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> namesWithK = names.stream().filter(name >

name.startsWith("K")).toList();

1. 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: Funktion, mit Parameter und Rückgabe true oder false
* Function : Funktion, mit Parameter und etwas zurückgibt
* Consumer: Funktion, mit Parameter aber nichts zurückgibt
* Producer: Funktion ohne Parameter aber mit Rückgabe

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

Da beide Funktionen keinen Rückgabewert haben müssen sie durch Seiteneffekte einen Zustand einnehmen

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

Da die Threads von parallelen Streams nicht synchron sind, ist dort eine Sortierung nicht möglich. Sie muss am Ende vorgenommen werden

1. Achtung: Erklären Sie was falsch oder problematisch ist und warum.
2. Set<Integer> seen = new HashSet<>();

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

* Collection kann in diesem Fall nicht parallel gestreamt werden, da es sonst zu Konkurrenz Problemen kommt

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

Der parallele Stream ist unnötig, da er die Performance herunterzieht

1. 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)

Es wird kein Ergebnis angezeigt, da „Cannot resolve method ‘pos’ in ‘Object’“

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

„faul“ sind.

1. 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 dritte Argument kombiniert (addiert) die Outputs von Parallelen Streams

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

sequentielle Streams: Vorhandene Ordnung unterscheidet sich bei jedem Mal

parallele Streams: tut nichts da Stream bereits umgeordnet ist

1. Fallen
2. IntStream stream = IntStream.of(1, 2);

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

Der Stream wird nach einem Durchlauf beendet

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

.forEach(System.out::println);

Stream läuft unendlich da i immer erhöht wird

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

.distinct() //.parallel()?

.limit(10)

.forEach(System.out::println);

Limit wird nicht erreicht, da %2 immer zwischen 0 und 1 Ergebnisse liefert

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

.boxed()

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

list.stream()

.peek(list::remove)

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

Während durchlauf wird Liste bearbeitet, was zu Fehlermeldungen führt

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