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”);

names.stream()

.filter(s -> startsWith(“K”))

.collect(Collectors.toList());

2. Welche 4 Typen von Functions gibt es in Java 8 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: Funktion, die einen Parameter besitzt und true oder false zurückgibt

Function: Funktion, die einen Parameter besitzt und einen Wert zurückliefert

Consumer: Funktion, die Parameter besitzt aber nichts zurückgibt

Producer: Funktion ohne Parameter, die Werte zurückgibt

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

forEach() und peek() liefern keinen return Wert zurück, sondern arbeiten stattdessen mit Seiteneffekten, um State, Flow oder Variablen zu verändern.

(forEach() ist eine terminal Funktion und peek() eine intermediate Funktion)

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

Bei parallelen Streams liefert jeder Stream einen sortierten Stream zurück, diese können aber nicht zu einem sortierten Stream zusammengefügt 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; })

Es greifen mehrere parallele Streams auf die Collection zu, das kann zu Datenverlust führen.

Außerdem fehlt eine terminal Funktion, d.h. der Stream würde nichts machen.

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

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

Aufgrund des synchronized Keywords darf immer nur ein Stream auf die Collection zugreifen, dadurch wird Datenverlust verhindert, aber auch die parallelen Streams ausgebremst, sie bringen nichts mehr.

Außerdem fehlt auch hier eine terminal Funktion.

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.

* Aufgrund der lazy-evaluation wird nichts passieren, weil eine terminal Funktion fehlt
* Es ist gut, dass Streams lazy sind, weil das viel Zeit spart

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?

Das 3. Argument ist notwendig, um die Ergebnisse der parallelen Streams zusammenzufügen.

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);

Ein Stream kann nur einmal aufgerufen werden, beim Versuch denselben Stream mehrmals aufzurufen, kommt eine Exception.

b) IntStream.iterate(0, i -> i + 1)

.forEach(System.out::println);

Erzeugt einen unendlichen Stream, hört nichtmehr auf.

c) IntStream.iterate(0, i -> ( i + 1 ) % 2)

.distinct() //.parallel()?

.limit(10)

.forEach(System.out::println);

Endet auch in einem endlosen Stream, weil nur 0 und 1 gespeichert wird (es werden nur Einsen und Nullen erzeugt), sodass das Limit von 10 nie erreicht wird und immer nach weiteren Werten gesucht wird.

Wird noch ein .parallel() eingefügt, verschärft sich das Problem, weil mehrere Threads des Rechners blockiert werden und endlos den Stream ausführen.

d) List<Integer> list = IntStream.*range*(0, 10)

.boxed()

.collect(Collectors.*toList*());

list.stream()

.peek(list::remove)

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

Funktioniert nicht bzw. liefert unerwartetes Ergebnis, weil Objekte im Stream erst gelöscht und dann ausgegeben werden sollen (liefert gegen Ende null zurück und wirft dann eine Exception).

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