Parallele Programmierung
Thread Pools

Vorlesung 5

Prof. Dr. Luc Bläser

Letzte Vorlesung - Quiz

```
void acquireMultiLocks(Lock[] lockSet) {
  int i = 0;
  while (i < lockSet.length) {
    if (lockSet[i].tryLock()) {
      i++;
    } else {
      while (i > 0) {
        i--;
        lockSet[i].unlock();
      }
    }
  }
}
```

true, falls Sperre erhalten
false, falls von anderem gesperrt

Was für ein Problem hat diese Lösung?

Inhalt Heute

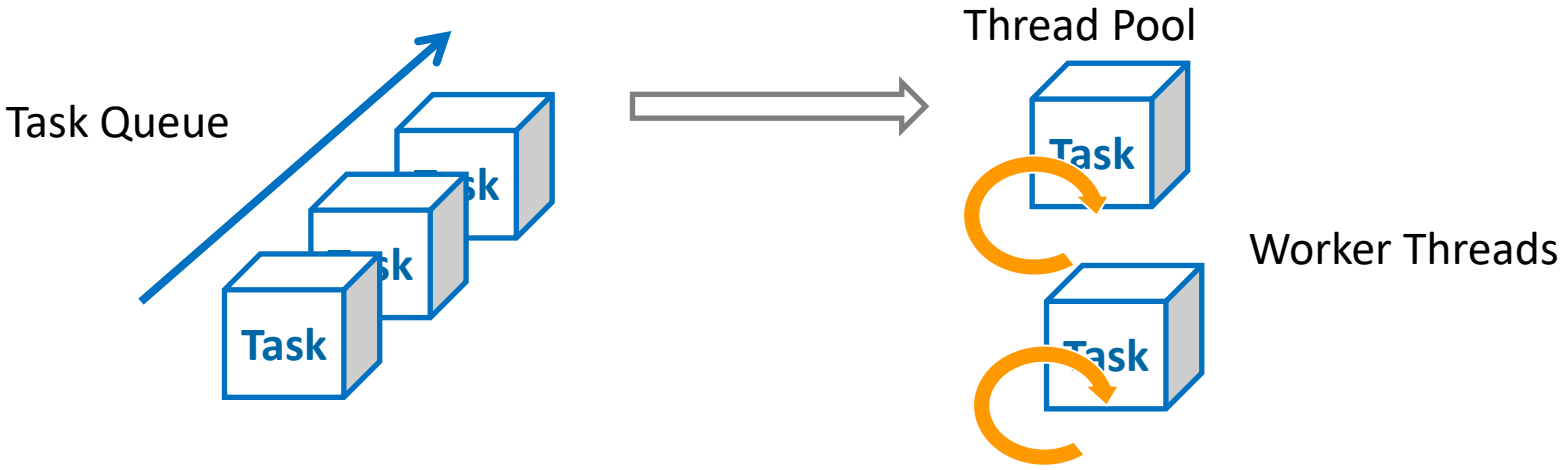
- Thread-Pool
  - Konzept und Funktionsweise
  - Vorteile und Einschränkungen
  - Fork & Join Pool
- Asynchrone Programmierung
  - CompletableFutures

Lernziele

- Thread Pool Konzept verstehen
- Java Fork Join Pool einsetzen können
- Asynchrone Programmierung in Java kennen

Thread Pool Konzept

- Tasks
  - Tasks implementieren potentiell parallele Arbeitspakete
  - Auszuführende Tasks werden in Warteschlange eingereiht
- Thread Pool
  - Beschränkte Anzahl von Worker-Threads
  - Holen Tasks aus der Warteschlange und führen sie aus



Thread Pool Vorteile

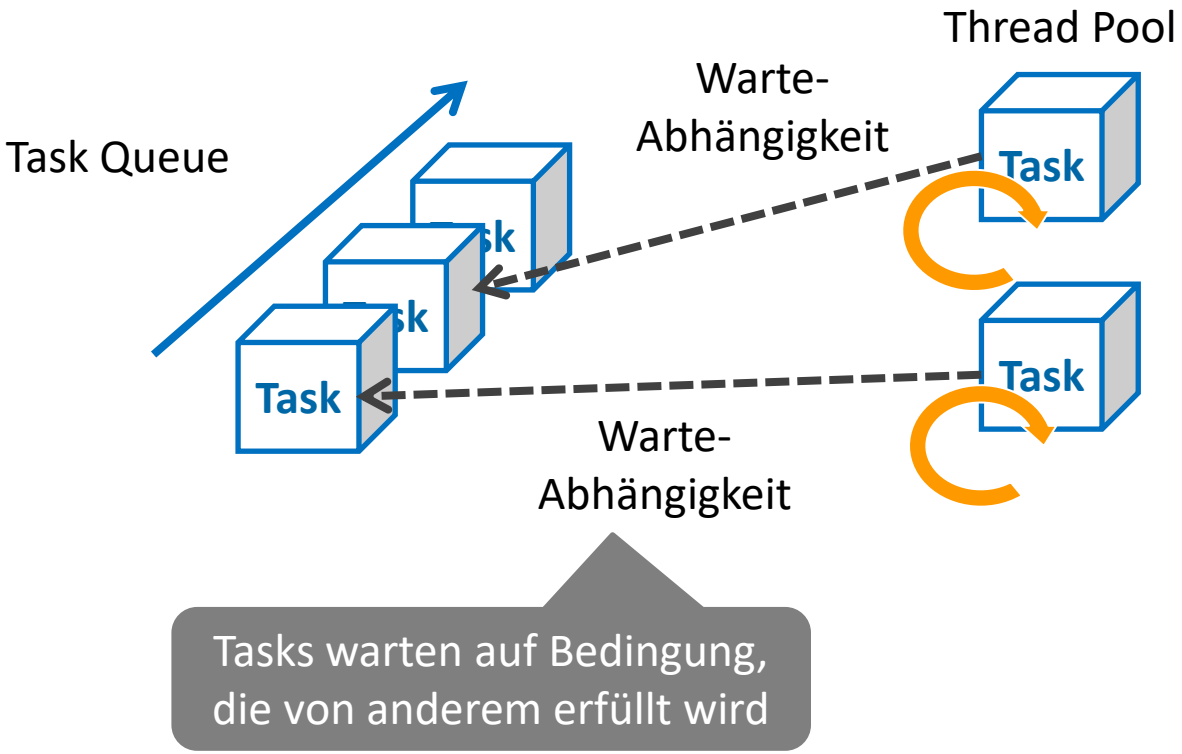
- Beschränkte Anzahl von Threads
  - Viele Threads verlangsamen System oder überschreiten verfügbaren Speicher
- Recycling der Threads
  - Spare Thread-Erzeugung und Freigabe
- Höhere Abstraktion
  - Trenne Task-Beschreibung («Problem Space») von Task-Ausführung («Machine Space»)
- Anzahl Threads pro System konfigurierbar
  - #Worker Threads = #Prozessoren + #I/O-Aufrufe

Neuer «Free Lunch»

- Programme mit Task modellieren
  - Laufen automatisch schneller auf parallelen Maschinen
  - Ermöglicht Ausschöpfung der Parallelität ohne hohe Thread-Kosten

Thread Pool Einschränkung

- Tasks dürfen nicht aufeinander warten



Deadlock

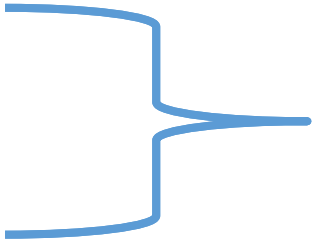
Run to Completion

- Task muss zu Ende laufen, bevor Worker Thread anderen Task ausführen kann
- Ausnahme: Geschachtelte Tasks (Sub-Tasks)

Wieso kann der Worker Thread nicht andere Tasks ausführen, wenn ein Task blockiert?

Java Thread Pool Unterstützung

- Fork-Join-Pool (seit Java 7 & 8)
  - Unterstützt rekursive Aufgaben
  - Effiziente Implementierung
- Einfache Executors (seit Java 5)
  - Keine rekursiven Aufgaben
  - Nicht besonders optimiert



Task Lancierung

```
var threadPool = new ForkJoinPool();
```

Moderner Thread Pool

```
Future<Integer> future = threadPool.submit(() -> {
  int value = …;
  // long calculation
  return value;
});
```

Task in Thread Pool einreihen

Tasks können Rückgabe haben

Wieso liefert submit keinen int-Wert?

Future Konzept

- Future repräsentiert ein zukünftiges Resultat
  - Proxy auf Resultat, dass evtl. noch nicht bekannt ist, weil Berechnung noch läuft
  - Muss Ende der Berechnung abwarten, bevor Resultat zurückgegeben wird

Future Verwendung

Blockiert nicht, lanciert Task ohne zu Warten

```
Future<T> future = threadPool.submit(...);
…
T result = future.get();
```

Blockiert, bis Task beendet ist

Fehler Propagierung

- Task mit unbehandelter Exception beendet
  - `get()` liefert dann `ExecutionException`
  - Ursprüngliche Exception ist darin geschachtelt (Cause)

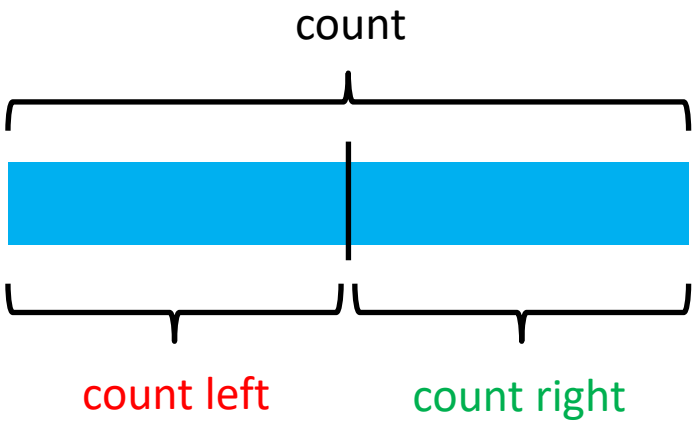
Zählen von Primzahlen

2 N

Wie viele Primzahlen gibt es zwischen 2 und N?

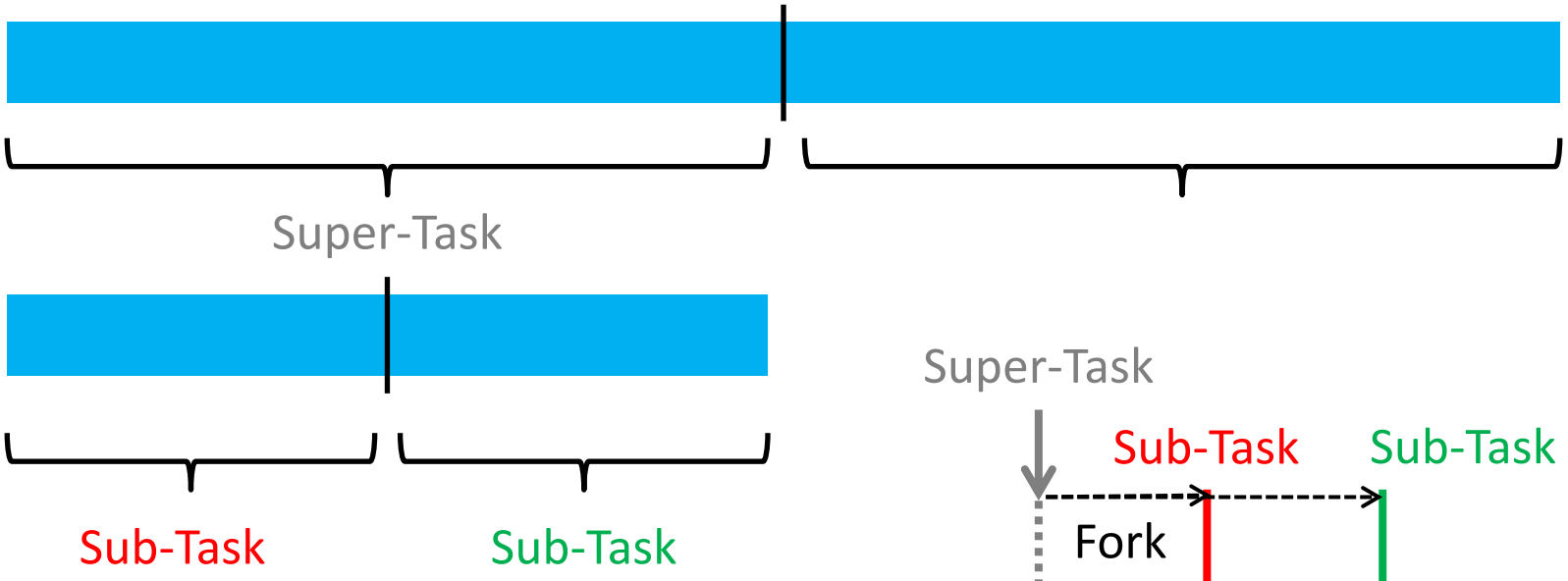
```
int counter = 0;
for (int number = 2; number < N; number++) {
   if (isPrime(number)) { counter++; }
}
```

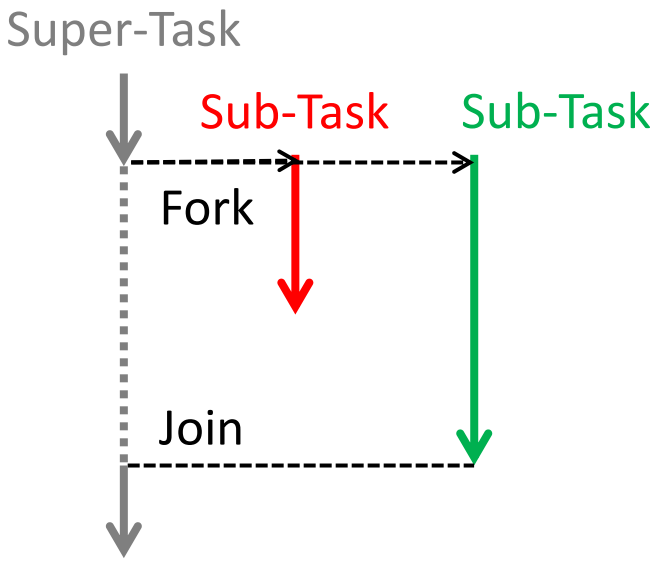
Paralleles Zählen




```
var left = threadPool.submit(() -> count(leftPart));
var right = threadPool.submit(() -> count(rightPart));
result = left.get() + right.get();
```

Rekursive Tasks





Ein Task kann Unter-Tasks starten und abwarten

Rekursives Zählen

```
class CountTask extends RecursiveTask<Integer> {
  // Constructor

  @Override
  protected Integer compute() {
    // if no or single element => return result
    // split into two parts
    var left = new CountTask(lower, middle);
    var right = new CountTask(middle, upper);
    left.fork(); right.fork();
    return right.join() + left.join();
  }
}
```

Rekursive Task Ausführung

- Expliziter Thread Pool

```
var threadPool = new ForkJoinPool();

int result = threadPool.invoke(new CountTask(2, N));
```

blockiert

- Standard Pool (Java 8)
  - `ForkJoinPool.commonPool()`

```
int result = new CountTask(2, N).invoke();
```

Fork Join Pool

- Rekursive Tasks
  - Tasks können Untertasks starten und abwarten
  - Verboten in alten Java Thread Pools (Executors) - Deadlock
- Tasks erben von `RecursiveTask<T>`
  - `T compute()`: Task Implementierung
  - `fork()`: Starte als Sub-Task in einem anderen Task
  - `T join()`: Warte auf Task-Ende und frage Resultat ab
  - `T invoke()`: Ein Sub-Task starten und abwarten
  - `invokeAll()`: Mehrere Sub-Tasks starten und abwarten
- `RecursiveAction`, falls kein Rückgabetyp
  - void statt T

Konkreter Ausbau

```
class CountTask extends RecursiveTask<Integer> {
  private final int lower, upper;

  public CountTask(int lower, int upper) {
    this.lower = lower; this.upper = upper;
  }

  protected Integer compute() {
    if (lower == upper) { return 0; }
    if (lower + 1 == upper) { return isPrime(lower) ? 1 : 0; }
    int middle = (lower + upper) / 2;
    var left = new CountTask(lower, middle);
    var right = new CountTask(middle, upper);
    left.fork(); right.fork();
    return right.join() + left.join();
  }
}
```

Keine Über-Parallelisierung

```
protected Integer compute() {
  if (upper - lower > THRESHOLD) {
    // parallel count
    int middle = (lower + upper) / 2;
    var left = new CountTask(lower, middle);
    var right = new CountTask(middle, upper);
    left.fork(); right.fork();
    return right.join() + left.join();
  } else {
    // sequential count
    int count = 0;
    for (int number = lower; number < upper; number++) {
       if (isPrime(number)) { count++; }
    }
    return count;
  }
}
```

Tuning mit Schwellwert durch Programmierer

Fork Join Pool Internals

- Automatischer Parallelitätsgrad
  - Default: #Worker Threads im Pool = #Prozessoren
  - Dynamisches Hinzufügen/Wegnehmen von Threads
  - Ziel: Genügend aktive/laufende Worker-Threads
  - Jedoch nicht garantiert (wäre bei I/O Tasks relevant)
- Common Pool
  - Verhindert Engpässe durch zu viele Thread Pools
  - Parallelitätsgrad z.T. tiefer als #Prozessoren

Mehr Details nächste Woche

Asynchrone Programmierung

Unnötige Synchronität

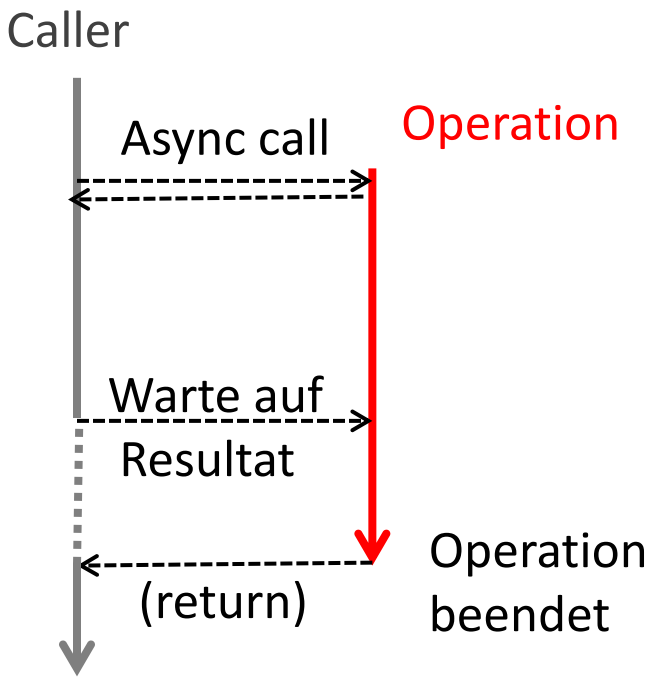
- Vielfach unnötig blockierende Methodenaufrufe
  - Langlaufende Rechnungen
  - I/O Aufrufe

```
long result = longOperation();
//other work
process(result);
```

Aufrufer unnötig blockiert

Asynchroner Aufruf

- Aufrufer soll während der Operation weiterarbeiten



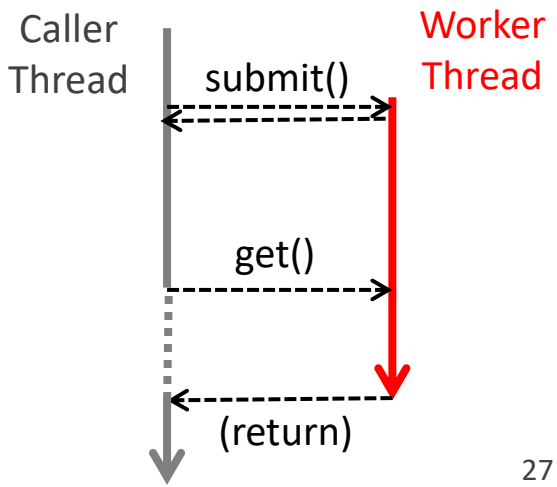
Asynchroner Aufruf (klassisch)

- Operation in Thread oder Thread Pool auslagern

Asynchrone Ausführung

```
Future<Long> future =
                    threadPool.submit(() -> longOperation());
//other work
process(future.get());
```

Resultat über Future



Moderne Asynchronität (Java 8)

- Starte asynchrone Aufgabe in Standard Pool
  - `ForkJoinPool.commonPool()`

```
CompletableFuture<Long> future =
    CompletableFuture.supplyAsync(() -> longOperation());
//other work
process(future.get());
```

`runAsync()`, falls keine Rückgabtyp

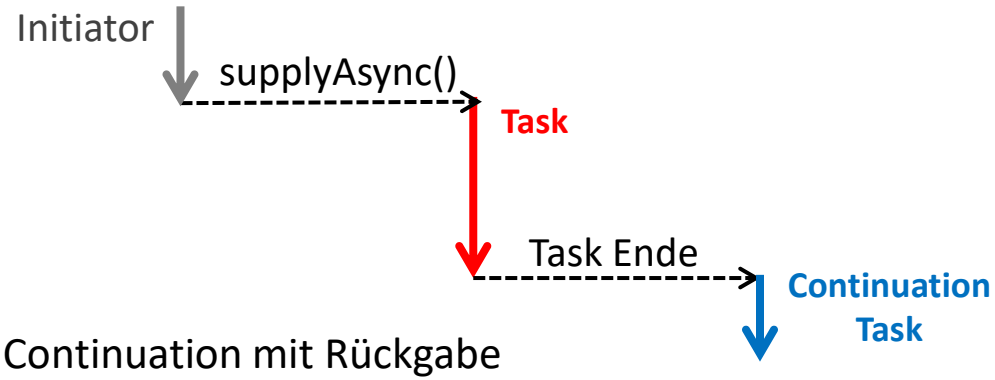
Ende des asynchronen Aufrufs

- Caller-zentrisch (Pull)
  - Caller wartet auf Task Ende und holt sich das Resultat
  - Future abfragen
- Callee-zentrisch (Push)
  - Asynchrone Operation informiert direkt über Resultat
  - Completion Callback (Continuation)

Continuation

- Folgeaufgabe an asynchrone Aufgabe anhängen
  - Ausführung, sobald vorgängiger Task fertig ist

```
CompletableFuture<Long> future =
    CompletableFuture.supplyAsync(() -> longOperation());
...
future.thenAccept(result -> System.out.println(result));
```



`thenApply()` für Continuation mit Rückgabe

Ausführung der Continuation

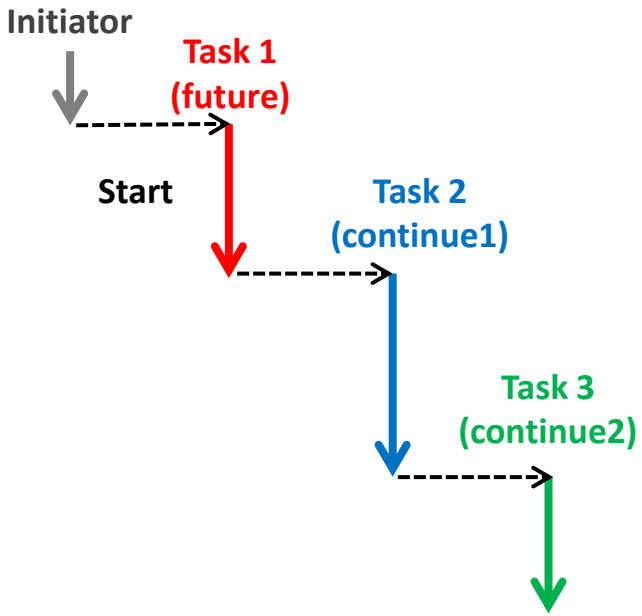
- Durch beliebigen Thread
  - Initiator, wenn Future bereits Resultat hat
  - Beliebiger Worker Thread
- Asynchrone Continuations
  - `thenAcceptAsync()` bzw. `thenApplyAsync()`

Continuation läuft potentiell in anderem Thread
=> Synchronisation beachten

Continuation-Style Programming

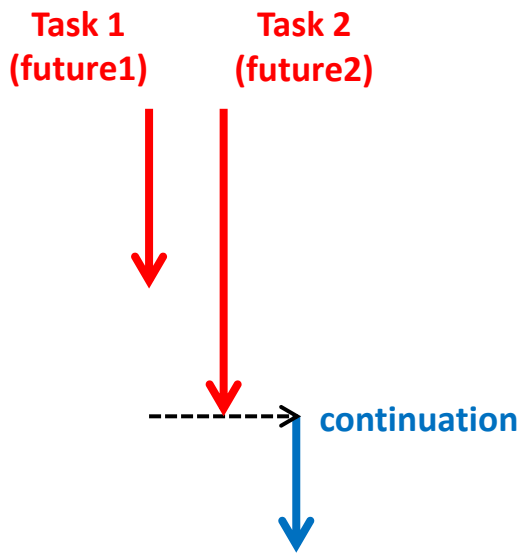
- Tasks verketten via deren `CompletableFuture`

```
future.
thenApplyAsync(second).
thenAcceptAsync(third);
```

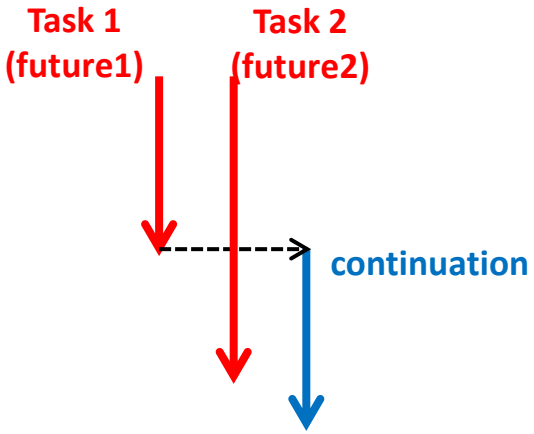


Multi-Continuation

```
CompletableFuture.allOf(future1, future2)
  .thenAcceptAsync(continuation);
```




```
CompletableFuture.any(future1, future2)
  .thenAcceptAsync(continuation);
```



Fire and Forget

- Task starten, ohne das Ende abzuwarten
  - Submitter ruft kein `get()` oder `join()` auf
  - Z.B. Continuation-Style

```
CompletableFuture.runAsync(() -> {
  …
});
```

Verschiedene Probleme

Daemon Workers

- Worker Threads in Fork-Join-Pool sind Daemon
- Anwendung kann vor Task-Ende stoppen

```
CompletableFuture.runAsync(() -> {
   …
   …
}
```

plötzliches Ende

Ignorierte Exceptions

- Exceptions in Fire & Forget Task werden ignoriert

```
CompletableFuture.runAsync(() -> {
  …
  throw e;
}
```

ignoriert

Rückblick: Lernziele

- Thread Pool Konzept verstehen
- Java Fork Join Pool einsetzen können
- Asynchrone Programmierung in Java kennen

Literatur

- B. Goetz et al. Java Concurrency in Practice, Addison Wesley, 2006
  - Kapitel 6/8: Thread Pools
- Java Tutorial: Executors, ForkJoinPool
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html
- Java CompletableFutures
  - http://www.infoq.com/articles/Functional-Style-Callbacks-Using-CompletableFuture