Java

Contents

[Parallelprogrammierung 3](#_Toc67409358)

[Amdahlsches Gesetz 3](#_Toc67409359)

[Memory Consistency Error 3](#_Toc67409360)

[Caching 3](#_Toc67409361)

[Reordering 4](#_Toc67409362)

[Happens-before-Beziehung 4](#_Toc67409363)

[Schlüsselwort volatile 5](#_Toc67409364)

[Executor 6](#_Toc67409365)

[ExecutorService 7](#_Toc67409366)

[submit 7](#_Toc67409367)

[Futures 7](#_Toc67409368)

[Berechnung mit ExecutorService und Futures 9](#_Toc67409369)

[Vorgehensweise 9](#_Toc67409370)

[Complex Case 9](#_Toc67409371)

[ScheduledExecutorService 10](#_Toc67409372)

[CompletableFutures 10](#_Toc67409373)

[Java Advanced 11](#_Toc67409374)

[Streams 11](#_Toc67409375)

[filter() 11](#_Toc67409376)

[map() 11](#_Toc67409377)

[reduce() 12](#_Toc67409378)

[findAny() 12](#_Toc67409379)

[findFirst() 12](#_Toc67409380)

[min(), max(), average() 12](#_Toc67409381)

[collect() 13](#_Toc67409382)

[Beispiel für Streams 14](#_Toc67409383)

[Parallel Streams 14](#_Toc67409384)

[Optional 14](#_Toc67409385)

[Wichtigste Methoden 14](#_Toc67409386)

[Alle Methoden 15](#_Toc67409387)

[Actoren 16](#_Toc67409388)

[Actor Model 16](#_Toc67409389)

[Actor 16](#_Toc67409390)

[Messages 16](#_Toc67409391)

[Akka 18](#_Toc67409392)

[Wichtige Momente 18](#_Toc67409393)

[Aktorsystem estellen 18](#_Toc67409394)

[Aktor Class erstllen 18](#_Toc67409395)

[Aktor ohne Parameter erstellen 18](#_Toc67409396)

[Aktor mit Parameter erstellen 18](#_Toc67409397)

[Child Actor mit context() erstellen 18](#_Toc67409398)

[Aktor createReceive() example 19](#_Toc67409399)

[match() 19](#_Toc67409400)

[matchAny() 19](#_Toc67409401)

[unhandled() 19](#_Toc67409402)

[tell() 20](#_Toc67409403)

[sender() 20](#_Toc67409404)

[self() 20](#_Toc67409405)

[Alle Objekte sind Aktoren 20](#_Toc67409406)

[Folien 21](#_Toc67409407)

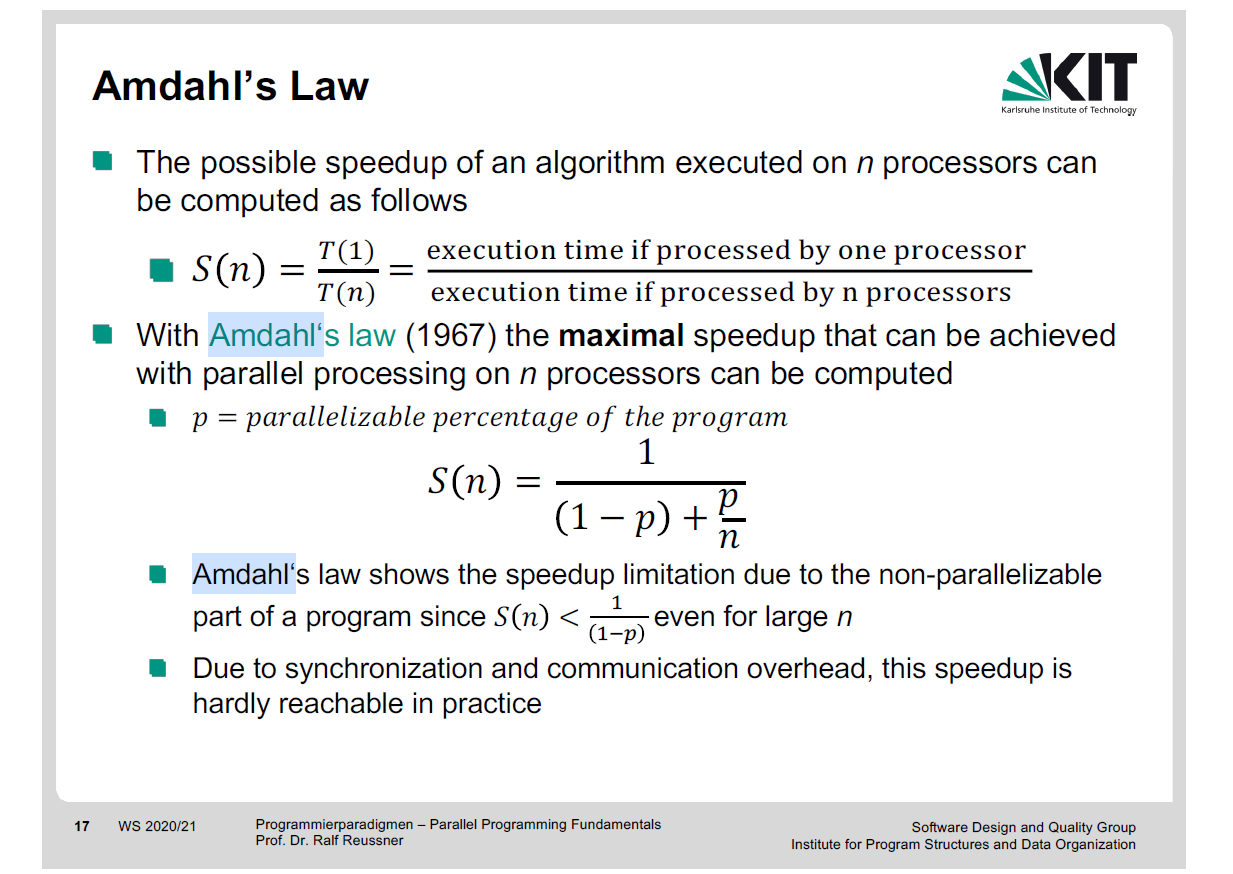
[Design by Contract 25](#_Toc67409408)

[Aufgaben 25](#_Toc67409409)

[Amdahlsches Gesetz aus Ü9 25](#_Toc67409410)

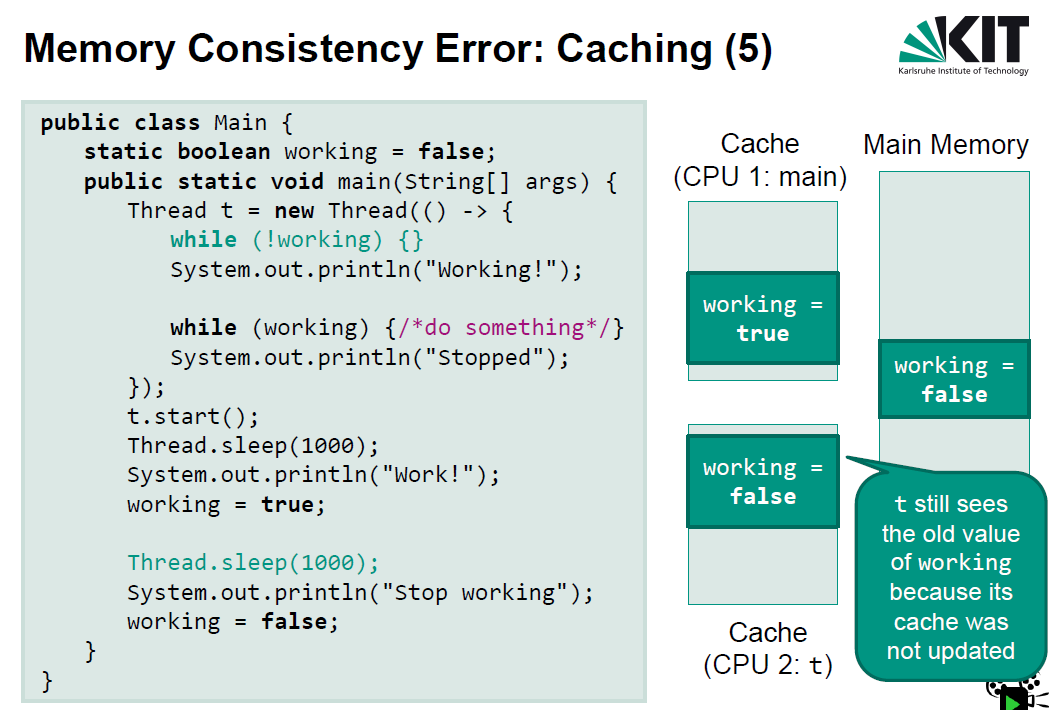
# Parallelprogrammierung

## Amdahlsches Gesetz

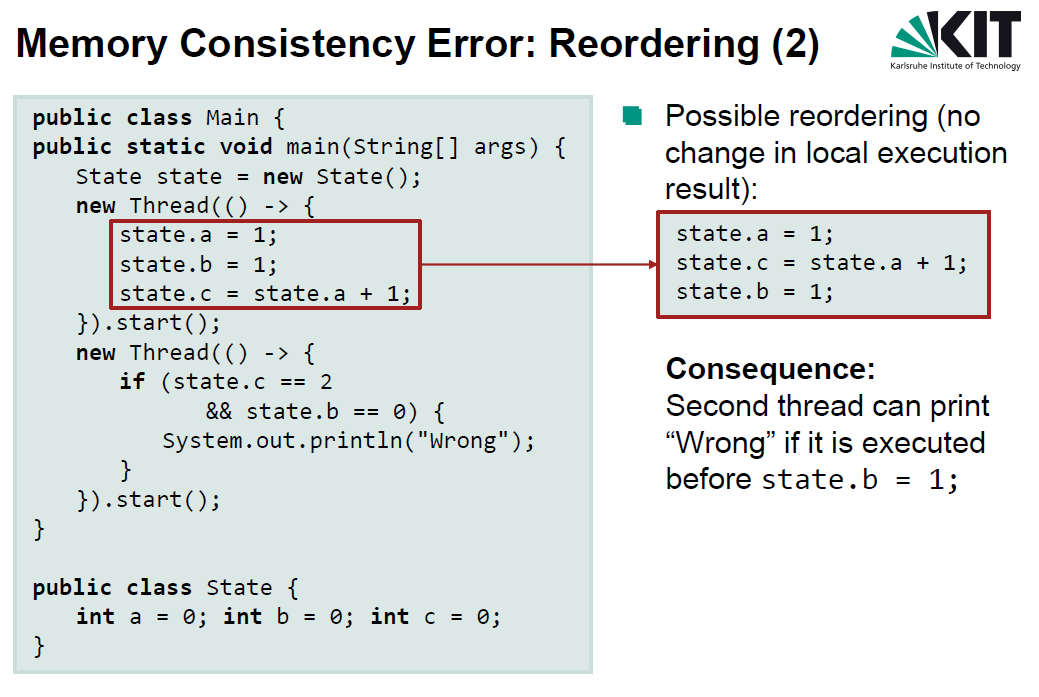


## Memory Consistency Error

### Caching

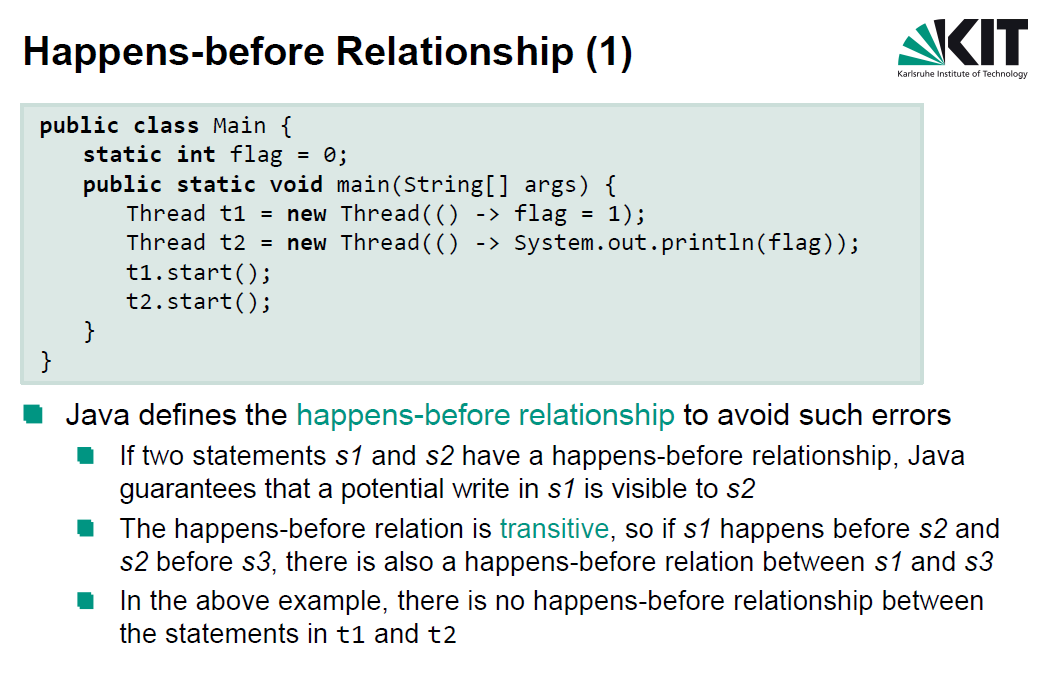


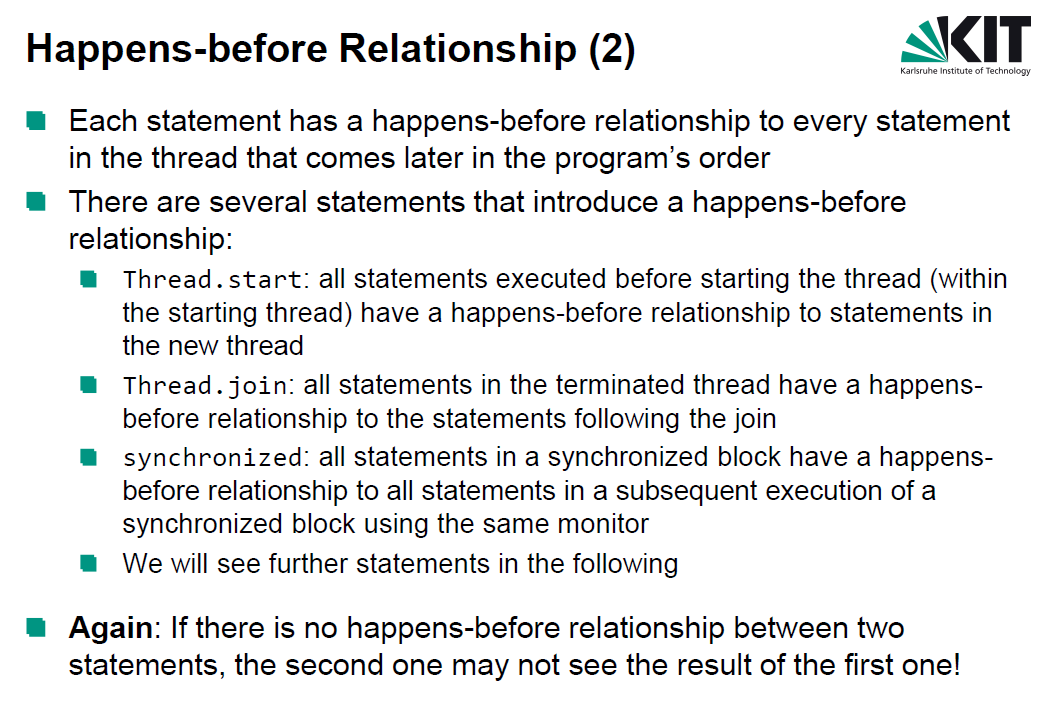
### Reordering



## Happens-before-Beziehung

Hilft, die Memory Consistency Errors wegen Reordering zu vermeiden



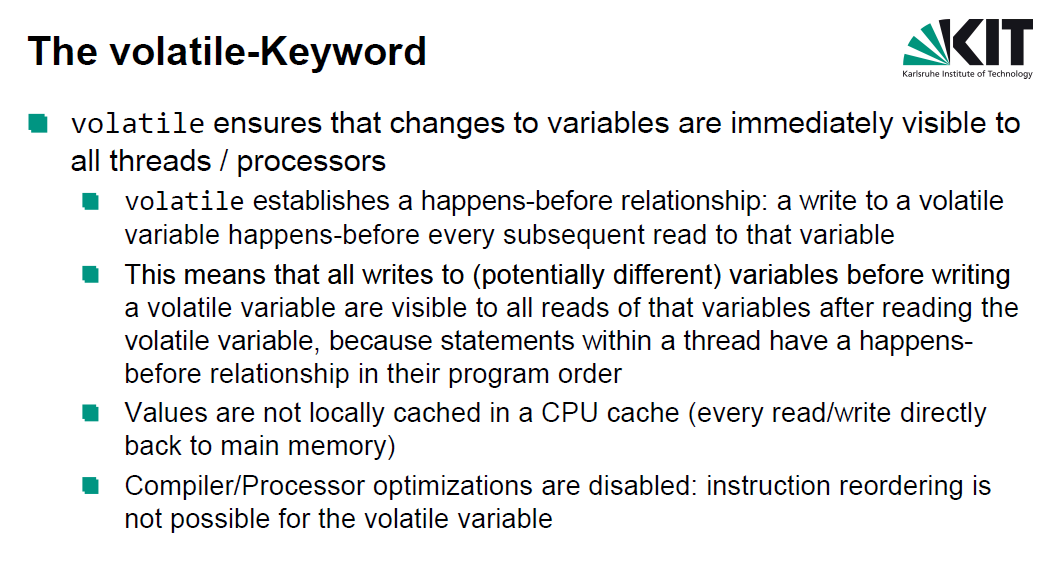


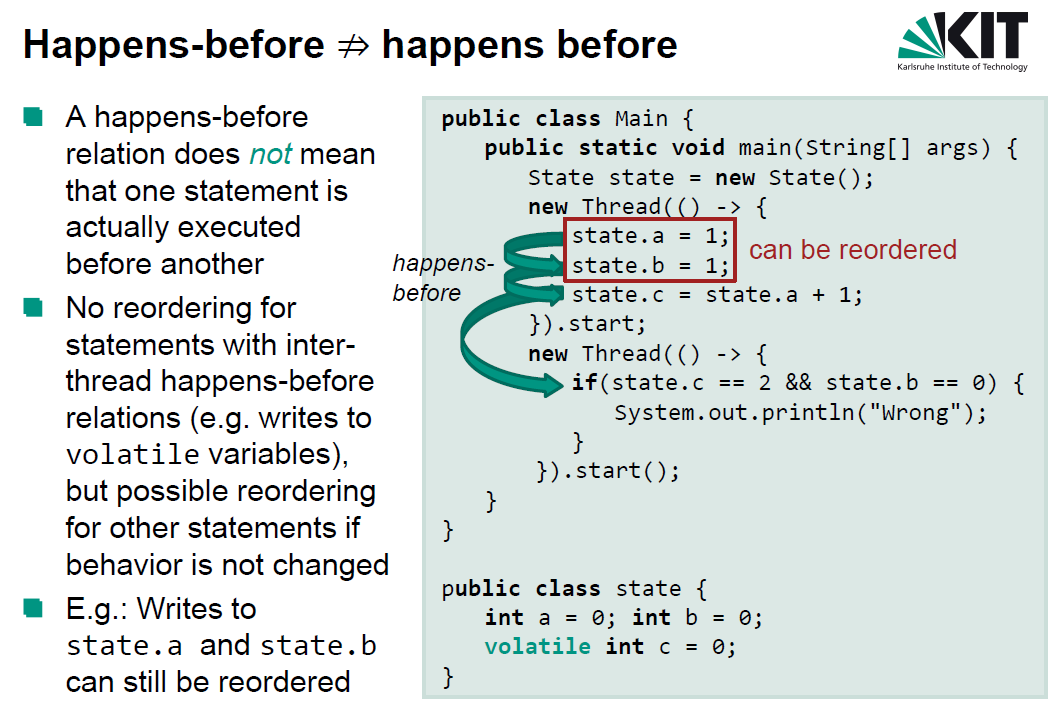
## Schlüsselwort volatile

Volatile Variable wird immer in RAM gespeichert und nicht in Cache

Keine Reordering für volatile

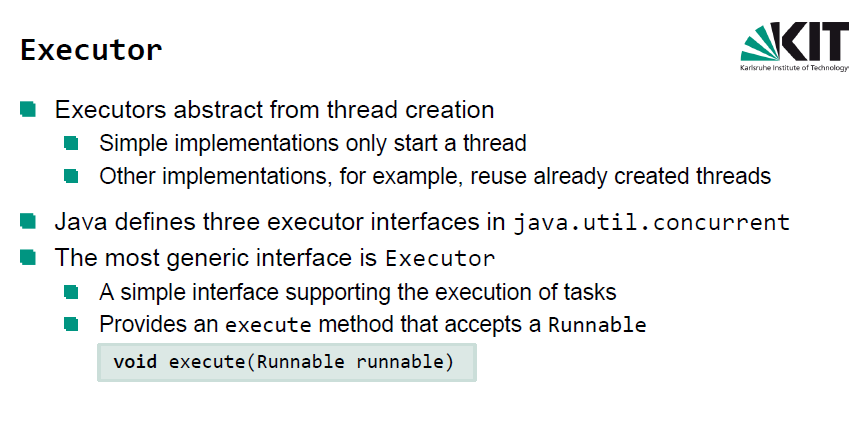
Hilft, die Memory Consistency Error wegen Caching (Variable in RAM) und Reordering (happens-before-Beziehung) zu vermeiden



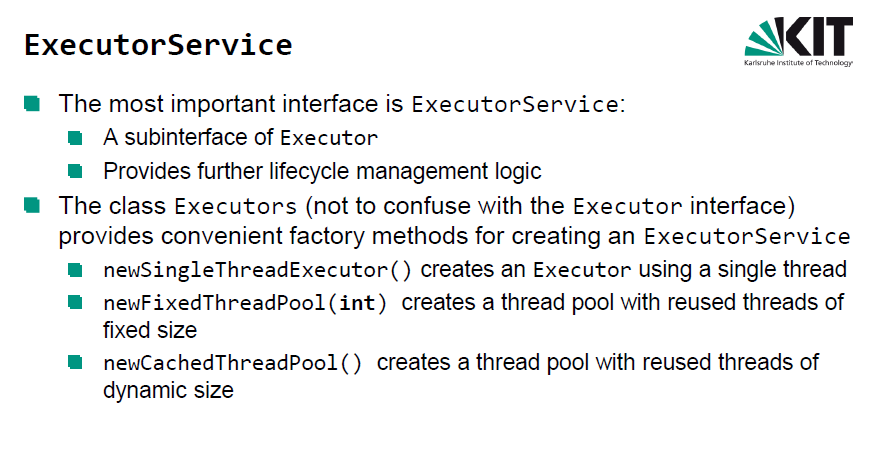


Executor

Executor ist ein Woker von ExecutorService

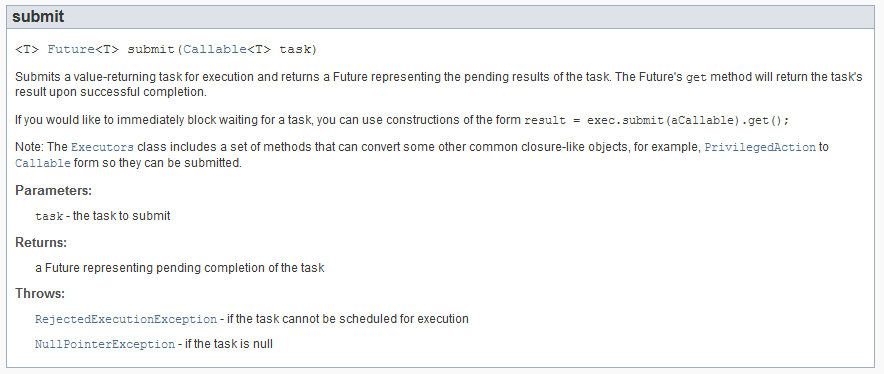


## ExecutorService



### 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.

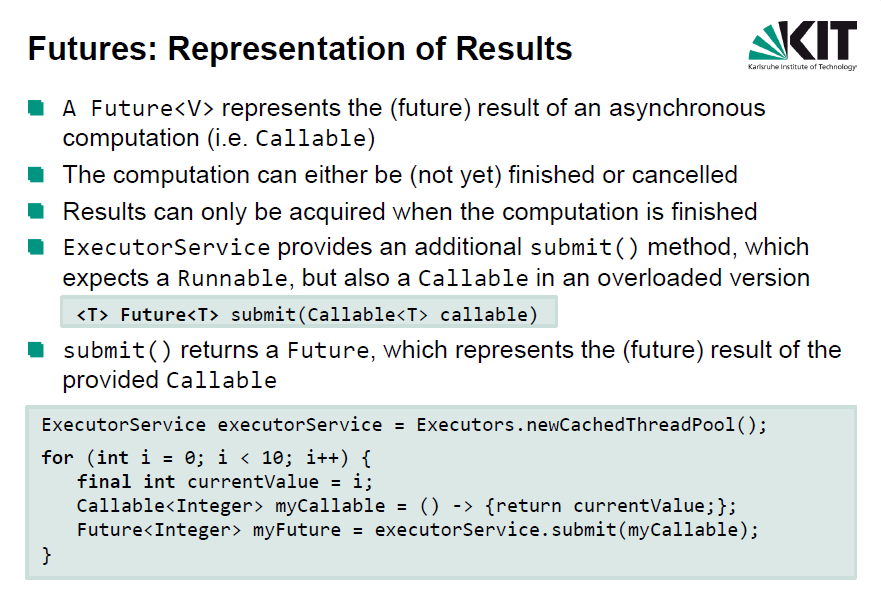
calculateX(elementIdx) ist eine Methode, die ein Integer zurückgibt.

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

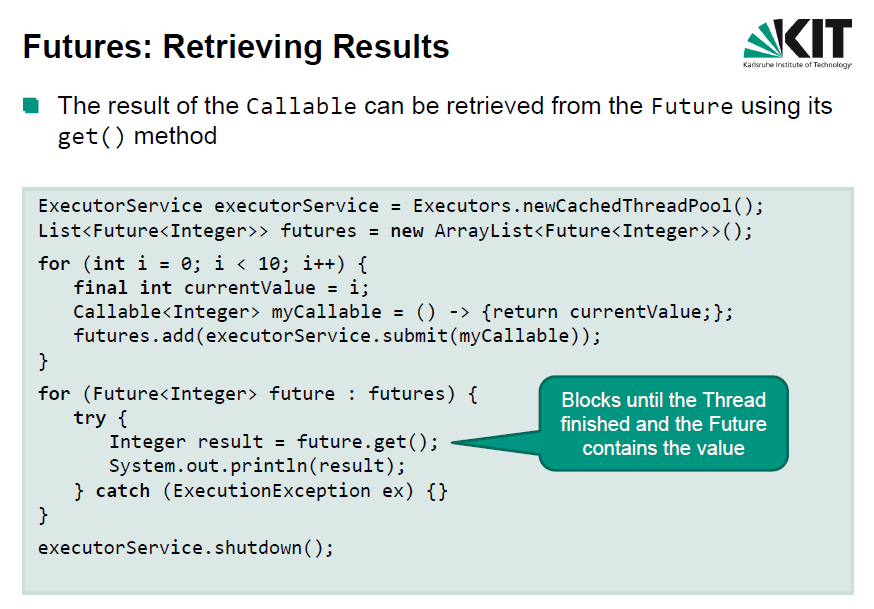
...

Integer result = future.get();

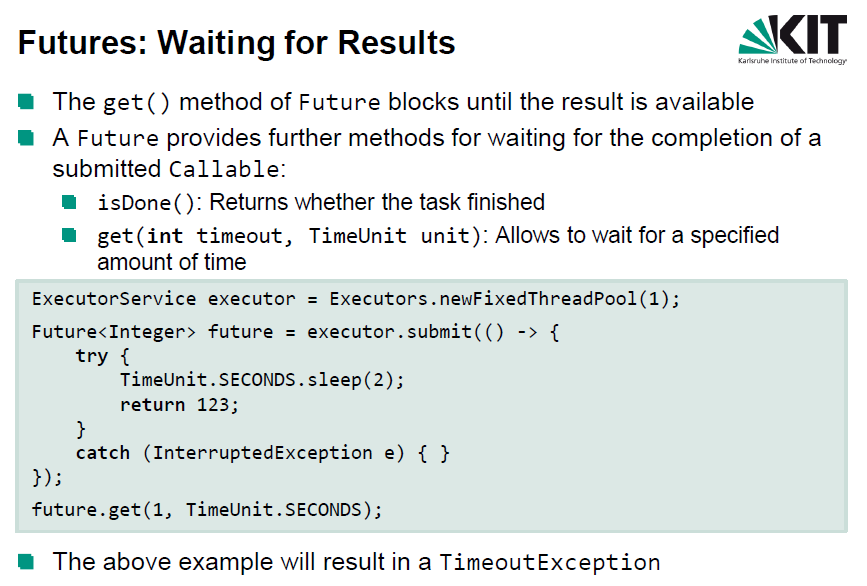
## Futures



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) {

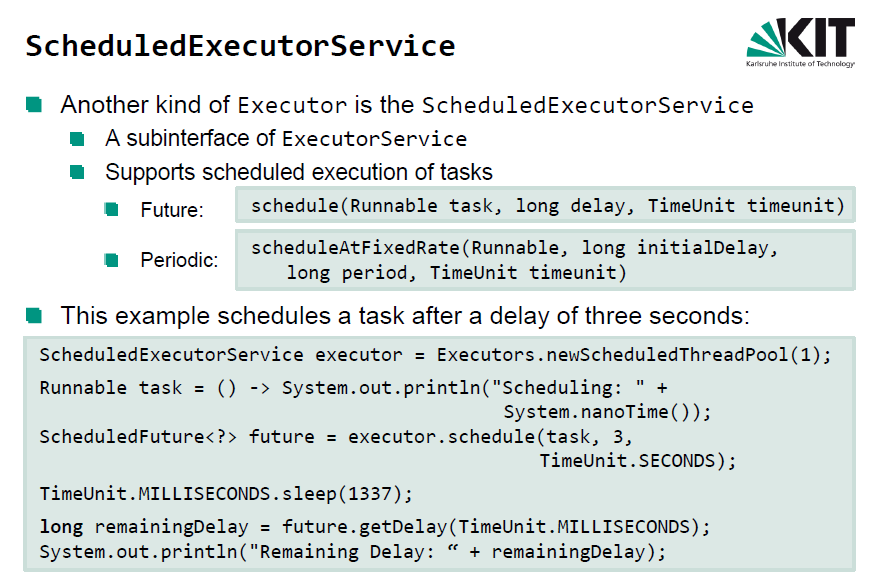
count += future.get();

}

service.shutdown()

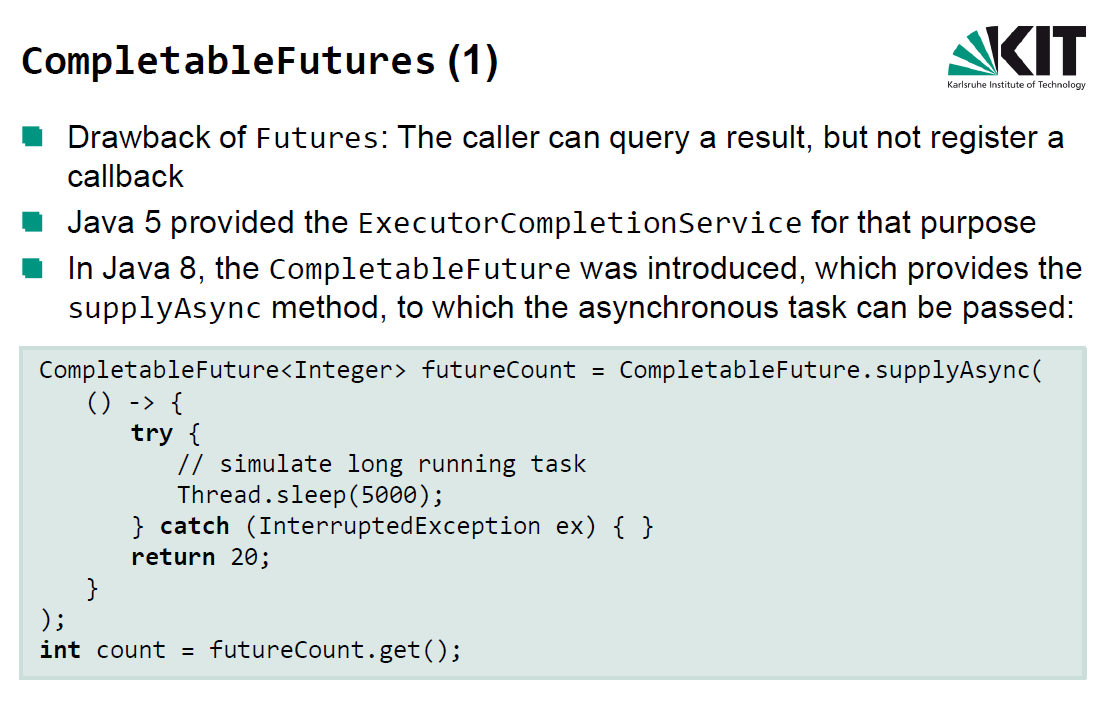
## ScheduledExecutorService

ExecutorService mit Scheduling-Mechanismus



## CompletableFutures

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



# 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.

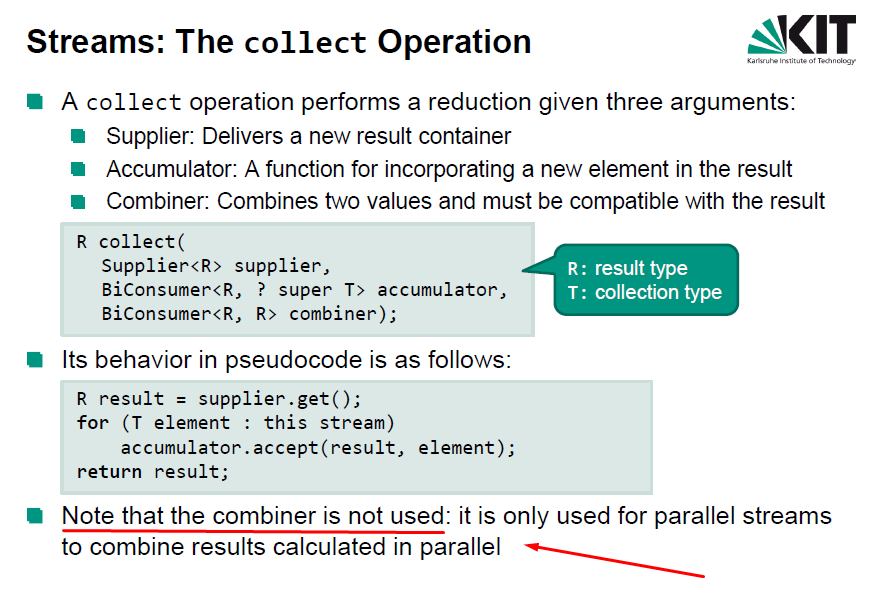
an Optional describing the maximum element of this stream, or an empty Optional if the stream is empty.””

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()

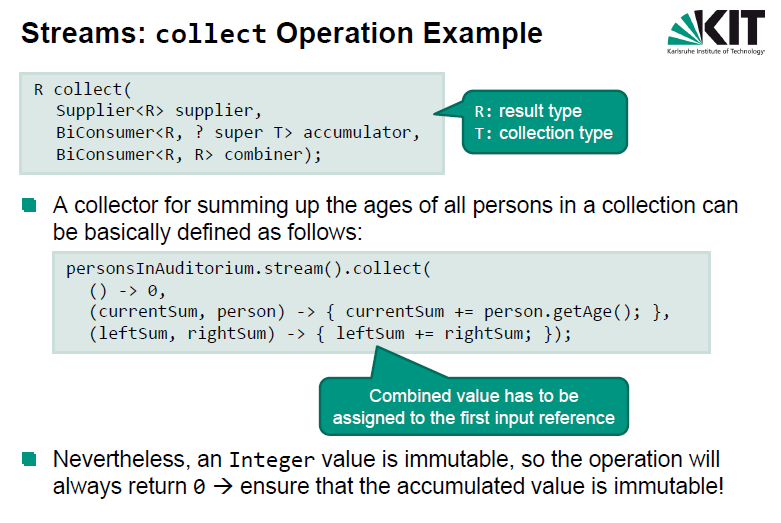


Supplier: Null-Element der Operation

Accumulator: das neue Element hinzufügen und gleichzeitig Ergebnis aktualisieren

Combiner: Zusammenführung von Ergebnissen von parallelen Berechnungen

BiConsumer: hat 2 Werte und verknüpft sie

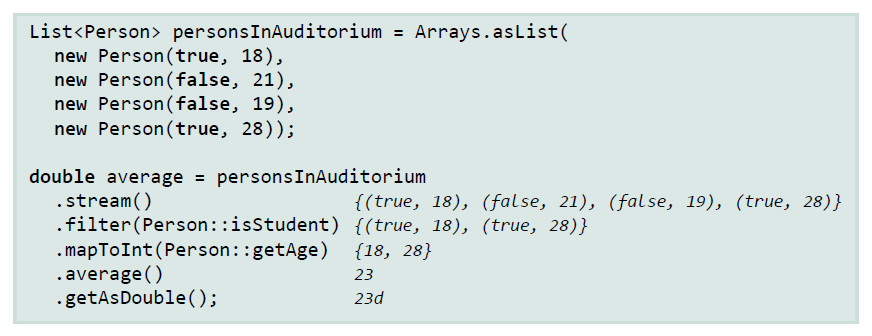


() -> 0 „Null-Element“

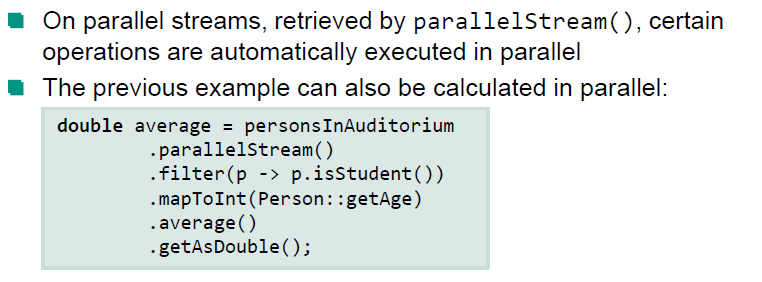
In currentSum wird Age von allen personen akkumulliert.

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

### Beispiel für Streams



## Parallel Streams



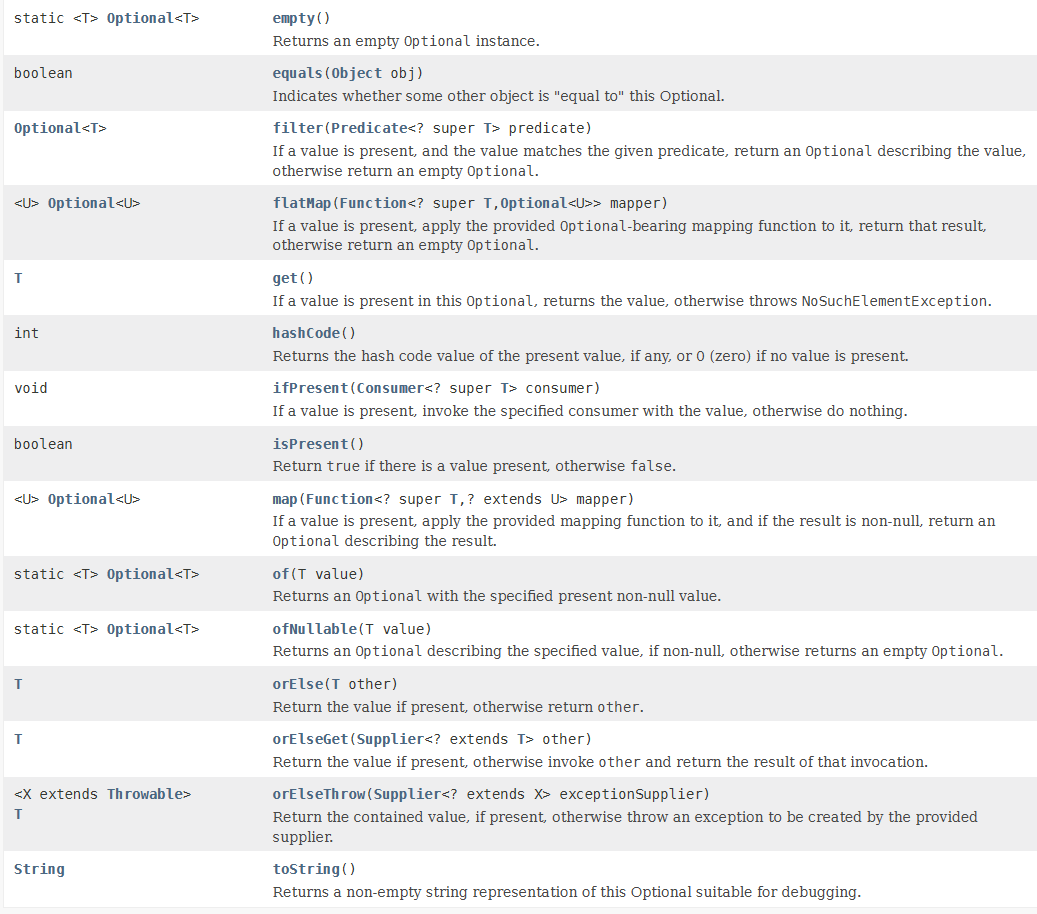
## Optional

### 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.

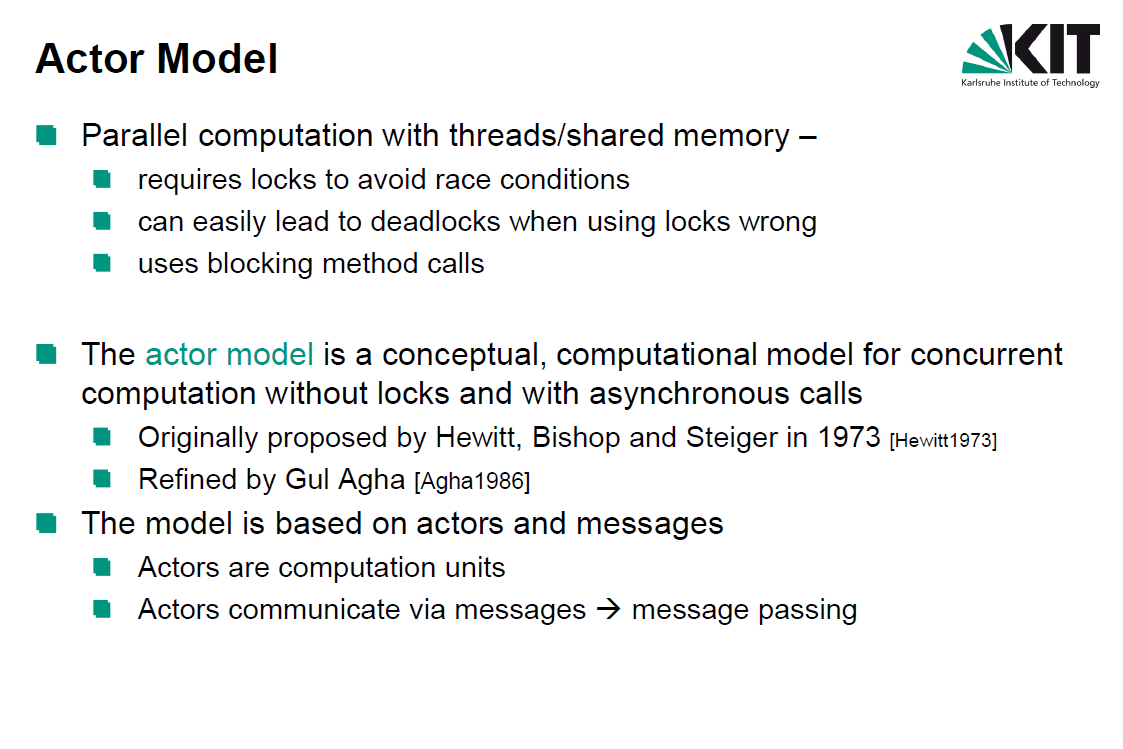
### Alle Methoden



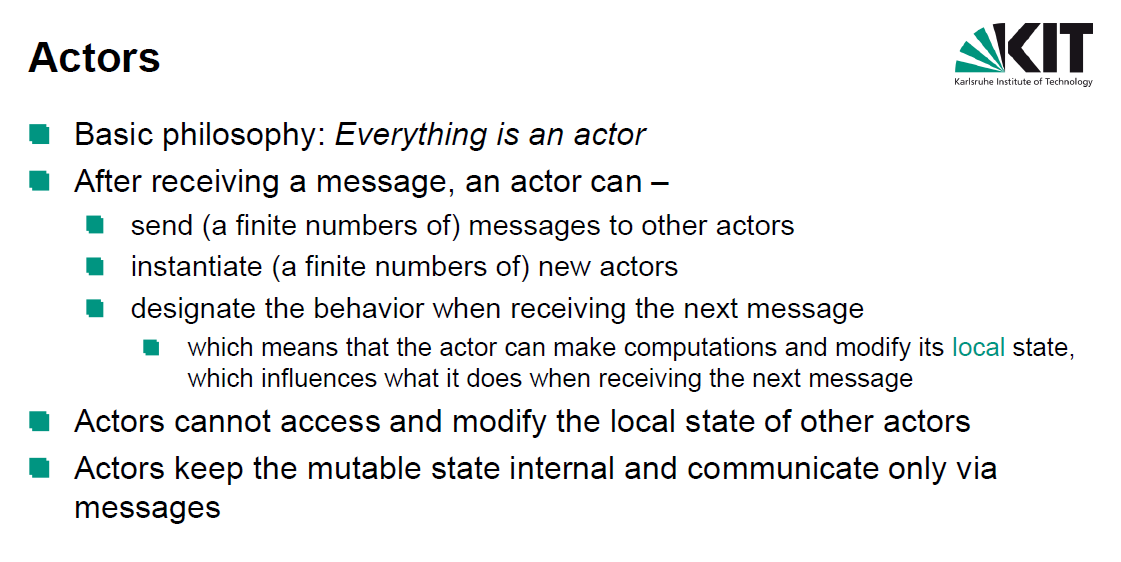
|  |
| --- |
|  |

# Actoren

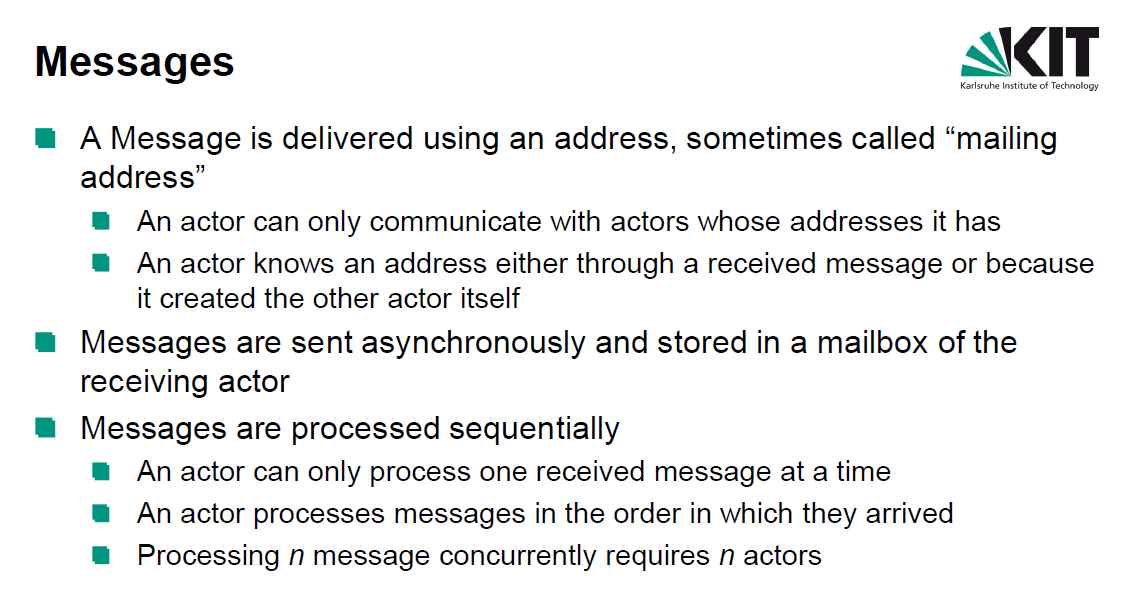
## Actor Model

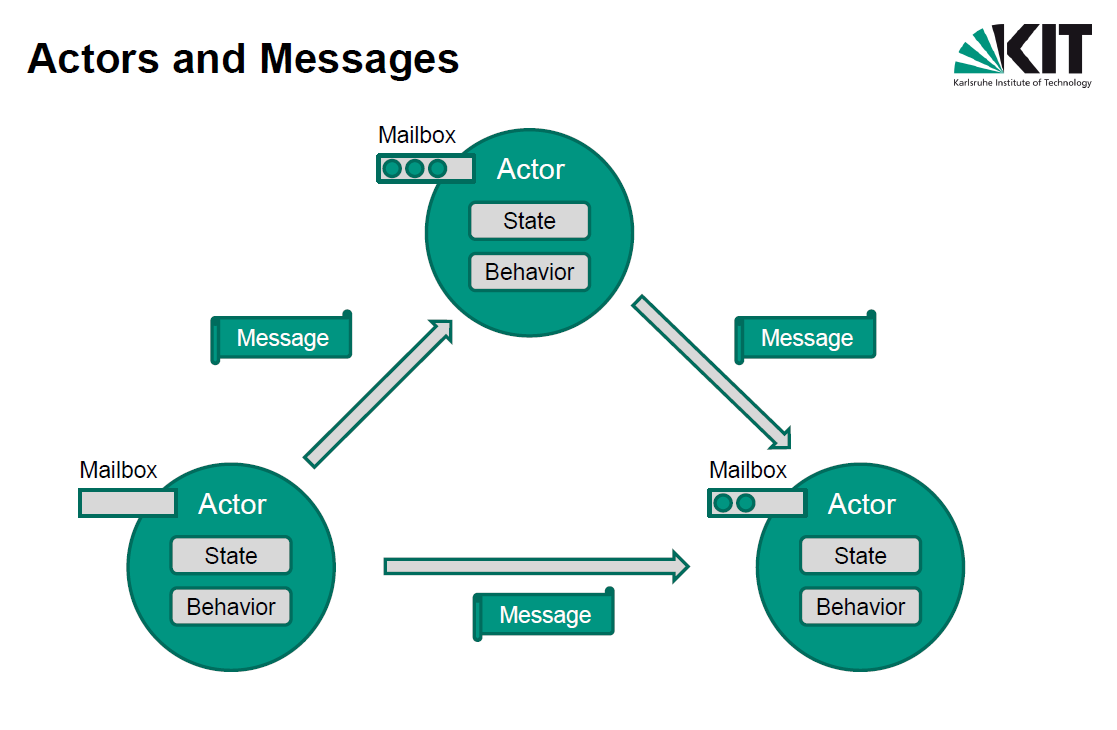


## Actor



# Messages





# Akka

## Wichtige Momente

### Aktorsystem estellen

ActorSystem actorSystem = ActorSystem.create("TeaseLisa");

### Aktor Class erstllen

extends **AbstractActor**, implementiere methode **public Receive createReceive()**

public class PingPong extends AbstractActor {  
 ActorRef partner;  
 String name;  
  
 public PingPong(ActorRef sendTo, String name) {  
 this.partner = sendTo;  
 this.name = name;  
 }

@Override  
public Receive createReceive() {  
 return receiveBuilder()  
 .match(Integer.class, message -> beat(message))  
 .build();  
}

### Aktor ohne Parameter erstellen

ActorRef lisa = actorSystem.actorOf(Props.create(Kid.class));

### Aktor mit Parameter erstellen

ActorRef philosoher = actorSystem.actorOf(  
 Props.create(Philosopher.class, table, name));

Struktur: Props.create(class, args);

### Child Actor mit context() erstellen

Man verwendet getContext().getSystem(), um actorSystem zu finden

public class Kid extends AbstractActor {  
 private int cursed = 0;  
 private ActorRef mommy;  
  
 public Kid() {  
 mommy = getContext().getSystem()

.actorOf(Props.create(Mommy.class));  
 }

### Aktor createReceive() example

Falls message ein String ist und message==“printHello“ ist, wird „Hello World!“ gedruckt

Otherwise unhandled(message)



### match()

Nur dann, wenn class passt und bedingung erfüllt (optional)

match(class, bedingung, aktion);

match(class, aktion);

### matchAny()

Beliebige message.

mathcAny(aktion);

### unhandled()

Mache nichts, default case

### tell()

sender sends a message to actor:

actor.tell(message, sender);

ping.tell(0, ActorRef.noSender());

message ist 0, sender() in ping gibt Null-Sender zurück

ping.tell(“abc”, pong);

message ist „abc“, sender() in pong gibt ActorRef von pong zurück

### sender()

Mit sender kann man ActorRef auf Sender vom Message bekommen, falls es beim tell() gesetzt wurde, ActorRef.noSender, falls es nicht gesetzt wurde.

if (partner != null) {  
 partner.tell(message, self());  
} else {  
 sender().tell(message, self());  
}

### self()

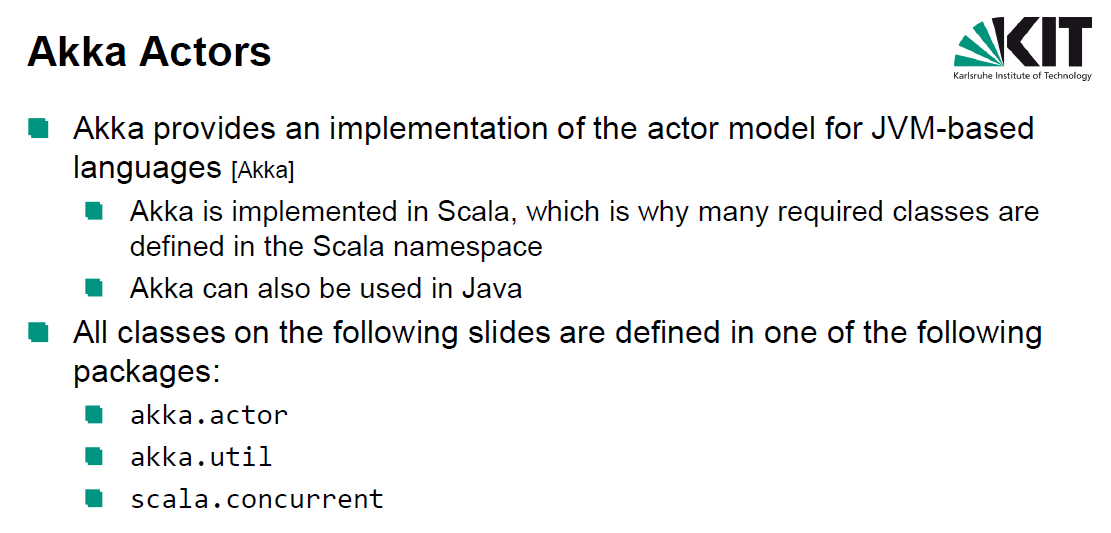
“”returns ActorRef of current Actor””

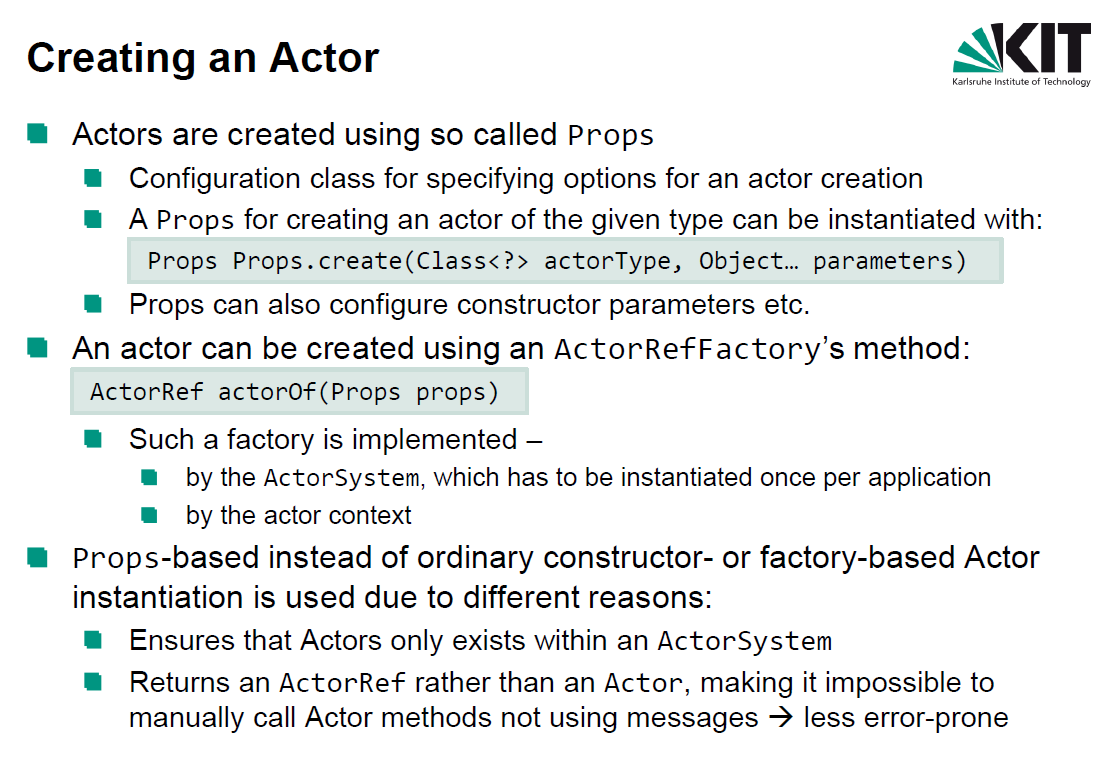
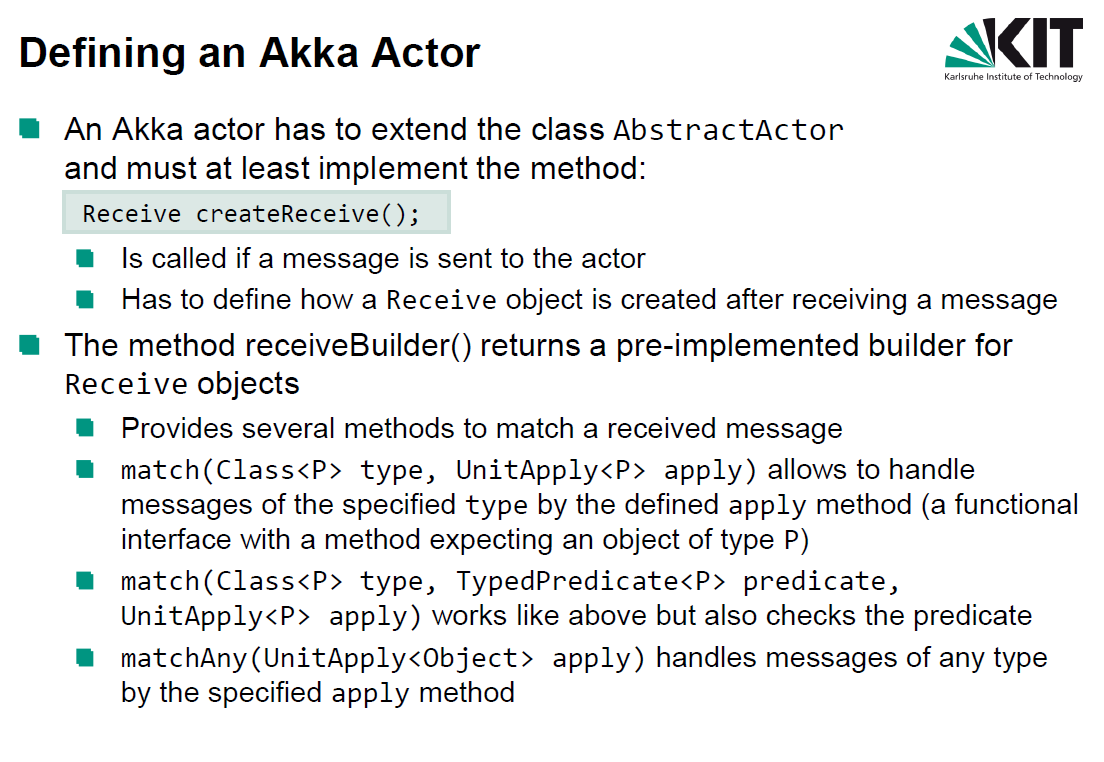
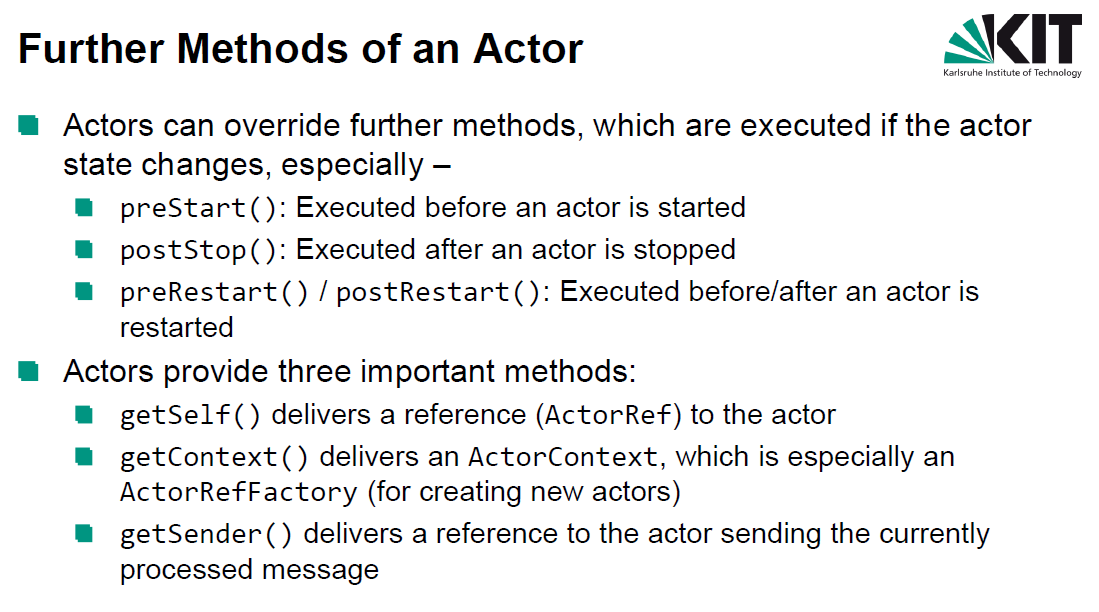
Kann man für tell verwenden

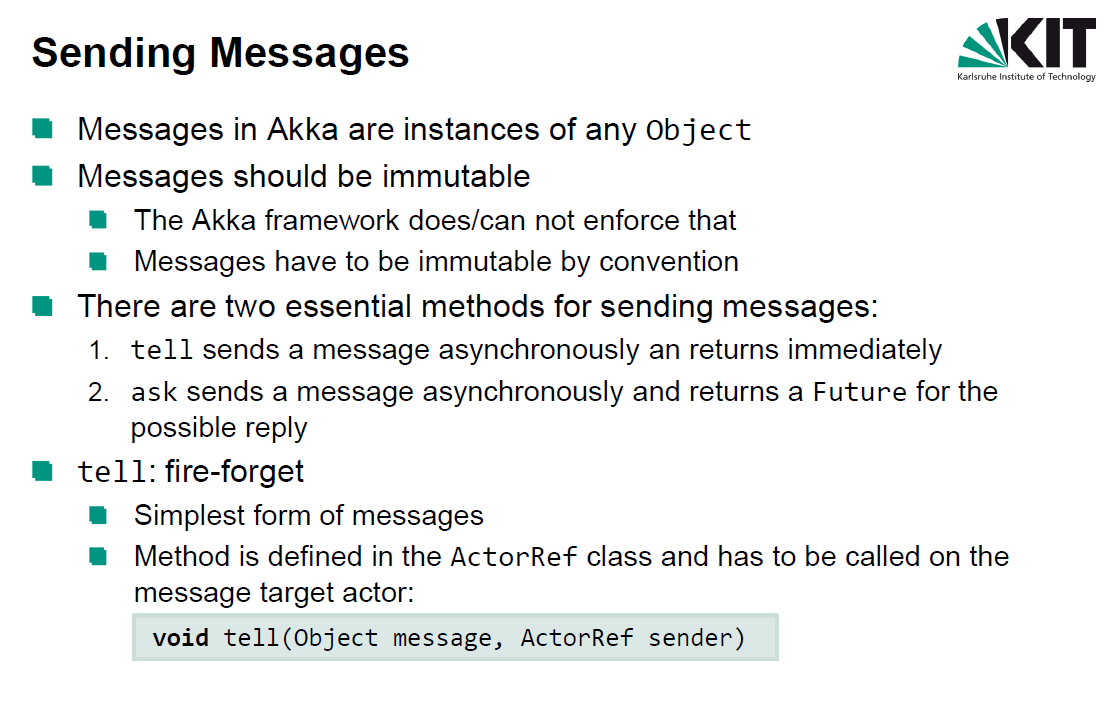
### Alle Objekte sind Aktoren

Speischende Philosopher: Alle Philosopher sind Aktoren und Tisch ist auch ein Aktor

## Folien



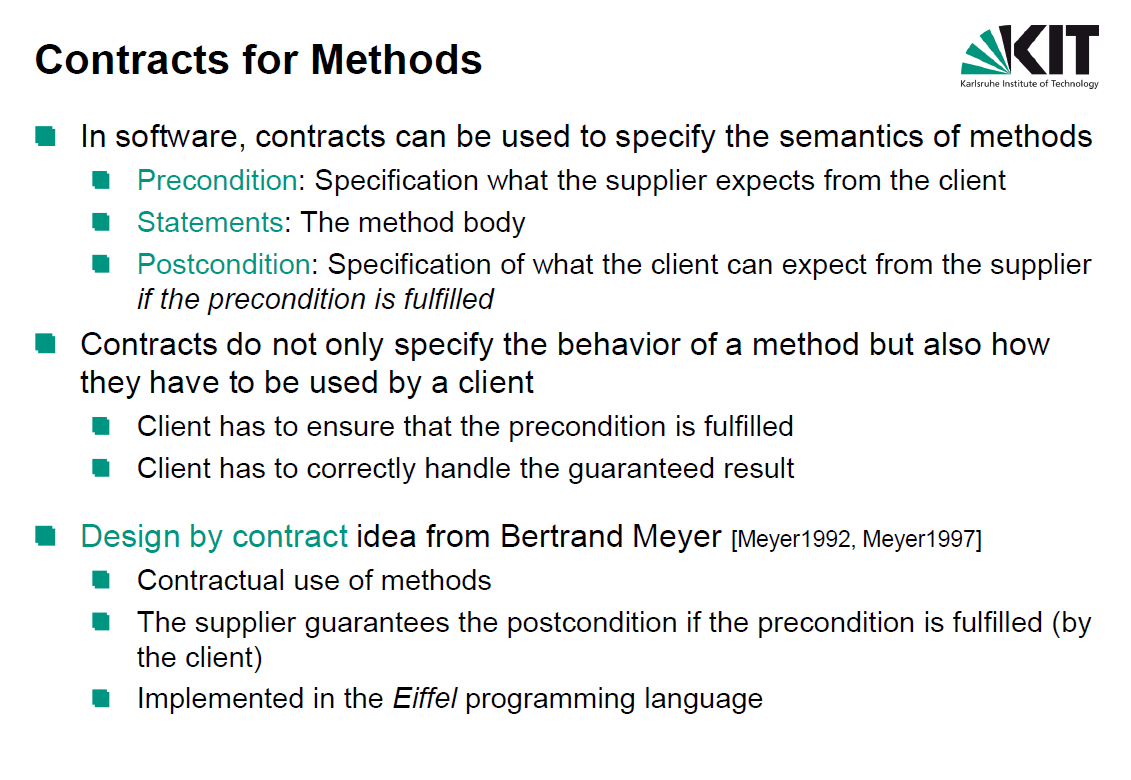




# Design by Contract

## Hauptidee

**“”The supplier guarantees the postcondition if the precondition is fulfilled (by the client)””**



## Non-Redundancy

The body of a routine shall not test for the routine’s precondition.

Beispiel

### Quellcode

public class Employee {

private boolean isEmployed;

public Employee() {

this.isEmployed = false;

}

protected void hire() {

this.isEmployed = true;

}

protected void fire() {

this.isEmployed = false;

}

public boolean isEmployed() {

return isEmployed;

}

}

### Mit asserts:

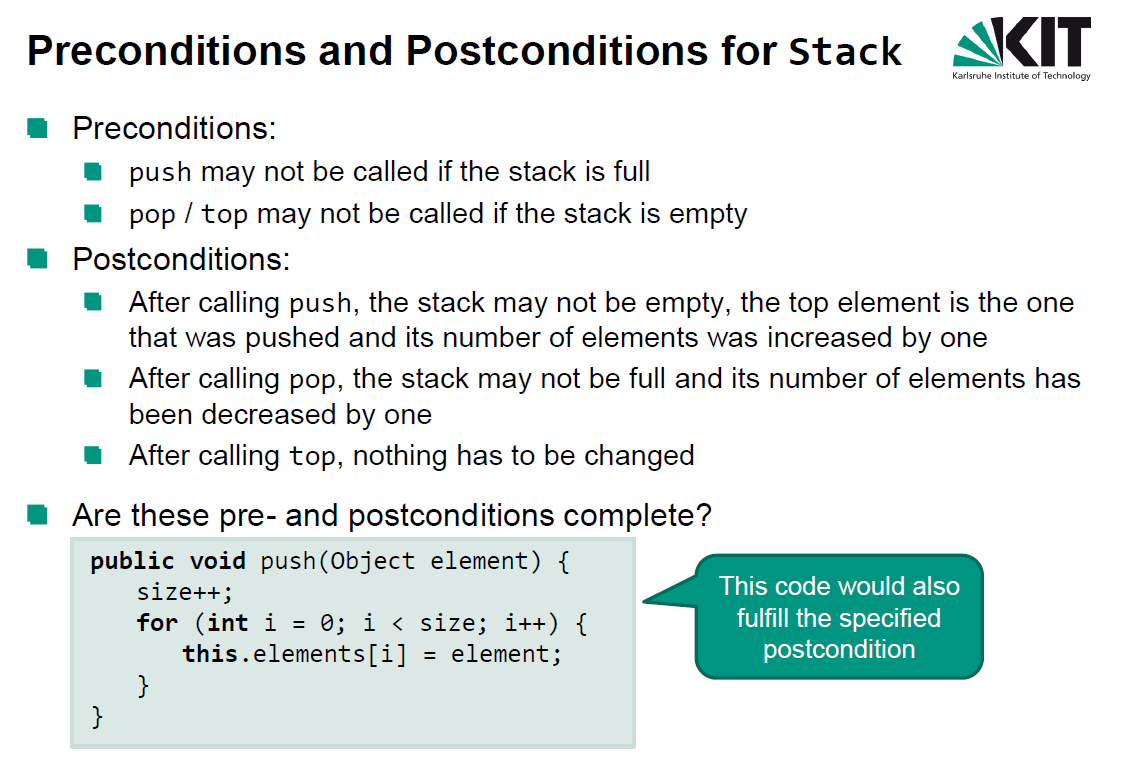
1. Alle Vorbedingungen (Null, Zustand, alte Liste speichern)

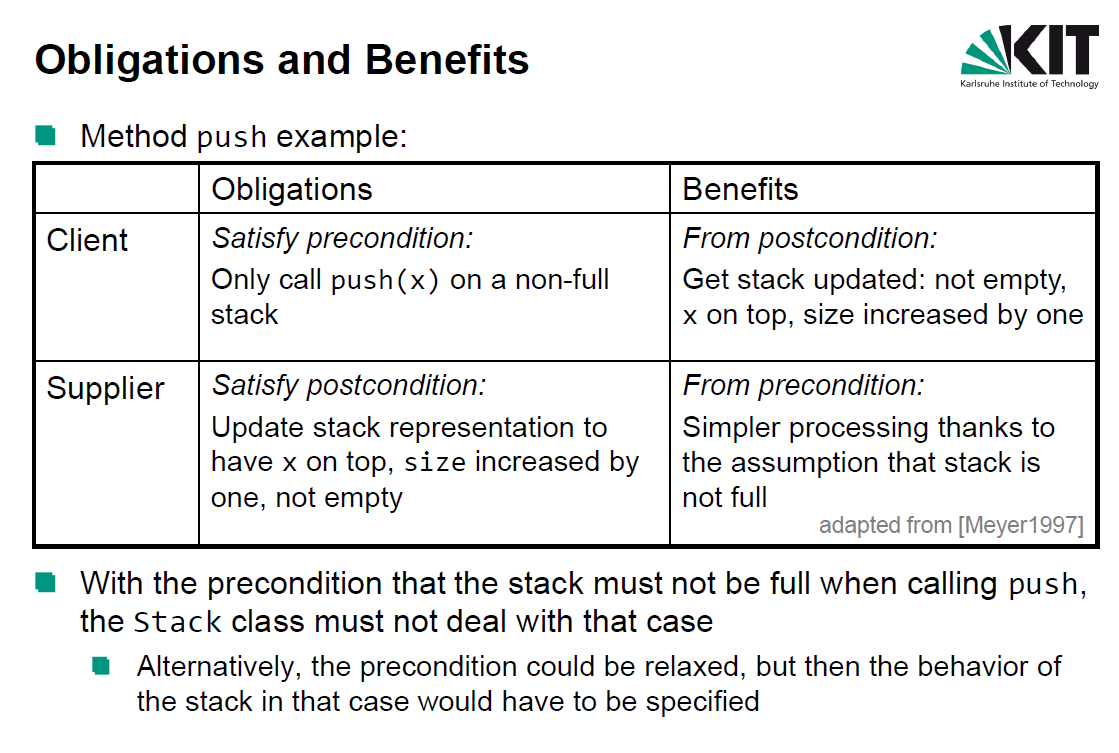
2. Aktion

3. Alle Nachbedingungen

@Override  
public void hire(Employee employee) {  
 assert ( employee!= null);  
 assert (!employees.contains(employee));  
 assert (!employee.isEmployed());  
 List<Employee> oldEmployees = new ArrayList<>(employees);  
  
 employee.hire();  
 employees.add(employee);  
  
 assert (employees.contains(employee));  
 assert (employee.isEmployed());  
 assert (employees.containsAll(oldEmployees));  
 assert (employees.size() == oldEmployees.size() + 1);  
  
}  
  
@Override  
public void fire(Employee employee) {  
 assert (employee!= null);  
 assert (employees.contains(employee));  
 assert (employee.isEmployed());  
 List<Employee> oldEmployees = new ArrayList<>(employees);  
  
 employee.fire();  
 employees.remove(employee);  
  
 assert (!employees.contains(employee));  
 assert (!employee.isEmployed());  
 assert (oldEmployees.containsAll(employees));  
 assert (employees.size() == oldEmployees.size() - 1);  
}

## Folien





# 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.

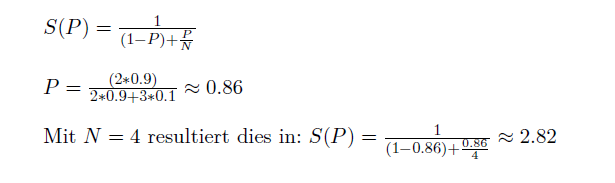
Leser sind nicht blockierend, benötigt 2 Sekunden

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)