Nachdenkzettel: Software-Entwicklung 2, Streams processing

Sara Tietze (st093), Merve Özdemir(mo064), Pia Schilling (ps149)

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”);
```

```
List<String> filtered = names.stream()
        .filter(s -> s.startsWith("K"))
        .collect(Collectors.toList());

System.out.println(filtered.toString());
```

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.

- filter()
- map()
- println()
- stream()

Die Access-Methode heißt Stream.of.

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

Weil die Funktionen die an forEach() und peek() weitergegeben werden nicht zustandslos sein sollten. Da die Funktionen durch den vorhandenen Zustand, Seiteneffekte hervorrufen können, operieren die beiden in dem Fall über Seiteneffekte.

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

Es wird immer auf die gesamte Collection zugegriffen. Die gesamte Collection wird umgangssprachlich an Station X gesammelt, sortiert und weiter gestreamt. Bei anderen Functions in Streams sind alle Daten verteilt auf “Stationen” in den Verarbeitungsstufen. Wenn die Datenmenge groß ist und man einen parallelen Stream benutzt sollte man auf sort () verzichten, da diese ein Programm verlangsamen kann.

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;
      })
```

Paralleler Stream mit Seiteneffekt. Der parallel stream bearbeitet die Liste, was zu unerwarteten Ergebnissen führen kann.

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

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

Es kann zu falschen Ergebnissen kommen, da der ein Wert evtl schon hinzugefügt wurde der andere Thread/Stream dies aber noch nicht weiß.

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));
```

Kein Ergebnis, da die Methode pos() nicht verfügbar ist.

Wenn Sie schon am Grübeln sind, erklären Sie doch bei der Gelegenheit warum es gut ist, dass Streams„faul“ sind.
So wird schneller ein Ergebnis erzielt, bspw. also sobald ein Ergebnis erzielt ist bricht die Verarbeitungskette ab → Stichwort Performance

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);
```

Weil wir mit einem parallel-stream arbeiten.

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?

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

Unordered “entfernt” die Bedingung, dass der Stream geordnet bleiben muss.

Bei sequentiellen Streams macht das keinen Performance-Unterschied. Bei parallelen Streams kann es (muss aber aber nicht) zu einer effizienteren Abarbeitung des Streams führen.

9. Fallen

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

forEach ist eine Terminal operation, nach dieser ist keine weitere Verkettung möglich.

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

Endlosschleife. Es wird hochgezählt ohne eine obere Grenze zu haben.

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

Endlosschleife. Es wird nie 10 verschiedene Zahlen geben, sondern immer nur null oder 1. Der Stream wird nie also nie aufhören.

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

```
    .forEach(System.out::println);
    from: Java 8 Friday:
```

Die Liste wird bearbeitet während darüber iteriert wird, dies wirft eine Exception.

http://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-thestreams-api/