Java 8 Stream API

Efthimios Alepis

Introduction

- Package java.util.stream provides classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections
- The key abstraction introduced in this package is **stream**
- The classes Stream, IntStream, LongStream, and DoubleStream are streams over objects and the primitive int, long and double types
- Streams are Monads, thus playing a big part in bringing functional programming to Java

- No storage. A stream is not a data structure that stores elements
- Functional in nature. An operation on a stream produces a result, but does not modify its source
- Laziness-seeking. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization
- Possibly unbounded. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time
- Consumable. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source

Streams Vs Collections

Obtaining a Stream

- From a Collection via the stream() and parallelStream() methods
- From an array via Arrays.stream(Object[])
- From static factory methods on the stream classes, such as Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator)
- The lines of a file can be obtained from BufferedReader.lines()
- Streams of file paths can be obtained from methods in Files
- Streams of random numbers can be obtained from Random.ints()

Stream operations and pipelines

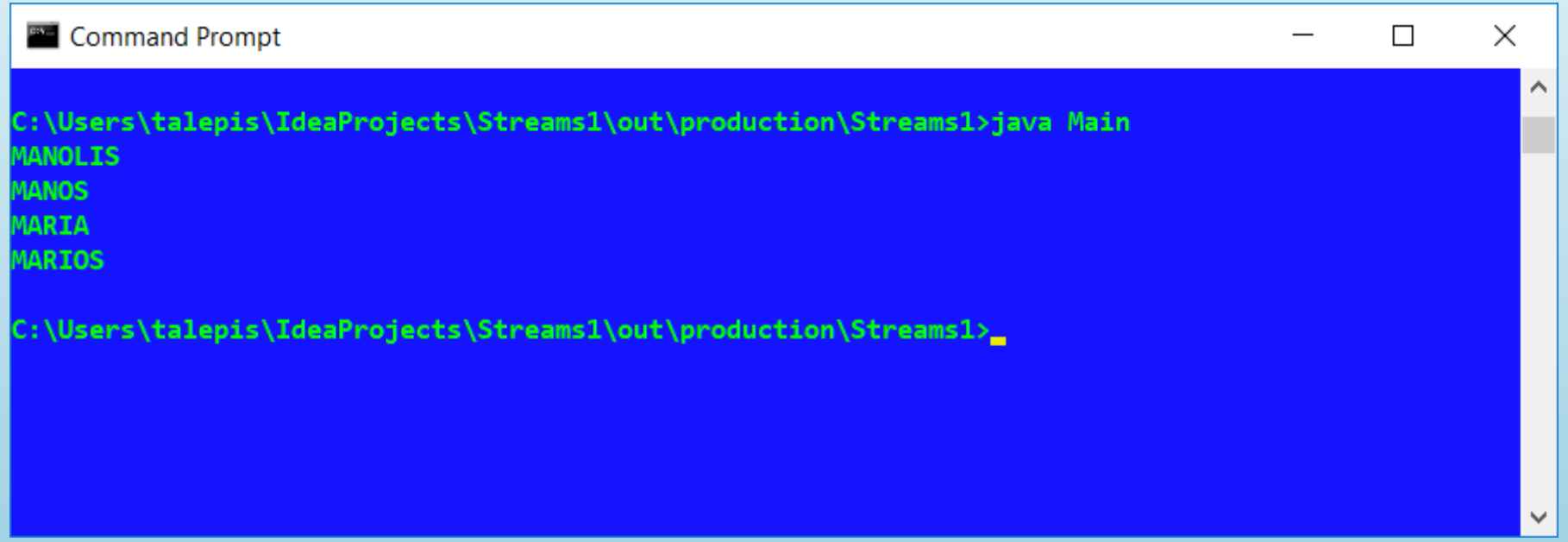
- Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source, followed by zero or more intermediate operations such as Stream.filter or Stream.map, and a terminal operation such as Stream.forEach or Stream.reduce
- Intermediate operations return a new stream. They are always lazy: executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate
- Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used

Stream Parallelism

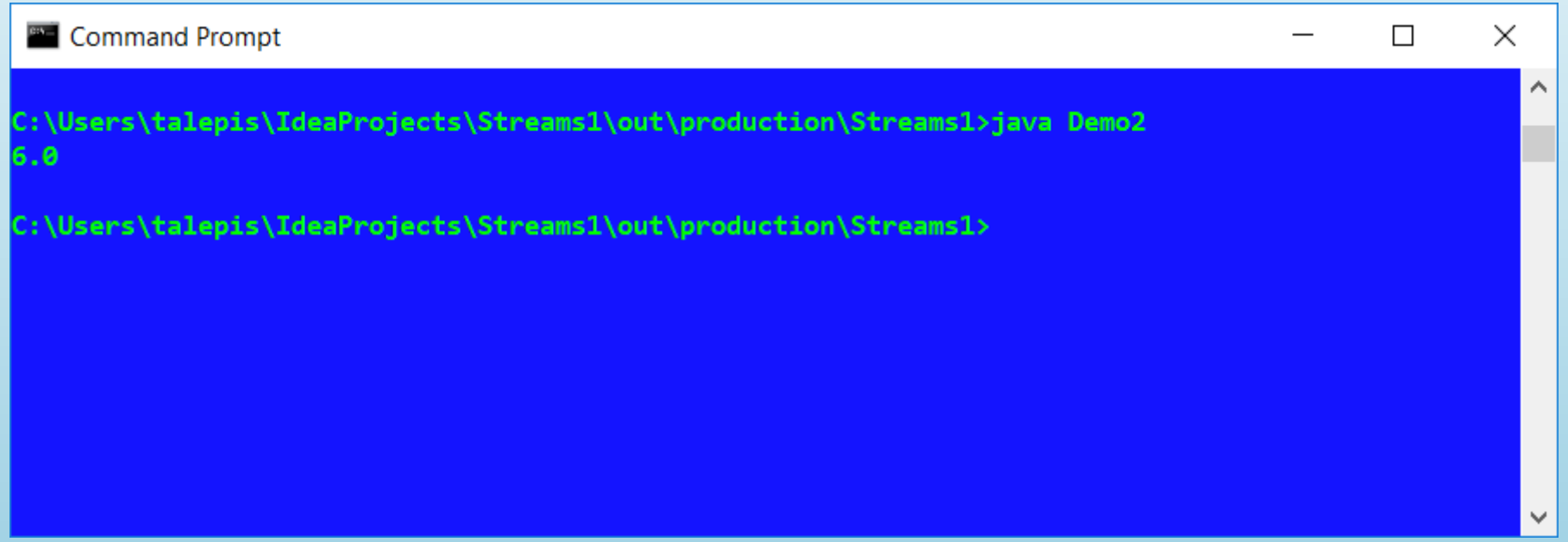
- Processing elements with an explicit for-loop is inherently serial
- All streams operations can execute either in serial or in parallel
- The stream implementations in the JDK create serial streams unless parallelism is explicitly requested
- For example, Collection has methods Collection.stream() and Collection.parallelStream(), which produce sequential and parallel streams respectively

Simple Example

```java
import java.util.Arrays;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<String> myList =
                Arrays.asList("Manolis", "Efthimios", "Maria", "Christina",
                        "Marios", "Manos", "Dimitris", "Costas");
        myList  .stream()
                .filter(s -> s.startsWith("M"))
                .map(String::toUpperCase)
                .sorted()
                .forEach(System.out::println);
    }
}
```


Command Prompt
C:\Users\talepis\IdeaProjects\Streams1\out\production\Streams1>java Main
MANOLIS
MANOS
MARIA
MARIOS
C:\Users\talepis\IdeaProjects\Streams1\out\production\Streams1>

```java
import java.util.Arrays;

public class Demo2 {
    public static void main(String[] args) {
        Arrays.stream(new int[] {1, 2, 3, 4})
                .map(n -> 2 * n + 1)
                .average()
                .ifPresent(System.out::println);
    }
}
```


Command Prompt
C:\Users\talepis\IdeaProjects\Streams1\out\production\Streams1>java Demo2
6.0
C:\Users\talepis\IdeaProjects\Streams1\out\production\Streams1>

Stream Creation

```
Stream<String> streamEmpty = Stream.empty();
```

- The empty() method should be used in case of a creation of an empty stream
- Its often the case that the empty() method is used upon creation to avoid returning null for streams with no element:

```
public Stream<String> streamOf(List<String> list) {
   return list == null || list.isEmpty() ? Stream.empty() : list.stream();
}
```

Empty Stream

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

Stream of Collection

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Stream of Array

```
Stream<String> streamBuilder =
  Stream.<String>builder().add("a").add("b").add("c").build();
```

- ✓ When builder is used the desired type should be additionally specified in the right part of the statement, otherwise the build() method will create an instance of the Stream<Object>

Stream.builder()

```
Stream<String> streamOfString =
  Pattern.compile(", ").splitAsStream("a, b, c");
```

Stream of String

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
  Files.lines(path, Charset.forName("UTF-8"));
```

Stream of File

- It is possible to instantiate a stream and to have an accessible reference to it as long as only intermediate operations were called
- Executing a terminal operation makes a stream inaccessible

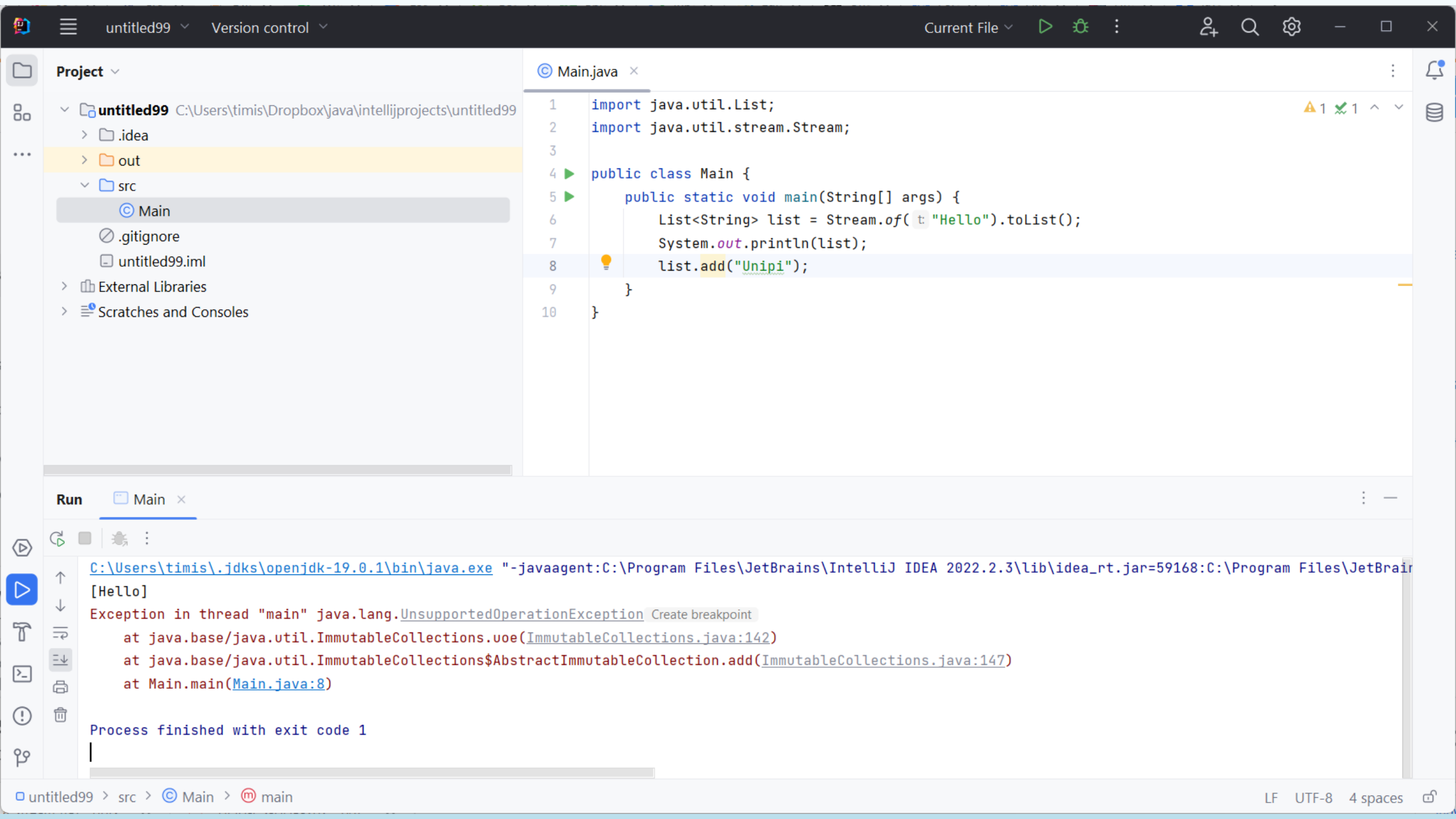
```
Stream<String> stream =
  Stream.of("a", "b", "c").filter(element -> element.contains("b"));
Optional<String> anyElement = stream.findAny();
```

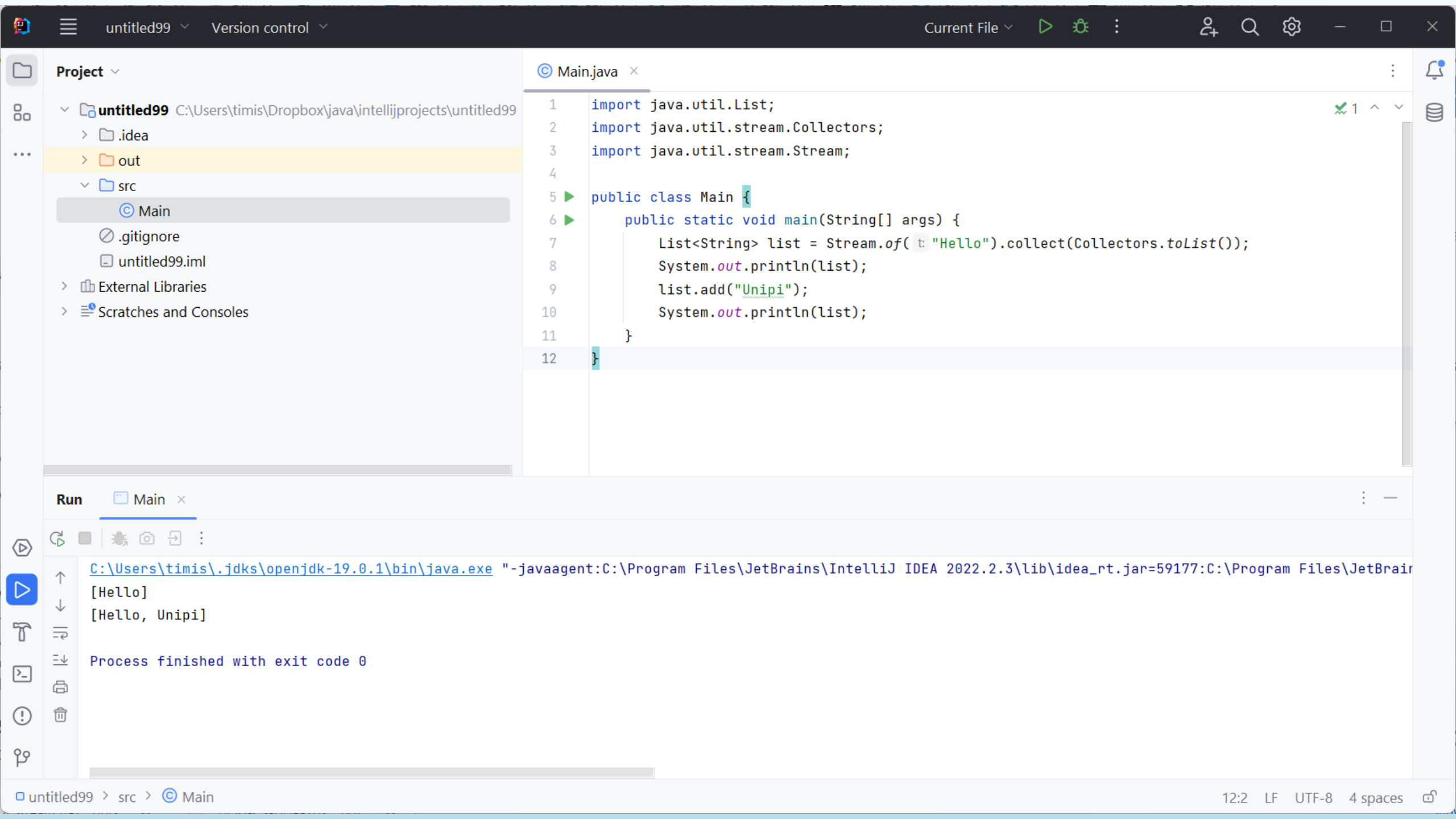
- It is very important to remember that **Java 8 streams can’t be reused**

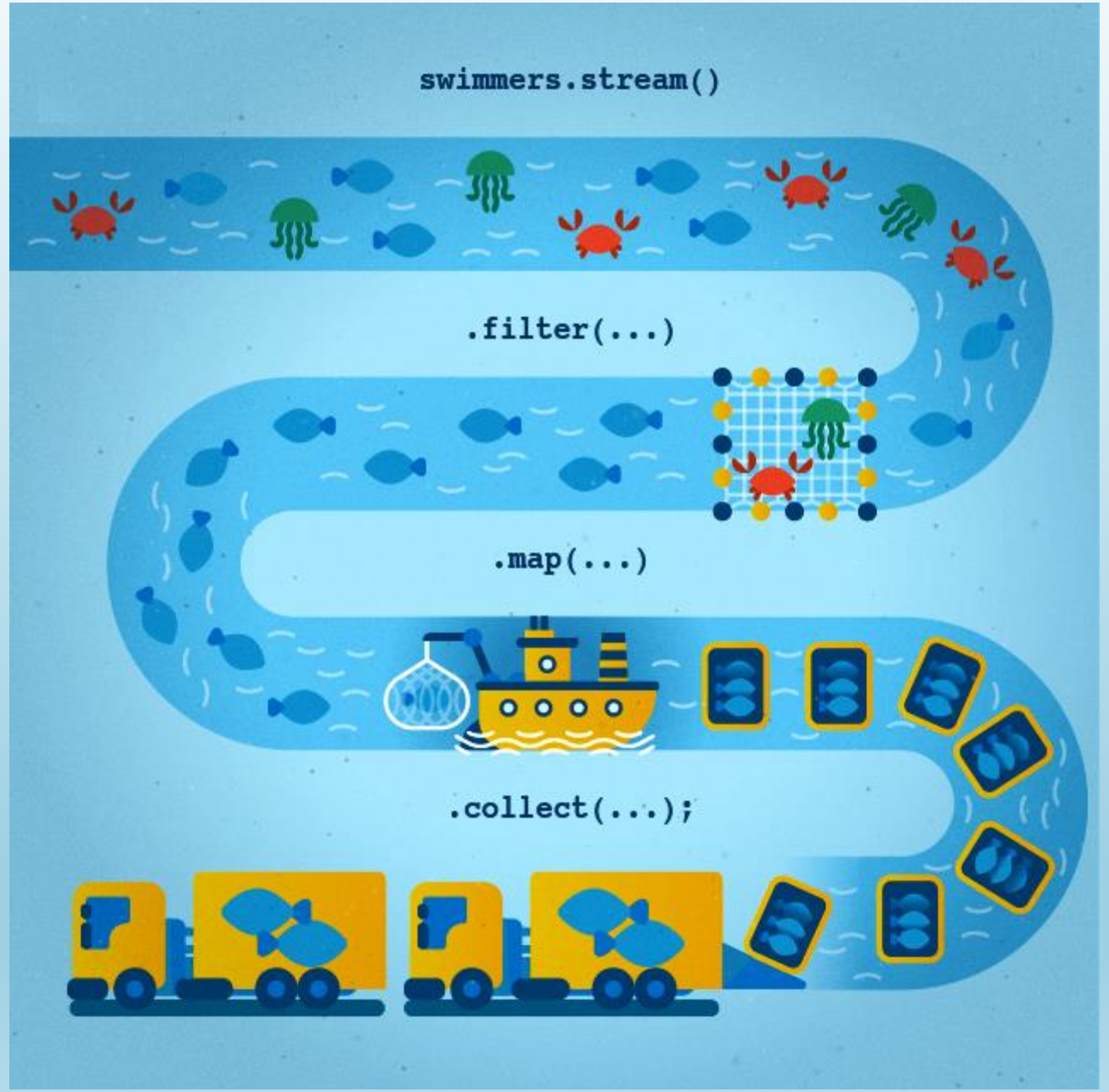
Referencing a Stream

Method	Guarantees unmodifiability	Allows nulls
`collect(toList())`	No	Yes
`collect(toUnmodifiableList())`	Yes	No
`toList()`	Yes	Yes

Collecting Streams as Lists


untitled99
Version control
Current File
Project
untitled99 C:\Users\timis\Dropbox\java\intellijprojects\untitled99
.idea
out
src
Main
.gitignore
untitled99.iml
External Libraries
Scratches and Consoles
Main.java
1 import java.util.List;
2 import java.util.stream.Stream;
3
4 public class Main {
5 public static void main(String[] args) {
6 List<String> list = Stream.of(t: "Hello").toList();
7 System.out.println(list);
8 list.add("Unipi");
9 }
10 }
Run
Main
C:\Users\timis\.jdks\openjdk-19.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.3\lib\idea_rt.jar=59168:C:\Program Files\JetBrai
[Hello]
Exception in thread "main" java.lang.UnsupportedOperationException Create breakpoint
at java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.java:142)
at java.base/java.util.ImmutableCollections$AbstractImmutableCollection.add(ImmutableCollections.java:147)
at Main.main(Main.java:8)
Process finished with exit code 1
untitled99 > src > Main > main
LF
UTF-8
4 spaces


untitled99
Version control
Current File
Project
untitled99 C:\Users\timis\Dropbox\java\intellijprojects\untitled99
.idea
out
src
Main
.gitignore
untitled99.iml
External Libraries
Scratches and Consoles
Main.java
1 import java.util.List;
2 import java.util.stream.Collectors;
3 import java.util.stream.Stream;
4
5 public class Main {
6 public static void main(String[] args) {
7 List<String> list = Stream.of(t: "Hello").collect(Collectors.toList());
8 System.out.println(list);
9 list.add("Unipi");
10 System.out.println(list);
11 }
12 }
Run
Main
C:\Users\timis\.jdks\openjdk-19.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.3\lib\idea_rt.jar=59177:C:\Program Files\JetBrair
[Hello]
[Hello, Unipi]
Process finished with exit code 0
untitled99 > src > Main
12:2 LF UTF-8 4 spaces


swimmers.stream()
.filter(...)
.map(...)
.collect(...);

Conclusions

- The Stream API is a powerful but simple to understand set of tools for processing sequence of elements
- It allows us to reduce a huge amount of boilerplate code, create more readable programs and improve app's productivity when used properly
- Hint: *don't leave instantiated streams unconsumed as that may lead to memory leaks*