“System Design” – Reactive Programming

Lambdas / Streams / Publisher-Subscriber [06]

1 Lambdas

1.1 Comparator

A list will be populated with random numbers:

```
int n = 10;
Random r = new Random();
List<Integer> list = new ArrayList<>();
for(int i = 0; i < n; i++)
     list.add(r.nextInt(n));
System.out.println(list);
```

The method `list.sort(Comparator<? super E> c)` sorts the given list according to the comparator.

Create and supply a comparator (for natural sort order of `int`)

- as implementing class
- as anonymous class
- as lambda-expression

The interface `Comparator<T>` defines the method `int compare(T o1, T o2)` with the following semantics: it returns a value < 0, if `o1` is considered “smaller” than `o2`, 0 if they are considered equal and a value > 0 if `o1` is considered “larger” than `o2`. The natural sort-order of int can be this can be achieved by achieved by simply subtracting the parameters, which will yield the above results. For a detailed documentation of the Comparator see: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Now change the lambda-expression in a way, that

- the list is sorted in reverse order
- odd numbers are considered smaller than even numbers

1.2 Function Declaration

Define a lambda of the type `IntUnaryOperator` which calculates the factorial. Bonus: try to define it recursively.

2 Streams

2.1 Creation of Streams

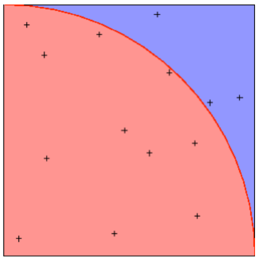
Calculate the squares of numbers from 1 to 10 with the help of streams. Use `IntStream` as the data source. In a first version, output the numbers on the screen. In a second version, create a list of the square numbers. Hint: For collecting you need the intermediate operation `boxed()`, which converts primitives to wrappers, as collections cannot deal with primitives.

Create a stream of 10 random doubles. In a first version use `Stream.generate(…)`. For generating a random number use `ThreadLocalRandom.current().nextDouble()` for the supplier. For a second implementation use `ThreadLocalRandom.current().doubles()` which directly returns a random stream.

2.2 π Calculation

Redo last week's task with the help of streams, i.e.: Calculate π with the help of the Monte-Carlo-Method (you find the frame for this task in Moodle):

- shoot into search space(x, y) within domain [0..1; 0..1]
- check, whether the shot lies within the unit circle's quarter
- the ratio of shots per hits *4 evaluates to π.



Use the given class-frame in Moodle and do the calculation of hits with a ***single stream statement***. Hint: instead out incrementing hits, it is more efficient to count the hits.

After measuring the time the calculation takes, change it to a parallel stream.

2.3 Grouping

Given is the following array String-Integer-Pairs:

```
Object[][] data = {{"A", 1}, {"A", 2}, {"B", 1}, {"B", 2}, {"B", 3}, {"C", 2}, {"C", 3}};
```

Create a `Map<String, Set<Object>>` which takes the first element of each pair as key and collect the corresponding numbers into a set. Hint: use the Collectors' mapping-method (see: https://docs.oracle.com/javase/10/docs/api/java/util/stream/Collectors.html#mapping(java.util.function.Function,java.util.stream.Collector))

Now group the elements by the Integer-Value resulting in a `Map<Integer, Set<Object>>`.

3 Publisher / Subscriber

In the following exercise you should create a simple reactive program. The application is based on the talk by Venkat Subramaniam: Reactive Programming in Java (https://www.youtube.com/watch?v=f3acAsSZPhU) – although it uses a different technology.

In Moodle you find two Maven-Projects for this exercise. Import these projects in your workspace: „Import …“ → „Existing Projects into Workspace“. Select the project and use option „Copy projects into workspace“.

The server-project implements a simple REST-Server which returns simulated stock data – this project should stay unchanged. You can start the server with the `StartServer` class. The server provides the a service (for Stock) based on a service originally provided by www.worldtradingdata.com [1]

You should extend the client-project. In this project you find the classes `Stock`, `Stocks` and `StockServiceAccess` – again these classes should stay unchanged. `StockServiceAccess.fetch(String symbol):Stock` retrieves the current stock-data of a supplied stock symbol (e.g. `AAPL` for Apple). You may try it out with its main-method.

You should create a publisher, which receives a list of symbols and publishes the stock data for these symbols. A subscriber should subscribe to this publisher and should receive these values.

3.1 Creating the Basic Application

Initially you create a “Hot Publisher” which emits stock data and a subscriber which receives and prints this data.

Creating the Publisher (in the package `flow`) (see slide 36)

- As a publisher create your own class `StockPublisher` which implements the `Publisher`-Interface for the type `Stock`.
- On construction the publisher receives a list of symbols to be fetched from the server. In the constructor you should first create `SubmissionPublisher` (predefined by Java), store it as a private field and use it as a delegate. For emitting the message create a new thread and start it immediately in the constructor. The thread (in an endless loop) should fetch for each given symbol a `Stock`-Object and submits it to the internal publisher. After each loop wait for a second.
- The `subscribe`-method should delegate the subscription to the internal publisher.

Creating the Subscriber (in the package `flow`) (see slide 27)

- As a subscriber, you can simply use the SampleSubscriber of the lecture

Host the following operations in the `main`-method of the `Main`-class. *(in the package `main`)*

- This Publisher should receive the list of symbols when created; e.g.:

```
List<String> symbols = Arrays.asList("GOOGL","AMZN", "INTC");
Publisher<Stock> feed = new StockPublisher(symbols);
```

[1] Unfortunately, this service is no longer available in this form – however, the simulation is sufficient for the exercise and allows an unlimited number of calls and the simulation of errors, which was not possible with the original service.

- Then create your subscriber

  ```
  Subscriber<Stock> subscriber = new BasicSubscriber<>();
  ```

- Now subscribe to the publisher:

  ```
  feed.subscribe(subscriber);
  ```

- The subscriber should immediately emit the stock prices; e.g.

  ```
  created ...
  Ready to emit ...
  Subscriber: subscribed
  Subscriber: got GOOGL : 1039.790000
  Subscriber: got AMZN : 1609.950000
  Subscriber: got INTC : 47.680000
  Subscriber: got GOOGL : 1039.790000
  Subscriber: got AMZN : 1609.950000
  …
  ```

3.2 Dealing with Errors and End of Stream

Instead of using the regular server, use `StartServerWithHazards` which returns sometimes null (which indicates the end of the stream) or a message which forces the client to throw an exception (which indicates an error while retrieving).

Run your application with the hazardous server – the results should be rather ugly.

Now fix your Publisher:

- If the fetch-operation should throw any kind of exception, catch it and call `closeExceptionally` for your internal publisher.
- If the fetch-operation returns null call `close` for your internal publisher.
- In both cases, end the thread by leaving the loop.

Try it again, your application should now behave more sensible; e.g.:

```
created ...
Ready to emit ...
Subscriber: subscribed
Subscriber: got GOOGL : 1040.410000
Subscriber: got AMZN : 1610.080000
Subscriber: error java.lang.RuntimeException: something went wrong
```

or

```
created ...
Ready to emit ...
Subscriber: subscribed
Subscriber: got GOOGL : 1040.410000
Subscriber: got AMZN : 1609.750000
Subscriber: done
```

3.3 Realizing a “Cold Publisher”

For this use the regular server again. Your current publisher acts as a “hot” publisher, thus starting the publishing immediately, even if no subscriber has subscribed. A “cold” publisher starts publishing, once a subscriber has subscribed. Implement this strategy – i.e. start the thread, once the first subscriber subscribes. You can check if a Thread is not yet started with the expression `thread.getState() == Thread.State.NEW`

3.4 Realizing a Filter-Processor

Create a FilterProcessor which filters the results according to a given predicate. As a blueprint you can use the `TransformerProcessor` of the lecture. The Processor should receive a predicate as parameter, which filters the accepted Stock. E.g.:

```
Processor<Stock, Stock> processor
      = new TransformerFilter<>(s -> s.price >= 1000);
```

Now put the processor in-between your previous publisher and subscriber:

```
feed.subscribe(processor);
processor.subscribe(subscriber);
```

The output-stream should only show the filtered stock:

```
created ...
Ready to emit ...
Subscriber: subscribed
Subscriber: got GOOGL : 1039.210000
Subscriber: got AMZN : 1607.230000
Subscriber: got GOOGL : 1039.210000
Subscriber: got AMZN : 1607.230000
Subscriber: got GOOGL : 1039.210000
Subscriber: got AMZN : 1607.230000
Subscriber: got GOOGL : 1039.210000
Subscriber: got AMZN : 1607.230000
…
```