Java

Contents

[Parallelprogrammierung 2](#_Toc67326418)

[Amdahlsches Gesetz 2](#_Toc67326419)

[Memory Consistency Error 2](#_Toc67326420)

[Caching 2](#_Toc67326421)

[Reordering 3](#_Toc67326422)

[Happens-before-Beziehung 3](#_Toc67326423)

[Schlüsselwort volatile 4](#_Toc67326424)

[Executor 5](#_Toc67326425)

[ExecutorService 6](#_Toc67326426)

[submit 6](#_Toc67326427)

[Futures 7](#_Toc67326428)

[Berechnung mit ExecutorService und Futures 8](#_Toc67326429)

[Vorgehensweise 8](#_Toc67326430)

[Complex Case 9](#_Toc67326431)

[ScheduledExecutorService 9](#_Toc67326432)

[Java Advanced 10](#_Toc67326433)

[Streams 10](#_Toc67326434)

[filter 10](#_Toc67326435)

[map 10](#_Toc67326436)

[reduce 10](#_Toc67326437)

[findAny 11](#_Toc67326438)

[findFirst 11](#_Toc67326439)

[min, max, average 11](#_Toc67326440)

[collect 11](#_Toc67326441)

[Beispiel für Streams 12](#_Toc67326442)

[Parallel Streams 13](#_Toc67326443)

[Optional 13](#_Toc67326444)

[Wichtigste Methoden 13](#_Toc67326445)

[Alle Methoden 14](#_Toc67326446)

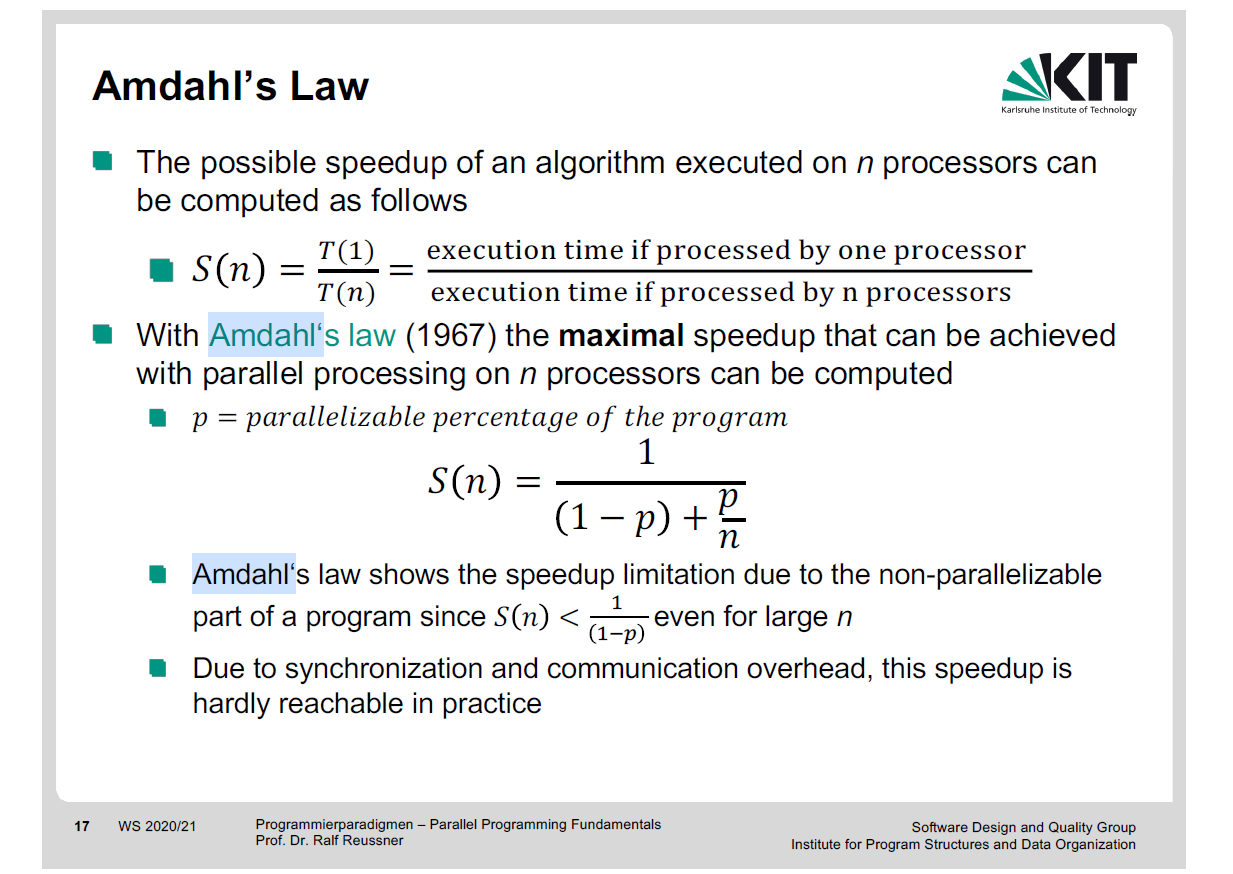
[Design by Contract 14](#_Toc67326447)

[Aufgaben 14](#_Toc67326448)

[Amdahlsches Gesetz aus Ü9 14](#_Toc67326449)

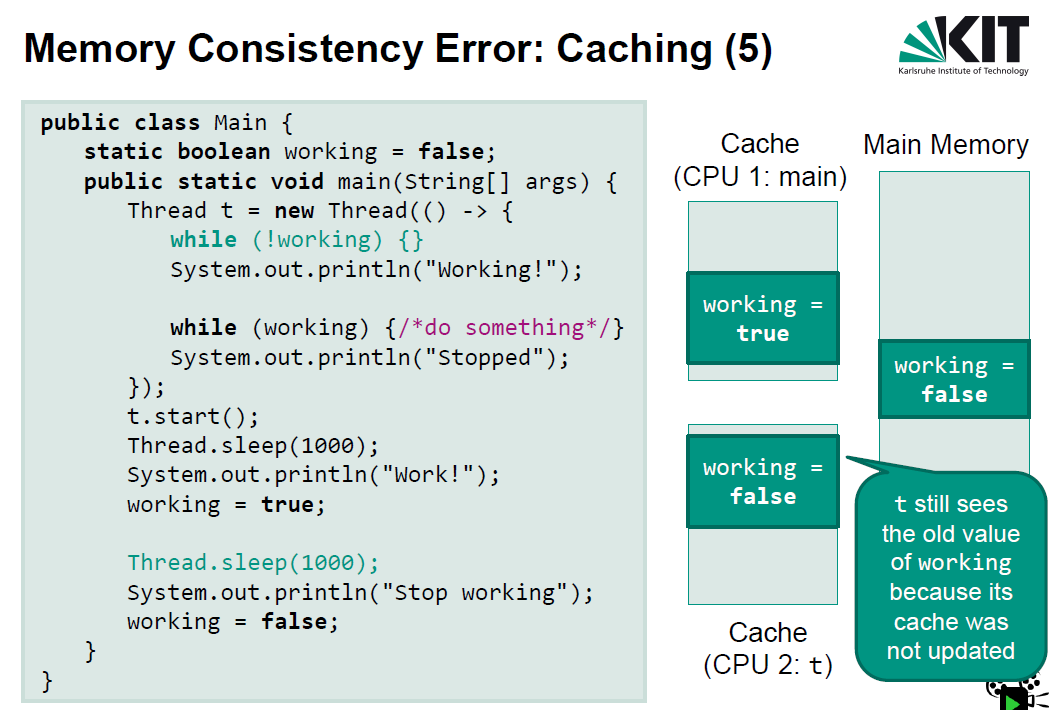
# Parallelprogrammierung

## Amdahlsches Gesetz

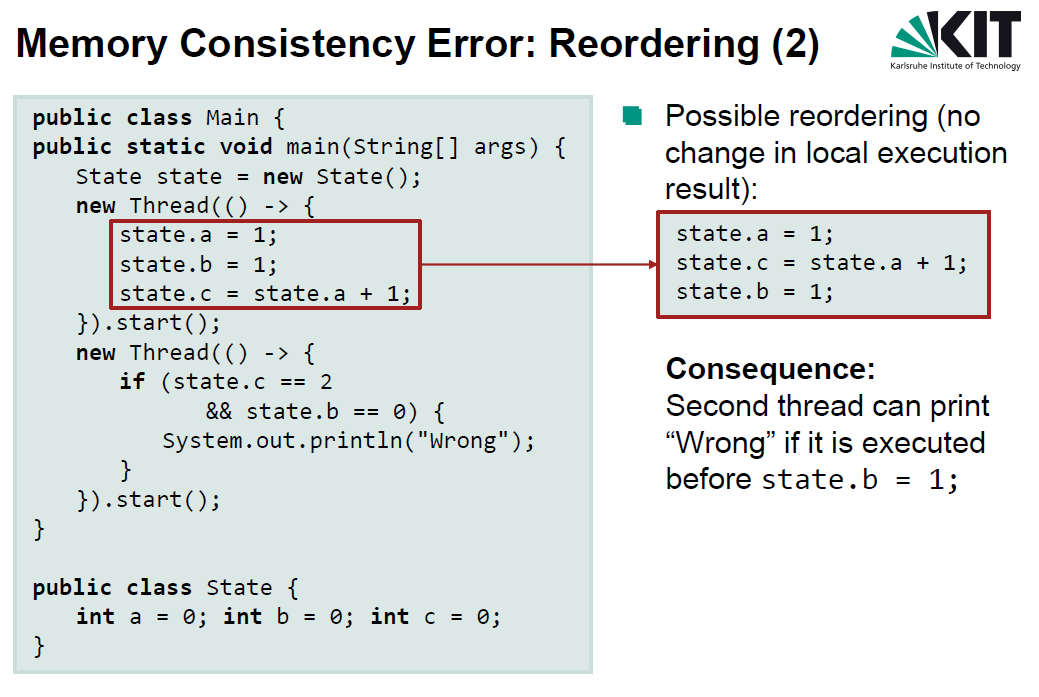


## Memory Consistency Error

### Caching

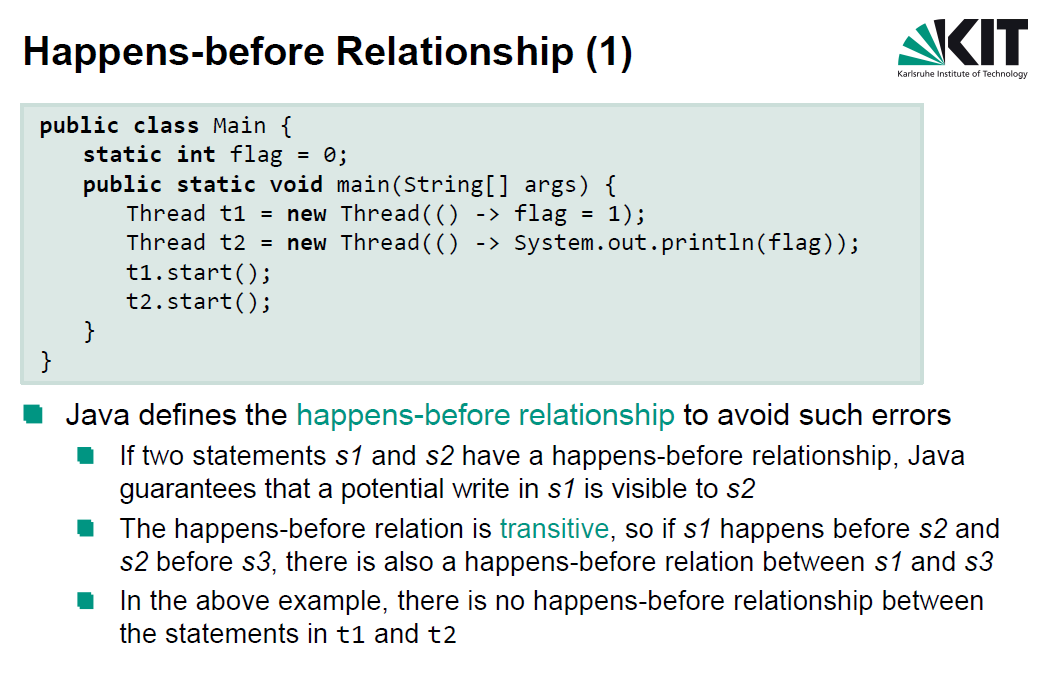


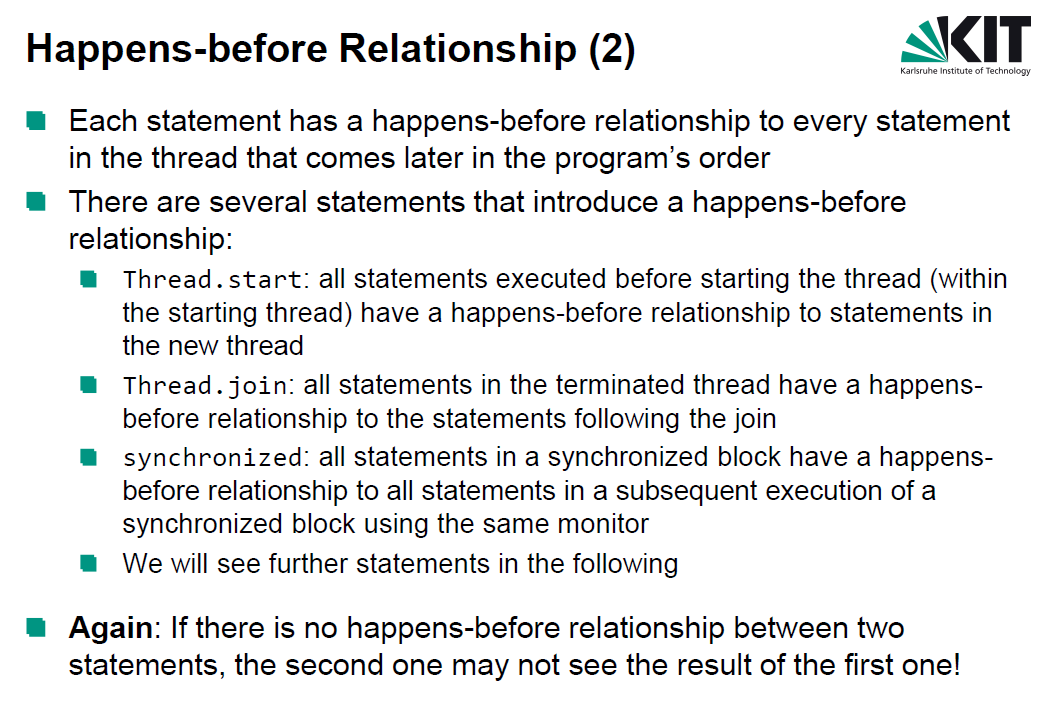
### Reordering



## Happens-before-Beziehung

Hilft, die Memory Consistency Errors wegen Reordering zu vermeiden



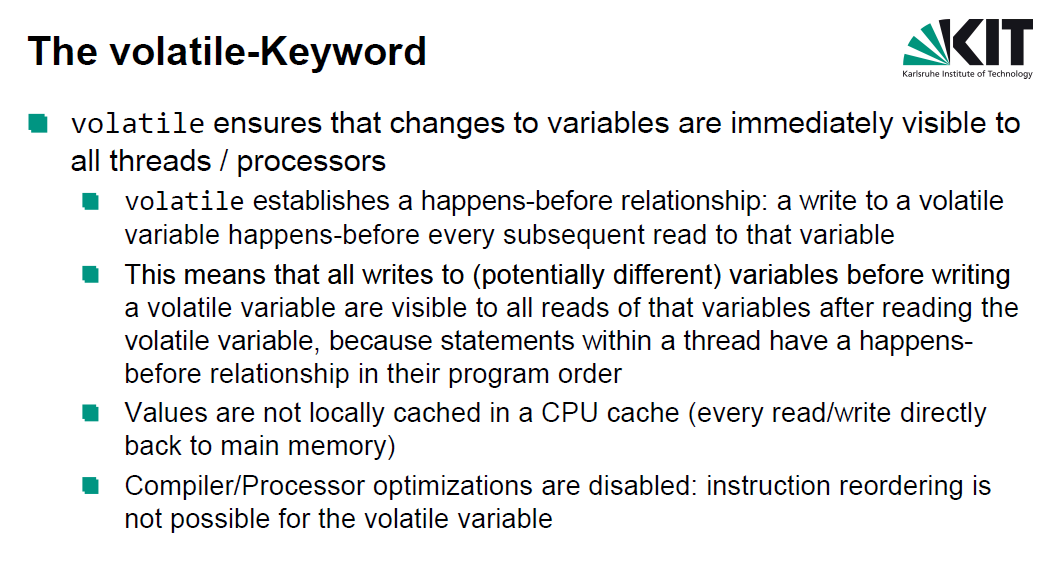


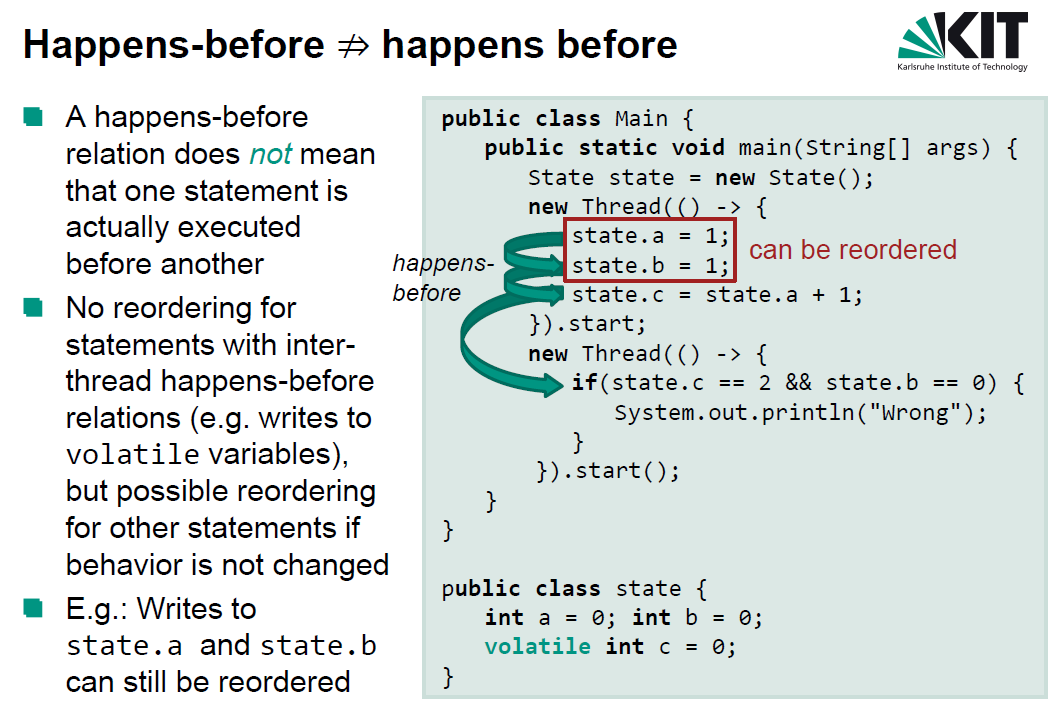
## Schlüsselwort volatile

Volatile Variable wird immer in RAM gespeichert und nicht in Cache

Keine Reordering für volatile

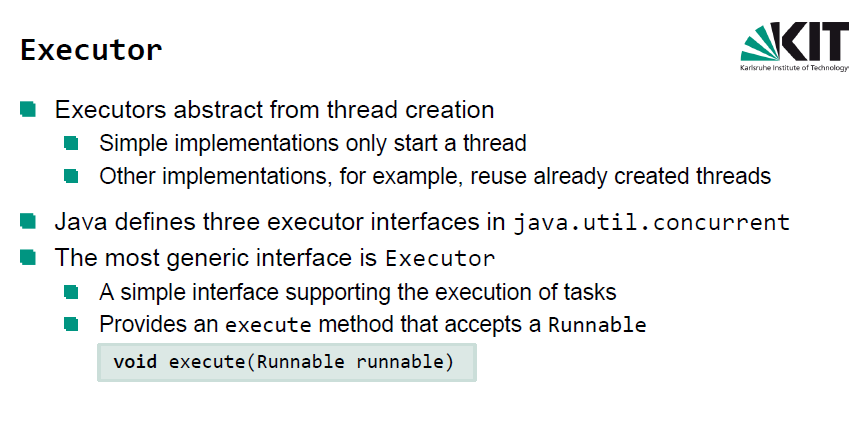
Hilft, die Memory Consistency Error wegen Caching (Variable in RAM) und Reordering (happens-before-Beziehung) zu vermeiden



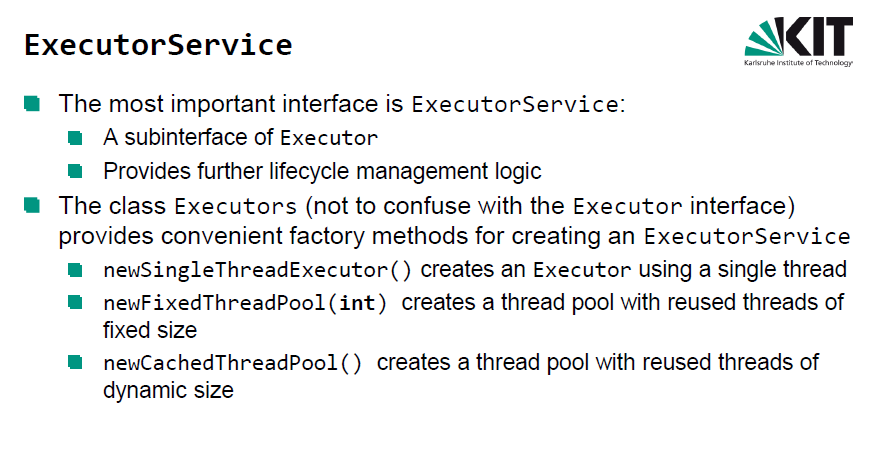


Executor

Executor ist ein Woker von ExecutorService

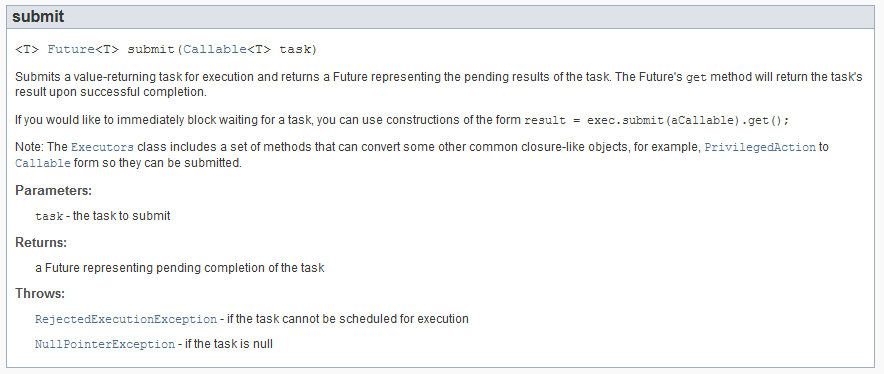


## ExecutorService



### submit

Future <T> submit(task):



task == lambda Funktion

Verwendung mit Lambda:

service.submit(() -> calculateX(elementIdx));

gibt Future<Integer> zurück, der später mit future.get() in einer Integer Variable geschrieben sein kann.

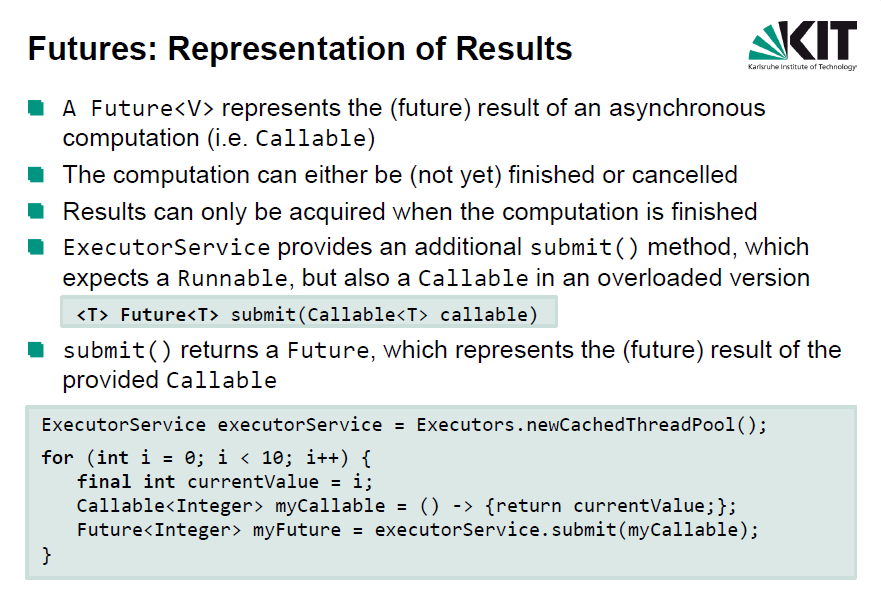
calculateX(elementIdx) ist eine Methode, die ein Integer zurückgibt.

Fututre<Integer> future = service.submit(() -> calculateX(elementIdx));

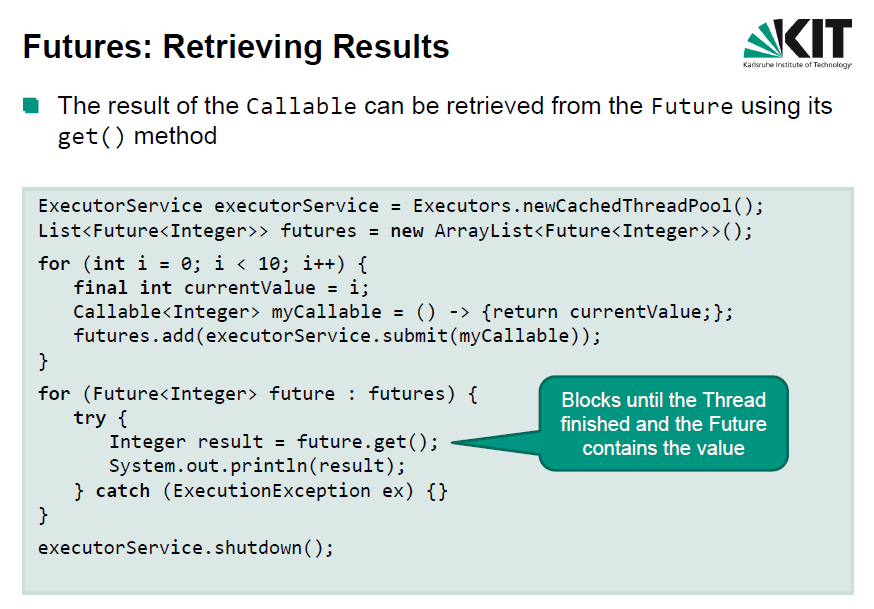
...

Integer result = future.get();

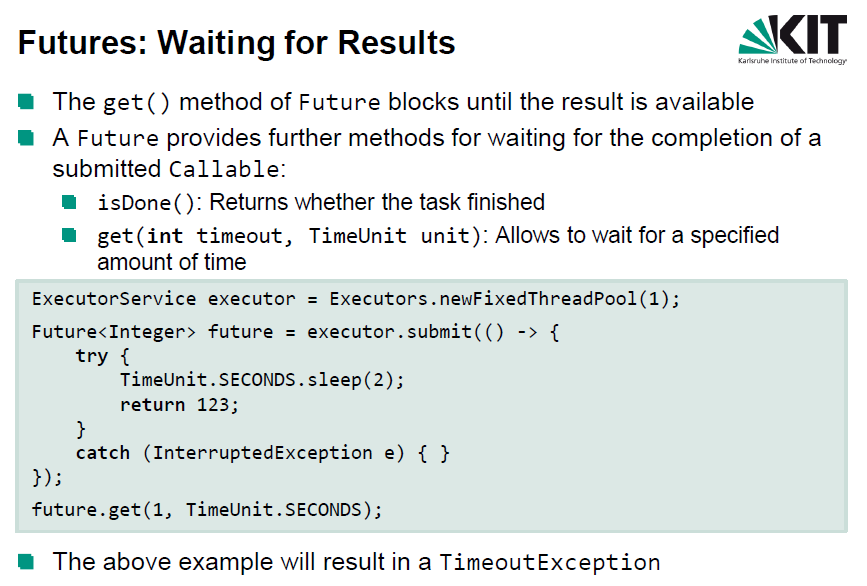
## Futures



get() blockiert Thread



get(30, TimeUnit.SECONDS) -> warte max 30 Sekunden auf eine Antwort. Falls keine -> TimeoutException



## Berechnung mit ExecutorService und Futures

### Vorgehensweise

1. Zuerst mus man ein ExecutorService erstellen und Anzahl von Executors einstellen.

ExecutorService service = Executors.newFixedThreadPool(amountThreads);

2. Dann wird eine Liste von Futures erstellt

List<Future<Integer>> futures = new ArrayList<Future<Integer>>();

3. Iterire über Eingabedaten und fülle die Liste von Futures.

**Wichtig**: Anzahl der Iterationen (range von i) == Anzahl der Threads

for (int i = 0; i < input.size(); i += i) {

final int elemeintIdx = i;

futures.add(service.submit(() -> calculateX(elementIdx)));

}

4. Dann iteriere über liste von Futures und berechne das Programmergebnis:

int count = 0;

for(Future<Integer> future: futures) {

count += future.get();

}

### Complex Case

final int target = Integer.parseInt(args[0]);

final int amountThreads = Integer.parseInt(args[1]);

ExecutorService service = Executors.newFixedThreadPool(amountThreads);

for (int i = SEARCH\_BEGIN; i < target; i += BLOCK\_SIZE) {

final int from = i;

final int until = i + BLOCK\_SIZE;

futures.add(service.submit(() -> countPrimes(from, until)));

}

int count = 0;

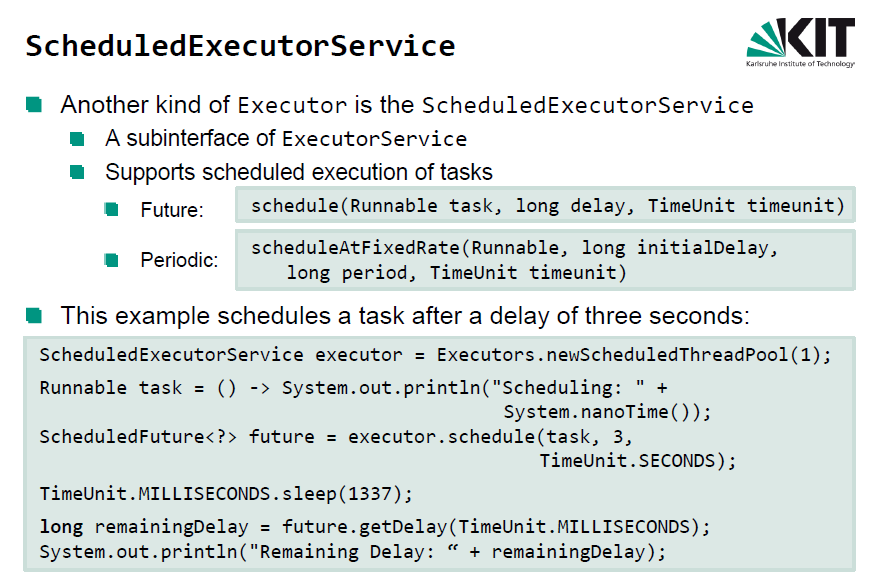
for(Future<Integer> future: futures) {

count += future.get();

}

## ScheduledExecutorService

ExecutorService mit Scheduling-Mechanismus



## CompletableFutures

TODO

# 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);

### 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]

### reduce

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

int result = numbers.stream().reduce(0, Integer::sum);

int sum = numbers.stream().reduce(0, (x,y) -> x + 5y);

int result = users.stream().reduce(0, (partialAgeResult, user) -> partialAgeResult + user.getAge());

### 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.“”

### findFirst

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

### 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.

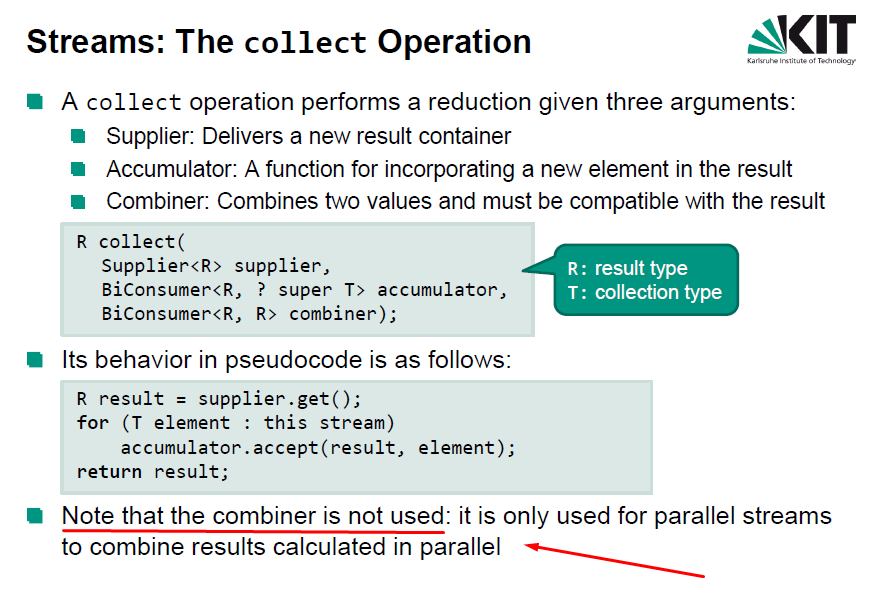
an Optional describing the maximum element of this stream, or an empty Optional if the stream is empty.””

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

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

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

### collect

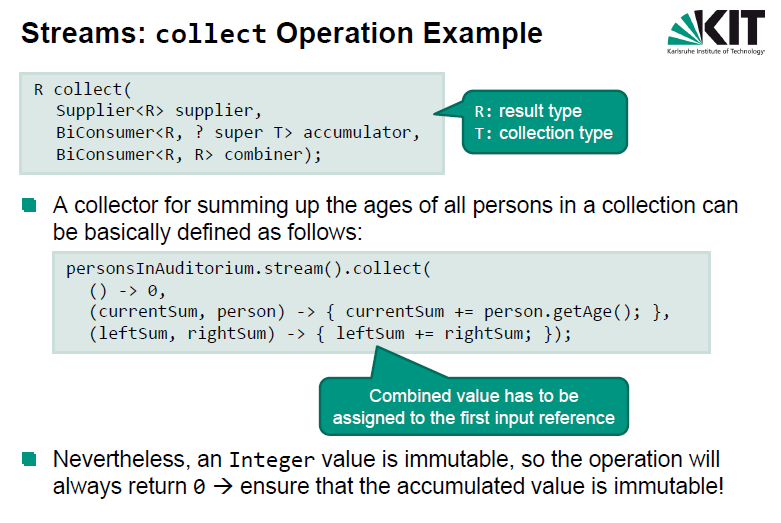


Supplier: Null-Element der Operation

Accumulator: das neue Element hinzufügen und gleichzeitig Ergebnis aktualisieren

Combiner: Zusammenführung von Ergebnissen von parallelen Berechnungen

BiConsumer: hat 2 Werte und verknüpft sie

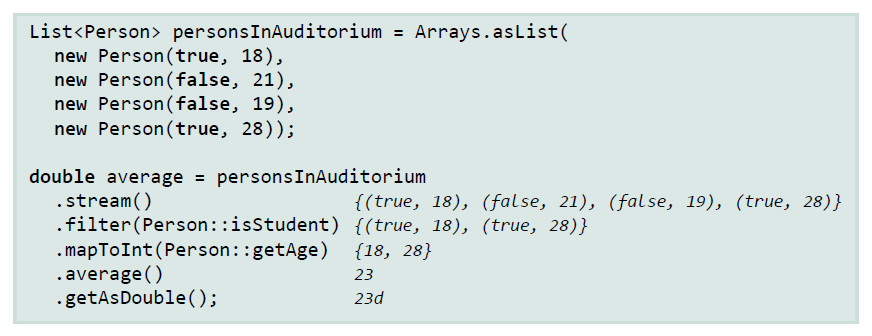


() -> 0 „Null-Element“

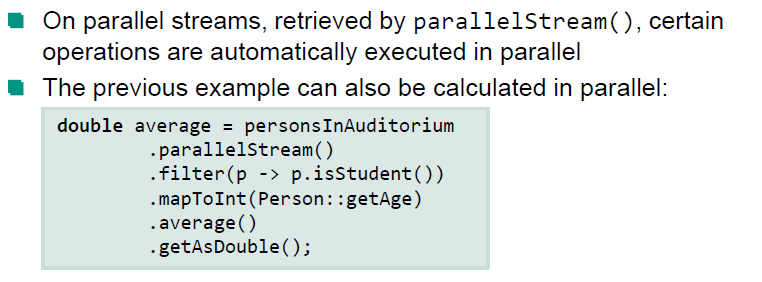
In currentSum wird Age von allen personen akkumulliert.

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

### Beispiel für Streams



## Parallel Streams



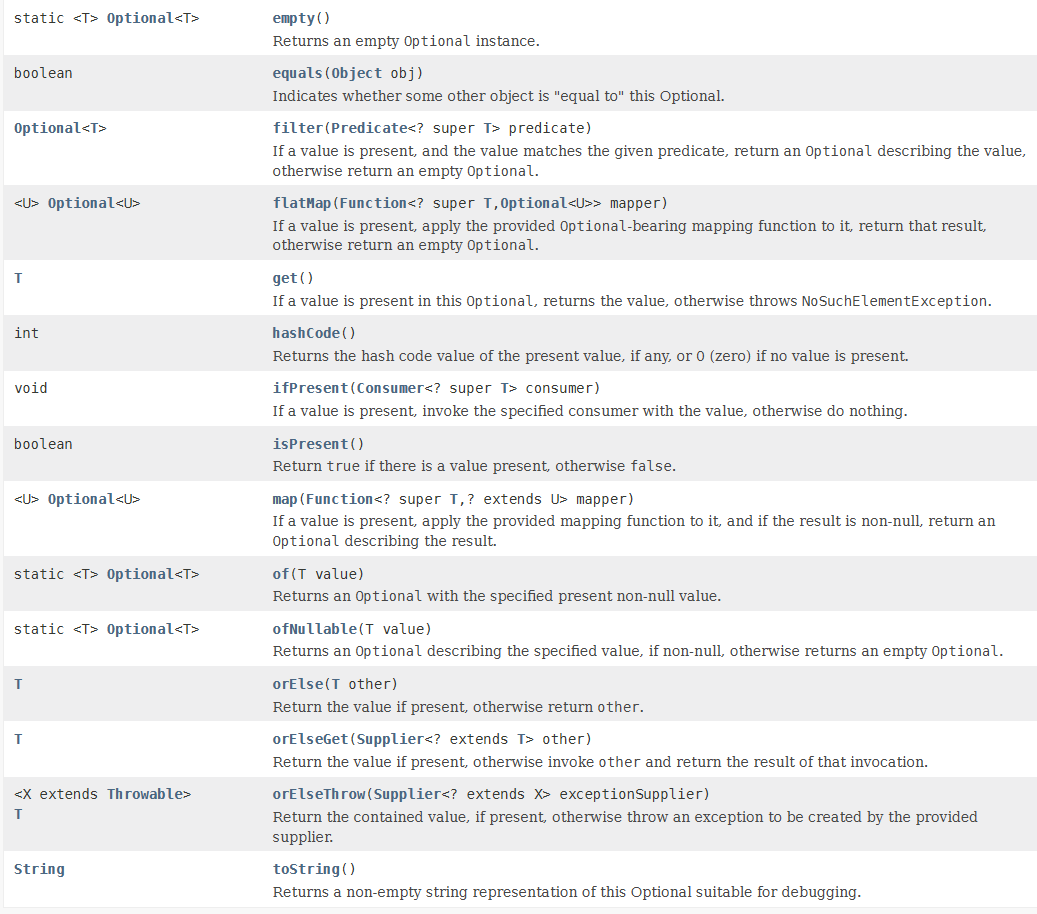
## Optional

### 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



|  |
| --- |
|  |

# Actor Model

# Design by Contract

# Aufgaben

## Amdahlsches Gesetz aus Ü9

Aufgabe: geg. Thread Pool. Jeder Leser und Schreiber werden repräsentiert durch einen Thread. 90% Threads sind Leser, 10% sind Schreiber.

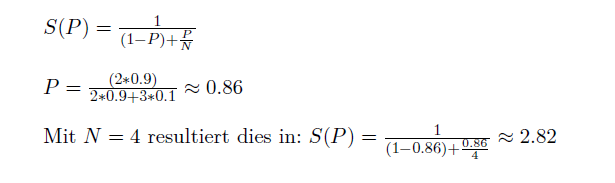
Leser sind nicht blockierend, benötigt 2 Sekunden

Schreiber ist blockierend für alle Schreiber und Lese, benötigt 3 Sekunden.

Lösung:

*P*: Anteil eines Programms, der parallelisiert werden kann

*N*: Anzahl der Prozessoren



P = P(Leser)/P(Gesamt)