

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");
List<names> kNames = names.stream().filter(name -> names.contains("K"));
collect(Collectors.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.

Predicate : filter() Function : map() Consumer : println()
Producer : stream()

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

λ -Ausdruck darf Zustand haben, allerdings ist man selbst für Thread-sicherheit des Zustandes verantwortlich

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

Keine Garantie der richtigen Reihenfolge. Für jedes Element kann eine Aktion zu jeder Zeit und jeder Thread ausgeführt werden.

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; })
*ein Element kann 2x vorkommen
entl. unterschiedliche Map*

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

6. Ergebnis?

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

```
names.stream()
```

```
.map(x -> x.toUpperCase())
```

```
.mapToInt(x -> x.toInt(1))
```

```
.filter(x -> x < 5)
```

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

zwischenliegende Operationen werden nicht ausgewertet bis die Operation beendet wird

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));
```

Combine Funktion kombiniert Teilergebnisse um ein neuer Teilergebnis zu erzeugen. combiner ist nötig um parallel laufenden geteilten Variablen zw. zu führen.

```
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?

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

Stream beachtet die Reihenfolge nicht mehr => Operationen werden optimiert

9. Fallen

- a) IntStream stream = IntStream.of(1, 2);

stream.forEach(System.out::println);

stream.forEach(System.out::println);
 for each: terminal function => nach aufrufen geschlossen

L → 2. kann nicht benutzt werden
- b) IntStream.iterate(0, i -> i + 1)

.forEach(System.out::println);
 unendlich sequenziell geordneter stream

L → for each & relativ ausgeführt, da kein limit()
- c) IntStream.iterate(0, i -> (i + 1) % 2)

.distinct()
 Kannst & da distinct() nur max. 2 zulässt und es

.limit(10)
 parallel()?

.forEach(System.out::println);
 mehr als 10 kommt.
- d) List<Integer> list = IntStream.range(0, 10)

.boxed()

.collect(Collectors.toList());
 ConcurrentModificationException, da IntStream verändert wird, während sie ausgeführt wird.

list.stream()

.peek(list::remove)

.forEach(System.out::println);