Nachdenkzettel: Software-Entwicklung 2, Streams processing

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”);

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

final List<String> filteredNames = names.stream().filter(name →
name.startsWith("K")).toList();
```

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.

Predicates: boolean-valued functions of one argument.
Suppliers: functions that supplies data with no argument
Consumers: functions that consumes data with no return value
Operators: functions that receive and return the same value type

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

forEach() and peek() have no return value, to do something they have to use side effects by e.g. using System.out::println

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

parallel streams need to be sorted at the end (e.g. forEachOrdered()). Otherwise, the order of operations cannot be guaranteed as this would harm the way parallellism works.

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; })

The function in map has a side effect. Generally this is no problem, though considered bad style. But when executing it parallelly like above, there can be concurrency issues.

b) Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());
someCollection.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })

Using a synchronizedSet circumvents the problem from above. But still, it is bad style and now, using a parallel stream is useless as the synchronization is blocking the set.

6. Ergebnis?

```
List<String> names = Arrays.asList(“1a”, “2b”, “3c”, “4d”, “5e”);
 names.stream()
        .map(x → x.toUppercase())
        .mapToInt(x → x.pos(1))
        .filter(x → x < 5)
```

This outputs nothing because

a) there is no terminal function (e.g. „forEach()“).
b) if there where forEach(System.out::println) there won’t be any entry with an int value below 5 (A -> 65, B -> 66, ...)

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

As soon as the terminal function has finished the processing ends. That’s good because when we execute findFirst() we don’t want it to do any further operations after finding the first match.

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?
```

The third argument is the so called combiner that is needed to combine the aggregated values.

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

Sequential streams: when using stream.unordered() the output won't be deterministic as the order can differ from one execution to another.

Parallel streams: Some operations can be implemented more efficiently when the ordering constraint is relaxed. E.g.: filtering duplicates, ...

9. Fallen

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

„stream" is closed after first execution.

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

„iterate" is never stopped.

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

Won't end as there are only two possible values for iterate(0, i -> (i + 1) % 2)

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

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

Modifies the collection list and this errors will occurr.

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