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");
names.stream()
           .filter( s→s.startsWith("K"))
           .forEach(s→System.out.print(s);
```

2. Welche 4 Typen von Functions gibt es in Java8 und wie heisst ihre Access-Methode?
Producer: Funktion ohne Parameter, die Werte zurückgibt (stream())
Consumer: Funktion, die Parameter besitzt aber nichts zurückgibt (println())
Predicate: Funktion, die einen Parameter besitzt und true oder false zurückgibt (filter())
Function: Funktion, die einen Parameter besitzt und einen Wert zurückliefert (map())

3. forEach() and peek() operieren nur über Seiteneffekte. Wieso?
ForEach() und peek() funktionieren nur über Seiteneffekte, da sie nichts zurückgeben, sollten als nicht zustandslos sein.

4. sort() ist eine interessante Funktion in Streams. Vor allem wenn es ein paralleler Stream ist. Worin liegt das Problem?
Es muss jedes Element bis hin zum sort() gelenkt werden (≠ Lazy Streams). Die Geschwindigkeiten der Threads sind unabhängig voneinander, unterscheiden sich so also. Deshalb ist eine parallele Verarbeitung sowie auch eine Sortierung unmöglich.

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; })
```

Hier wird die wird die Collection parallel gestreamt. Das ist nicht so gut gelöst, da sich die Ergebnisse unterscheiden könnten.

b)

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

Es kann ein falsches Ergebnis ausgelöst werden, da der Stream nicht genau wei ob in der Map schon Werte sind.

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)
```

Compilerfehler: **Cannot resolve method 'toUppercase' in 'String'**
Cannot resolve method 'pos' in 'Object'
Es gibt kein Ergebnis, da .pos() nicht verfügbar ist.

Wenn Sie schon am Grübeln sind, erklären Sie doch bei der Gelegenheit warum es gut ist, dass Streams „faul“ sind.
Die Anwendung kann beschleunigt werden, da die “faulen” Streams Performanter sind. Der nächste Phase kann beginnen ohne dass die vorherige beendet wurde. Nachteil von dem gannzen ist, das Seiteneffekte entstehen können.

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?
Bei einem parallelen Stream wird die Ursprungscollection in mehrere Teilcollections zerlegt. Dabei erhält jeder Thread erhält eine Teilcollections und somit jeweils ein Teilergebnis. Das dritte Argument ist zur Addition der Teilergebnisse, somit also das Gesamtergebnis.

8. Was ist der Effekt von stream.unordered() bei sequentiellen Streams und bei parallelen streams?
sequentiellen Stream: die bestehende Ordnung wird umgeordnet
parallelen Stream: wenn Stream bereits geordnet, ändert sich nichts an der Ordnung

9. Fallen

a)
```
IntStream stream = IntStream.of(1, 2);
      stream.forEach(System.out::println);
      stream.forEach(System.out::println);
```
forEach() ist eine terminal Operation → Verkettung nicht möglich

b)
```
IntStream.iterate(0, i -> i + 1)
      .forEach(System.out::println);
```
Endlosschleife, da Hochzählen ohne obere Schranke

c)
```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
            .distinct()                    //.parallel()?
            .limit(10)
            .forEach(System.out::println);
```
Endlosschleife, da Es nie das Limit 10 erreicht wir, denn mit %2 nur Werte zwischen 0 und 1 entstehen können

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

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

wirft eine Exception, da Liste wird gleichzeitig bearbeitet und iteriert

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