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");
        names.stream()
                .filter(name -> name.startsWith("K"))
                .forEach(System.out::println);
}
```

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.

- Consumer<T>
    - accept()
    - andThen()

  Haben keinen return Wert
- Producer<T>
  Nehmen keine Werte an
- Supplier<T>
    - get()
- Predicate<T>
    - test()

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

Da ForEach() und peek() ein void returnen.
Da die Aktion for-each in einer Klasse enthalten ist, die das Consumer interface implementiert.

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

Beim sortieren müssen alle Elemente verwendet werden. Gerade bei größeren Listen führt es zu schlechter performance.

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; })

Ist nicht Thread-safe. Da mehrere Threads auf das HashSet zugreifen. Führt zu Datenkorruption.

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

Wurde Thread-safe gemacht. Jetzt können alle Threads nur noch nacheinander auf das HashSet zugreifen. Keine Datenkorruption mehr, dafür deutlich schlechtere performance.

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)

.mapToInt(x → x.pos(1) hat nicht funktioniert.

Haben es umgeschrieben zu.

```
names.stream()
                .map(x -> x.toUpperCase())
                .filter(x -> Integer.parseInt(x.substring(0, 1)) < 5);
```

So bekommen wir als Ergebnis, jeder String bei dem die Zahl kleiner als 5 ist.

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

da Zwischenoperationen nur ausgewertet werden, wenn die Terminaloperation aufgerufen 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));

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

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

Da am Anfang aufgeteilt wurde, müssen die Argumente wieder zusammengefügt werden.

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

Die Reihenfolge der Elemente aufheben.
Bei Streams die parallel laufen, gibt es häufiger Probleme mit der Reihenfolge.
Mit unordered() kann ich den Stream mitteilen, dass die Reihenfolge egal ist.

9. Fallen

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

Funktioniert nicht, da nach dem ersten mal „forEach(System.out::println);“ der Stream sich automatisch geschlossen hat und darauf dann nicht mehr zugegriffen werden kann.

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

Das int i wird bei jedem Durchgang um 1 erhöht, beendet sich aber nicht. Haben einen unendlichen loop.

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

i erhöht sich wieder bei jedem Durchgang. Da Modulo 2 bekommen wir nur 0 und 1 raus. Sobald wir 2 erreicht haben, werden keine neuen Werte mehr geprinted, da diese identisch sind. Limit(10) wird nie erreicht. Deswegen läuft der Stream unendlich.

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

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

wir greifen im debug die Liste an.
Auf die Collection sollte im debug nicht zugegriffen werden.

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