

Lambdas and Streams

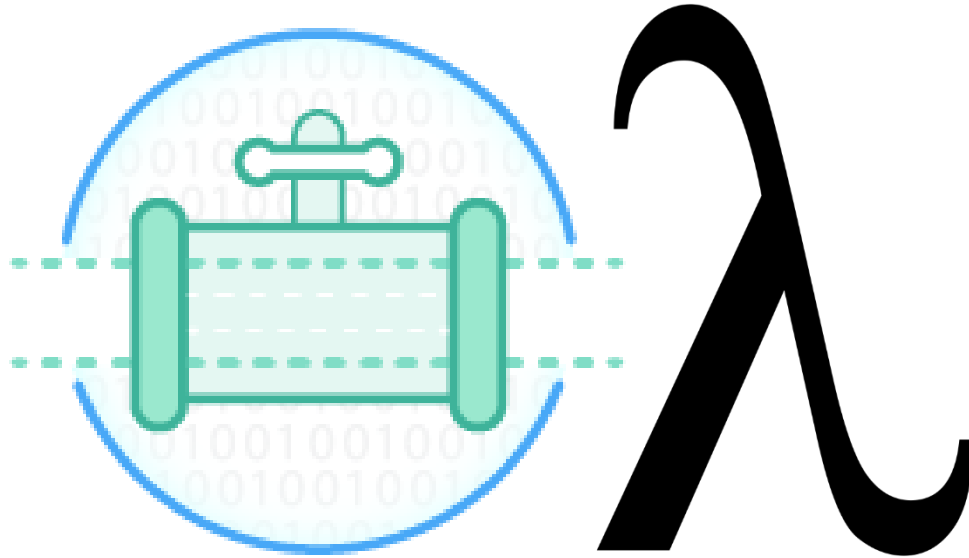


Table of Contents

- 1. Introduction to Lambdas and Streams**
 - What is a Lambda?
 - What is a Stream?
- 2. Stream Operations**

Introduction to Lambdas and Streams

What is a Lambda?

What is a Stream?

The most important principle in Programming?

KEEP IT SHORT & SIMPLE

KISS

Declarative programming using Lambdas
allows achieving KISS

Imperative vs. Declarative

Imperative Programming

- You tell the computer **how** to perform a task.

Declarative Programming


- You tell the computer **what you want**, and you let the computer (i.e. the compiler or runtime) figure out for itself the best way to do it.
- Also known as **Functional Programming**



What is a Lambda?

- Lambda is very similar to *simple methods*. It has:
 - A return type
 - Parameters
 - A body
- They don't have a name (anonymous method)
- They have no associated object
- They can be passed as parameters:
 - As *code* to be executed by the receiving method
- Syntax defined (introduced in Java 8) as:

Parameters  Body

 Arrow token

Lambda Expressions

- Lambda expressions can be used with `filter` and `map` methods :

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

- 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.
- 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 Lambda Expressions

Syntax *Parameter variables -> body*

Omit parentheses
for a single parameter.

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

The body can be
a single expression.

Parameter variables

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

Optional parameter type

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

These functions
have two parameters.

Use braces and
a return statement for
longer bodies.

`(v, w) ->`

`{`

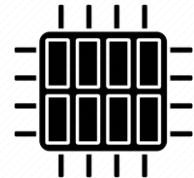
`int difference = v.length() - w.length();`

`return difference;`

`}`



What is a Stream?

- Stream API is used to process collections of objects.
- A stream is a **sequence** of objects that supports various methods which can be *pipelined* to produce the desired result.
- They don't store their own data
 - The data comes from elsewhere such as List, Arrays or I/O channels (disk or network)
- Can split work over multiple processors. 
- Streams were designed to work well with lambda expressions:

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

- Any collection can be turned into a stream:

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



© microgen/iStockphoto.

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 program the *how* (visit each element in turn, if it is long, increment a variable).
- "What, not how" is powerful:
 - Operations can be executed in parallel.



© pullia/iStockphoto.

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 }
```

Producing Streams

- Several utility methods yield streams:

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

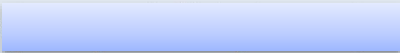
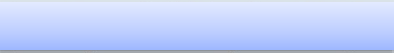
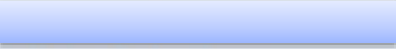

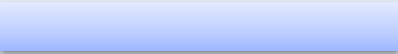
- You can make infinite streams:

```
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

Example	Result
<code>Stream.of(1, 2, 3)</code>	A stream containing the given elements. You can also pass an array.
<code>Collection<String> coll = . . . ; coll.stream()</code>	A stream containing the elements of a collection.
<code>Files.lines(<i>path</i>)</code>	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.
<code>Stream<String> stream = . . . ; stream.parallel()</code>	Turns a stream into a parallel stream.
<code>Stream.generate(() -> 1)</code>	An infinite stream of ones 
<code>Stream.iterate(0, n -> n + 1)</code>	An infinite stream of Integer values 
<code>IntStream.range(0, 100)</code>	An IntStream of int values between 0 (inclusive) and 100 (exclusive) - 
<code>Random generator = new Random(); generator.ints(0, 100)</code>	An infinite stream of random int values drawn from a random generator 
<code>"Hello".codePoints()</code>	An IntStream of code points of a string 

Stream Operations

Filter, Map, Reduce, and others

Filter

Keep elements that satisfy a condition

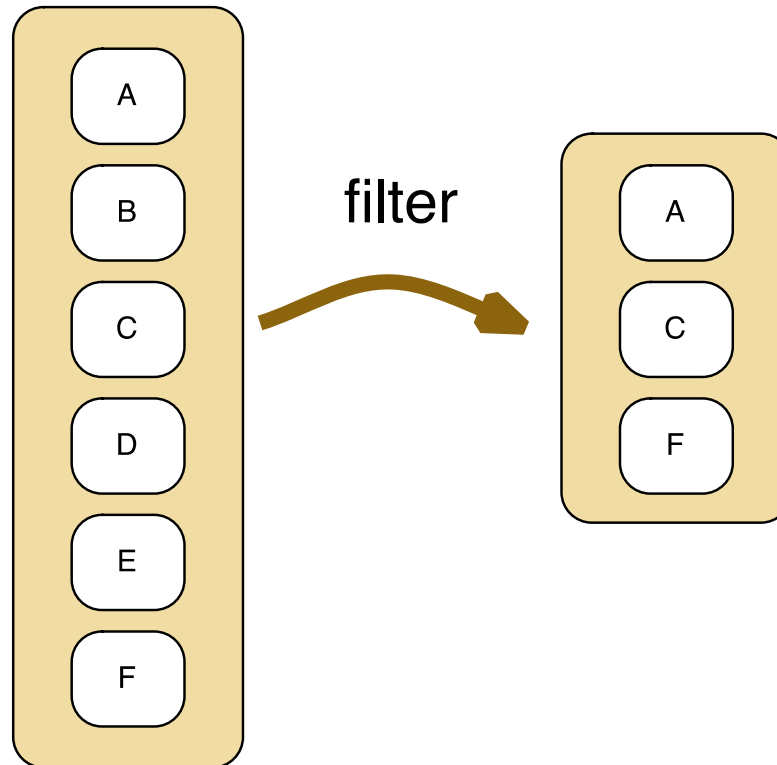
// Java 7

```
List<String> result =  
    new ArrayList<String>();  
  
for (String str : myList) {  
    if (str.length() > 5) {  
        result.add(str);  
    }  
}
```

// Java 8

```
Stream<String> filtered =  
    myStream.filter  
        (s -> s.length() > 5);
```

Filter



Find

Return one element satisfying a condition

```
// Java 7
String result = null;

for (String str : myList) {
    if (str.length() == 5) {
        result = str;
        break;
    }
}
```

```
// Java 8
Optional<String> result =
    myStream
        .filter
            (s -> s.length() == 5)
        .findAny();
```

Map

Transform elements by applying a Lambda to each element

// Java 7

```
List<Integer> lens =  
    new ArrayList<Integer>();
```

```
for (String str : myList) {  
    lens.add(str.length());  
}
```

// Java 8

```
Stream<Integer> lens =  
    myStream.map(s -> s.length());
```

Map Examples



© DragonImages/iStockphoto.

- `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
```

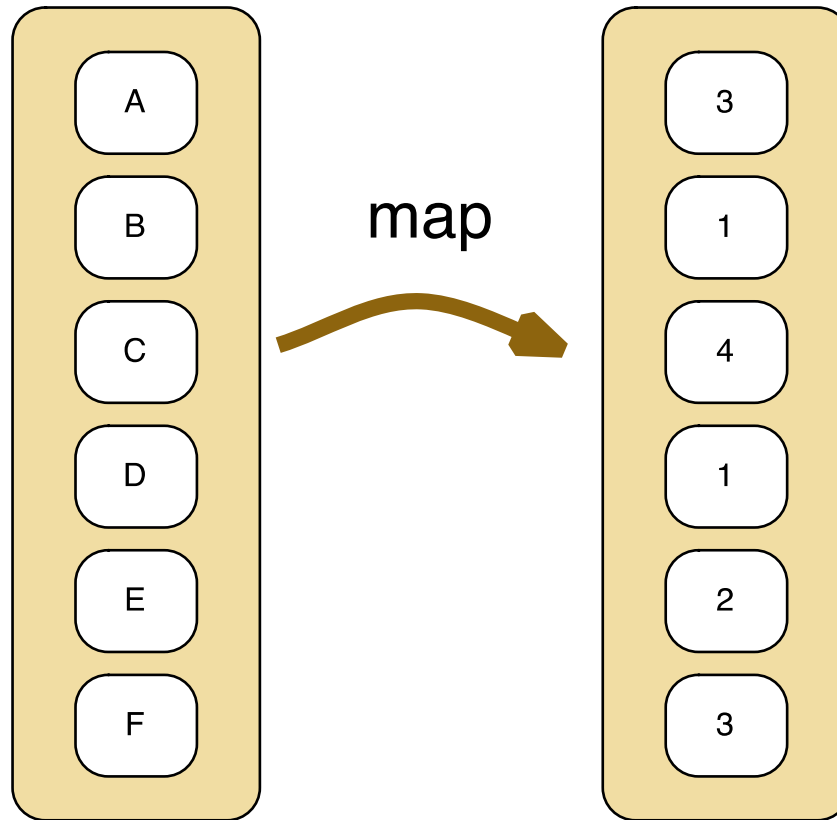
StreamDemo.java

```
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...]
```

Map



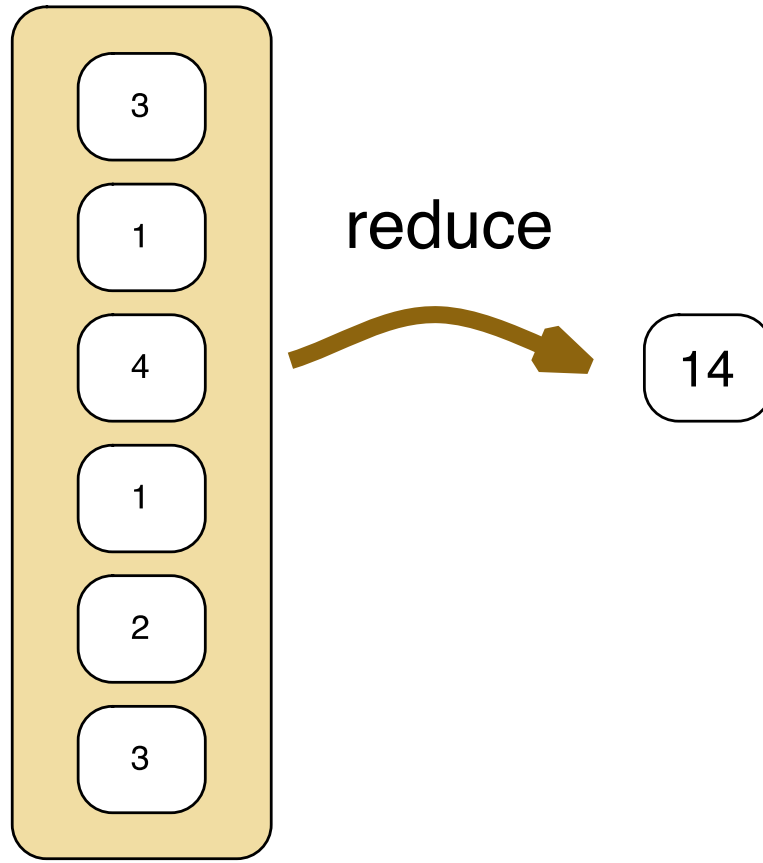
Reduce

Apply an accumulator function to each element of the list to reduce them to a single value.

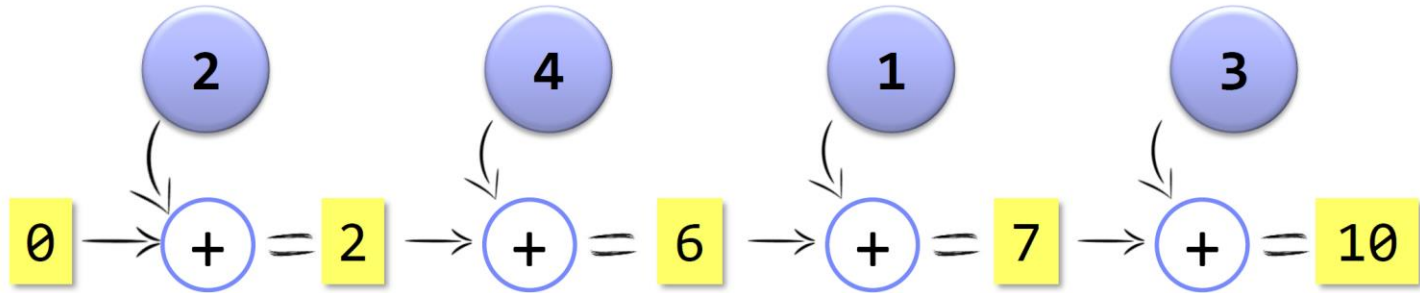
```
// Java 7
int totalLen = 0;
for (String str : myList) {
    totalLen += str.length();
}
```

```
// Java 8
int totalLen =
    myStream
        .reduce(0,
            (sum,s) -> sum + s.length());
```

Reduce



Reduce



.reduce(0, +)

Convenience Reducers

Min, Max, Sum, Average, Count, etc.

```
// Java 7
int totalLen = 0;
for (String str : myList) {
    totalLen += str.length();
}

double avgLen =
    ((double)totalLen)
    / myList.size();
```

```
// Java 8
OptionalDouble avgLen =
    myStream
        .mapToInt
            (s -> s.length())
        .average();
```

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

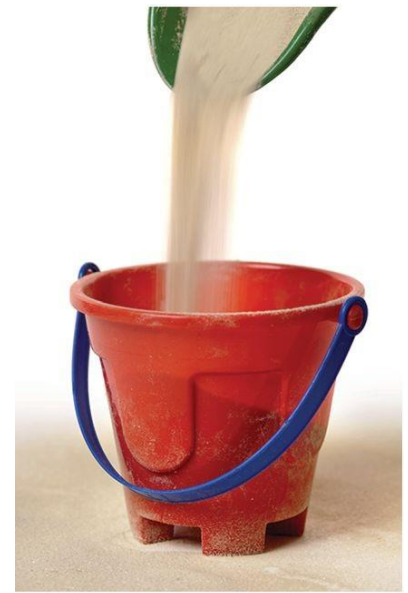
```
List<String> result = stream.collect(Collectors.toList());  
Set<String> result = stream.collect(Collectors.toSet());
```

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
```



© Jamesmcq24/iStockphoto.

Flat Map

Do a map and flatten the result by one level

// Java 7

```
List<Character> chars =  
    new ArrayList<Character>();  
  
for (String str : myList) {  
    for(char ch : str.toCharArray()){  
        chars.add(ch);  
    }  
}
```

// Java 8

```
Stream<Character> chars =  
    myStream  
        .flatMapToInt  
            (s -> s.chars())  
        .mapToObj(i -> (char) i);
```

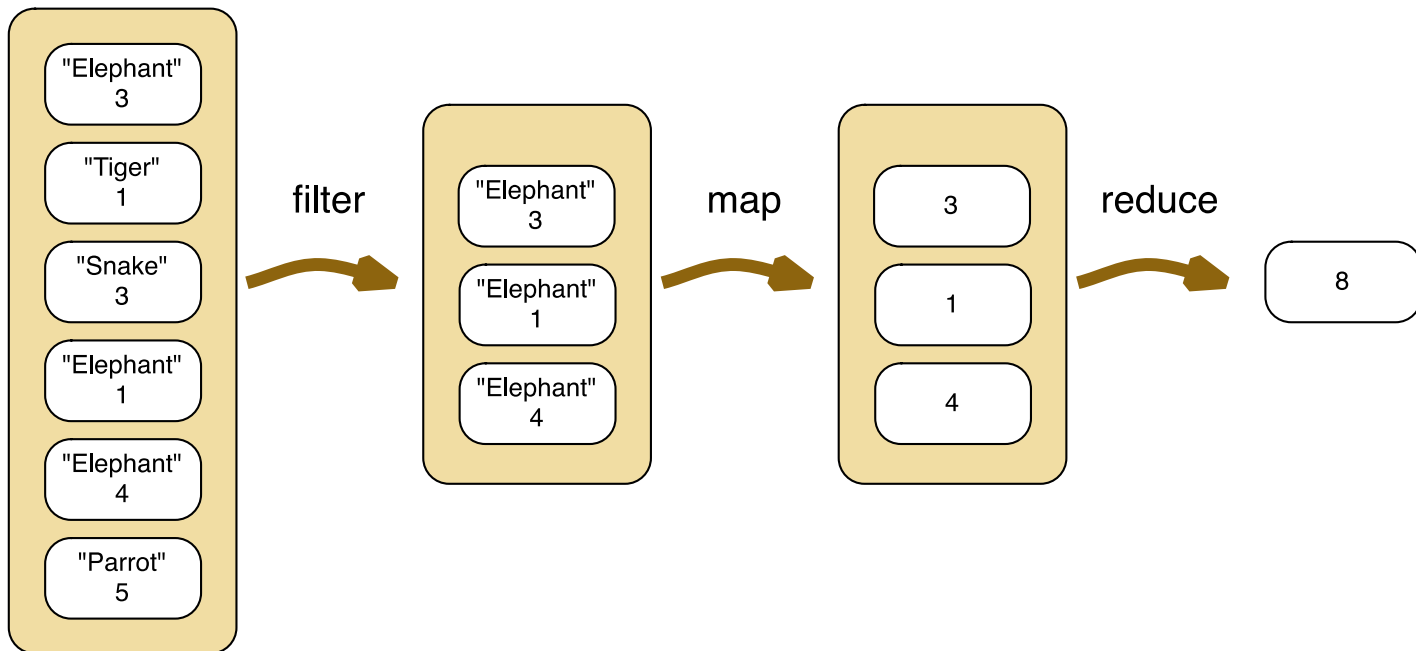
Sorting

Example: Sort strings by length (longest to shortest) and then alphabetically

Java 8

```
Stream<String> result =  
    myStream  
        .sorted(  
            Comparator.comparing(String::length)  
                .reversed()  
                .thenComparing(  
                    Comparator.naturalOrder()  
                )  
        )  
    ;
```

A pipeline of operations



```
filter(name is elephant).map(count).reduce(add up)
```

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 Operators - Summary

Example	Comments
<code>stream.filter(<i>condition</i>)</code>	A stream with the elements matching the condition.
<code>stream.map(<i>function</i>)</code>	A stream with the results of applying the function to each element.
<code>stream.mapToInt(<i>function</i>)</code> <code>stream.mapToDouble(<i>function</i>)</code> <code>stream.mapToLong(<i>function</i>)</code>	A primitive-type stream with the results of applying a function with a return value of a primitive type
<code>stream.limit(n)</code> <code>stream.skip(n)</code>	A stream consisting of the first n, or all but the first n elements.
<code>stream.distinct()</code> <code>stream.sorted()</code> <code>stream.sorted(<i>comparator</i>)</code>	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

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())`

How to start thinking in the functional style

- Avoid “for” loops
 - If you find yourself using one, there’s probably a better way to do what you want.
- Use fewer variables
 - See if you can connect multiple steps together into an expressive “chain” of method calls.
- Avoid “side effects”
 - When you run a function, it shouldn’t inadvertently change the value of its input or some other variable.