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”);

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.

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

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

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; })

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.pos(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.

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?

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

9. Fallen

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

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

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

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

Antworten:

1. names.stream().filter(s -> s.startsWith(“K“)).forEach(s -> System.out::println);

2. 1: Consumer: Akzeptiert einzelnen Input, kein Return (zum Beispiel .println())
   2: Supplier: Ohne Parameter, Return (zum Beispiel .stream())
   3: Predicate: Verwendet zum filtern von Daten (zum Beispiel .filter())
   4: Function: Nimmt ein Argument und produziert damit ein Resultat (zum Beispiel .map())

3. forEach() und peek() sind dafür da, um etwas am Array über den Stream zu verändern. Wenn man zum Beispiel mit forEach über ein Array iteriert, ist immer davon auszugehen, dass sich das Array dadurch verändert und nicht nur die Werte des Arrays ausgegeben werden.

4. sort() entscheidet basierend auf der Länge eines Arrays ob es parallel oder sequentiell sortieren soll. Ist das Array zu lang wird parallel sortiert, wodurch es zu einer Sortierung des Arrays kommt, die nicht so gewollt ist (es wird Arrays.parallelSort() verwendet, statt dem normalen sort())

5. a.) Methode hat einen Seiteneffekt. Es kommt durch die parallele Modifizierung zu einem möglichen Updateverlust
   b.) diese Implementierung sollte funktionieren

6. Nichts passiert, da Terminal Options fehlen

   Es ist gut, dass Streams „faul“ sind da somit wirklich nur die im Stream festgelegten Elemente bearbeitet werden und die Verarbeitung abbricht, sobald das erwünschte Ergebnis erreicht ist (man spart sich iterations)

7. Man benötigt das dritte Argument in der reduce() Methode um die eigentliche „Salary“ der beiden verglichenen Objekte abzurufen. Ohne dies zu machen vergleicht man nur die beiden Objekte untereinander und es kommt zum Error

8. Ist der Stream sequentiell iteriert nach und nach über ein Set und ändert somit nicht die Sortierung. Ist der Stream jedoch parallel ändert sich die Sortierung
   Beispielcode:

```
Set<Integer> set = new TreeSet<>(Arrays.asList(-9, -5, -4, -2, 1, 2, 4, 5,
7, 9, 12, 13, 16, 29, 23, 34, 57, 102, 230));
set.stream().unordered().forEach(s ->System.out.print(s + " "));
//Ergebnis: -9 -5 -4 -2 1 2 4 5 7 9 12 13 16 23 29 34 57 102 230
System.out.println("\n");
set.stream().unordered().parallel().forEach(s -> System.out.print(s + " "));
//Ergebnis: 9 12 7 5 13 -9 23 29 34 -4 -5 16 -2 1 57 102 230 4 2
```

9. A) Der Stream wurde bereits verwendet und kann somit nicht noch einmal ausgeführt werden. Es muss erst ein neuer Stream gecalled werden, damit man erneut System.out::println machen kann
   B) Der Stream hat keine limitierte Laufzeit und läuft somit unendlich lange. Man kann das ganze verhindern mit „.limit()“
   C) .distinct erwartet nicht, dass nur zwei verschiedene Werte übergeben werden, wodurch der Stream erneut unendlich lange läuft. Verwendet man dann auch noch .parallel() lastet man schnell den ganzen CPU aus.
   D) Es kommt zu einer ConcurrentModificationException