Nachdenkzettel: Streams

Mini games group

1. Filtern sie die folgende Liste mit Hilfe eines Streams nach Namen mit „K“ am Anfang:

final List<String> names = Arrays.asList(“John”, “Karl”, “Steve”, “Ashley”, “Kate”);

names.stream().filter(i → i.startsWith(“K”)).forEach(i → System.out.print(i));

2. Welche 4 Typen von Functions gibt es in Java8 und wie heisst ihre Access-Methode?

Tipp: Stellen Sie sich eine echte Funktion vor (keine Seiteneffekte) und variieren Sie die verschiedenen Teile der Funktion.

Function accepts any type and returns any type (Stream.map)

Predicate accepts any type and returns boolean (Stream.filter)

Consumer takes any type but returns nothing (println)

Supplier takes no argument and returns any type (Stream.generate)

3. forEach() and peek() operieren nur über Seiteneffekte. Wieso?

Both functions are consumers, that means: one should write some side effects because they return nothing.

4. sort() ist eine interessante Funktion in Streams. Vor allem wenn es ein paralleler Stream ist. Worin liegt das Problem?

Sort cannot be parallel because it needs all the elements.

5. Achtung: Erklären Sie was falsch oder problematisch ist und warum.

a) Set<Integer> seen = new HashSet<>();

someCollection.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })

Hashset cannot access from interfaces safely, it may be corrupted. So, it does not have parallel access.

b) Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());

someCollection.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })

SynchronizedSet is safe, map has a side effect despite being a function and not a consumer.

Incorrect usage of streams. It is doing nothing.

6. Ergebnis?

List<String> names = Arrays.asList("1a", "2b", "3c", "4d", "5e");

names.stream()

.map(x → x.toUppercase())

.mapToInt(x → Integer.parseInt(x.substring(0,1)))

.filter(x → x < 5)

.forEach(System.out::println)

makes all the letters to upper case

Remove characters different than numbers

filters all numbers greater than 4

prints all numbers from 1 to 4

Wenn Sie schon am Grübeln sind, erklären Sie doch bei der Gelegenheit warum es gut ist, dass Streams „faul" sind.

7. Wieso braucht es das 3. Argument in der reduce Methode?

```
List<Person> persons = Arrays.asList(
        new Person("Max", 18, 4000),
        new Person("Peter", 23, 5000),
        new Person("Pamela", 23, 6000),
        new Person("David", 12, 7000));

        int money = persons
                .parallelStream()
                .filter(p -> p.salary > 5000)
                .reduce(0, (p1, p2) -> ( p1 + p2.salary), (s1, s2)-> (s1 + s2));

log.debug("salaries: " + money);
```

Tipp: Stellen Sie sich eine Streamsarchitektur vor (schauen Sie meine Slides an).

Am Anfang ist eine Collection. Sie haben mehrere Threads zur Verfügung. Mit was fangen Sie an? Dann haben die Threads gearbeitet. Was muss dann passieren?

parallelStream uses that argument to work

8. Was ist der Effekt von stream.unordered() bei sequentiellen Streams und bei parallelen streams?

Sequential and parallel streams have the same properties with respect to ordering. The stream.unordered() method may be applied to a stream to indicate to the implementation that the order is not important. Parallel streams will have, in some cases, a better performance if they do not have to respect the ordering of the stream elements.

9. Fallen

a)
```
IntStream stream = IntStream.of(1, 2);
stream.forEach(System.out::println);
stream.forEach(System.out::println);
```

forEach cannot be used again on the stream, you already used it in the second line.

b)
```
IntStream.iterate(0, i -> i + 1)
        .forEach(System.out::println);
```

it will be an infinite stream

c)
```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
        .distinct()                          //.parallel()?
        .limit(10)
        .forEach(System.out::println);
```

The stream is also infinite. It will print 0,1 (distictly) but it will never complete and always run.

d)
```
List<Integer> list = IntStream.range(0, 10)
        .boxed()
        .collect(Collectors.toList());

        list.stream()
        .peek(list::remove)
        .forEach(System.out::println);
```

we remove elements while working on the stream. It should have an exception and not use side effects like “peek”.

from: Java 8 Friday: **http://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/**