Java Streams and Fork Join

Contents

[Java Advanced 1](#_Toc68456774)

[Streams 1](#_Toc68456775)

[filter() 1](#_Toc68456776)

[map() 2](#_Toc68456777)

[reduce() 2](#_Toc68456778)

[findAny() 2](#_Toc68456779)

[findFirst() 2](#_Toc68456780)

[min(), max(), average() 3](#_Toc68456781)

[collect() 3](#_Toc68456782)

[Beispiel für Streams 4](#_Toc68456783)

[Parallel Streams 5](#_Toc68456784)

[Optional 5](#_Toc68456785)

[Wichtigste Methoden 5](#_Toc68456786)

[Alle Methoden 6](#_Toc68456787)

[Fork Join 7](#_Toc68456788)

[Beispiel 8](#_Toc68456789)

[Thread-Safe Classes 9](#_Toc68456790)

[Aufgaben 9](#_Toc68456791)

# 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()

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(), max(), average()

“”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.””

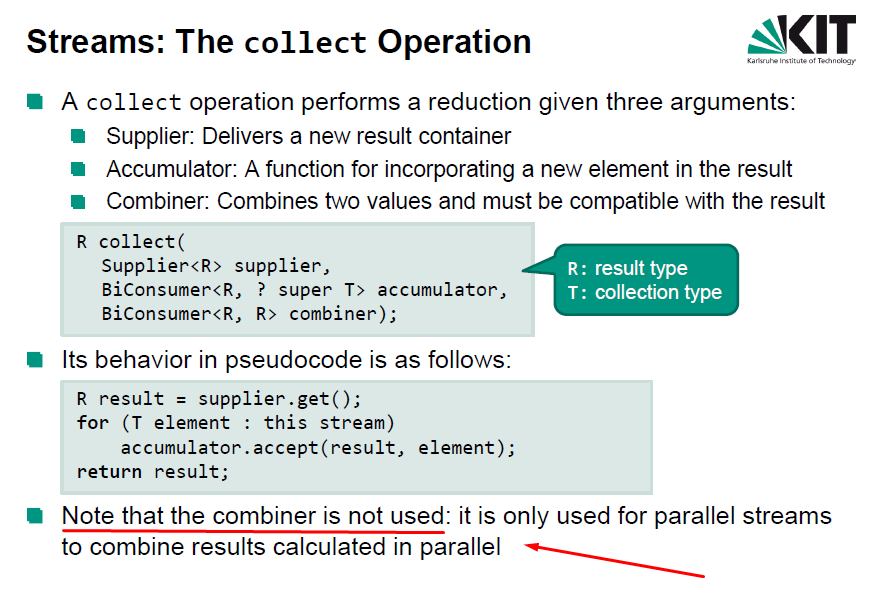
Für Integer, Duble usw. soll man kein Comparator hinzufügen, sondern einfach max() verwenden:

int max = numbers.stream().max();

**Returns Optional**

**Analog funktioniert es bei min() und average()**

### collect()

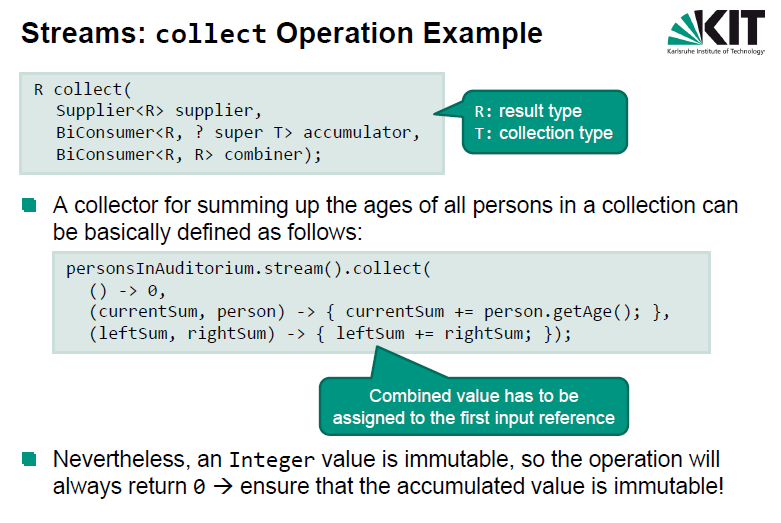


Supplier: Null-Element der Operation

Accumulator: das neue Element hinzufügen und gleichzeitig Ergebnis aktualisieren

Combiner: Zusammenführung von Ergebnissen von parallelen Berechnungen

BiConsumer: hat 2 Werte und verknüpft sie

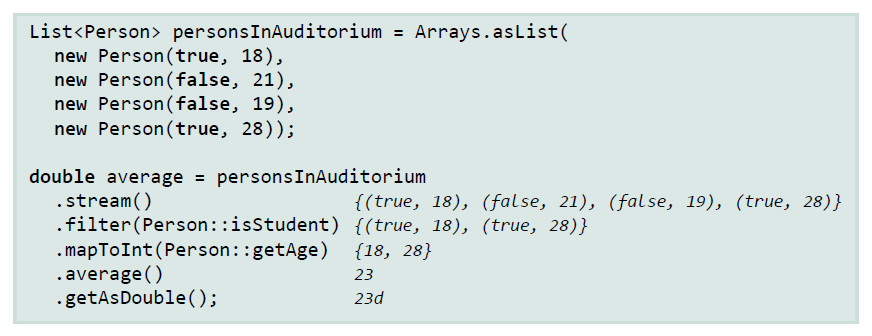


() -> 0 „Null-Element“

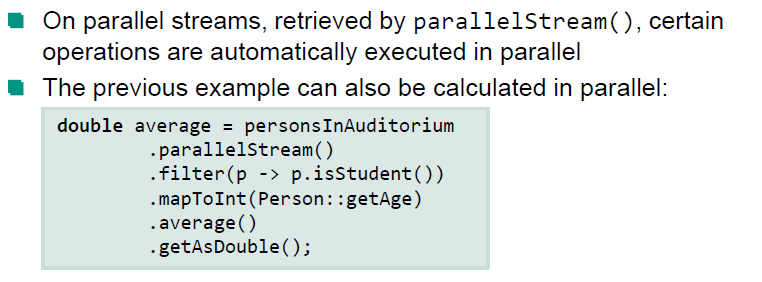
In currentSum wird Age von allen personen akkumulliert.

(leftSum, rightSum) -> ... einfach so, muss man nicht wissen

### Beispiel für Streams



## Parallel Streams



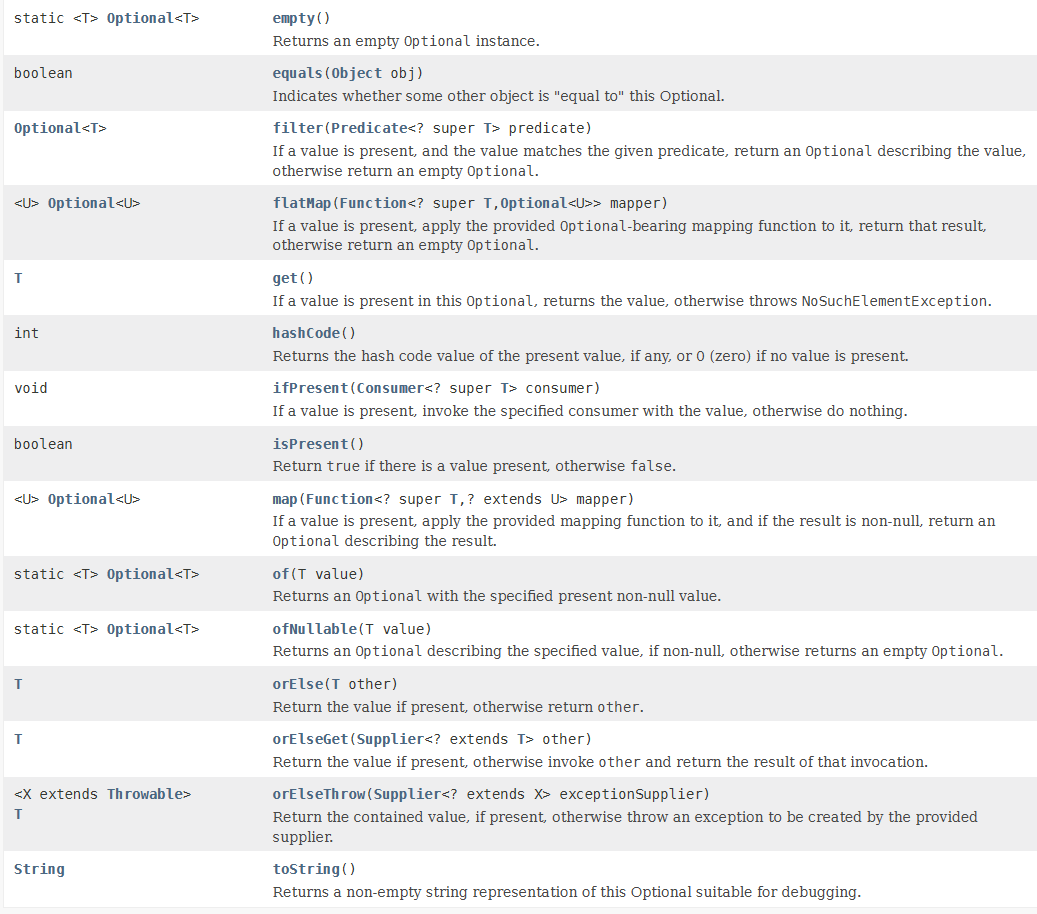
## Optional

### Wichtigste Methoden

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

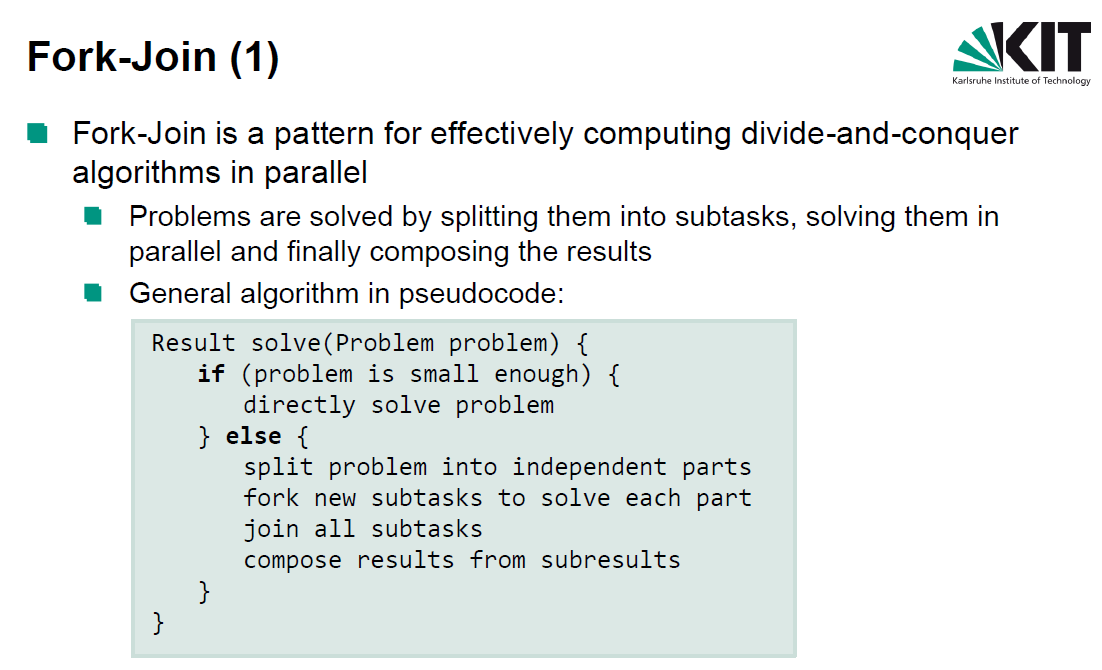
T get(): If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

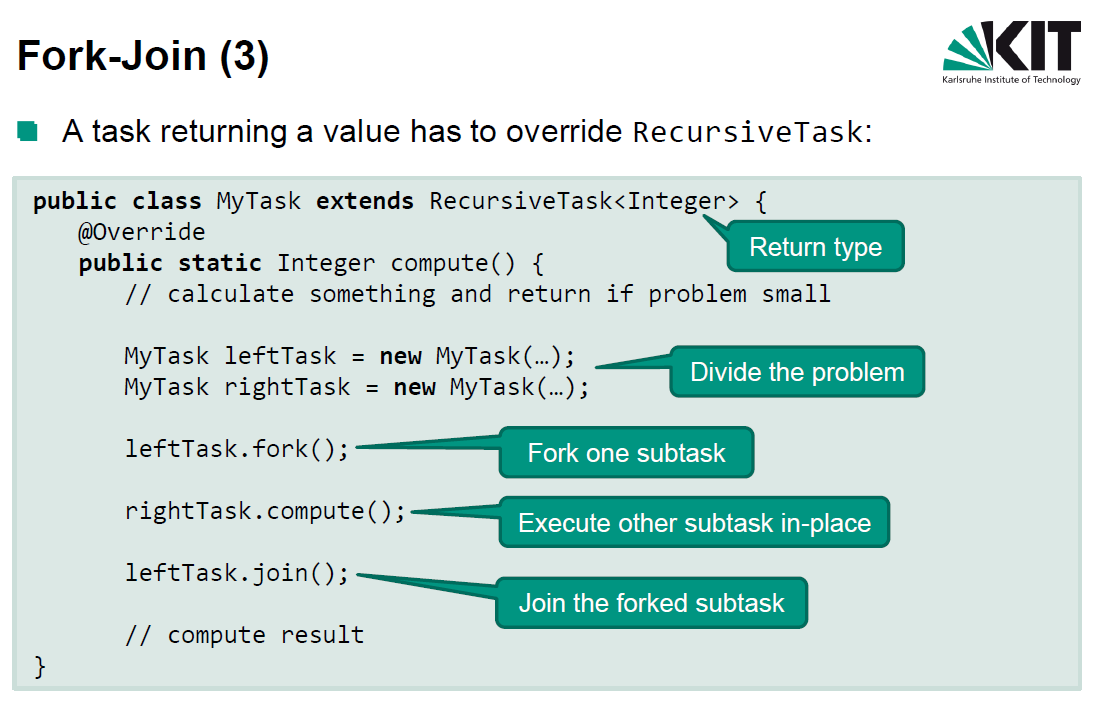
### Alle Methoden



|  |
| --- |
|  |

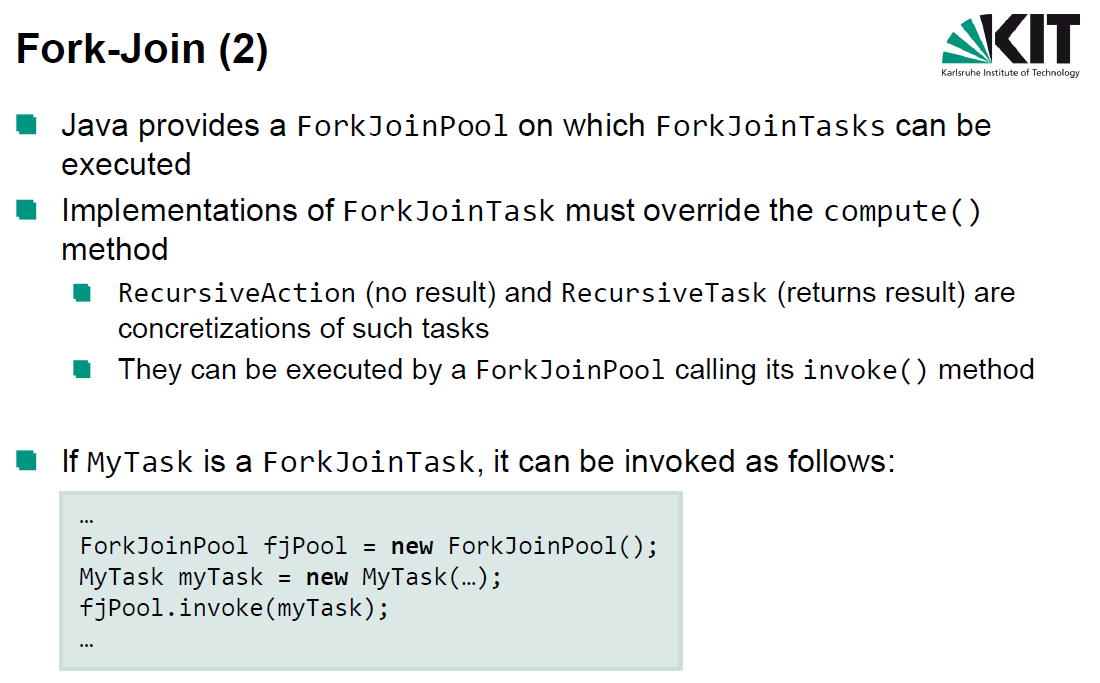
# Fork Join





**extends RecursiveTaks<Type>**

**@Override compute()**



### Beispiel

class DocumentSearchTask extends RecursiveTask<Long> {

...

@Override

protected Long compute() {

...

return occurrencesCount(document, searchedWord);

}

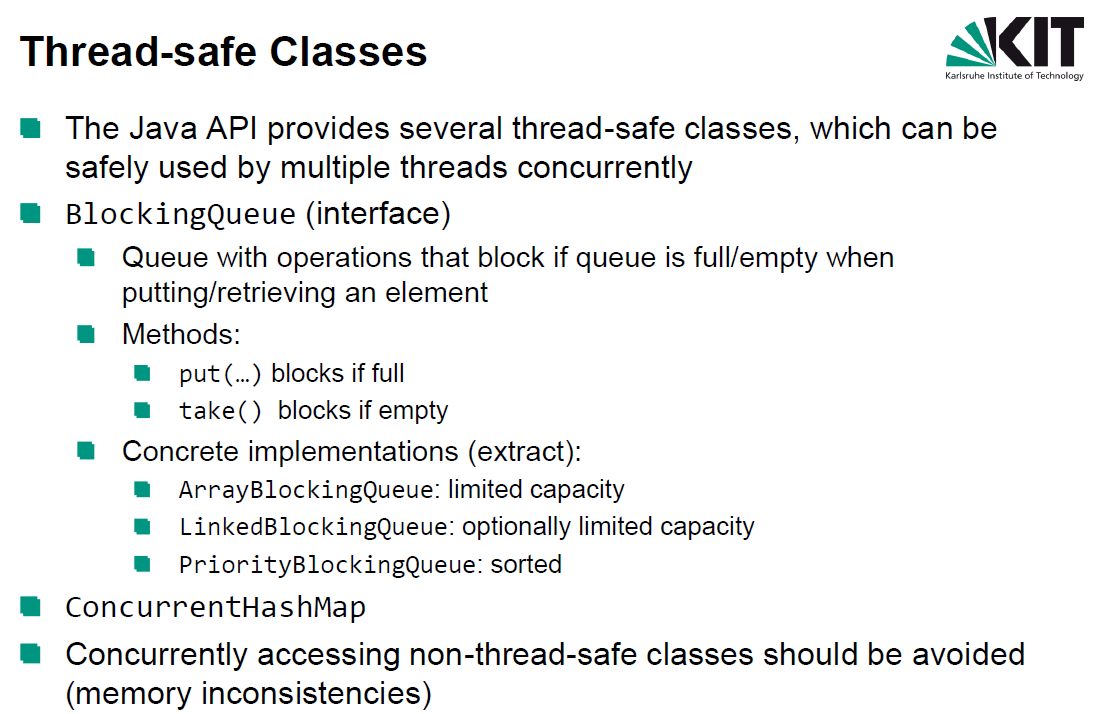
}

ForkJoinPool forkJoinPool = new ForkJointPool();

DocumentSearchTask customRecursiveTask = new DocumentSearchTask(...)

long result = forkJoinPool.invoke(customRecursiveTask);

### Thread-Safe Classes



# Aufgaben