CS209A – Stream

Key Content

- What is a Stream
- Understand the necessity of Stream.
- Learn how to work with Stream.

1. Before Exercise

We know in software engineering a well-known problem is that no matter what you do, user requirements will change. We have rewrite some versions of code for a sort problem. But the user requirements always change.

Now you are given a task that you only sort the elements that smaller than 50 and you are asked sort by (value-index), then how do you change the code?

2. What is a Stream

Stream sounds similar to **InputStream** and **OutputStream** from Java I/O. But **Stream** is a completely different thing. Streams are Monads, thus playing a big part in bringing *functional programming* to Java:

*In functional programming, a **monad** is a structure that represents **computations defined as sequences of steps**. A type with a monad structure defines what it means to **chain operations**, or nest functions of that type together.*

The Streams API in Java 8 lets you write code that's

- ***Declarative*** — More concise and readable
- ***Composable*** — Greater flexibility
- ***Parallelizable*** — Better performance

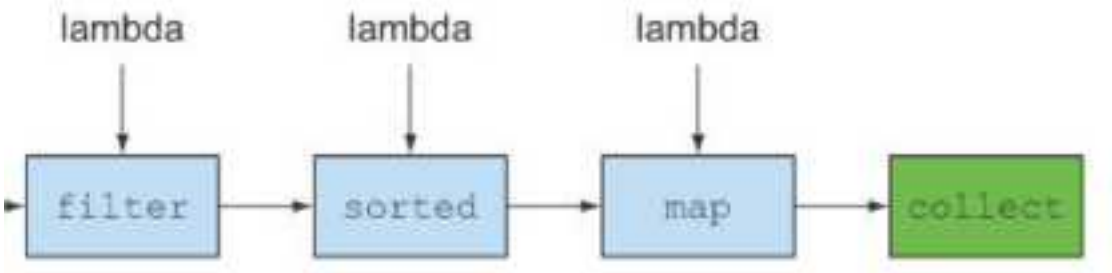
3. How streams work

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements:

```
ArrayList<Element> arrayList = new ArrayList<Element>(n);
for (int i = 0; i < n; i++) {
    arrayList.add(new Element(i, in.nextInt()));
}
```

```
//input:
// value: 5 3 66 22 9 1 77 88 99
// index: 0 1 2   3 4 5 6   7   8
arrayList.stream().filter(e->e.value<50)   //value: 5 3 22 9 1 index:0 1   3   4   5
      .map(e->new Element(e.index,e.value - e.index)) //value: 5 2 19 5 -4 index:0 1   3   4   5
      .sorted(methodList.get(method)) //value: -4 2 5 5 19 index:5 1   0   4   3
      .forEach(e->System.out.printf("%d ", e.index)); //print
```

Stream operations are either **intermediate** or **terminal**. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. **Terminal operations are either void or return a non-stream result**. In the above example *filter, map and sorted* are intermediate operations whereas *forEac*h is a terminal operation. For a full list of all available stream operations see the **Stream Javadoc**. Such a chain of stream operations as seen in the example above is also known as **operation pipeline**.



Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both **non-interfering** and **stateless**.

Non-interfering:

A function is non-interfering when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify arrayList by adding or removing elements from the collection.

Stateless:

A function is stateless when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

3.1 Creating streams

Streams can be created from various data sources, especially collections.

StreamCreating.java shows a lot of examples how Streams be created from various data sources.

3.2 How stream operations are processed

An important characteristic of intermediate operations is laziness. Look at this sample where a terminal operation is missing:

```
Stream.of("CS", "209", "A").filter(s -> {
                System.out.println("filter: " + s);
                return true;
        });
```

When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.

More example please refer **StreamProcessingOrder.java**.

3.3 Reusing streams

Streams cannot be reused. As soon as you call any terminal operation the stream is closed:

```
Stream<String> stream =
      Stream.of("CS", "209", "A")
            .filter(s -> s.startsWith("C"));

stream.anyMatch(s -> true);       // ok
stream.noneMatch(s -> true);      // exception
```

Calling noneMatch after anyMatch on the same stream results in the following exception:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated
upon or closed
      at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
      at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
      at StreamProcessingOrder.main(StreamProcessingOrder.java:95)
```

To overcome this limitation we have to to create a new stream chain for every terminal operation we want to execute, e.g. we could create a **stream supplier** to construct a new stream **with all intermediate operations** already set up:

```
Supplier<Stream<String>> streamSupplier =
      () -> Stream.of("CS", "209", "A")
                  .filter(s -> s.startsWith("C"));

streamSupplier.get().anyMatch(s -> true);     // ok
streamSupplier.get().noneMatch(s -> true);    // ok
```

3.4 Advanced Operations

Streams support plenty of different operations. The most important operations like filter or map. I leave it up to you to discover all other available operations (see Stream Javadoc). Instead let's dive deeper into the more complex operations collect, flatMap and reduce.

StreamExample.java shows how to use these advanced stream operations.