Java Streams and Fork Join

Contents

Java Advanced

Streams

filter()

Passt nur die Elemente weiter, die ein Predicat erfüllen:

```
people.stream().filter(Person::isStudent)
```

```
numbers.stream().filter(n -> n > 42)
```

Funktioniert auch mit komplexen Blöcken, am besten aber eine separate Methode schreiben, die ein bool zurückgibt und diese Methode in filter übergeben.

```
Person result2 = persons.stream()
    .filter(p -> {
      if ("jack".equals(p.getName()) && 20 == p.getAge()) {
        return true;
      }
        return false;
      }
).findAny().orElse(null);
```

Returns Stream

map()

Modifiziert alle Elemente des Streams

```
List<Integer> num = Arrays.asList(1,2,3,4,5);
List<Integer> collect1 = num.stream().map(n -> n *2)
     .collect(Collectors.toList());
System.out.println(collect1); //[2, 4, 6, 8, 10]
```

Returns Stream

reduce(funktion)

Analog zu dem Fold in Haskell: reduce(Akkumulator, Operation)

```
int result = numbers.stream().reduce(0, Integer::sum).get();
int sum = numbers.stream().reduce(0, (x,y) -> x + 5y).get();
int result = users.stream().reduce(0, (partialAgeResult, user) -> partialAgeResult
+ user.getAge()).get();
```

reduce liefert Optional zurück -> .get() am Ende!

findAny()

Ein Element aus Stream bekommen als Optional. Stream leer -> Optional leer.
“”In a non-parallel operation, it will most likely return the first element in the Stream but there is no guarantee for this.“”

Returns Optional

findFirst()

Das erste Element aus Stream bekommen als Optional. Stream leer -> Optional leer

Returns Optional

min(comparator), max(comparator), average(comparator)

“”Optional<T> max(Comparator<? super T> comparator)
Returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.

an Optional describing the maximum element of this stream, or an empty Optional if the stream is empty.””

Für Integer, Duble usw. soll man kein Comparator hinzufügen, sondern einfach max() verwenden:

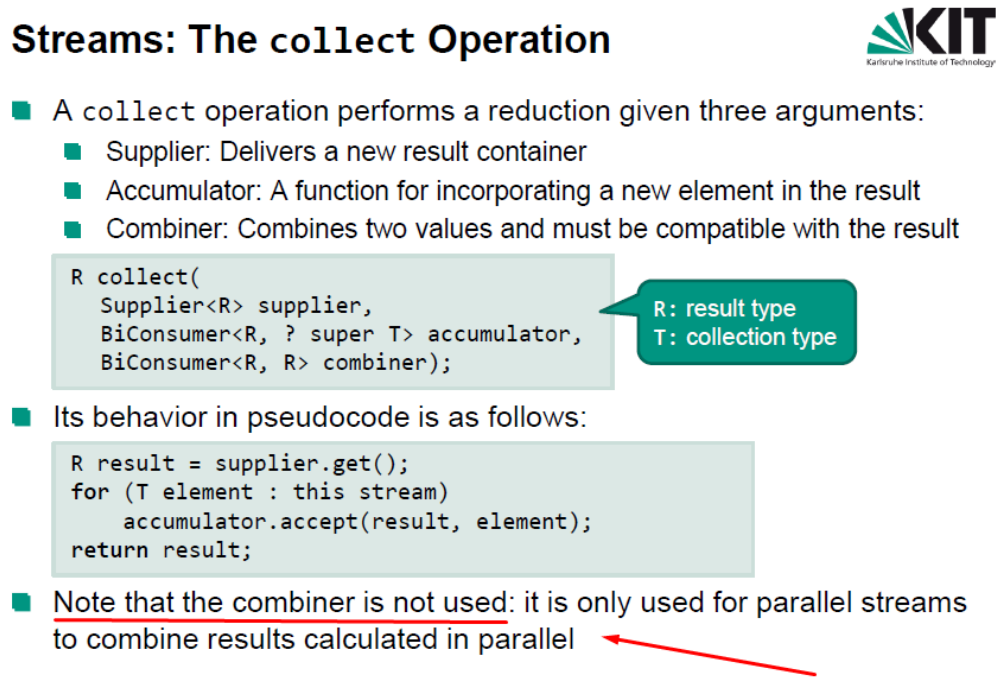
```
int max = numbers.stream().max() -> Funktioniert nur mit primitiven Datentypen;
```

Besser: reduce(Integer::max)

Returns Optional
Analog funktioniert es bei min() und average()

Gefährlich bei nicht-Primitiven, da man immer comparator eingeben muss!

collect()



Supplier: Null-Element der Operation
Accumulator: das neue Element hinzufügen und gleichzeitig Ergebnis aktualisieren
Combiner: Zusammenführung von Ergebnissen von parallelen Berechnungen

BiConsumer: hat 2 Werte und verknüpft sie

Streams: `collect` Operation Example

```
R collect(
   Supplier<R> supplier,
   BiConsumer<R, ? super T> accumulator,
   BiConsumer<R, R> combiner);
```

R: result type
T: collection type

- A collector for summing up the ages of all persons in a collection can be basically defined as follows:

```
personsInAuditorium.stream().collect(
   () -> 0,
   (currentSum, person) -> { currentSum += person.getAge(); },
   (leftSum, rightSum) -> { leftSum += rightSum; });
```

Combined value has to be assigned to the first input reference

- Nevertheless, an `Integer` value is immutable, so the operation will always return `0` → ensure that the accumulated value is immutable!

() -> 0 „Null-Element“
In currentSum wird Age von allen personen akkumulliert.
(leftSum, rightSum) -> ... einfach so, muss man nicht wissen

Beispiel für Streams

```
List<Person> personsInAuditorium = Arrays.asList(
   new Person(true, 18),
   new Person(false, 21),
   new Person(false, 19),
   new Person(true, 28));

double average = personsInAuditorium
   .stream()                  {(true, 18), (false, 21), (false, 19), (true, 28)}
   .filter(Person::isStudent) {(true, 18), (true, 28)}
   .mapToInt(Person::getAge)  {18, 28}
   .average()                 23
   .getAsDouble();            23d
```

Parallel Streams

- On parallel streams, retrieved by `parallelStream()`, certain operations are automatically executed in parallel
- The previous example can also be calculated in parallel:

```
double average = personsInAuditorium
        .parallelStream()
        .filter(p -> p.isStudent())
        .mapToInt(Person::getAge)
        .average()
        .getAsDouble();
```

Optional

Wichtigste Methoden

`boolean isPresent()`: Return true if there is a value present, otherwise false.

`T get()`: If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

`T orElse(alternative)`: if value is not present in Optional, returns the alternative, otherwise value from Optionnal

Alle Methoden

`static <T> Optional<T>`	**`empty()`** Returns an empty `Optional` instance.
`boolean`	**`equals(Object obj)`** Indicates whether some other object is "equal to" this Optional.
`Optional<T>`	**`filter(Predicate<? super T> predicate)`** If a value is present, and the value matches the given predicate, return an `Optional` describing the value, otherwise return an empty `Optional`.
`<U> Optional<U>`	**`flatMap(Function<? super T,Optional<U>> mapper)`** If a value is present, apply the provided `Optional`-bearing mapping function to it, return that result, otherwise return an empty `Optional`.
`T`	**`get()`** If a value is present in this `Optional`, returns the value, otherwise throws `NoSuchElementException`.
`int`	**`hashCode()`** Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
`void`	**`ifPresent(Consumer<? super T> consumer)`** If a value is present, invoke the specified consumer with the value, otherwise do nothing.
`boolean`	**`isPresent()`** Return `true` if there is a value present, otherwise `false`.
`<U> Optional<U>`	**`map(Function<? super T,? extends U> mapper)`** If a value is present, apply the provided mapping function to it, and if the result is non-null, return an `Optional` describing the result.
`static <T> Optional<T>`	**`of(T value)`** Returns an `Optional` with the specified present non-null value.
`static <T> Optional<T>`	**`ofNullable(T value)`** Returns an `Optional` describing the specified value, if non-null, otherwise returns an empty `Optional`.
`T`	**`orElse(T other)`** Return the value if present, otherwise return `other`.
`T`	**`orElseGet(Supplier<? extends T> other)`** Return the value if present, otherwise invoke `other` and return the result of that invocation.
`<X extends Throwable> T`	**`orElseThrow(Supplier<? extends X> exceptionSupplier)`** Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.
`String`	**`toString()`** Returns a non-empty string representation of this Optional suitable for debugging.

Fork Join

Fork-Join (1)

- Fork-Join is a pattern for effectively computing divide-and-conquer algorithms in parallel
  - Problems are solved by splitting them into subtasks, solving them in parallel and finally composing the results
  - General algorithm in pseudocode:

```
Result solve(Problem problem) {
   if (problem is small enough) {
      directly solve problem
   } else {
      split problem into independent parts
      fork new subtasks to solve each part
      join all subtasks
      compose results from subresults
   }
}
```

Fork-Join (3)

- A task returning a value has to override `RecursiveTask`:

```
public class MyTask extends RecursiveTask<Integer> {
   @Override
   public static Integer compute() {
      // calculate something and return if problem small

      MyTask leftTask = new MyTask(…);
      MyTask rightTask = new MyTask(…);

      leftTask.fork();

      rightTask.compute();

      leftTask.join();

      // compute result
}
```

Return type

Divide the problem

Fork one subtask

Execute other subtask in-place

Join the forked subtask

extends RecursiveTaks<Type>
@Override compute()

Fork-Join (2)

- Java provides a `ForkJoinPool` on which `ForkJoinTasks` can be executed
- Implementations of `ForkJoinTask` must override the `compute()` method
  - `RecursiveAction` (no result) and `RecursiveTask` (returns result) are concretizations of such tasks
  - They can be executed by a `ForkJoinPool` calling its `invoke()` method

- If `MyTask` is a `ForkJoinTask`, it can be invoked as follows:

```
…
ForkJoinPool fjPool = new ForkJoinPool();
MyTask myTask = new MyTask(…);
fjPool.invoke(myTask);
…
```

Beispiel

```
class DocumentSearchTask extends RecursiveTask<Long> {
...
    @Override
    protected Long compute() {
      ...
      return occurrencesCount(document, searchedWord);
    }
}

ForkJoinPool forkJoinPool = new ForkJointPool();
DocumentSearchTask customRecursiveTask = new DocumentSearchTask(...)
long result = forkJoinPool.invoke(customRecursiveTask);
```

Thread-safe Classes

- The Java API provides several thread-safe classes, which can be safely used by multiple threads concurrently
- `BlockingQueue` (interface)
  - Queue with operations that block if queue is full/empty when putting/retrieving an element
  - Methods:
    - `put(…)` blocks if full
    - `take()` blocks if empty
  - Concrete implementations (extract):
    - `ArrayBlockingQueue`: limited capacity
    - `LinkedBlockingQueue`: optionally limited capacity
    - `PriorityBlockingQueue`: sorted
- `ConcurrentHashMap`
- Concurrently accessing non-thread-safe classes should be avoided (memory inconsistencies)