Java Parallelprogrammierung

Contents

[Parallelprogrammierung 2](#_Toc68456564)

[Amdahlsches Gesetz 2](#_Toc68456565)

[Memory Consistency Error 2](#_Toc68456566)

[Caching 2](#_Toc68456567)

[Reordering 3](#_Toc68456568)

[Happens-before-Beziehung 3](#_Toc68456569)

[Schlüsselwort volatile 4](#_Toc68456570)

[Executor 5](#_Toc68456571)

[ExecutorService 6](#_Toc68456572)

[submit 6](#_Toc68456573)

[Futures 6](#_Toc68456574)

[Berechnung mit ExecutorService und Futures 8](#_Toc68456575)

[Vorgehensweise 8](#_Toc68456576)

[Complex Case 9](#_Toc68456577)

[ScheduledExecutorService 9](#_Toc68456578)

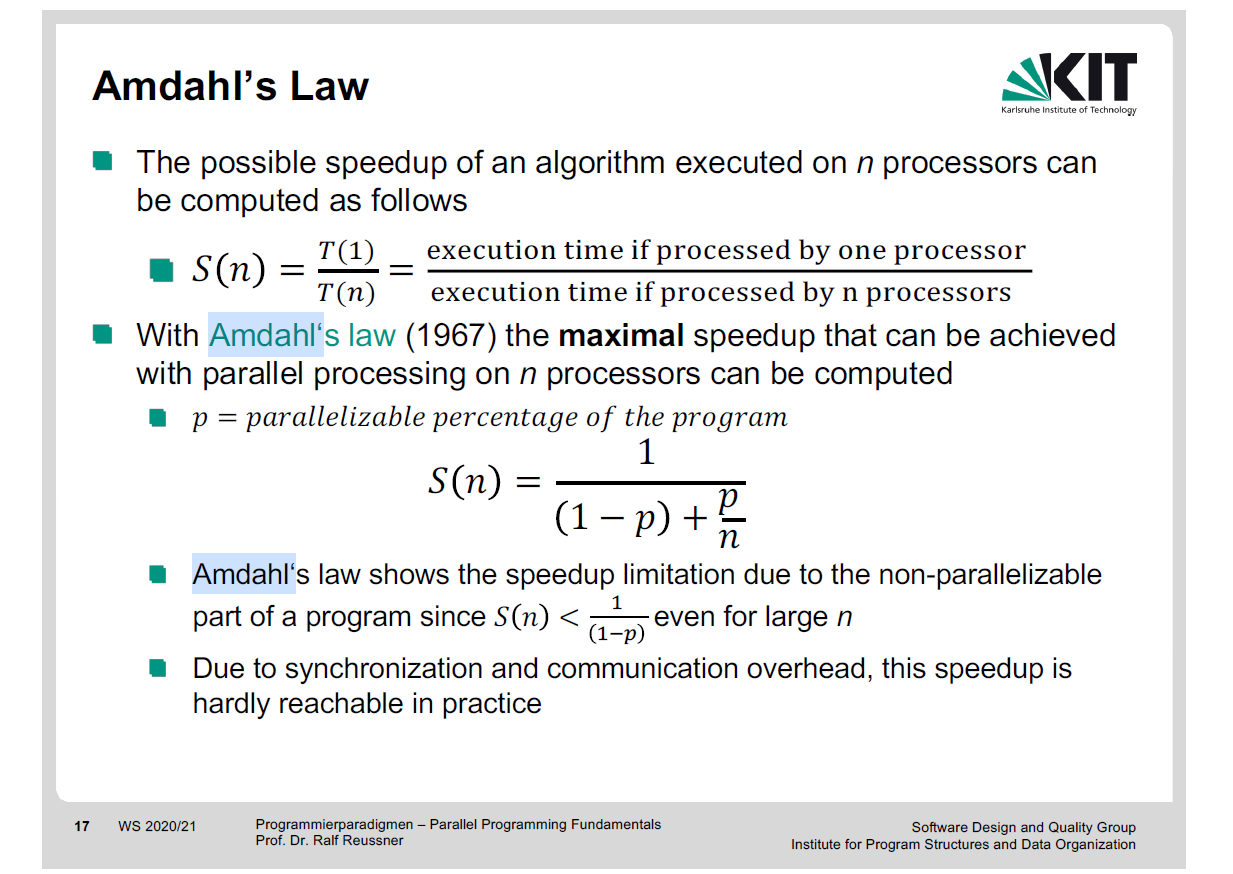
[CompletableFutures 9](#_Toc68456579)

[Aufgaben 10](#_Toc68456580)

[Amdahlsches Gesetz aus Ü9 10](#_Toc68456581)

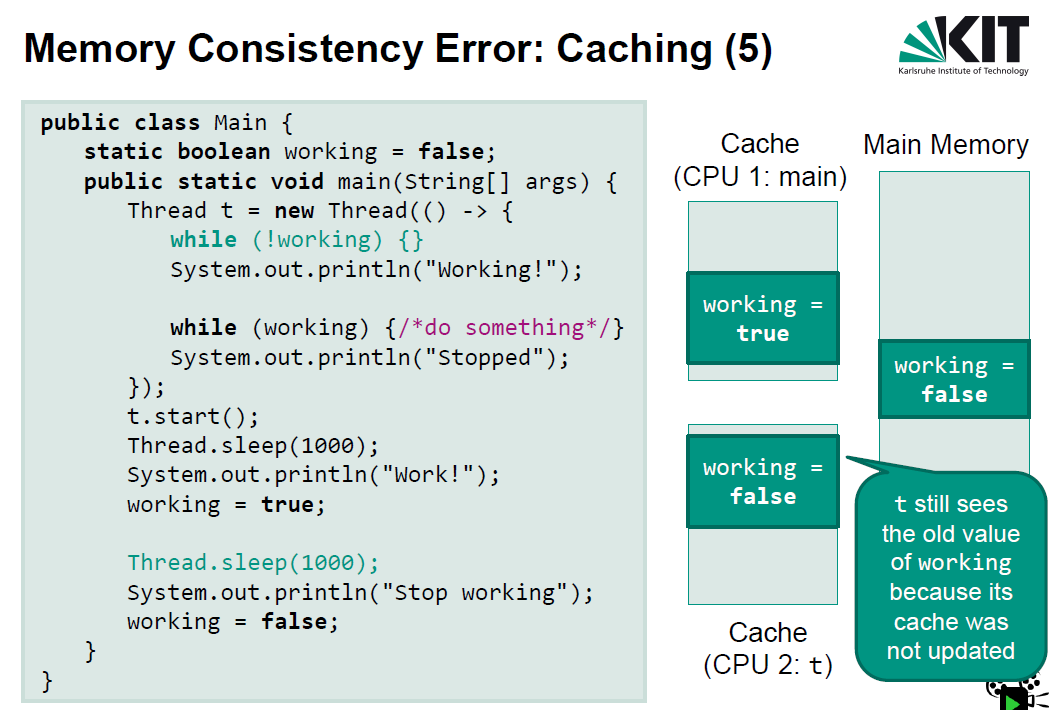
# Parallelprogrammierung

## Amdahlsches Gesetz

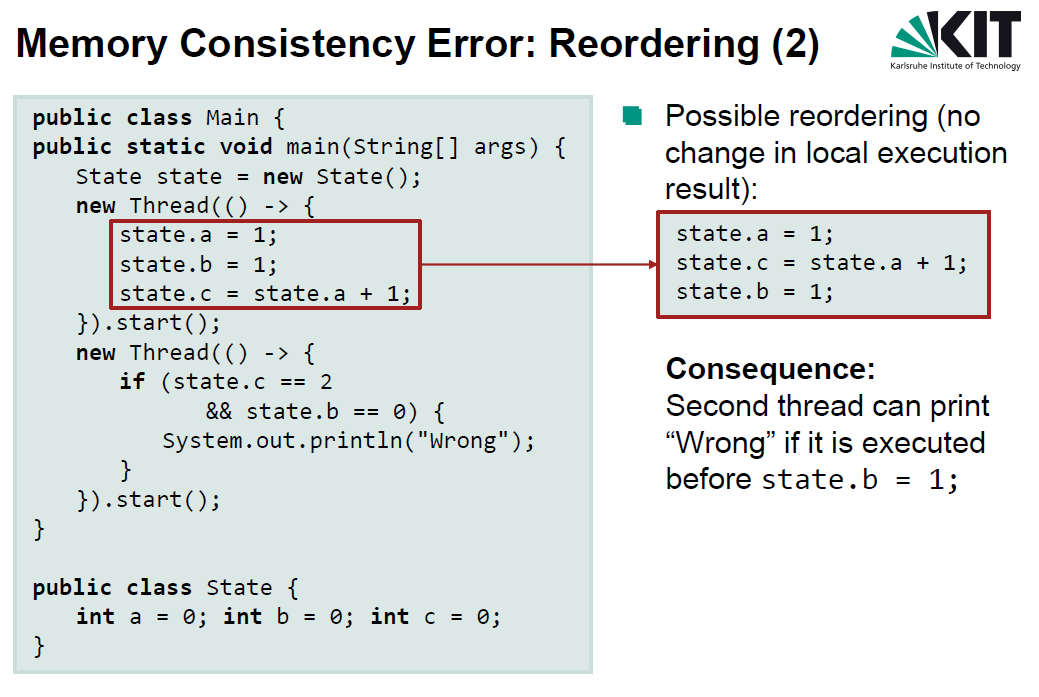


## Memory Consistency Error

### Caching

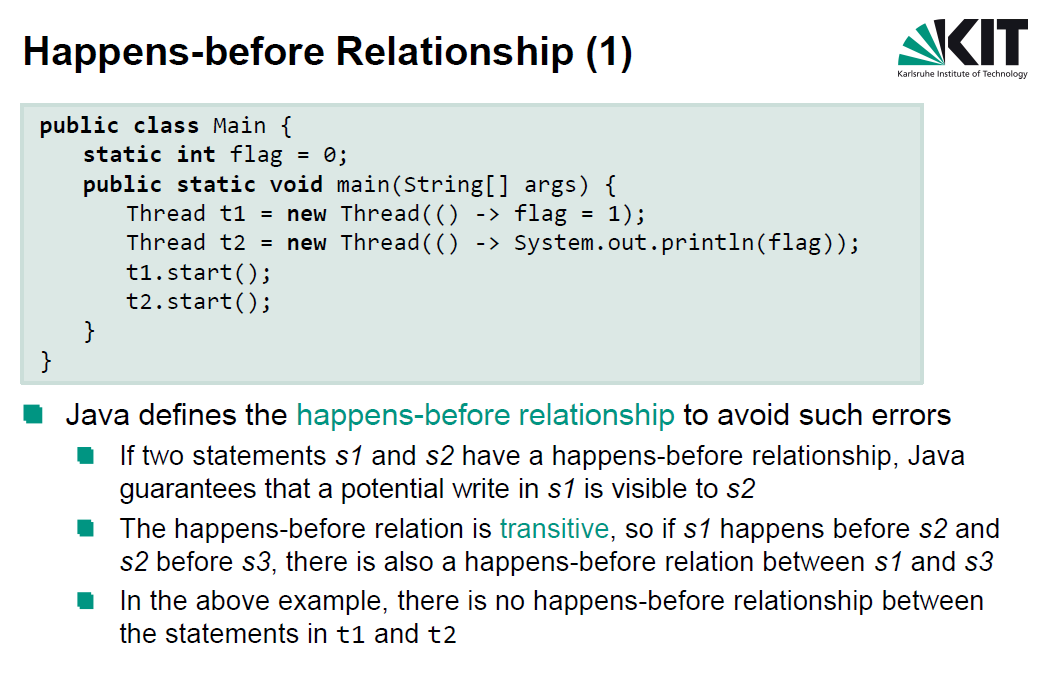


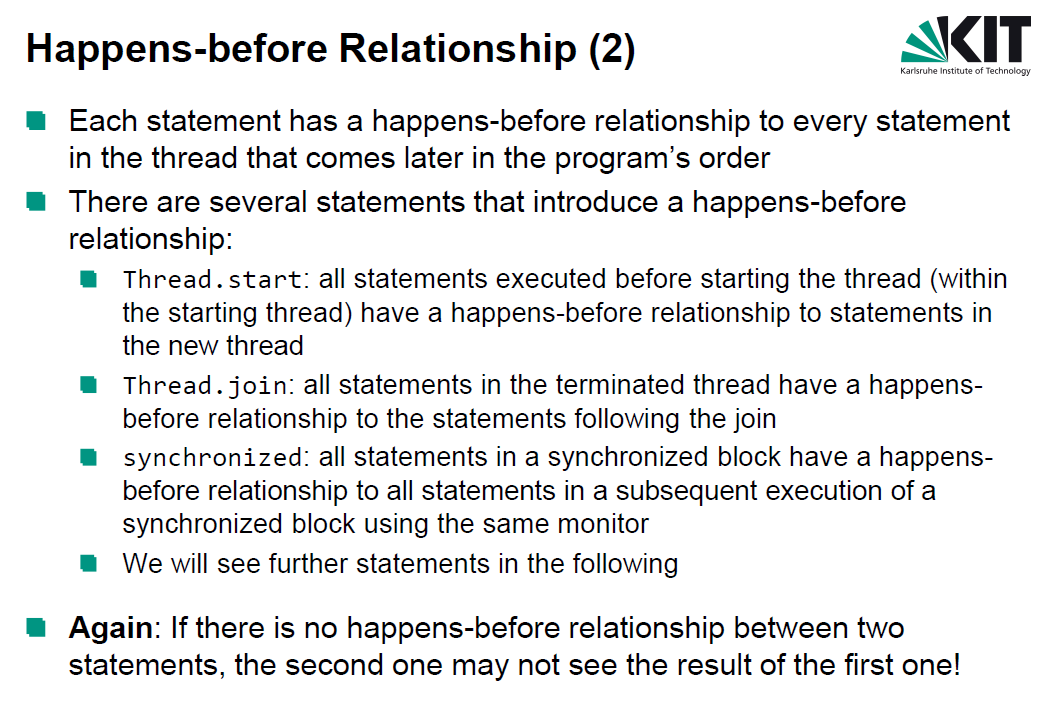
### Reordering



## Happens-before-Beziehung

Hilft, die Memory Consistency Errors wegen Reordering zu vermeiden



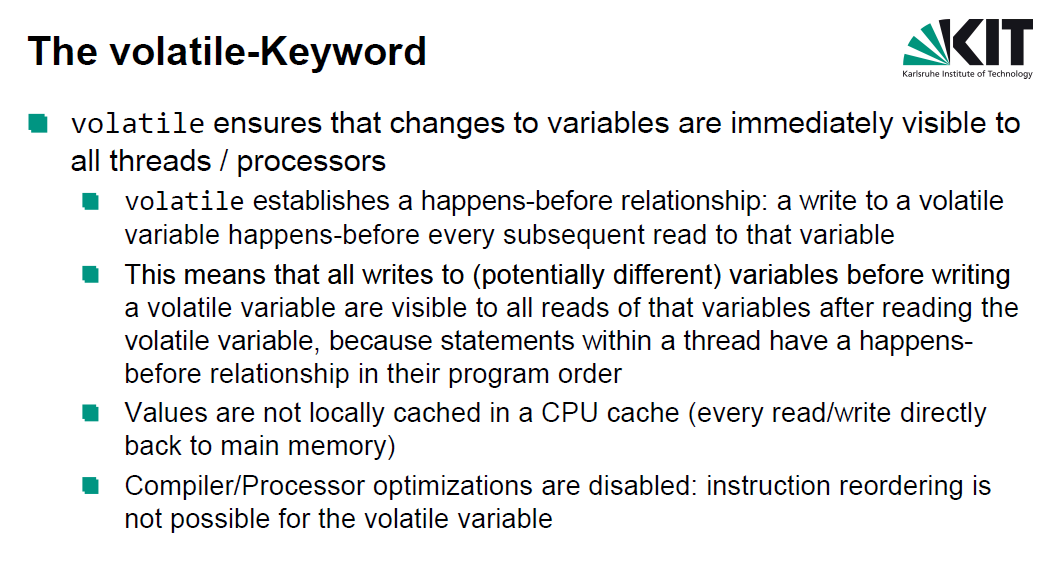


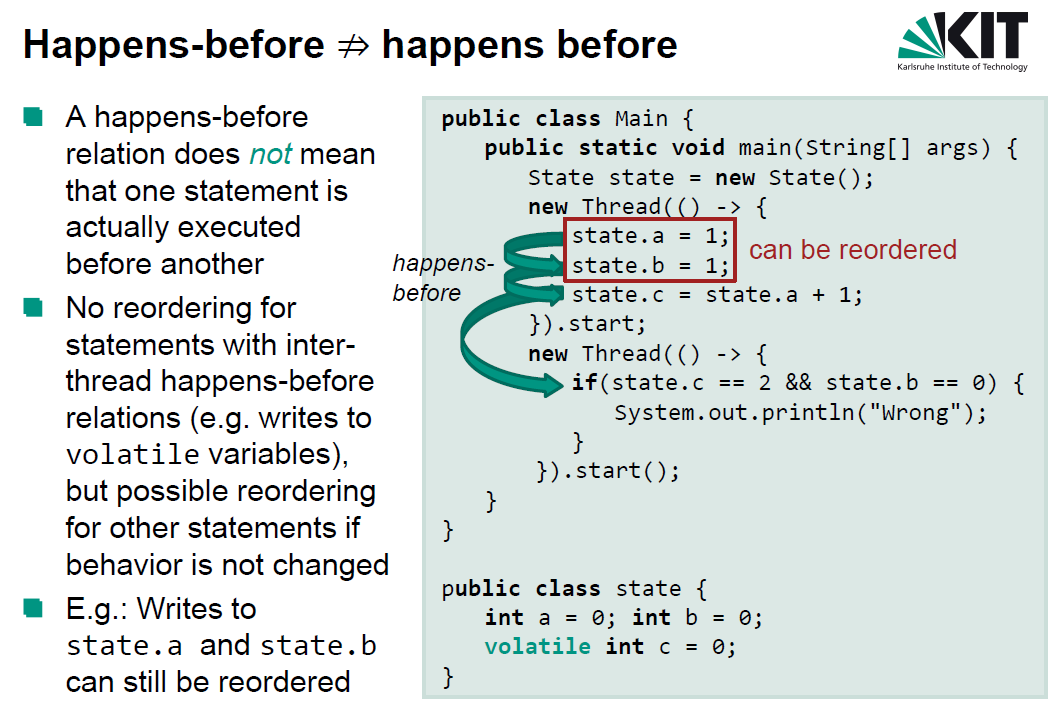
## Schlüsselwort volatile

Volatile Variable wird immer in RAM gespeichert und nicht in Cache

Keine Reordering für volatile

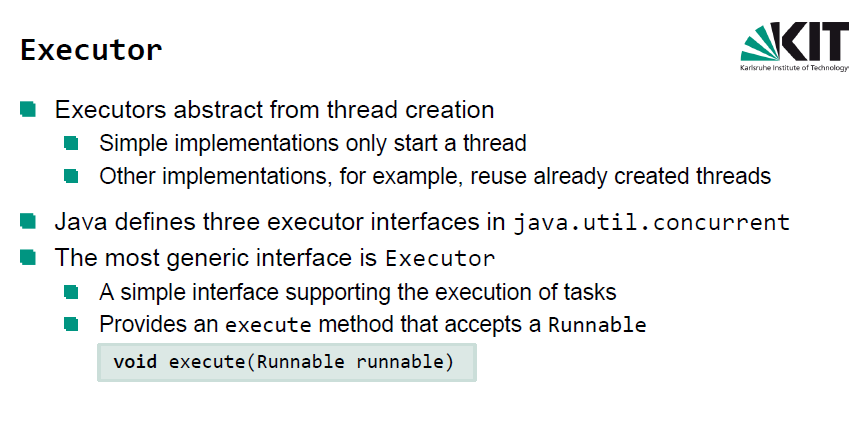
Hilft, die Memory Consistency Error wegen Caching (Variable in RAM) und Reordering (happens-before-Beziehung) zu vermeiden



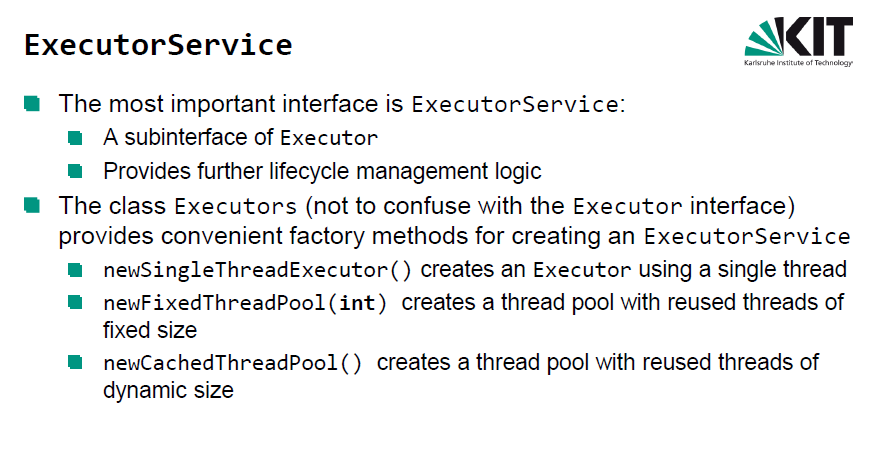


Executor

Executor ist ein Woker von ExecutorService

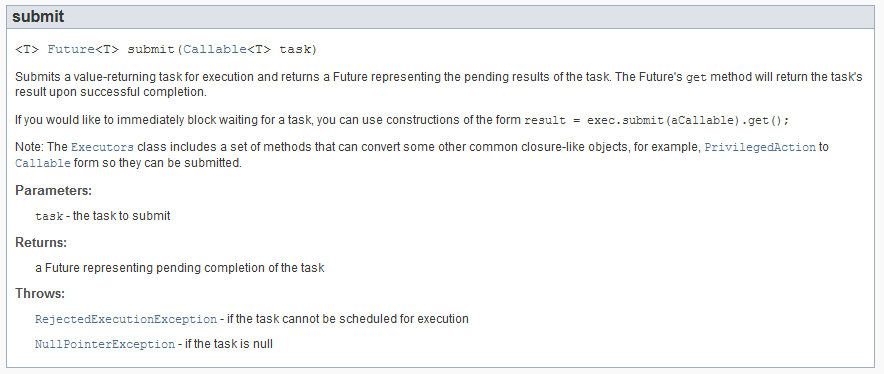


## ExecutorService



### 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.

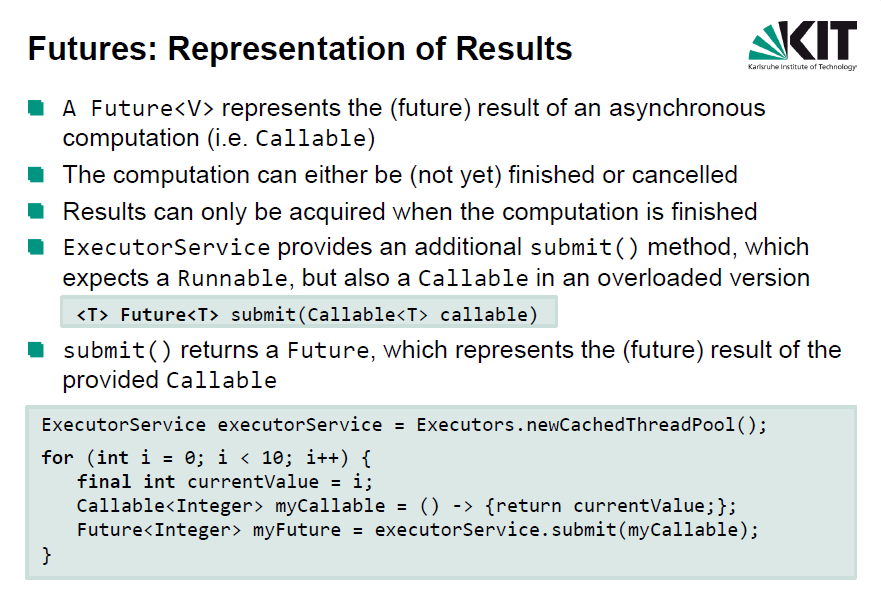
calculateX(elementIdx) ist eine Methode, die ein Integer zurückgibt.

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

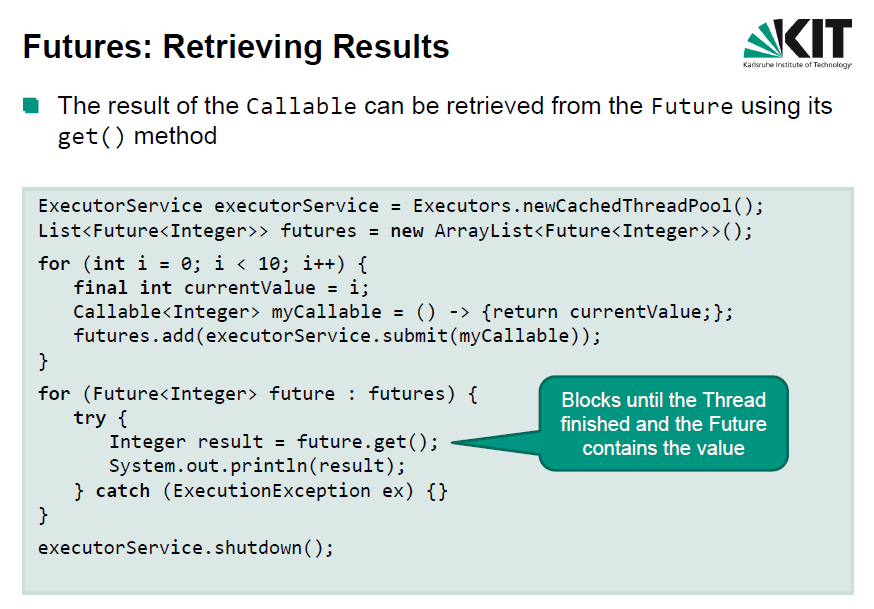
...

Integer result = future.get();

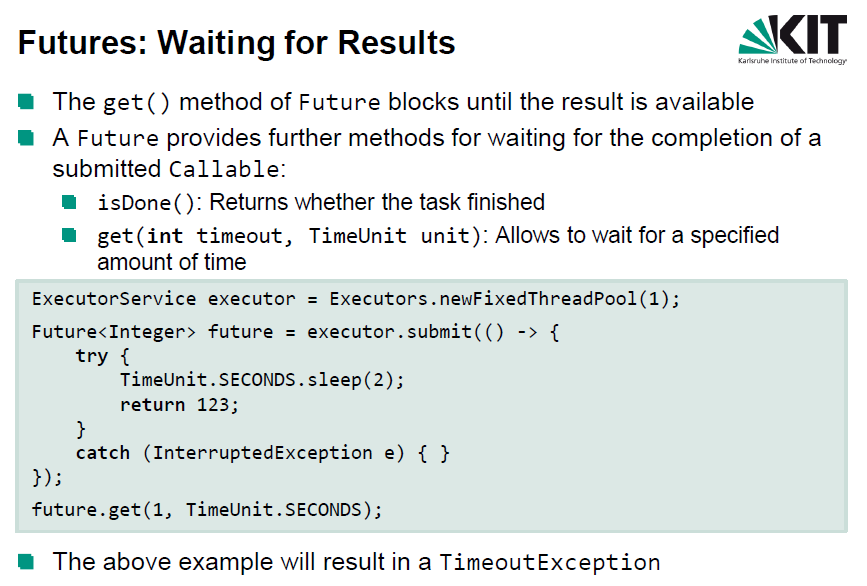
## Futures



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();

}

5. Schalte service ab

service.shutdown()

### 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) {

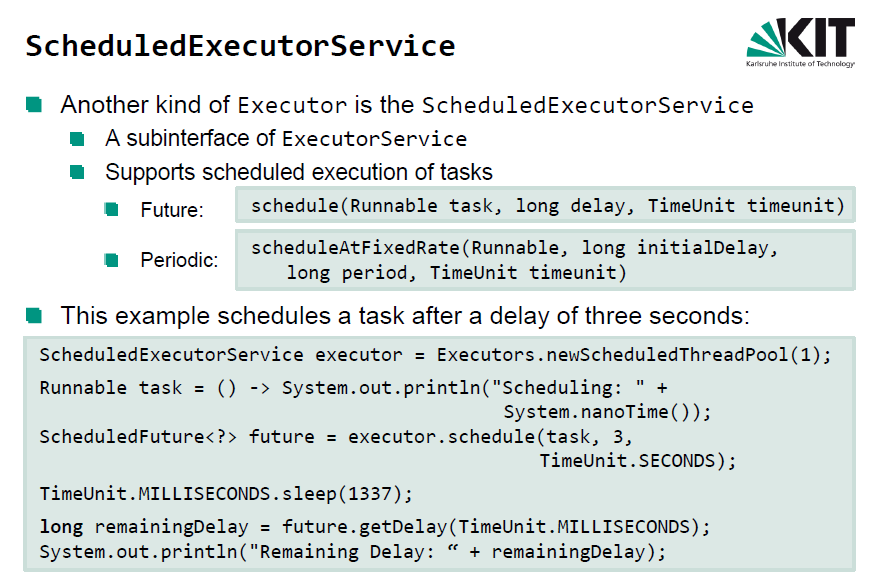
count += future.get();

}

service.shutdown()

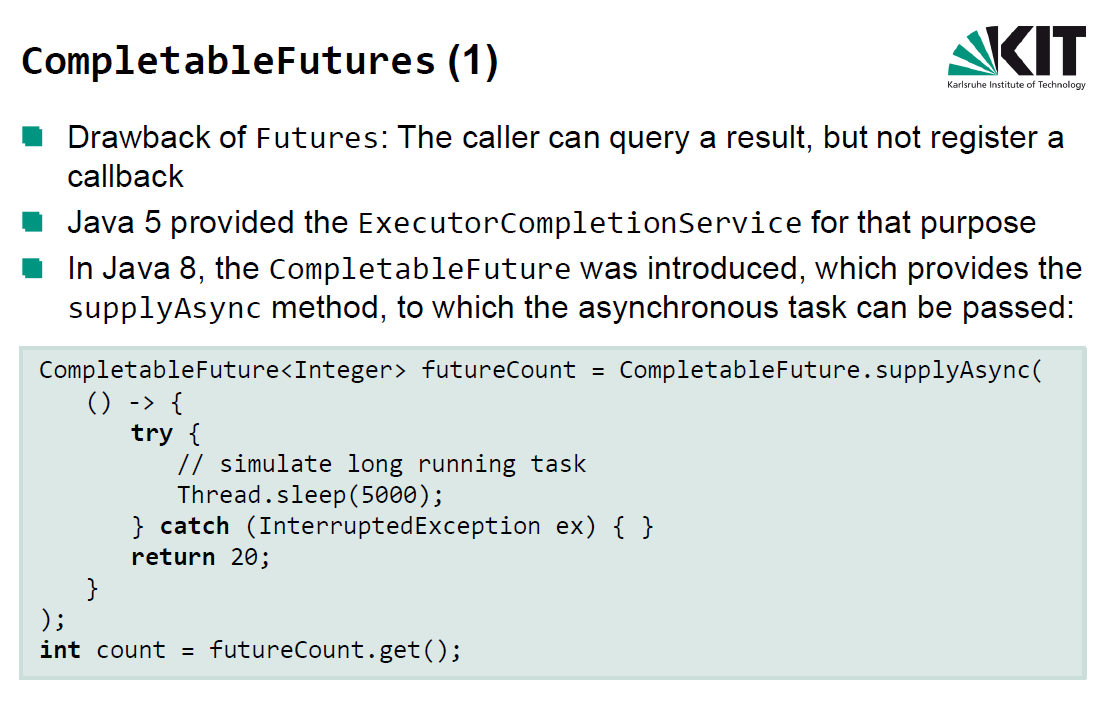
## ScheduledExecutorService

ExecutorService mit Scheduling-Mechanismus



## CompletableFutures

futureCount.get() führt zusätzlich die Funktion aus supplyAsync() aus (? - nicht ganz klar, abe aber eher unwichtig)



# 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.

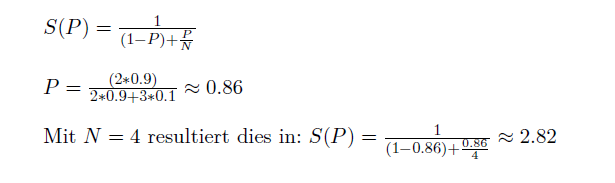
Leser sind nicht blockierend, benötigt 2 Sekunden

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)