Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Multithreading auf höherem Niveau

- Thread vs Task
- Thread-Pools,
- Executor-Framework, Fork-Join-Framework
- Futures, Completable Futures, JavaFXConcurrent

JUC / *java util concurrency*

Java 5: Einführung des package *java.util.concurrency* kurz JUC

Wesentliche Erweiterung der Concurrency-Features von Java

Ziel: Verbesserte Anwendungsentwicklung durch höhere (abstraktere) Hilfsmittel

Wesentliche Bestandteile:

- **Kollektionsklassen mit Synchronisation und/oder Nebenläufigkeit**
- **Atomare Klassen**
- **Synchronisierer**
- **Exceutor-Framework**

Excecutor-Framework

- Bestandteil von JUC : Package java.util.concurrent
- Sinn: Nebenläufigkeit auf höherem Abstraktionsniveau
- Entkopplung von
  - Aufgabe (Task) und
  - Aufgaben-Erlediger (Thread)
- Flexible Thread-Policy möglich:

  Die Zuordnung von Aufgaben zu Threads
  - single-threaded,
  - Thread pro Task
  - Thead-Pool
  - …

  wird von der Anwendungslogik getrennt und kann unabhängig von ihr an die aktuellen Anforderungen angepasst werden.

Tasks und Threads: Aufgeben und ihre Erlediger

Thread

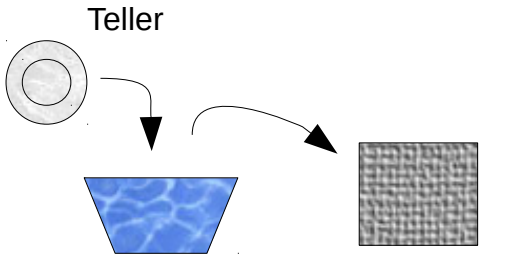
Arbeiter / Maschine / Ausführungseinheit für bestimmte oder wechselnde Aufgaben

Task

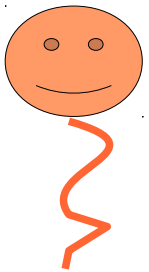
Aufgabe, das was zu tun ist

Tasks und Threads wurden früher identifiziert:

Ein Thread hat eine definierte feste Aufgabe (Task)



Task Spülen



Thread Spüler

Traditionelle Identifikation von *Task* und *Thread*

Tasks und Threads

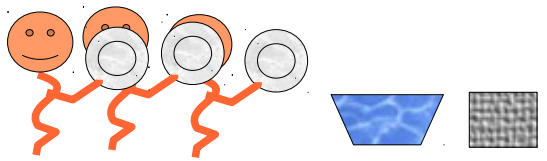
Tasks und Threads können in vielfältiger Art kombiniert werden:

- **sequentiell**
  ein Thread erledigt alle Aufgaben
- **Thread pro Task**
  für jede anfallende Aufgabe wird ein Thread erzeugt
- **Thread-Pool**
  eine Menge von Threads erledigt die anfallenden Aufgabe

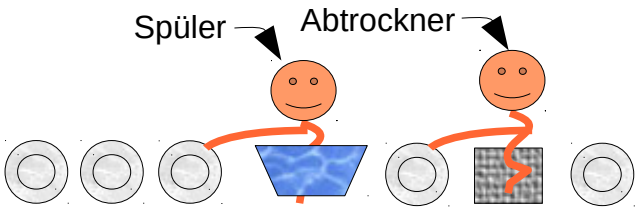
Sind Tasks und Threads getrennt
dann können sie in wechselnder Art kombiniert werden

- **je nach System-Konfiguration /- Auslastung**
- **je nach Last**

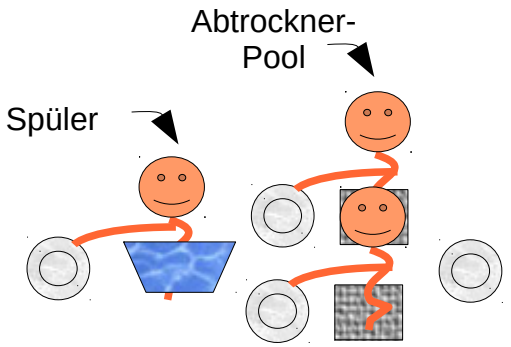
Die Zuordnung von Threads zu Tasks kann angepasst werden, ohne dass der Quellcode verändert werden muss



Task = Anfrage + Thread pro Task (sequentiell)



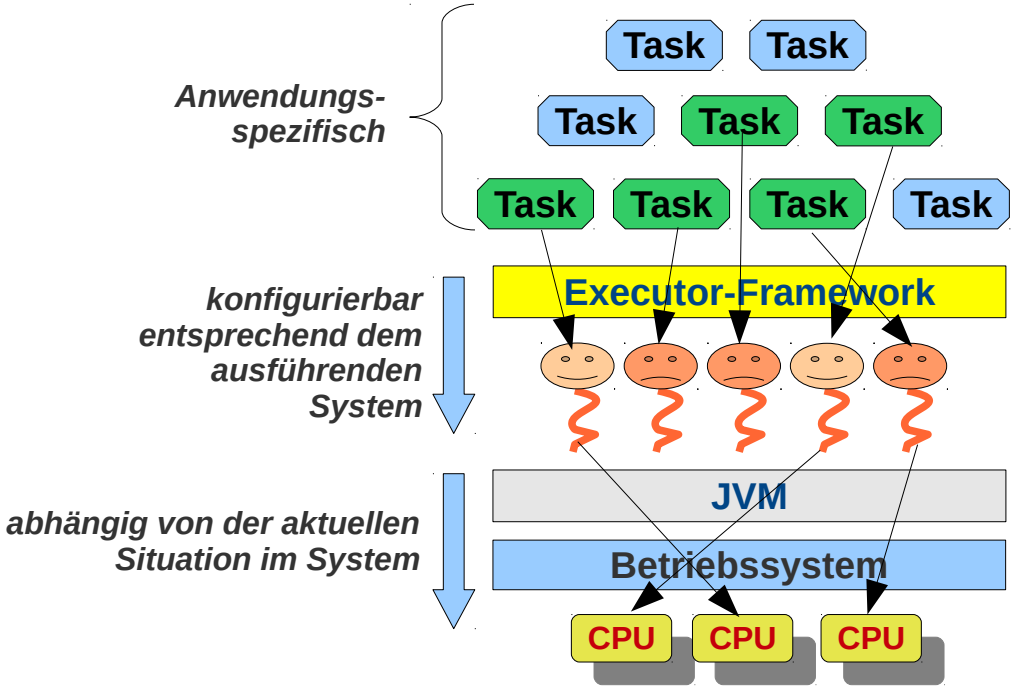
Task = Art der Tätigkeit + Thread pro Task



Task = Art der Tätigkeit + Thread-Pool für eine Task

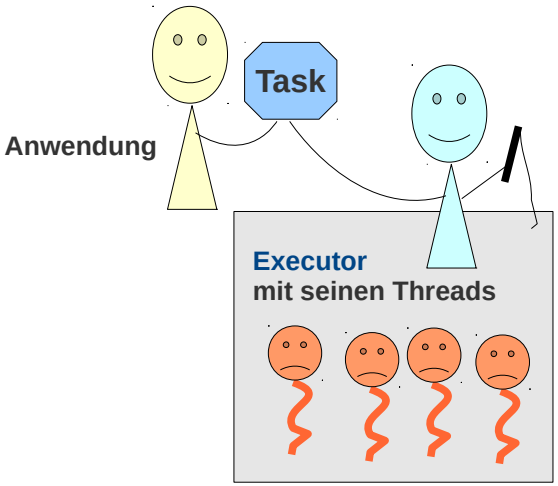
Executor-Framework

sollte stets der direkten Verwendung von Threads vorgezogen werden



Executor

- Ein Interface für unterschiedliche Threadpools
  d.h. für Erlediger von Aufgaben / Tasks
- kann Aufgaben mit Threads in unterschiedlicher Threading-Strategie erledigen



Executor

– **Beispiel:**

```
class SequentialExecutor extends Executor {

  override def execute(r: Runnable):Unit = r.run

}
```

Ein Executor führt Runnables aus.
Der einfachst-mögliche Executor führt den übergebenen Task im Kontext des Auftraggeber aus.

```
class ThreadPerTaskExecutor extends Executor {

  override def execute(r: Runnable): Unit =
           new Thread(r) start

}
```

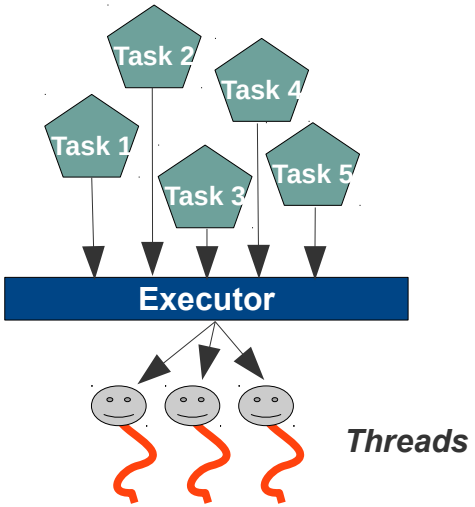
Ein ebenfalls einfacher Executor führt den übergebenen Task im Kontext eines neuen Threads aus.

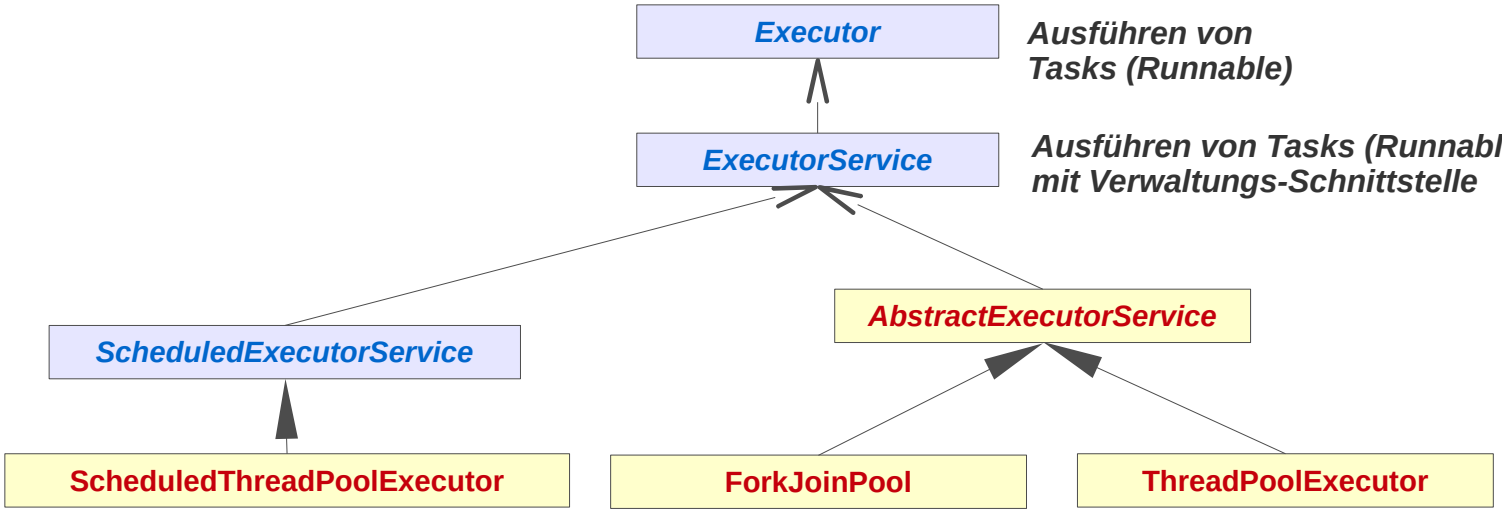
```
object Executor_Main extends App {

  def factorizationTask(x: Long) : Runnable =
    new Runnable {
      override def run(): Unit = {
        println(Factorization.factors(x))
      }
    }

  val executor: Executor = new SequentialExecutor

  executor.execute(factorizationTask(1000099000))
}
```

Anwendungscode:
Threading-Strategie ist leicht zu ändern durch Austausch des Executors

JUC Executor-Klassen

- Durch JUC werden einige Executor-Klassen zur Verfügung gestellt
- Bei allen handelt es sich um ThreadPools mit bestimmten Eigenschaften
- Alle implementieren auch das Interface ExecutorService
- Ein ExceutorService stellt Hilfsmittel zur Verwaltung der laufenden Tasks / Threads zur Verfügung
- Scala definiert keine eigenen / weiteren Exceutor-Klassen





JUC Executor-Klassen

Interfaces

- *Executor*

  Tasks ausführen
- *ExecutorService*

  Exceutor mit zusätzlichen Verwaltungsoptionen für laufende Tasks

Implementierungen

- ThreadPoolExecutor

  Task durch einen Thread aus einem Pool ausführen
- AbstractExecutorService

  Basiskalsse / Default-Implementierung für ExecutorService
- ScheduledThreadPoolExecutor

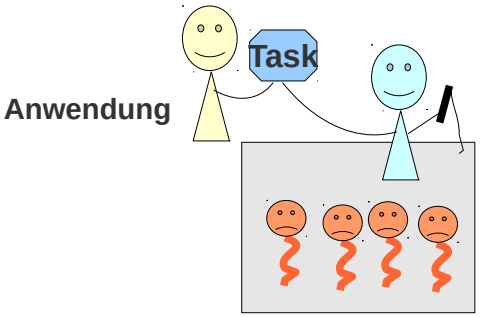
  Executer mit Einplanungen:
  - verzögerte,
  - periodische Task-Ausführung
- ForkJoinPool (ab Java 7)

  Thread-Pool, der für Tasks optimiert ist, die selbst wieder Tasks erzeugen („work stealing“)

JUC Executor-Klassen

Beispiel

```
import java.util.concurrent.Executor
import java.util.concurrent.Executors

object Executor_2_Main extends App {

  val nThreads = Runtime.getRuntime().availableProcessors()*2

  val threadPoolExecutor = Executors.newFixedThreadPool(nThreads)

  def factorizationTask(x: Long) : Runnable = new Runnable {
      override def run(): Unit = {
        println(Factorization.factors(x))
      }
    }

  for (i <- 1000099 until 1000199)
    threadPoolExecutor.execute(factorizationTask(i))

  threadPoolExecutor.shutdown()
}
```



Thread-Pool Executer

JUC: Callable und Future

Callable ~ Runnable mit Ergebnis

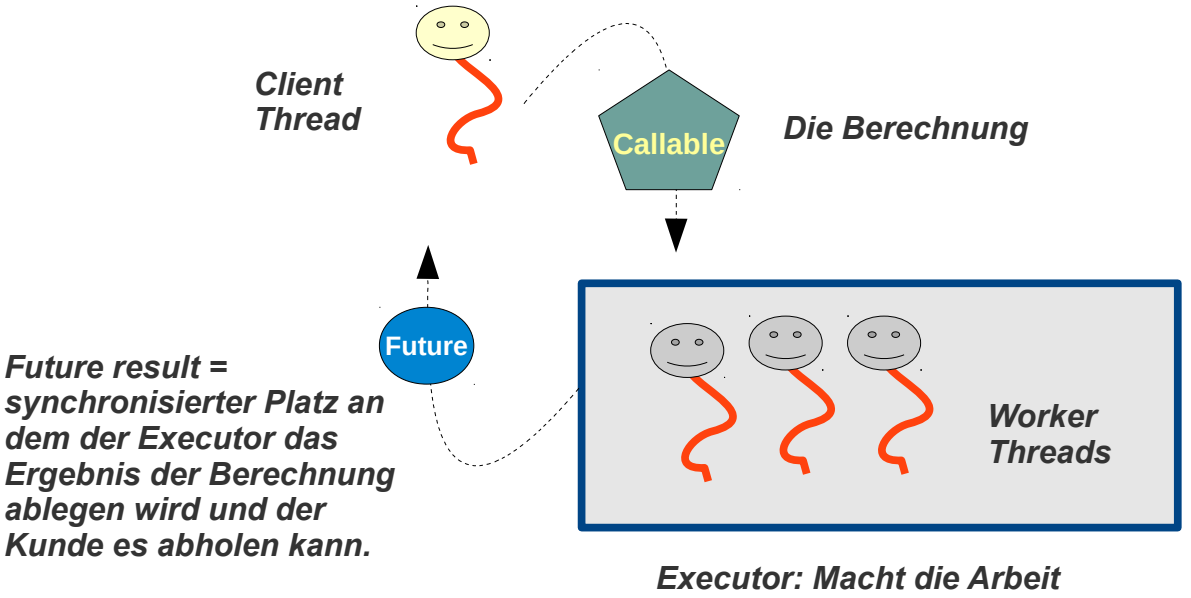
Mit Java 5 als Teil des Executor-Frameworks eingeführt

Repräsentiert eine Berechnung die an einen Executor übergeben werden kann.

Futures ~ repräsentieren laufende Berechnung (“strukturierte Callbacks”)

Mit Java 5 als Teil des Executor-Frameworks eingeführt

Erlaubt den Zugriff auf Ergebnisse, die in der Zukunft vorliegen werden



JUC: Callable und Future

Beispiel:

```
import java.util.concurrent.Callable
import java.util.concurrent.Future
import java.util.concurrent.Executors
import collection.mutable.ArrayBuffer

object Callable_Main extends App {

  val nThreads = Runtime.getRuntime().availableProcessors()*2;
  val threadPoolExecutor = Executors.newFixedThreadPool(nThreads);

  def factorizationCallable(x: Long) : Callable[List[Long]] =
    new Callable[List[Long]] {
      override def call(): List[Long] = Factorization.factors(x)
    }

  val futureResults = new ArrayBuffer[Future[List[Long]]]()

  for (i <- 1000099 until 1000199)
    futureResults += threadPoolExecutor.submit(factorizationCallable(i))

  for (f <- futureResults) {
    println(f.get())
  }

  threadPoolExecutor.shutdown()

}
```

JUC-Future-Task

Basis-Implementierung des Future-Interface

„A cancellable asynchronous computation. This class provides a base implementation of Future, with methods to start and cancel a computation, query to see if the computation is complete, and retrieve the result of the computation.“

Future-Task Beispiel 1, Abbrechbare Faktorisierung in FX-GUI 1: Berechnung starten

```
var future: Option[Future[List[Long]]] = None

buttonCompute.setOnAction(
  new EventHandler[ActionEvent] {
    def handle (ae: ActionEvent): Unit = {

      val futureTask: FutureTask[List[Long]] =
          new FutureTask(new Callable[List[Long]] {
            override def call(): List[Long] = { Factorization.factors(100000000) }
          })

      executor.execute(futureTask)

      future = Some(futureTask)

      executor.execute(
        new Runnable {
          override def run(): Unit = {
            var res = ""
            try {
              res = future.get.get.toString()
            } catch {
              case ce: CancellationException =>
                res = "canceled"
            }
            future = None
            Platform.runLater(
              new Runnable {
                override def run(): Unit = { tfOutput.setText(""+res) }
              })
          }
        }
      )
    }})
```

asynchrone Berechnung definieren

asynchrone Berechnung starten

asynchron auf Ergebnis warten

UI-Thread Ergebnis anzeigen lassen

Future-Task Beispiel 1, Abbrechbare Faktorisierung in FX-GUI 1: Berechnung stoppen

```
buttonCancel.setOnAction(
    new EventHandler[ActionEvent] {
       def handle (ae: ActionEvent): Unit = {
         if (future.isDefined) {
            future.get.cancel(true);
            future = None
          } else {
            Platform.runLater(
              new Runnable {
                override def run(): Unit = {
                    tfOutput.setText("Nothing to be canceled")
                }
              }
            )
          }
       }
    }

  )
```

findet eine Berechnung statt?

asynchrone Berechnung abbrechen

UI-Thread Meldung anzeigen lassen

JUC-Future-Management

Beispiel 2 (1) :

```
import java.util.concurrent.Executors
import java.util.concurrent.Callable
import java.util.concurrent.Future
import java.util.concurrent.TimeoutException

import scala.concurrent.duration._

object Executor_3_Main extends App {
  val THREAD_NUM = 8
  val executor = Executors.newFixedThreadPool(THREAD_NUM);

  val lx = List[Long](10240, 256, 1024, 922000120, 58, 222012000, 12)
  val futures: scala.collection.mutable.Map[Long, Future[List[Long]]]
    = scala.collection.mutable.Map()

  for (x <- lx) {
    futures(x) = executor.submit(
        new Callable[List[Long]] {
          override def call(): List[Long] = Factorization.factors(x)
        })}

  // do something else
  Thread.sleep(1000)

  // pick results
  var pending : List[Long] = List()
  for (x <- lx) {
    try {
      println(s"$x -> " + futures(x).get(500, MILLISECONDS))
    } catch {
    case e: TimeoutException =>
      println(s"$x is still in progress")
      pending = x :: pending
    }
  }
```

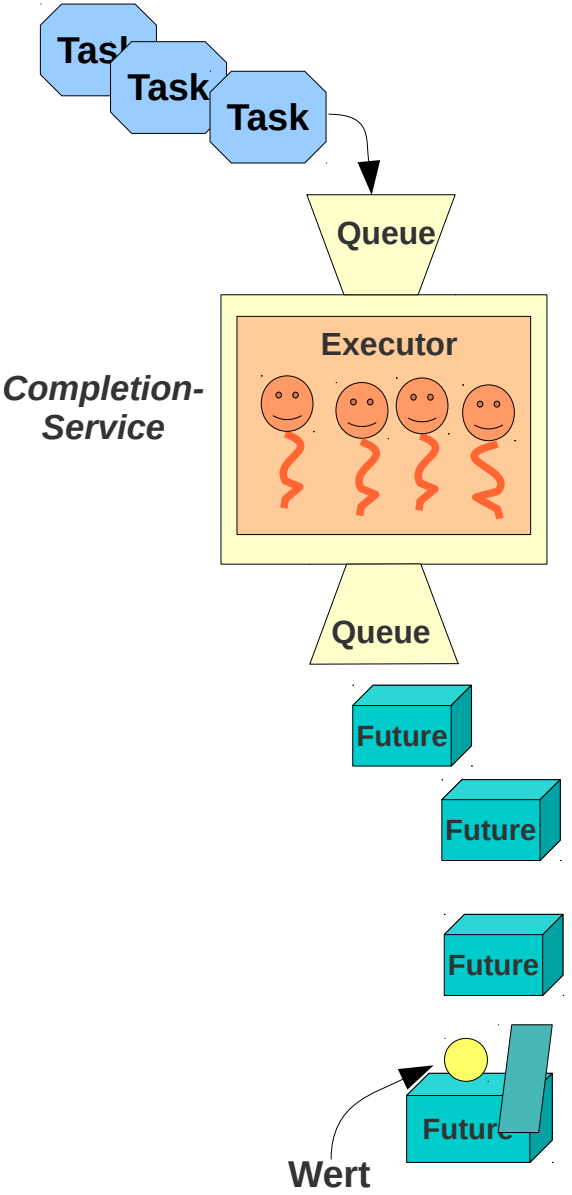
JUC-Future-Management

Beispiel 2 (2) :

```
  // do something else
  Thread.sleep(10000)

  println("some time later")

  // and now
  for (x <- pending) {
    if (futures(x).isDone()) {
      println(s"$x -> "+futures(x).get)
    } else {
      println(s"$x is still in progress")
    }
  }

  Thread.sleep(1000) // wait some more time

  println("all things must come to an end")

  for (x <- pending) {
    if (!futures(x).isDone()) {
      println(s"Factorization of $x is canceled")
      futures(x).cancel(true)
    }
  }

  executor.shutdown()
}
```

JUC CompletionService

- Schnittstelle aus *java.util.concurrent* die eine Serie von Aufgaben mit Ergebnis repräsentiert
- entkoppelt die Erzeugung von Futures von der Verwendung ihrer Ergebnisse
- ergänzt einen Executor um eine Warteschlange für die Ergebnisse



JUC CompletionService

Beispiel:

```
import java.util.concurrent.Executors
import java.util.concurrent.ExecutorCompletionService
import java.util.concurrent.Callable

object CompletionService_Main extends App {

  val executor = Executors.newFixedThreadPool(5)

  val completionS = new ExecutorCompletionService[List[Long]](executor)

  for (i <- 10100 until 10200) {
     completionS.submit(new Callable[List[Long]] {
         override def call() : List[Long] = Factorization.factors(i)
     })
  }

  for (i <- 10100 until 10200) {
    println(completionS.take().get)
  }

  executor.shutdown();

}
```

Executor-Lebenszyklus

Zustände

- **running** — Threads sind aktiv oder/und können Arbeit annehmen
- **shutting down** — Aufgaben in Arbeit werden beendet, neue Aufgaben werden nicht angenommen
- **terminated**

Management-Operationen

- `void shutdown()`
  Rücksichtsvolles Runter-Fahren: Begonnenes wird beendet
- `List<Runnable> shutdownNow()`
  Sofortiges Runter-Fahren mit Abbruch der Tasks
- `boolean isShutDown()`
- `boolean awaitTermination(long timeOut, TimeUnit unit)`
  wartet bis das Runter-Fahren beendet oder der Timer abgelaufen ist

ForkJoinPool: Java-7 Ergänzung des JUC-Executor-Frameworks

- seit Java 7 offizieller Bestandteil von Java
- Spezialisierter weiterer Thread-Pool
- Ziel: weitere Entkopplung von Tasks und Threads

  Threads können sich von blockierten Tasks lösen

  und andere übernehmen
- Threadpools generell gut geeignet z.B. für Tasks ohne starken Bezug zu einander
- ForkJoinPool gut geeignet für Tasks mit starken gegenseitigen Abhängigkeiten

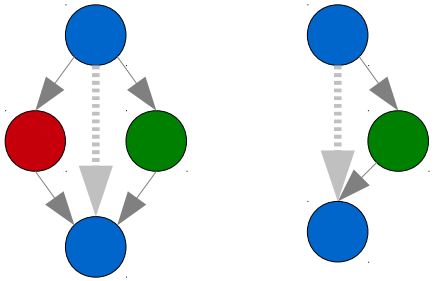
Threads: beschränkte Ressource in Threadpools

Beispiel

Parallelisierung er rekursiven Berechnung der Fibonacci-Zahlen

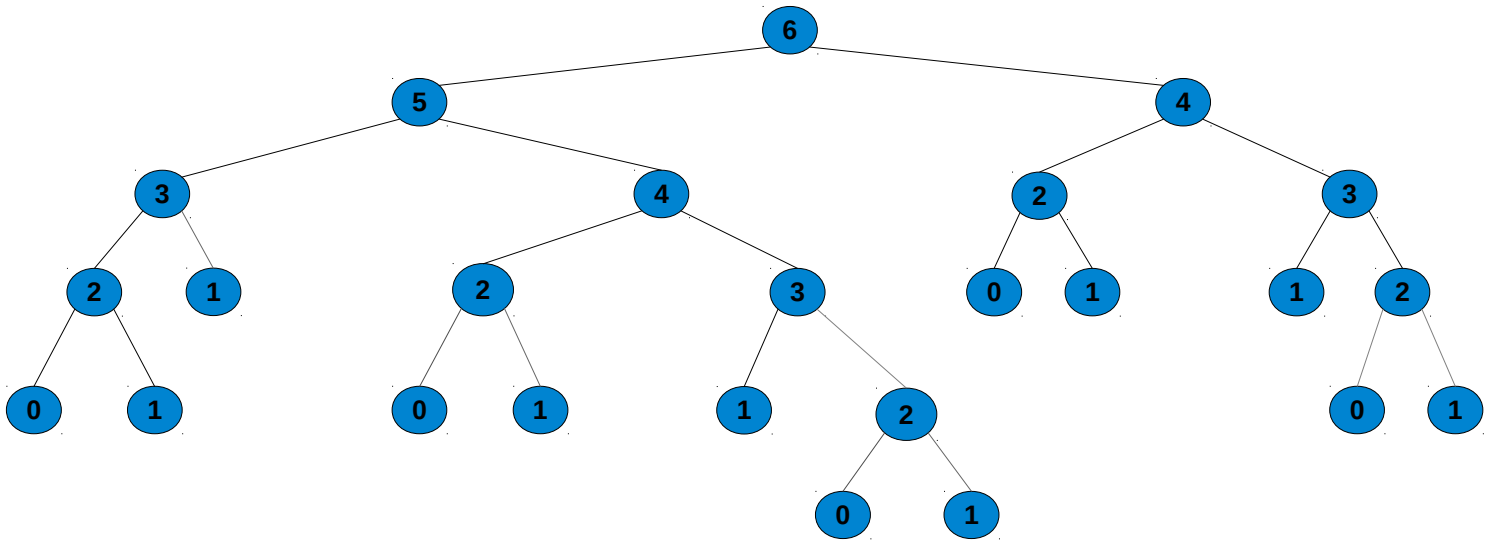
f(n) = f(n-1) + f(n-2)

- **Parallelisierung 1: Zwei Sub-Threads aufspannen**
  f(n) = { v1 = f(n-1) || v2 = f(n-2) join; v = v1+v2 }
- **Parallelisierung 2: Einen Sub-Thread aufspannen**
  f(n) = { v1 = f(n-1) || v2 = f(n-2) join; v = v1+v2 }



Problem:

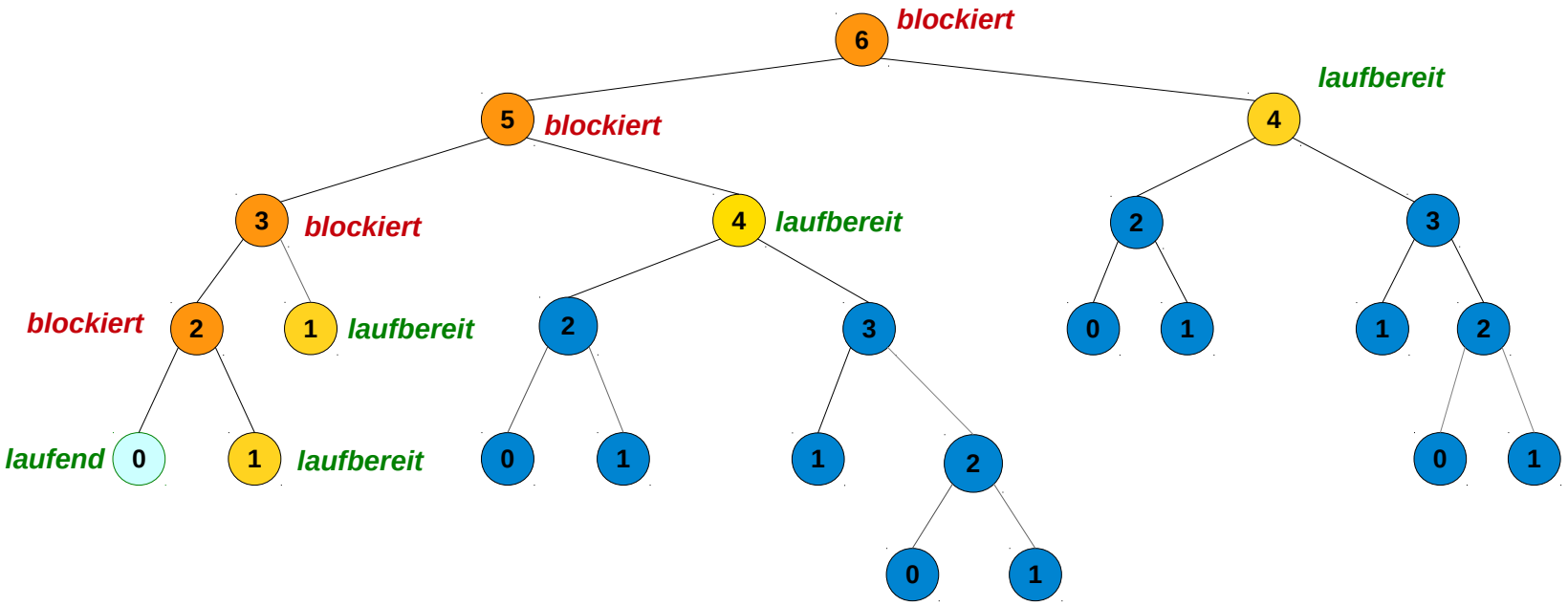
Berechnung von f(n) benötigt viele gleichzeitig existierende Threads von denen fast alle auf ihre Sub-Threads warten: Deadlock leicht möglich



Wie viele Threads muss ein Threadpool zur Verfügung stellen, damit f(6) berechnet werden kann?

Threads: beschränkte Ressource in Threadpools

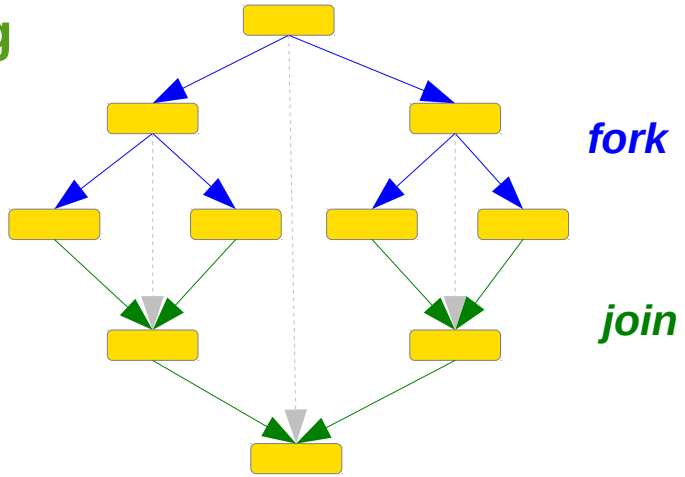
Parallel-Berechnung (z.B. bei einer rekursiven Funktion) benötigt einen sehr großen Thread-Pool auch wenn die Threads zu einem großen Teil inaktiv sind, weil sie auf das Ende von Sub-Threads warten.



Beispiel: Thread-Erzeugung bei der parallelen Berechnung der rekursiven Fibonacci-Funktion. Bei einem Thread-Pool unzureichender Größe passiert nichts mehr.

Fork-Join-Framework und Parallelberechnung

Beispiel: RecursiveTask vs Callable (1)




```
import java.util.concurrent.ForkJoinPool
import java.util.concurrent.RecursiveTask

case class FibFJ(n: Int) extends RecursiveTask[Long] {

  def compute(): Long =
     if (n < 2) n  else {
       val f1 = FibFJ(n-1)
       val f2 = FibFJ(n-2)
       f1.invoke + f2.invoke;
     }
}


object ForkJoin_Main extends App {

  val n = 25
  val parallelism = 8
  val pool: ForkJoinPool = new ForkJoinPool(parallelism);
  val fn = FibFJ(n)

  println(pool.invoke(fn))

}
```

→ 75025

Fork-Join-Framework: RecursiveTask

Beispiel: RecursiveTask vs Callable (2)

```
import java.util.concurrent.Callable
import java.util.concurrent.ExecutorService

case class Fib(n: Int, executor: ExecutorService) extends Callable[Long] {

  override def call(): Long =
     if (n < 2) {
       n
     } else {
       val f1 = executor.submit(Fib(n-1, executor))
       val f2 = executor.submit(Fib(n-2, executor))
       f1.get() + f2.get()
     }
}

object ForkJoin_Main extends App {

  val n = 25
  val parallelism = 8
  val pool: ForkJoinPool = new ForkJoinPool(parallelism);
  val fn = Fib(n, pool)

  println(pool.submit(fn).get)

}
```

→ `java.lang.OutOfMemoryError`

Bei einem Pool mit fixer Threadzahl wäre es zu einer Blockade gekommen

Fork-Join-Framework RecursiveAction

Beispiel: RecursiveAction Berechnung ohne Ergebnis

```
import java.util.concurrent.RecursiveAction
import java.util.concurrent.ForkJoinPool

case class Sort(from: Int, to: Int, a: Array[Int])
     extends RecursiveAction {

  Override def compute() {
     if (from < to) {
       val m = (from + to) / 2
       val m1 = Sort(from, m, a)
       val m2 = Sort(m+1, to, a)
       m1.fork()
       m2.fork()
       m1.join()
       m2.join()
       Sort.merge(from, m, to, a)
     }
  }

}

object Sort {
   def merge(fromIndex: Int, middleIndex: Int, toIndex: Int, toBeSorted: Array[Int]) { … }
}
```

```
object RecursiveAction_Main extends App {
  val poolsize = 8
  val pool: ForkJoinPool = new ForkJoinPool(poolsize)

  val a = Array[Int](0,1,2,9,3,8,4,5,4,6,7,8,9)

  val sortTask = Sort(0, a.length-1, a)
  pool.invoke(sortTask)
  sortTask.join()

  println(a.toList)
}
```

Completable Future

Ergänzung des JUC-Frameworks, seit Java 8 offizieller Bestandteil von Java

Future vs CompletableFuture:

Future<T>

Interface java.util.concurrent.Future<T> seit Java 1.5
hat die Methoden

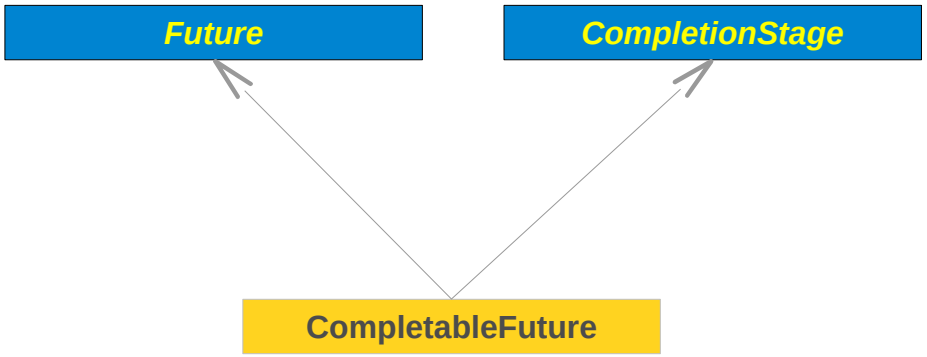
get	Ergebnis abholen
cancel	Abbrechen
isCanceled / isDone	Zustand testen

CompletableFuture<T>

Klasse java.util.concurrent.CompletableFuture<T>
Implementierung der Interfaces

- Future<T>
  Asynchrone Berechnung plus
- CompletionStage<T>
  Interface java.util.concurrent.CompletionStage<T> (seit Java 8)
  repräsentiert einen Schritt (*stage*) einer (asynchronen) Berechnung,
  der mit anderen Berechnungsschritten in vielfältiger Art
  kombiniert werden kann.

Completable Future



API: ***„A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.“***

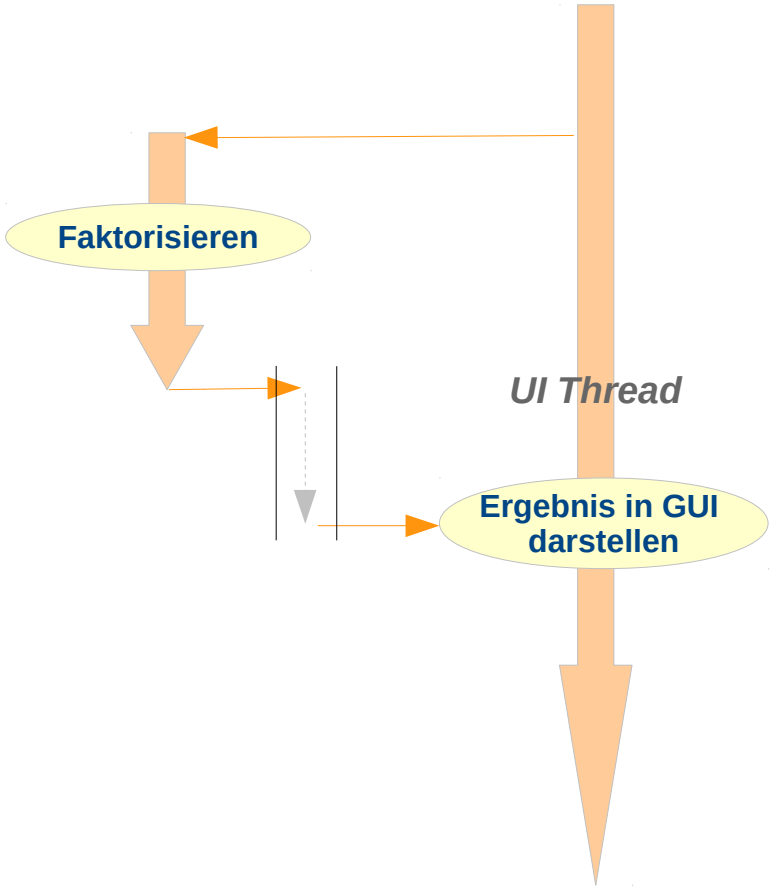
Completable Future

API: *„A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.“*

Completion Stage

Interface zur Kombination von (asynchronen) Verarbeitung-Schritten

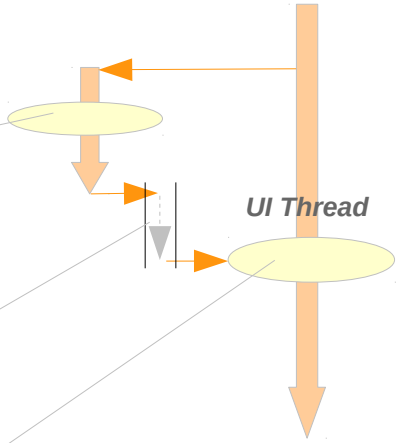
Beispiel, asynchrone Aktivitäten verketten



Completable Future

asynchrone Aktivitäten verketten: 1. mit JUC-Futures

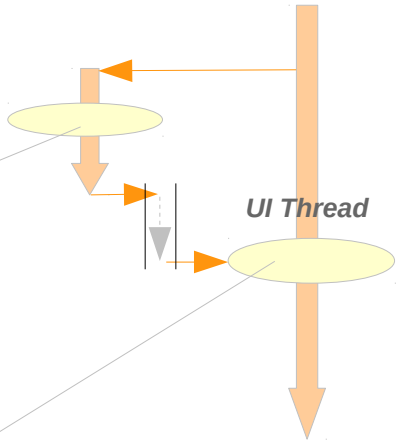
```
buttonCompute.setOnAction(
  new EventHandler[ActionEvent] {
    def handle (ae: ActionEvent): Unit = {
      val futureTask: FutureTask[List[Long]] =
          new FutureTask(new Callable[List[Long]] {
            override def call(): List[Long] = {
              val res = Factorization.factors(100000000)
              res
            }
          }
        )
      executor.execute(futureTask)

      executor.execute(
        new Runnable {
          override def run(): Unit = {
            var res = ""
            try {
              res = future.get.get.toString()
            } catch {
              case ce: CancellationException =>
                res = "canceled"
            }
            future = None
            Platform.runLater(
              new Runnable {
                override def run(): Unit = {
                  tfOutput.setText(""+res)
                }
              })
          }
      })
    }
})
```




```
val executor: ExecutorService = Executors.newFixedThreadPool(NumberOfcores)
```

Completion Stage

asynchrone Aktivitäten verketten: 2. mit CompletableFuture (CompletionStage)




```
buttonCompute.setOnAction(
  new EventHandler[ActionEvent] {
    def handle (ae: ActionEvent): Unit = {
      val completableFuture = CompletableFuture.supplyAsync(
          new Supplier[List[Long]] { // Supplier statt Callable, ist quasi das selbe
            override def get() : List[Long] = {
              Factorization.factors(100000000)
            }
          },
          executor
      )

      completableFuture.thenAcceptAsync(
        new Consumer[List[Long]] {
          override def accept(res: List[Long]) : Unit = {
            println("in thenAccept trying to set result "+ res)
             tfOutput.setText(""+res)
          }
        },
        uiExecutor
      )
    }
  }
)
```

```
val executor: ExecutorService = Executors.newFixedThreadPool(NumberOfcores)

val uiExecutor = new Executor {
  override def execute(r: Runnable): Unit = {
    Platform.runLater(r)
  }
}
```

Completable Future

Completion Stage

Interface zur Kombination von (asynchronen) Verarbeitung-Schritten

Methoden siehe API-Doku,

Beispiel

`CompletionStage<Void>`	**`thenAcceptAsync(Consumer<? super T> action, Executor executor)`** Returns a new CompletionStage that, when this stage completes normally, is executed using the supplied Executor, with this stage's result as the argument to the supplied action.

```
completableFuture.thenAcceptAsync(
  new Consumer[List[Long]] {
    override def accept(res: List[Long]) : Unit = {
      println("in thenAccept trying to set result "+ res)
       tfOutput.setText(""+res)
    }
  },
  uiExecutor
)
```

Completable Future – Complete

Completable Future beenden

Beispiel: Cancel-Implementierung

```
buttonCancel.setOnAction(
  new EventHandler[ActionEvent] {
     def handle (ae: ActionEvent): Unit = {
          completableFuture.complete(List()); // complete it with a value, computation may continue to run
     }
  }
)
```

Beenden mit einem alternativen Wert: der leeren Liste

Completable Future – Cancel

Completable Futures: Beendbar aber nicht unterbrechbar

Completable Futures

- **können „von aussen“ mit einem Ergebnis versorgt („*completed*“) werden**
- **sie können aber nicht wirklich unterbrochen werden !**

cancel

```
public boolean cancel(boolean mayInterruptIfRunning)
```

If not already completed, completes this CompletableFuture with a `CancellationException`. Dependent CompletableFutures that have not already completed will also complete exceptionally, with a `CompletionException` caused by this `CancellationException`.

Specified by:

`cancel` in interface `Future<T>`

Parameters:

`mayInterruptIfRunning` - this value has no effect in this implementation because interrupts are not used to control processing.

Returns:

`true` if this task is now cancelled

Eine Berechnung, die gestartet wurde, kann nicht mehr unterbrochen werden.

JavaFX-Concurrent

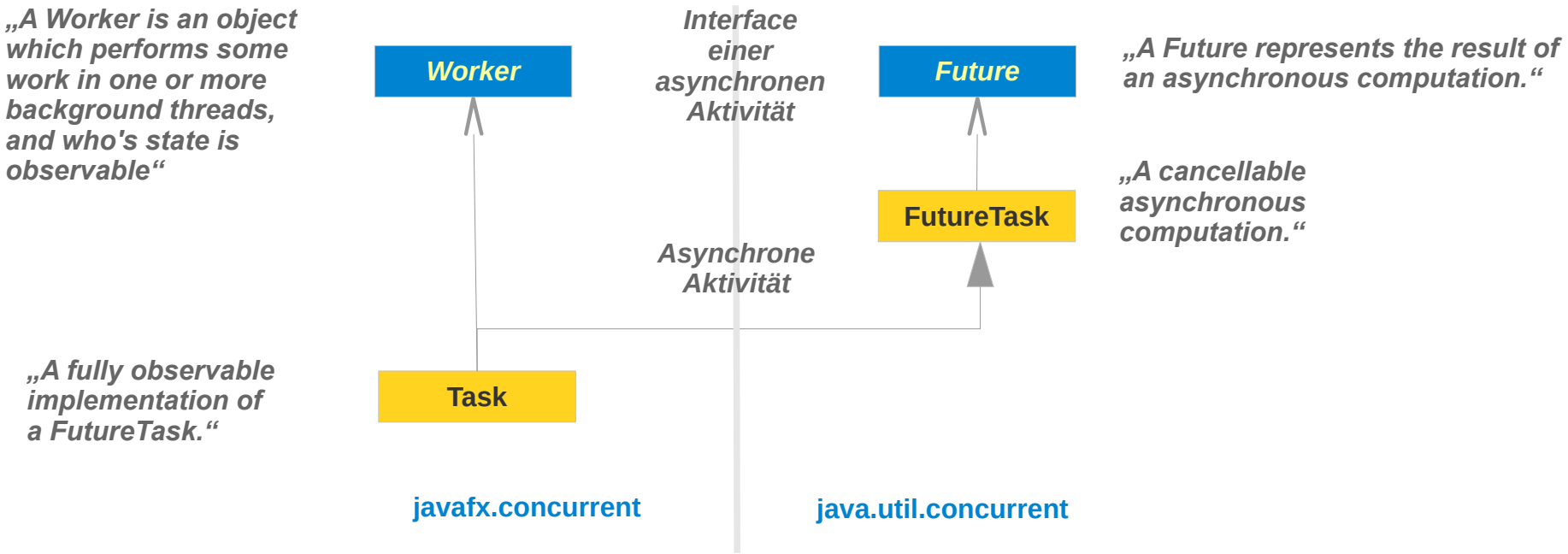
Ergänzung / Erweiterung von Java-Util-Concurrent

Interface javafx.concurrent.Worker / Class javafx.concurrent.Task

auf FX zugeschnittene Varianten von java.util.concurrent.Future / java.util.concurrent.Future

Ein Worker repräsentiert

- **eine asynchron (zum *Application-Thread*) auszuführende Aktivität**
- **deren Aktivität vom UI aus beobachtet und gesteuert werden kann**



JFXC Work / Task

Work / Task:

Ein Future (eine asynchrone Berechnung) mit weiteren Möglichkeiten

- **beobachtbar**
  Der Zustand und Fortschritt ist beobachtbar
- **cancel**
  kann mit *cancel* gestoppt werden, auch wenn schon in Bearbeitung
- **complete**
  kein Setzen des Ergebnisses von aussen (complete)
- **staging**
  kein allgemeines „Staging“, d.h. Verketten und Kombinieren von Tasks
- **Eventhandler für Zustandsübergänge**
  können registriert werden

`setOnCancelled(EventHandler<WorkerStateEvent> value)`
The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.

`setOnFailed(EventHandler<WorkerStateEvent> value)`
The onFailed event handler is called whenever the Task state transitions to the FAILED state.

`setOnRunning(EventHandler<WorkerStateEvent> value)`
The onRunning event handler is called whenever the Task state transitions to the RUNNING state.

`setOnScheduled(EventHandler<WorkerStateEvent> value)`
The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.

`setOnSucceeded(EventHandler<WorkerStateEvent> value)`
The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.