
   // Error—the stream can no longer be used
```

- To avoid the error, use “pipeline notation”:

```
result = create(...)
   .transform1(...)
   .transform2(...)
   .terminalOperation(...)
```

- If you want to do what the example tried to do, you need two streams:

```
List<String> result1 = list.stream()
   .limit(50)
   .collect(Collectors.toList()); // This stream can no longer be used
List<String> result2 = list.stream() // Create another stream
   .skip(50)
   .limit(50)
   .collect(Collectors.toList());
```

Lambda Expressions

- Have seen lambda expressions in `filter` and `map` methods such as:

```
w -> w.length() > 10
```

- Like a static function.
- Left side of `->` operator is a parameter variable.
- Right side is code to operate on the parameter and compute a result.
- When used with a type like `Stream<String>`, compiler can determine type.
- Otherwise, you can specify the type, like:

```
(String w) -> w.length() > 10
```

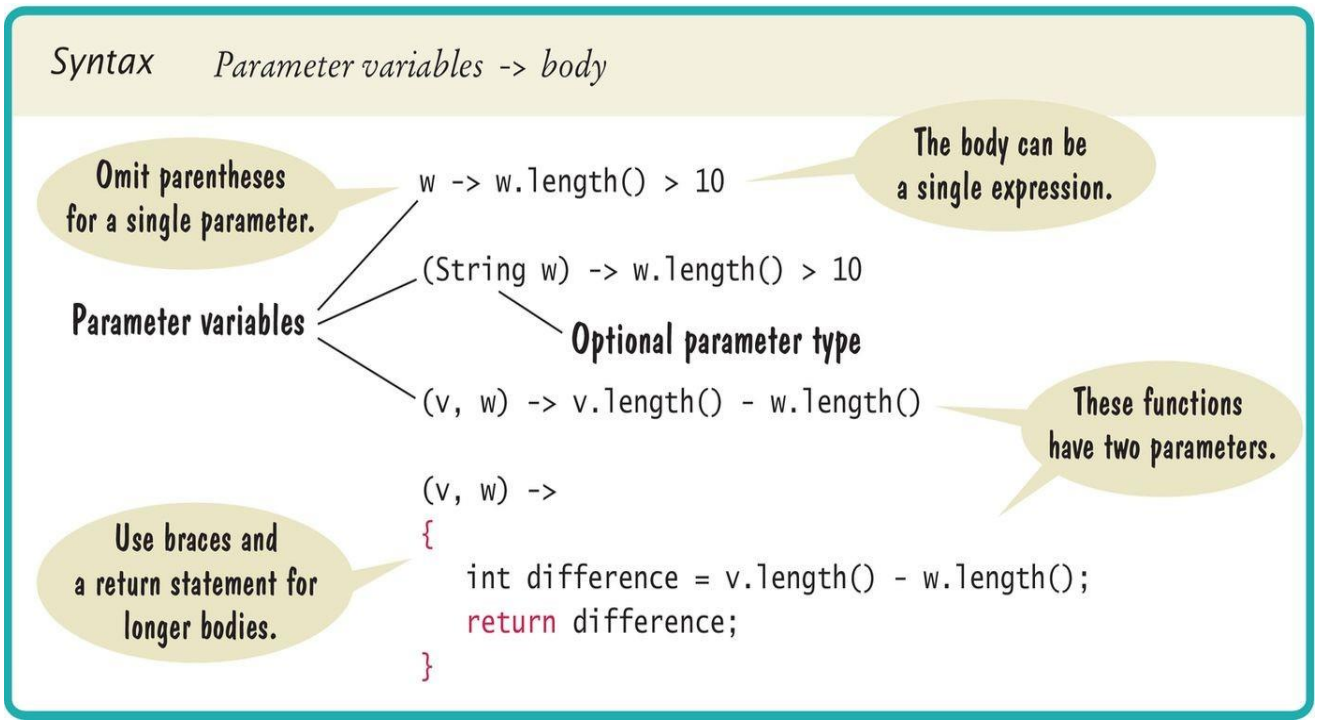
- Multiple parameters are enclosed in parentheses:

```
(v, w) -> v.length() - w.length()
```

- This expression can be used with the `sorted` method in the `Stream` class to sort strings by length:

```
Stream<String> sortedWords = distinctWords.sorted( (v, w) -> v.length() -  w.length());
    // "a", "how", "much", "wood", "could", "chuck"
```

Syntax 19.1 Lambda Expressions



Self Check 19.21

Write a lambda expression for a function that computes the average of two numbers.

Answer:

```
(x, y) -> (x + y) / 2.0
```

Self Check 19.22

Write a lambda expression that tests whether a word starts and ends with the same letter.

Answer:

```
w -> w.substring(0, 1).equals(
    w.substring(w.length() - 1))
```

Self Check 19.23

What does this lambda expression do?

```
s -> s.equals(s.toUpperCase())
```

Answer: It is a predicate that tests whether a string is in uppercase.

Self Check 19.24

Assuming that `words` is a `Stream<String>`, what is the result of this call?

```
words.filter(s -> s.equals(s.toUpperCase()))
```

Answer: It is a stream consisting of all words that are entirely in uppercase.

Self Check 19.25

Assuming that `words` is a `Stream<String>`, what is the result of this call?

```
words.map(s -> s.equals(s.toUpperCase()))
```

Answer: It is a `Stream<Boolean>` with values `Boolean.TRUE` and `Boolean.FALSE` (the wrappers for `true` and `false`), depending on whether the elements of words were entirely in uppercase or not.

Method Expressions

- Common to have lambda expressions that just invoke a method.
- Use *method expression*: *ClassName*`::`*methodName*:

```
String::toUpperCase
```

- Parameters are added “at the right places”:

```
(String w) -> w.toUpperCase()
```

- If method has a parameter, the lambda expression gets two parameters:

```
String::compareTo
```

  is the same as:

```
(String s, String t) -> s.compareTo(t)
```

- Also works with static methods:

```
Double::compare
```

is the same as:

```
(double x, double y) -> Double.compare(x, y)
```

- Can have an object to the left of `::` symbol:

```
System.out::println
```

is the same as:

```
x -> System.out.println(x)
```

Constructor Expressions

- Like method expression, with special method name `new`.
- For example,

```
BankAccount::new
```

  is equivalent to a lambda expression that invokes the `BankAccount` constructor.

- Which constructor?
- Depends on context—could be:

```
() -> new BankAccount()
```

  or:

```
b -> new BankAccount(b)
```

- Constructor expressions can construct arrays:

```
String[]::new
```

- Same as:

```
(n: int) -> new String[n]
```

- Used to overcome limitation of Java generics—can't construct array of generic type.

Higher-Order Functions

- A “function” that consumes and/or produces “functions”.
- Example: `filter` consumes function:

```
public static <T> List<T> filter(List<T> values, Predicate<T> p)
{
   List<T> result = new ArrayList<>();
   for (T value : values)
   {
      if (p.test(value)) { result.add(value); }
   }
   return result;
}
```

- Here, `Predicate` is a standard functional interface:

```
public interface Predicate<T>
{
   boolean test(T arg);
}
```

- Typical use:

```
List<String> longWords = filter(wordList, w -> w.length() > 10);
```

Higher-Order Functions

- Suppose we want to find all strings that contain the word `"and"`:

```
List<String> andWords = filter(wordList, w -> w.indexOf("and") >= 0);
```

- What if we want to find another word?
- Can write a method that yields the predicate for an arbitrary target:

```
public static Predicate<String> contains(String target)
{
   return s -> s.indexOf(target) >= 0;
}
```

- Pass the result to a method expecting a `Predicate`:

```
List<String> andWords = filter(wordList, contains("and"));
```

- `contains` is also a higher-order function.

Method Expressions and Comparators

- `Comparator.comparing` makes a comparator from an extractor function:

```
Comparator<String> comp = Comparator.comparing(t -> t.length())
```

- Same as:

```
Comparator<String> comp = (v, w) -> v.length() - w.length();
```

- Note that the extractor function makes a single method call.
- Write as method reference:

```
Comparator.comparing(String::length)
```

- Reads nicely: the comparator that compares strings by their length.
- Can add a secondary comparison with `thenComparing`:

```
Collections.sort(countries,
   Comparator.comparing(Country::getContinent
   )
      .thenComparing(Country::getName));
```

Countries are compared first by continent.

If the continents are the same, they are compared by name.

- Easy to read, easy to write.

  Thanks to lambda expressions, method expressions, higher order functions.

The `Optional` Type

- In Java, common to use `null` to denote absence of result.
- Drawback: `NullPointerException` when programmer forgets to check.

```
String result = oldFashionedMethod(searchParameters);
   // Returns null if no match
int length = result.length();
   // Throws a NullPointerException when result is null
```

- Stream library uses `Optional` type when a result may not be present.
- Example: First string of length > 10:

```
words.filter(w -> w.length() > 10).findFirst()
```

- What if there is none? An `Optional<String>` either contains a string or an indication that no value is present.

```
Optional<String> optResult = words
   .filter(w -> w.length() > 10)
   .findFirst();
```

Optional Values

Table 4 Working with `Optional` Values

Example	Comments
`result = optional.orElse("");`	Extracts the wrapped value or the specified default if no value is present.
`optional.ifPresent(v -> Process v);`	Processes the wrapped value if present or does nothing if no value is present.

`if (optional.isPresent())`
`{`
`  Process optional.get()`
`}`
`else`
`{`
`  Handle the absence of a value.`
`}`
	Processes the wrapped value if present, or deals with the situation when it is not present.

`double average = pstream.average()` `  .getAsDouble();`	Gets the wrapped value from a primitive-type stream—see Section 19.8.

`if (there is a result)`
`{`
`  return Optional.of(result);`
`}`
`else`
`{`
`  return Optional.empty();`
`}`
	Returns an `Optional` value from a method.

Working with `Optional`

- Work with it, not against it. (That is, don't treat it like a potentially `null` reference.)
- `orElse` extracts the value or an alternative if there is none:

```
int length = optResult.orElse("").length();
```

- `ifPresent` passes on the value to a function; does nothing if there is none:

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

- If neither of these works, use `isPresent` to test if there is a value and `get` to get it:

```
if (optResult.isPresent())
{
   System.out.println(optResult.get());
}
else
{
   System.out.println("No element");
}
```

Returning `Optional`

- Declare return type as `Optional<T>`.
- If there is a result, return `Optional.of(result)`.
- Otherwise return `Optional.empty()`.

```
public static Optional<Double> squareRoot(double x)
{
   if (x >= 0) { return Optional.of(Math.sqrt(x)); }
   else { return Optional.empty(); }
}
```

Self Check 19.26

Set `word` to the first word in the list `wordList` containing the letter a, or to the empty string if there is no match.

Answer:

```
String word = wordList.stream()
    .findFirst(w -> w.contains("a"))
    .orElse("");
```

Self Check 19.27

Repeat Self Check 26 using `ifPresent`.

Answer:

```
String word = "";
wordList.stream()
    .findFirst(w -> w.contains("a"))
    .ifPresent(v -> { word = v; });
```

Note that the previous solution was better because it did not involve any “side effect”.

Self Check 19.28

Repeat Self Check 26 using `isPresent`.

Answer:

```
Optional<String> optResult = wordList.stream()
    .findFirst(w -> w.contains("a"));
    String word = "";
    if (optResult.isPresent())
    {
        word = optResult.get();
    }
```

Self Check 19.29

Set `word` to the tenth word in the list `wordList` containing the letter a, or to the empty string if there is no such string. Don't use `collect`.

Answer:

```
String word = wordList.stream()
    .filter(w -> w.contains("a"))
    .skip(9)
    .findFirst()
    .orElse("");
```

Self Check 19.30

Write a method `reciprocal` that receives a parameter x of type `double` and returns an `Optional<Double>` containing 1 / x if x is not zero.

Answer:

```
public static Optional<Double> reciprocal(double x)
{
    if (x == 0) { return Optional.empty(); }
    else { return Optional.of(1 / x); }
}
```

Other Terminal Operations

- `findAny` is like `findFirst`, but is faster on parallel streams.

```
result = words
   .parallel()
   .filter(w -> w.length() > 10)
   .filter(w -> w.endsWith("y"))
   .findAny()
   .orElse("");
```

- `max`/`min` require a comparator and return an `Optional` (since the input may be empty):

```
Optional<String> result = words.max((v, w) -> v.length() - w.length());
```

- `allMatch`/`anyMatch`/`noneMatch` check a predicate:

```
boolean result = words.allMatch(w -> w.contains("e"));
// result is true if all words contain the letter e
```

Self Check 19.31

Rewrite the example for the `findAny` operation at the beginning of this section so that the filter method is only called once.

Answer:

```
result = words.parallel()
    .filter(w -> w.length() > 10
        && w.endsWith("y"))
    .findAny()
    .orElse("");
```

Self Check 19.32

How can you check whether any words start with the letter q and end with the letter y without calling `findAny`?

Answer:

```
boolean result = words.anyMatch(
    w.startsWith("q") && w.endsWith("y"));
```

Self Check 19.33

What is wrong with the following code?

```
Stream<String> qys = wordList.stream()
    .filter(w -> w.startsWith("q"))
    .filter(w -> w.endsWith("y"));
    if (qys.count() > 0)
    {
        System.out.println(qys.findAny().get());
    }
```

Answer: Once you invoke the terminal operation `qys.count()`, you can no longer invoke any operations on the stream.

Self Check 19.34

How can you get two words starting with q and ending with y?

Answer: You can't call `findAny` twice, so you should use `limit` to limit the stream to two results and then `collect` it to an array or list.

```
List<String> result = words
    .filter(w -> w.startsWith("q")
        && w.endsWith("y"))
    .limit(2)
    .collect(Collectors.toList());
```

Self Check 19.35

An operation *short circuits* if it stops looking at inputs that can no longer change the result. For example, the Boolean operator && short circuits when the first operand is `false`. Which of `allMatch`, `anyMatch`, and `noneMatch` can short circuit?

Answer: They all can short circuit: `allMatch` returns `false` as soon as it finds an element that doesn't match, and `anyMatch` and `noneMatch` return as soon as they find an element that matches, with return values `true` and `false` respectively.

Primitive-Type Streams

- Inefficient to have streams of number wrappers such as `Stream<Integer>`.
- For example, `numbers.map(x -> x * x)` requires unboxing and boxing for each element.
- `IntStream`, `DoubleStream`, `DoubleStream` work with `int`, `long`, `double` values without boxing.
- No stream classes for `byte`, `short`, `long`, `char`, or `float`.

Creating Primitive-Type Streams

- Can create from individual numbers, or an array:

```
IntStream stream = IntStream.of(3, 1, 4, 1, 5, 9);
int[] values = . . .;
stream = IntStream.of(values);
```

- `range` yields a contiguous range of integers:

```
IntStream stream = IntStream.range(a, b);
   // Stream contains a, a + 1, a + 2, ..., b -1
```

- Random generator yields infinite stream of random numbers:

```
Random generator = new Random();
IntStream dieTosses = generator.ints(1, 7);
```

- String yields stream of Unicode code points:

```
IntStream codePoints = str.codePoints();
```

Aside: This is the best way of getting the code points of a string in the Java API!

Much better than `charAt` which only yields 16-bit code units of the variable-length UTF-16 encoding.

Mapping Primitive-Type Streams

- `IntStream.map` with an `int -> int` function yields another `IntStream`.

```
IntStream stream = IntStream.range(0, 20)
   .map(n -> Math.min(n, 10));
   // A stream with twenty elements 0, 1, 2, ..., 9, 10, 10, ..., 10
```

- When the function yields objects, use `mapToObj`:

```
String river = "Mississippi";
int n = river.length();
Stream<String> prefixes = IntStream.range(0, n)
   .mapToObj(i -> river.substring(0, i));
   // "", "M", "Mi", "Mis", "Miss", "Missi", ...
```

- Also have `mapToDouble`, `mapToLong` if the function yields `double`, and `long` values.
- Think of `IntStream.range(a, b).mapXXX` as an equivalent of a `for` loop that yields a value in each iteration.
- Use `mapToInt`/`mapToLong`/`mapToDouble` with streams of objects when the map function yields primitive type values.
  - Use `boxed` to turn into a stream of objects.

Processing Primitive-Type Streams

- Stream methods for primitive-type streams have modified parameters/return types.
- For example, `IntStream.toArray` returns an `int[]` and doesn't require a constructor.
- Four additional methods `sum`, `average`, `max`, and `min` (without comparators).
- Last three return `OptionalInt`/`OptionalLong`/`OptionalDouble`:

```
double average = words
   .mapToInt(w -> w.length())
   .average()
   .orElse(0);
```

Computing Results

Table 5 Computing Results from a Stream<T>

Example	Comments
`stream.count()`	Yields the number of elements as a `long` value.
`stream.findFirst()` `stream.findAny()`	Yields the first, or an arbitrary element as an `Optional<T>`—see Section 19.6.
`stream.max(`*comparator*`)` `stream.min(`*comparator*`)`	Yields the largest or smallest element as an `Optional<T>`—see Section 19.7.

`pstream.sum()`
`pstream.average()`
`pstream.max()`
`pstream.min()`
	The sum, average, maximum, or minimum of a primitive-type stream—see Section 19.8.

`stream.allMatch(`*condition*`)`
`stream.anyMatch(`*condition*`)`
`stream.noneMatch(`*condition*`)`
	Yields a `boolean` variable indicating whether all, any, or no elements match the condition—see Section 19.7.

`stream.forEach(`*action*`)`	Carries out the action on all stream elements—see Section 19.7.

Self Check 19.36

Given a list of `BankAccount` objects, use streams to find the sum of all balances.

Answer:

```
double sum = accounts.stream()
    .mapToDouble(a -> a.getBalance())
    .sum();
```

Self Check 19.37

Given a list of `BankAccount` objects, use streams to find the average balance.

Answer:

```
double average = accounts.stream()
    .mapToDouble(a -> a.getBalance())
    .average()
    .orElse(0);
```

Self Check 19.38

Given a list of words, find the length of the longest one.

Answer:

```
int longestLength = wordList.stream()
    .mapToInt(w -> w.length())
    .max()
    .orElse(0);
```

Self Check 19.39

Given a list of words, find the length of the shortest word starting with the letter z.

Answer:

```
int longestLength = wordList.stream()
    .filter(w -> w.startsWith("z"))
    .mapToInt(w -> w.length())
    .min()
    .orElse(0);
```

Grouping Results

- So far, results were either a value or a collection.
- Sometimes, want to split result into groups.
- Example: Group all words with the same first letter together.
- Use:

```
stream.collect(Collectors.groupingBy(function))
```

The function produces a key for each element.

The result is a a map.

Map values are collections of elements with the same key.

```
Map<String, List<String>> groups = Stream.of(words)
   .collect(Collectors.groupingBy(
   w -> w.substring(0, 1))); // The function for extracting the keys
```

- `groups.get("a")` is a list of all words starting with a.

Processing Groups

- Nice to split result into groups.
- Even nicer: Can process each group.
- Pass a collector to `Collectors.groupingBy`.
- Example: Group into sets, not lists:

```
Map<String, Set<String>> groupOfSets = Stream.of(words)
   .collect(Collectors.groupingBy(
      w -> w.substring(0, 1), // The function for extracting the keys
      Collectors.toSet())); // The group collector
```

- The `groupingBy` collector collects *the stream* into groups.
- The `toSet` collector collects *each group* int a set.

section_9/GroupDemo.java

```
1  import java.util.List;
2  import java.util.Map;
3  import java.util.Optional;
4  import java.util.Set;
5  import java.util.stream.Stream;
6  import java.util.stream.Collectors;
7
8  public class GroupDemo
9  {
10    public static void main(String[] args)
11    {
12       String[] words = ("how much wood would a woodchuck chuck "
13           + "if a woodchuck could chuck wood").split(" ");
14
15       Map<String, List<String>> groups = Stream.of(words)
16          .collect(Collectors.groupingBy(
17             w -> w.substring(0, 1)));
18       System.out.println("Lists by first letter: " + groups);
19
20       Map<String, Set<String>> groupOfSets = Stream.of(words)
21          .collect(Collectors.groupingBy(
22             w -> w.substring(0, 1), // The function for extracting the keys
23             Collectors.toSet())); // The group collector
24       System.out.println("Sets by first letter: "
25          + groupOfSets);
26
27       Map<String, Long> groupCounts = Stream.of(words)
28          .collect(Collectors.groupingBy(
29             w -> w.substring(0, 1),
30             Collectors.counting()));
31       System.out.println("Counts by first letter: "
32           groupCounts);
33
34       Map<String, Optional<String>> groupLongest = Stream.of(words)
          .collect(
```

Program Run:

```
Lists by first letter: {a=[a, a], c=[chuck, could, chuck],
w=[wood, would, woodchuck, woodchuck, wood], h=[how], i=[if], m=[much]}
Sets by first letter: {a=[a], c=[could, chuck], w=[would, woodchuck, wood],
h=[how], i=[if], m=[much]}
Counts by first letter: {a=2, c=3, w=5, h=1, i=1, m=1}
Longest word by first letter: {a=Optional[a], c=Optional[chuck],
w=Optional[woodchuck], h=Optional[how], i=Optional[if], m=Optional[much]}
```

Self Check 19.40

Suppose words contains the strings "Mary", "had", "a", "little", "lamb". What are the contents of groups in the first example of this section?

Answer: It is a map:

```
{ "M" -> ["Mary"], "h" -> ["had"], "a" ->
["a"], "l" -> ["little", "lamb"] }.
```

Self Check 19.41

With the same contents for `words`, what are the contents of `groupCounts` in the third example?

Answer: It is a map:

```
{ "M" -> 1, "h" -> 1, "a" -> 1, "l" -> 2 }.
```

Self Check 19.42

Given a list of strings, make a map with keys 1, 2, 3, ..., so that the value for the key n is a list of all words of length n.

Answer:

```
Map<Integer, List<String>> groups =
    wordList.stream()
    .collect(Collectors.groupingBy(
        w -> w.length));
```

Self Check 19.43

Associate with each letter the average length of words in words that start with that letter.

Answer:

```
Map<String, Double> averages = Stream.of(words)
    .collect(Collectors.groupingBy(
        w -> w.substring(0, 1),
        Collectors.averagingInt(
            w -> w.length())));
```

Self Check 19.44

Associate with each letter a string containing all words in words starting with that letter, separated by commas.

Answer:

```
Map<String, String> wordsStartingWith =
    Stream.of(words)
        .collect(Collectors.groupingBy(
        w -> w.substring(0, 1),
        Collectors.joining(", ")));
```

Collecting Counts and Sums

- Use `Collectors.counting()` to count the group values:

```
Map<String, Long> groupCounts = Stream.of(words)
   .collect(Collectors.groupingBy(
      w -> w.substring(0, 1),
      Collectors.counting()));
```

- `groupCounts.get("a")` is the number of words that start with an a.
- To sum up some aspect of group values, use `summingInt`, `summingDouble`, and `summingLong`:

```
Map<String, Long> groupSum = countries.collect(
   Collectors.groupingBy(
      c -> c.getContinent(), // The function for extracting the keys
      Collectors.summingLong(
         c -> c.getPopulation()))); // The function for getting the summands
```

- `groupSum.get("Asia")` is the total population of Asian countries.

Collecting Average, Maximum, Minimum

- The `Collectors` methods `averagingInt`, `averagingDouble`, and `averagingLong` work just like `summingXxx`.
- Return 0 for empty groups (not an `Option`).
- Average word length grouped by starting character:

```
Map<String, Double> groupAverages = Stream.of(words)
   .collect(Collectors.groupingBy(
      w -> w.substring(0, 1),
      Collectors.averagingInt(String::length)));
```

- `maxBy`, `minBy` use a comparison function and return `Optional` results:

```
Map<String, Optional<String>> groupLongest = Stream.of(words)
   .collect(
      Collectors.groupingBy(
         w -> w.substring(0, 1), // The function for extracting the keys
         Collectors.maxBy(
            (v, w) -> v.length() - w.length()))); // The comparator function
```

Parallel Streams

- Use `parallelStream` on a collection:

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

- Or use `parallel` on any stream:

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

- When the terminal method executes, operations are parallelized.
- Intent: Same result as when run sequentially.
- Just faster because the work is distributed over available processors.
- Example:

```
long result = wordStream.parallel()
   .filter(w -> w.length() > 10)
   .count();
```

- The underlying data is partitioned in *n* regions.
  - The filtering and counting executes concurrently.
  - When all counts are ready, they are combined.

Effective Parallelization

- Streams from arrays and lists are *ordered*. Results are predictable, even on parallel streams.
- Use `findAny` instead of `findFirst` if you don't care about ordering.
- Call `unordered` to speed up `limit` or `distinct`:

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

- Use `groupingByConcurrent` to speed up grouping if you don't care about the order in which the values are processed:

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

Self Check 19.45

How do you compute the sum of all positive values in an array of integers?

Answer:

```
int sum = IntStream.of(values)
    .filter(n -> n > 0)
    .sum();
```

Self Check 19.46

How do you find the position of the *last* space in a string, using streams?

Answer: There are two possible approaches. You can collect all matches and then pick the last one.

```
int[] positions = IntStream
    .range(0, str.length)
    .filter(i -> str.charAt(i) == ' ')
    .toArray();
int lastPos = -1;
if (positions.length > 0) { lastPos =
    positions[positions.length - 1]; }
```

Or you can move backward through the string:

```
int n = str.length();
int lastPos = IntStream.range(0, n)
    .filter(i -> str.charAt(n - 1 - i) == ' ')
    .findFirst()
    .orElse(-1);
```

Self Check 19.47

How do you get the smallest area of any country from a list of `Country` objects, assuming that the `Country` class has a method `public double getArea()`? How do you get the country with that area?

Answer:

```
double smallest = countries.stream()
    .mapToDouble(c -> c.getArea())
    .min();
```

To get the country, you could now search:

```
Optional<Country> smallestCountry
    = countries.stream()
    .findAny(c -> c.getArea() == smallest);
```

But it is more efficient to search for the country with the minimal area. Then you need to specify a comparator.

```
Optional<Country> smallestCountry
    = countries.stream()
    .min((c, d) -> Double.compare(
        c.getArea(), d.getArea()));
```

Or, using Special Topic 19.4:

```
Optional<Country> smallestCountry
    = countries.stream()
    .min(Comparator.comparing(
        c -> c.getArea()));
```

Self Check 19.48

Someone proposes the following way to find the smallest element in a stream:

```
smallest = stream.sorted().limit(1).findAny().get();
```

Will it work? If so, is it a good idea?

Answer: Yes, it will work, provided there is at least one element in the stream. But it's not a good idea because sorting is much less efficient than computing the minimum.

Self Check 19.49

Why can't one use the `distinct` method to solve the problem of removing adjacent duplicates from a stream?

Answer: The `distinct` method removes all duplicates, not just adjacent ones.