Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

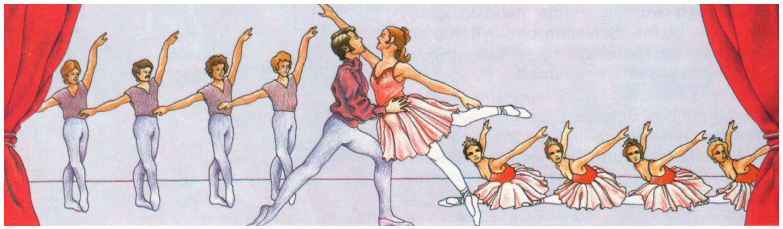
Asynchrone Verarbeitung: Futures in Scala

- Synchron vs Asynchron
- Asynchrone Verarbeitung mit Futures
- Verkettung / Kombination von Futures

Synchron vs Asynchron

Synchron – gemeinsam

zwei (oder mehr) Handlungen erfolgen gemeinsam / koordiniert



Asynchron – jeder für sich

zwei (oder mehr) Handlungen erfolgen entkoppelt, ohne Koordination / ohne gegenseitige Beeinflussung

Synchrone und Asynchrone Kommunikation

Synchrone Kommunikation

Sende- und Empfangsoperation erfolgen (mehr oder weniger) gleichzeitig
Der Sender blockiert so lange bis der Empfänger empfangen hat
Sender und Empfänger sind gekoppelt

Asynchrone Kommunikation

Sende- und Empfangsoperation erfolgen ohne direkten Bezug
Der Sender sendet und vergisst
Der Empfänger empfängt
Sender und Empfänger sind entkoppelt

Synchrone und Asynchrone Aufrufe

Synchrone Aufrufe

Aufruf einer Aktion (z.B. einer Methode) und deren Abarbeitung sind gekoppelt

Der Aufrufer blockiert (wartet) so lange bis die aufgerufene Aktion beendet ist

Synchrone Aufrufe (Funktions- oder Methodenaufrufe) sind der Normalfall bei sequentieller Verarbeitung

Asynchrone Aufrufe

Aufruf einer Aktion (z.B. einer Methode) und deren Abarbeitung sind entkoppelt

Der Aufrufer blockiert (wartet) nicht bis die aufgerufene Aktion beendet ist

Callback wird oft als eine Bezeichnung für einen asynchronen Aufruf verwendet.

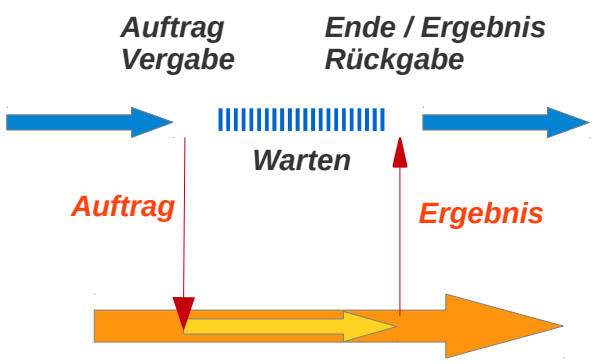
Synchrone und Asynchrone Auftragsbearbeitung

Synchrone / asynchrone Auftragsbearbeitung

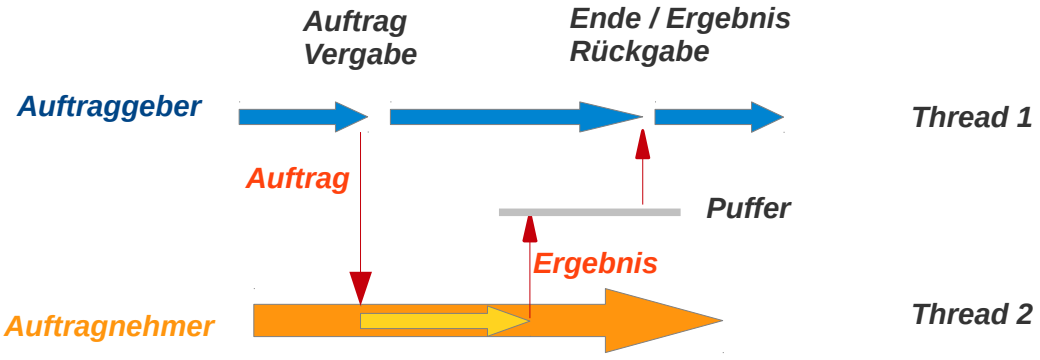
Die Auftragsbearbeitung kann synchron oder asynchron sein

- **Synchron**: Auftraggeber und Auftragnehmer sind gekoppelt: Der Auftraggeber wartet bis der Auftrag erledigt ist und nimmt eventuelle Ergebnisse entgegen
- **Asynchron**: Auftraggeber und Auftragnehmer sind entkoppelt: Der Auftraggeber wartet nicht bis der Auftrag erledigt ist.

  Eventuelle Ergebnisse müssen mit einer speziellen Aktion entgegengenommen werden.



Synchrone Aufragsverarbeitung



Asynchrone Aufragsverarbeitung

Synchrone und Asynchrone Auftragsbearbeitung

Methodenaufruf

Synchrone Auftragsbearbeitung

JUC – Executor

Asynchrone Auftragsbearbeitung mit Hilfe von Multithreading (Threadpools)

Synchrone und Asynchrone Auftragsbearbeitung

Problem

- Asynchron bedeutet „entkoppelt“ ohne Synchronisation. Oft müssen entkoppelte Aktivitäten doch irgendwann wieder synchronisiert werden. Z.B. Ergebnis einer asynchronen Berechnung abholen.
- Koordinierung asynchroner Aktivitäten:

  „Synchronisations-Hölle“ oder „Callback-Hölle“

Lösung: Abstraktion

- Trennung von
  - Auftrag und
  - Mechanismus zur Auftragsausführung
- Kapselung / Verwendung von Mustern
- Asynchrone Verarbeitung ist eine Standardaufgabe. Eine einfach zu verwendende Lösung sollte Standardangebot einer Plattform sein.
- Lösungen sind bekannt unter verschiedenen Namen
  - Future (juc.Future, juc.FutureTask, juc.CompletableFuture, jfxc.Task, ...)
  - Promise
  - ….

Diverse Konzepte asynchroner Verarbeitung in Java:

Java-5: JUC-Future

Auftraggeber und Auftragnehmer kommunizieren (bei Bedarf) über ein Future-Objekt

Java-6: ForkJoin mit RecursiveTask / RecursiveAction

Erweiterungen von Future: Spezialfall vor allem für Divide-and-Conquer-Algorithmen

Java 8: Completable Future

Completable Futures können vom Auftraggeber direkt mit einem Ergebnis versehen werden (sie werden damit beendet)

Completable Futures implementieren das Interface CompletionStage, das macht sie kombinierbar

JavaFX: Worker, Task, Service

Spezialisierte Version für interaktive graphische Anwendungen

Ziel der Future- / Worker- / Task-Abstraktion:

Asynchrone Berechnungen **einfach** und auf hoher Abstraktionsebene

- definieren / Starten
- miteinander Kombinieren
- managen / beobachten / unterbrechen / beenden

Damit Anwendungsprogrammierer – also ohne die Verwendung von Synchronisations-Mitteln – korrekten, effizienten und skalierenden nebenläufigen Code schreiben können

Futures in Scala

Elegante, ausdrucksstarke Weiterentwicklung von JUC-Futures

Beispiel, Asynchrone Faktorisierung:

```
import scala.concurrent._
import ExecutionContext.Implicits.global

object FutureEx1_Main extends App {

  val n = 7919

  // schedule async computation
  val futureFactors : Future[List[Long]] = Future{ factors(n*1024) }

  // do something else
  Thread.sleep(1000)

  println(futureFactors.value)

}
```

Wird asynchron (in einem Threadpool) berechnet

```
Some(Success(List(2, 7919)))
```

Futures in Scala

Die Verwendung impliziter Definitionen vereinfacht den Umgang mit Futures

```
import scala.concurrent._
import ExecutionContext.Implicits.global

object FutureEx1_Main extends App {
  val n = 7919
  val futureFactors : Future[List[Long]] = Future{ factors(n*1024) }
  Thread.sleep(1000)
  println(futureFactors.value)
}
```

Dies entspricht:

```
import scala.concurrent.{ Future, ExecutionContext }
import java.util.concurrent.{ ExecutorService, ForkJoinPool }

object FutureEx2_Main extends App {

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

  // Executor: oben implizit
  val executor: ExecutionContext = ExecutionContext.fromExecutorService(new ForkJoinPool(poolsize))

  val n = 7919
  val futureFactors : Future[List[Long]] = Future.apply( // oben: implizter Aufruf von apply
        {
          factors(n*1024)
        }
      )(executor)  // oben impliztes Argumet: Executor

  Thread.sleep(1000)

  println(futureFactors.value)
}
```

Futures in Scala

Verwendung impliziter Definitionen vereinfacht den Umgang mit Futures

scala.util.concurrent Object Future

```
def apply[T](body: ⇒ T)(implicit executor: ExecutionContext): Future[T]
```

Starts an asynchronous computation and returns a **Future** instance with the result of that computation.

```
trait  ExecutionContext extends AnyRef
```

An **ExecutionContext** can execute program logic asynchronously, typically but not necessarily on a thread pool.

Scala Futures: Ausführungskontext

Globaler Ausführungskontext

Die (asynchrone) Verarbeitung beruht auf einen Ausführungskontext – meist ein Threadpool
Verschiedene Ausführungskontexte können verwendet werden.
Jede Plattform bietet einen impliziten globalen Ausführungskonext.

```
ExecutionContext.Implicits.global
```

Der globale Ausführungskontext ist ein Threading-Mechanismus der aktuellen Plattform

- **ein Thread-Pool auf der JVM**
- **die Event-Queue bei Scala.JS**
  Scala.JS: Scala-Compiler mit JavaScript als „Maschinencode“
  siehe https://www.scala-js.org

Scala Futures: Ausführungskontext / Scala.JS

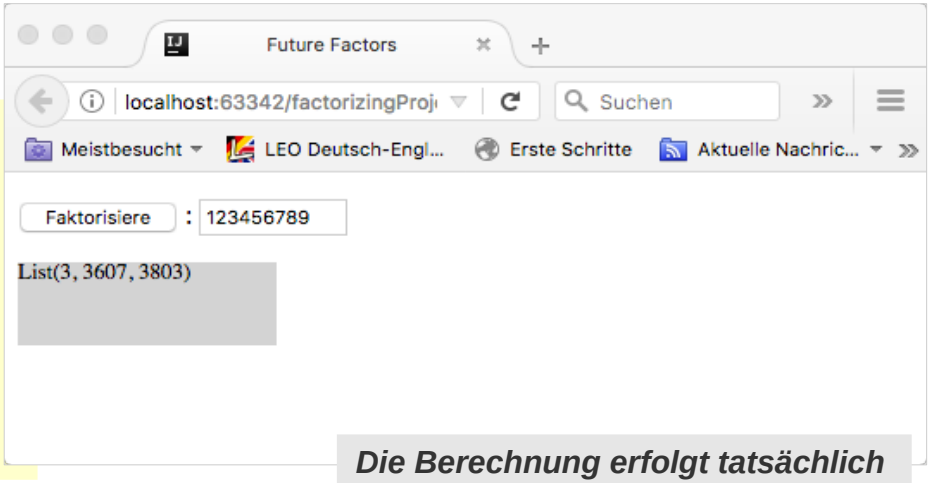
Scala.JS

Globaler Ausführungskontext

Beispiel Event-Queue als Globaler Ausführungskontext asynchroner Aktionen Scala.Js

Faktorisierung im Browser

```
@JSExport
def factors(x: String): Unit = {
  val futureFactors : Future[List[Long]] = Future{ factors(x.toLong) }
  futureFactors.onComplete {
    case Success(lst) =>
      document.getElementById("output").innerHTML = lst.toString()
    case Failure(ex) =>
      document.getElementById("output").innerHTML = ex.toString()
  }
}
```



Die Berechnung erfolgt tatsächlich synchron, der Browser blockiert bis die Fakultätsberechnung zu Ende ist!

JS ruft JS-Code der von Scala.JS erzeugt wurde.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Future Factors</title>
</head>
<body>
    <script type="text/javascript" src="./target/scala-2.12/factorizingproject-fastopt.js"></script>
    <script type="text/javascript">
      function compute(){
        var arg = document.getElementById('input').value;
        factorizing.LookIntoTheFuture().factors(arg);
      }
    </script>
    <p><button onclick="compute()">Faktorisiere</button> : <input id="input" size="10"></p>
    <p><div id="output" style="width:150px; height:50px; background:lightgrey; font-size:small; font:Courier"></div></p>
</body>
</html>
```

Scala Futures: Ausführungskontext / Scala.JS

Globaler Ausführungskontext

Beispiel Event-Queue als Globaler Ausführungskontext asynchroner Aktionen ScalaJs

```
package factorizing

import scala.util.{Success, Failure}
import scala.scalajs.js.JSApp
import scala.scalajs.js.annotation.JSExport
import scala.concurrent.{ExecutionContext, Future}
import ExecutionContext.Implicits.global
import org.scalajs.dom
import dom.document

object LookIntoTheFuture extends JSApp {

  def isPrime(n: Long): Boolean =
    Range.Long(2L, n/2+1, 1).count(n % _ == 0) == 0

  def factors(n: Long): List[Long] =
    Range.Long(2L, n/2+1, 1).filter( (i: Long) => { n%i == 0 && isPrime(i) } ).toList

  @JSExport
  def factors(x: String): Unit = {
    val futureFactors : Future[List[Long]] = Future{ factors(x.toLong) }
    futureFactors.onComplete {
      case Success(lst) =>
        document.getElementById("output").innerHTML = lst.toString()
      case Failure(ex) =>
        document.getElementById("output").innerHTML = ex.toString()
    }
  }

  def main(): Unit = {}
}
```

JavaScript Callback-Programmierung mit Scala-Futures.

(Achtung, Es geht nicht alles, was in Scala auf der JVM geht: Die Unterstützung der Nebenläufigkeit durch JavaScript ist beschränkt. Synchronisationen, z.B. wie z.B. das Warten auf das Ende einer asynchronen Aktion, werden nicht unterstützt.)

Es gibt nur einen Thread: Den Event-Dispatcher. Dieser Thread muss die Faktorisierung ausführen und in der Zeit kann er keine Ereignisse verarbeiten.

Polling: Status einer asynchronen Berechnung feststellen

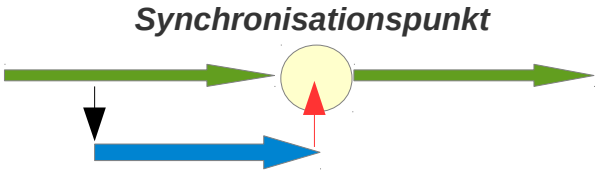
Der Status einer asynchronen Berechnung kann festgestellt werden mit:

- future.isCompleted : Boolean
- future.value : Option[Try[T]] mit den Werten:
  - None Berechnung läuft noch
  - Some(Success(...): Berechnung erfolgreich beendet
  - Some(Failure(...)) Berechnung mit Fehler beendet

Scala Futures: Polling / Stand der async. Berechnung feststellen

Warten auf das Ende einer Berechnung – 1

Der value eines Future-Objekts gibt Auskunft über den Stand der Berechnung

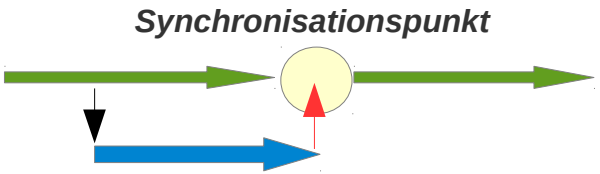



```
import scala.concurrent.{ Future, ExecutionContext, Await }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure, Try }
import scala.concurrent.duration._

object FutureEx3_Main extends App {
  val n = 99991
  val futureFactors : Future[List[Long]] = Future{ factors(n*1024) }

  var awaitedValue : Option[Try[List[Long]]] = None

  while (awaitedValue == None) {
    Thread.sleep(1000)
    println("let's see ... ")
    awaitedValue = futureFactors.value
  }

  futureFactors.value match {
    case Some(Success(lst)) => for (primefactor <- lst) println(primefactor)
    case Some(Failure(t))   => println("An error has occured: " + t.getMessage)
    case None               => /* should not occur */
  }
}
```

Warten solange value = None

Warten auf das Ende einer Berechnung – 2

Auf das Ende einer Berechnung kann gewartet werden



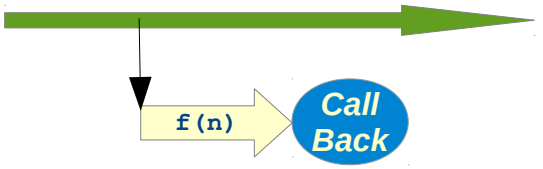

```
import scala.concurrent.{ Future, ExecutionContext, Await }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure, Try }
import scala.concurrent.TimeoutException
import scala.concurrent.duration._


object FutureEx3_Main extends App {
  val n = 99991

  val futureFactors : Future[List[Long]] = Future{ factors(n*10000) }

  try {
    Await.result(futureFactors, 20.second).foreach { println(_) }
  } catch {
    case e: TimeoutException => println("That takes more than 20 sec")
  }
}
```

Warten auf value != None

Scala Futures: Callbacks

Callbacks: Berechnungsergebnisse asynchron verarbeiten

- Auf das Ende einer Berechnung sollte möglichst nicht gewartet werden
- Asynchnrone Berechnungen sollten möglichst asynchron weiter verarbeitet werden
- Scala (2.12) Futures bietet folgende Methoden zur Verarbeitung von Ergebnissen asynchroner Operationen ():
  - onComplete
  - foreach
  - andThen
- Beispiel:



Callbacks werden an eine asynchrone Berechnung angehängt

```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }

object FutureEx4_Main extends App {
  val n = 99991

  val futureFactors : Future[List[Long]] = Future{ factors(n*1000) }

  futureFactors.onComplete {
      case Success(result) => println(result)
      case Failure(failure) ⇒ println("Failed because of " + failure)
    }

  // forget factors do something more relaxing
  Thread.sleep(10000)
}
```

Callbacks: onComplete

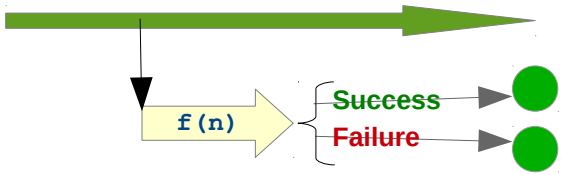
Wende eine Funktion auf das Ergebnis an

onComplete hat als Argument eine Funktion, die nach der Berechnung ausgeführt wird:

def onComplete[U](f: Try[T] =>U)

Typischerweise wird das Argument f von onComplete als Pattern-Match definiert

```
val futureLong = Future {_
     val s = scala.io.StdIn.readLine()
     s.toLong // may throw NumberFormatException
  }

futureLong.onComplete {
    case Success(v) => println(s"sucessfully read $v")
    case Failure(t) => println(s"read failed because of $t")
  }
```



Callbacks: foreach

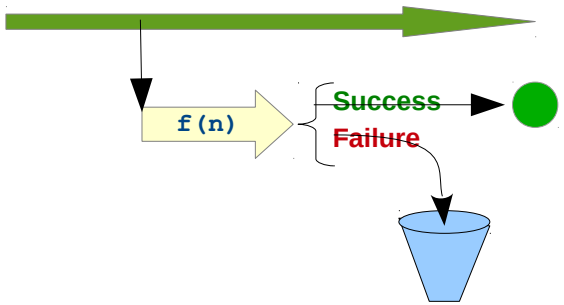
Wende eine Funktion auf das Ergebnis einer erfolgreichen Berechnung an

foreach hat als Argument eine Funktion, die auf das Ergebnis einer erfolgreichen Berechnung ausgeführt wird:

```
def foreach[U](f: T =>U)
```

Beispiel

```
val futureLong = Future {
      val s = scala.io.StdIn.readLine()
      s.toLong // may throw NumberFormatException
  }

futureLong.foreach { v => println(s"sucessfully read $v" }
```



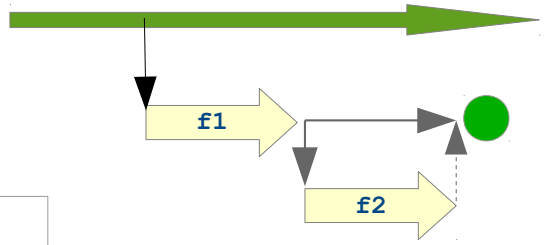
Callbacks: andThen

Wende eine Funktion auf das Ergebnis an einer asynchronen Berechnung an und liefere ein Future mit dem ersten Ergebnis (!) nachdem der Aufruf der Funktion abgeschlossen ist

andThen hat als Argument eine Funktion, die auf das Ergebnis Berechnung ausgeführt wird:

def foreach[U](f: T =>U)

Beispiel




```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{Failure, Success, Try}

object Future_Wait_App extends App {

  val f1 = Future {
    40
  }

  val f2 = f1 andThen {
    case Success(v) => { println( s"f1 ended with Success($v)"); v+2 }
    case Failure(e) => println(s"f1 Failure with $e")
  }

  f2 onComplete {
    case Success(v) => println(s"f2 completed with Success($v)")
    case Failure(e) => println(s"f2 completed with Failure $e")
  }

  Thread.sleep(50000)
}
```

andThen dient zur Festlegung von Reihenfolgen von Aktionen. Es dient nicht der Verkettung von Aktionen!

v ist das Ergebnis des Futures
v+2: diese Berechnung ist ohne Wirkung

```
f1 ended with Success(40)
f2 completed with Success(40)
```

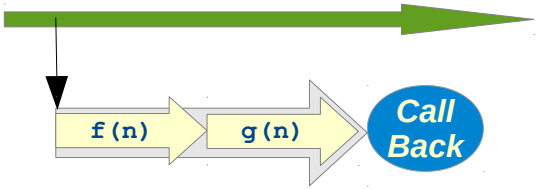
Scala Futures: Transformation von Futures

Transformationen

Futures können zu neuen Futures transformiert werden.

Eine Vielzahl von Transformationen sind möglich

map: Macht aus einem Future und einer Funktion, die einen Wert erzeugt, ein neues Future



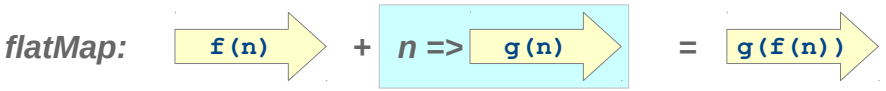

```
val f1 = Future { 40 }

val f2 = f1 map ( v => v+2 )

f2 onComplete {
  case Success(v) => println(s"f2 completed with Success($v)")
  case Failure(e) => println(s"f2 completed with Failure $e")
}
```

f2 completed with Success(42)

Scala Futures: Transformation von Futures

Transformationen

flatmap: Macht aus einem Future und einer Funktion, die ein Future erzeugt, ein neues Future




```
val f1 = Future { 40 }

val f2 = f1 flatMap (n => Future { n + 2 } )

f2 onComplete {
  case Success(v) => println(s"f2 completed with Success($v)")
  case Failure(e) => println(s"f2 completed with Failure($e)")
}
```

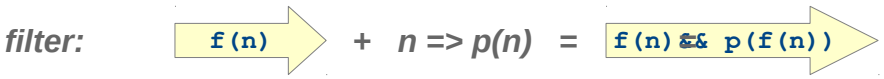
→ **f2 completed with Success(42)**

Scala Futures: Transformation von Futures

Transformationen

filter: Macht aus einem Future und einem Prädikat, ein Future das einen Wert liefert, der das Prädikat erfüllt, oder eine Exception

Beispiel:




```
val f1 = Future { 42 }

val f2 = f1 filter ( _ % 2 == 1)

f2 onComplete {
  case Success(v) => println(s"f2 completed with Success($v)")
  case Failure(e) => println(s"f2 completed with Failure($e)")
}
```

```
f2 completed with
Failure(java.util.NoSuchElementException:
Future.filter predicate is not satisfied)
```

Scala Futures: Transformation von Futures

Transformationen

transform: ~ map auf einem Try, also mit einer Option für Success und Failure

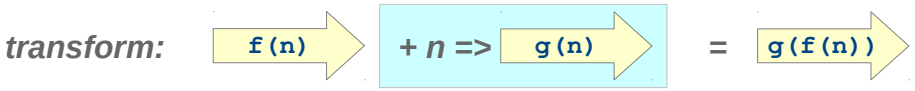

```
val f1 = Future { 42 }

val f2 = f1 filter ( _ % 2 == 1)

val f3 = f2 transform {
  case Success(v) => Success("42 is odd")
  case Failure(e) => Success("42 is even")
}

f3 onComplete {
  case Success(v) => println(s"f2 completed with Success($v)")
  case Failure(e) => println(s"f2 completed with Failure($e)")
}
```

f2 completed with Success(42 is even)

Scala Futures: Transformation von Futures

Transformationen

transformWith: ~ map auf einem Try, also mit einer Option für Success und Failure, das ein Future erzeugt




```
val f1: Future[Long] = Future { scala.io.StdIn.readLine().toLong }

val f2: Future[List[Long]] = f1 transformWith {
  case Success(v) => Future { Factorizer.factors(v) }
  case Failure(e) => f1.asInstanceOf[Future[List[Long]]]
}

f2 onComplete {
  case Success(lst) => println(lst.foldLeft("")( (acc:String, v:Long) => acc + " " + v))
  case Failure(e) => println(e)
}
```

Liese String, prüfe ob es ein Long ist und faktorisiere dann eventuell

Scala Futures: Transformation von Futures

Transformationen

zipWith: *zippe* zwei Futures und kombiniere die Paare mit einer Funktion

```
val f1 = Future { scala.io.StdIn.readInt() }
val f2 = Future { Factorizer.factors(1234432) }

val f3 = f1.zipWith(f2)(
    (v:Int, lst:List[Long]) =>
        lst.contains(v.toLong)
)

f3 foreach { b =>
  println(s"Your number ${if (b) "is" else "is not"} a factor of 1234432")
}
```

Liese eine Zahl und prüfe ob sie ein Faktor von 1234432 ist

Transformationen von Futures: Recover

Recover: Exception mit einem alternativen Wert behandeln

```
val cf = for (
     l1 <- Future{ scala.io.StdIn.readLine().toLong }.recover {
          case e : NumberFormatException => {
                  println("could not parse your first input " + e);
                  42L
             }
        };

     l2 <- Future{ scala.io.StdIn.readLine().toLong }.recover {
          case e : NumberFormatException => {
                  println("could not parse your second input " + e);
                  100000190L
             }
        };

     factors1 <- Future{ factors(l1) };
     factors2 <- Future{ factors(l2) };
     commonFactors <- Future { collection.SortedSet(factors1: _*) & collection.SortedSet(factors2: _*) }
  ) yield commonFactors
```

Für weitere Transformationen siehe API-Doku. !

Scala Futures: For-Comprehension

Futures sind monadisch

- Sie haben map / flatMap Methoden
- For-Comprehension ist möglich und wird automatisch in map / flatMap übersetzt

Beispiel:

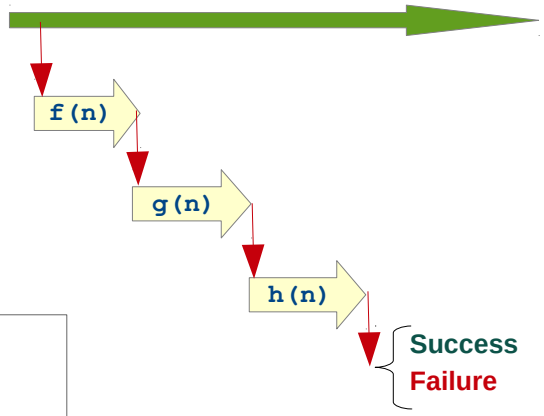
```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global

object MonadicFuture_Main extends App {

  val f1 = Future { 7 }
  val f2 = Future { 8 }

  val combinedFuture_1 =
    for (v1 <- f1;
         v2 <- f2
    ) yield v1 + v2

  val combinedFuture_2 =
    f1.flatMap { v1 =>
      f2.map { v2 =>
        v1 + v2
      }
    }

  combinedFuture_1.foreach { x=> println(s"for-comprehension result: $x") }
  combinedFuture_2.foreach { x=> println(s"map / flatMap result: $x") }

  Thread.sleep(1000)
}
```

```
map / flatMap result: 15
for-comprehension result: 15
```

Scala Futures: Transformation mit for-Ausdrücken

Futures sind ein monadischer Typ

– **Beispiel**




```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }

object FutureEx5_Main extends App {

  val futurePrimes = for (
          l1 <- Future { 99900 to 100000 };
          l2 <- Future { l1 map (x => (x, factors(x))) };
          l3 <- Future { l2 filter( x => x._2.size == 0 ) }
    ) yield l3

  futurePrimes.onComplete{
      case Success(result) => result.foreach(p => println(p._1))
      case Failure(failure) => println("Failed because of " + failure)
  }

  Thread.sleep(30000)
}
```

Scala Futures: Ergebnis asynchron verarbeiten

Futures sind ein monadischer Typ

Beispiel

```
def f(s: String): Long = s.toLong
def g(l: Long) : List[Long] = factors(l)
```

```
// forComprehension on List - 1
val primesL1 = for (
          l1 <- List("1091", "1092","1093", "1094", "1095", "1096", "1097", "1098", "1099", "1100");
          l2 = f(l1);
          l3 = g(l2);
          if l3.length < 2
    ) yield l2

println(primesL1)
```

Primzahlen sind die, für die die Liste der Primfaktoren weniger als 2 Elemente hat!

→ `List(1091, 1093, 1097)`

```
  def fL(s: String): List[Long] =
    try { List(s.toLong) } catch { case e: java.lang.NumberFormatException => List() }
  def gL(l: Long) : List[List[Long]] =
    if (l > 2) List(factors(l)) else List()
```

```
  // forComprehension on List - 2
  val primesL2 = for (
          l1 <- List("1091", "1092","Hugo", "1094", "1095", "1096", "1097", "1098", "1099", "1100");
          l2 <- fL(l1);
          l3 <- gL(l2);
          if l3.length < 2
    ) yield l2

  println(primesL2)
```

Hugo wird ignoriert.

→ `List(1091, 1097)`

Scala Futures: Ergebnis asynchron verarbeiten

Futures sind ein monadischer Typ

Beispiel: Option vs Future

```
def fO(s: String): Option[Long] =
    try { Some(s.toLong) } catch { case e: java.lang.NumberFormatException => None }

def gO(l: Long) : Option[List[Long]] = if (l > 2) Some(factors(l)) else None
```

```
// forComprehension on Option
val primeOption = for (
          l1 <- Some("1091");
          l2 <- fO(l1);
          l3 <- gO(l2);
          if l3.length < 2
    ) yield l2

println(primeOption)
```

→ `Some(1091)`

Option

```
def fF(s: String): Future[Long] = Future { s.toLong }
def gF(l: Long) : Future[List[Long]] = Future { factors(l) }
```

Future

```
// forComprehension on Future
val primeFuture = for (
          l1 <- Future { "1091" };
          l2 <- fF(l1);
          l3 <- gF(l2);
          if l3.length < 2
    ) yield l2

primeFuture.onComplete { x => println(x) }
```

→ `Success(1091)`

Scala Futures: Try

Futures und Exceptions: Try[T]

Try[T]: Hülle um

- Werte vom Typ T: Success(value: T)
- mit der Alternative Failure(error: Throwable)

Vergleichbar mit Option[T]: Hülle um

- Werte vom Typ T Some(value: T)
- mit der Alternative None

Try ist also gewissermaßen eine „Option mit einem Grund für None“

Ein Future (also eine asynchrone Aktion) wird typischerweise mit einem „Callback“ via onComplete abgeschlossen

Scala Futures: Try

Futures und Exceptions: Try[T]

Try[T] ist ein monadischer Typ:

ermöglicht Berechnungsverkettungen mit *For-Comprehension*

Beispiel: zwei long-Werte einlesen und die gemeinsamen Primfaktoren berechnen:

```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }
import scala.io.StdIn

object FutureEx8_Main extends App {

  val cf = for (
     l1 <- Future{ scala.io.StdIn.readLine().toLong };
     l2 <- Future{ scala.io.StdIn.readLine().toLong };
     factors1 <- Future{ factors(l1) };
     factors2 <- Future{ factors(l2) };
     commonFactors <- Future { collection.SortedSet(factors1: _*) & collection.SortedSet(factors2: _*) }
  ) yield commonFactors

  cf onComplete {
    case Success(factors) => println("successfully computed: " + factors)
    case Failure(t)  => println("computation failed because of " + t)
  }

  // go away and do something else
  Thread.sleep(20000)
}
```

Scala Futures: Try

Futures und Exceptions: Try[T]

Try[T] fängt nicht alle Exceptions

Fatale Exceptions werden nicht gefangen, sondern an die Anwendung durchgereicht

Fatale Exceptions sind :

- **InterruptedException**
- **LinkageError, VirtualMachineError, …**

```
val cf = for (
     L1 <-            Future{ … }
     l2 <-            Future{ …  throw new InterruptedException() … };
     Factors1 <-      Future{ … }
     Factors2 <-      Future{ … }
     commonFactors <-Future{ … }
  ) yield commonFactors

  cf onComplete {
    case Success(factors) => println("successfully computed: " + factors)
    case Failure(t)  => println("computation failed because of " + t)
  }
```

```
java.lang.InterruptedException
at slides_06.FutureEx9_Main$$anonfun$2$$anonfun$apply$1.apply(FutureEx9_Main.scala:15)
at slides_06.FutureEx9_Main$$anonfun$2$$anonfun$apply$1.apply(FutureEx9_Main.scala:15)
at scala.concurrent.impl.Future$PromiseCompletingRunnable.liftedTree1$1(Future.scala:24)
at scala.concurrent.impl.Future$PromiseCompletingRunnable.run(Future.scala:24)
at scala.concurrent.impl.ExecutionContextImpl$AdaptedForkJoinTask.exec(ExecutionContextImpl.scala:121)
at scala.concurrent.forkjoin.ForkJoinTask.doExec(ForkJoinTask.java:260)
at scala.concurrent.forkjoin.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1339)
at scala.concurrent.forkjoin.ForkJoinPool.runWorker(ForkJoinPool.java:1979)
at scala.concurrent.forkjoin.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:107)
```

Futures und Exceptions: Try[T]

Try[T] fängt nicht alle Exceptions

z.B.: InterruptedException:

- **Achtung: Interrupts werden zum Thread-Management benötigt, auch von den Threadpools die hinter den Futures wirken!**
- **Eine InterruptedException sollte nie bis auf die Thread-Ebene propagiert werden!**

Scala Futures: Verkettung

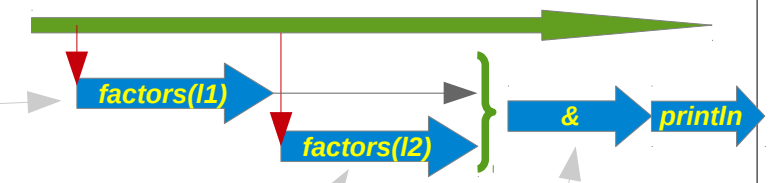
Futures sequentiell oder parallel verarbeiten

In einer for-Comprehension verkettete Futures werden sequenziell verarbeitet.

Beispiel:

```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }
import scala.io.StdIn

object FutureEx10_Main extends App {
    val cf = for (
     l1 <- Future{ 999710000 };
     l2 <- Future{ 999610000 };
     factors1 <- Future{      println(s"start factorizing $l1");
                              val f1 = factors(l1);
                              println(s"stop factorizing $l1");
                              f1    };
     factors2 <- Future{ println(s"start factorizing $l2");
                              val f2 = factors(l2);
                              println(s"stop factorizing $l2");
                              f2 };
     commonFactors <- Future { collection.SortedSet(factors1: _*) & collection.SortedSet(factors2: _*) }
  ) yield commonFactors

  cf onComplete {
    case Success(factors) => println("successfully computed: " + factors)
    case Failure(t)  => println("computaion failed because of " + t)
  }
  Thread.sleep(50000)
}
```

```
start factorizing 999710000
stop factorizing 999710000
start factorizing 999610000
stop factorizing 999610000
successfully computed: TreeSet(2, 5)
```



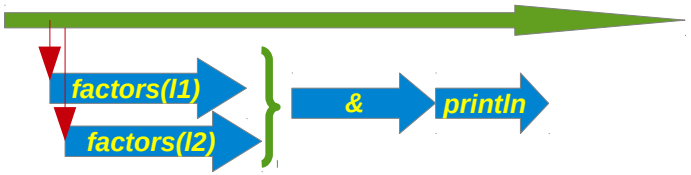
Futures sequentiell oder parallel verarbeiten

Parallelverarbeitung: Future-Objekt ausserhalb der Schleife erzeugen

Beispiel:

```
val l1 = 999710000
val l2 = 999610000
val fo1 = Future{
            println(s"start factorizing $l1")
            val f1 = factors(l1);
            println(s"stop factorizing $l1")
            f1
      }
val fo2 = Future{
            println(s"start factorizing $l2")
            val f2 = factors(l2);
            println(s"stop factorizing $l2")
            f2
      }

// 2 async computations are scheduled here

val cf = for (
   factors1 <- fo1;
   factors2 <- fo2;
   commonFactors <- Future { collection.SortedSet(factors1: _*) & collection.SortedSet(factors2: _*) }
) yield commonFactors

cf onComplete {
  case Success(factors) => println("successfully computed: " + factors)
  case Failure(t)  => println("computaion failed because of " + t)
}
```

```
start factorizing 999710000
start factorizing 999610000
stop factorizing 999610000
stop factorizing 999710000
successfully computed: TreeSet(2, 5)
```



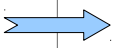
Monadische-Typen

Monadische Typen dürfen in einer For-Comprehension nicht vermischt werden

Beispiel: Liste von Zahlen faktorisieren: Entweder Liste oder Future

```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }

object MonadicMelange_Main extends App {

  // OK: for auf Liste
  val factorization_1 = for (
          x <- List(954130, 9541900, 95429000);
          y <- factors(x)
     ) yield (x, y)

  factorization_1.foreach( x => println(x._1 + " -> " + x._2))

  println()

  // OK: for auf Future
  val factorization_2 = for (
          x <- Future { 95429000 };
          y <- Future { factors(x) }
     ) yield y

  factorization_2 .foreach( x => println(x))

  Thread.sleep(5000)
}
```

```
// !OK
val factorization = for (
          x <- List(954130, 9541900, 95429000);
          y <- Future{ factors(x) }
   ) yield (x, y)
```



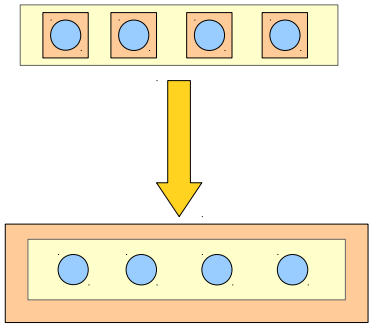
```
954130 -> 95413
9541900 -> 2
9541900 -> 5
9541900 -> 95419
95429000 -> 2
95429000 -> 5
95429000 -> 95429

List(2, 5, 95429)
```

Helfermethode Future . sequence

Helfermethode sequence: Container mit Futures => Futures von Container

wie traverse - ohne modifizierende Funktion

```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global

object TraverseEx_Main extends App {


  val factorization = Future.sequence(
            List[Long](954130, 9541900, 95429000).map(
                              (x:Long) => Future{ (x, factors(x)) }
                        )
       )

  factorization.foreach( x => println(x))

  Thread.sleep(5000)

}
```



```
List((954130,List(2, 5, 95413)), (9541900,List(2, 5, 95419)), (95429000,List(2, 5, 95429)))
```

Helfermethode Future . firstCompletedOf

Helfermethode firstCompletedOf: Wettlauf asynchroner Berechnungen sarten

kann für Timeout genutzt werden:

```
val v = scala.io.StdIn.readLong

val factorsFuture = Future { factors(v) }

val timeoutFuture = Future {
            Thread.sleep(1000)
            throw new TimeoutException("FactoizationTimeOut")
      }

val future = Future.firstCompletedOf(List(factorsFuture, timeoutFuture))

future onComplete {
  case Success(lst) => println(s"factors of $v = $lst")
  case Failure(t)   => println(s"factors of $v failed because of :" + t)
}
```

fallbackTo

Fehlerbehandlung durch ein alternatives Future

```
import scala.concurrent.{ Future, ExecutionContext }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }
import scala.io.StdIn

object FallBackTo_Main extends App {

  val cf = for (
     v <- Future{ scala.io.StdIn.readLine().toLong }.fallbackTo( Future{ 42L } ); // 42 fits anyway
     fact <- Future{ factors(v) }
  ) yield fact

  cf onComplete {
    case Success(factors) => println("successfully computed: " + factors)
    case Failure(t)  => println("computaion failed because of " + t)
  }

  Thread.sleep(10000)
}
```

recover / recoverWith

Fehlerbehandlung durch ein alternativen Wert / alternatives Future

- **recover: alternativer Wert für Fehlschlag**
- **RecoverWith: alternatives Future für Fehlschlag**

```
val f = Future { factors(-42) } recover {
    case e: IllegalArgumentException => List[Long]()
  }
```

```
val f = Future { factors(-42) } recoverWith {
    case e: IllegalArgumentException => Future { factors(998090000L) }
  }
```

Zusammenfassung: Futures

Erzeugung: Future { …. Berechnung …. }

Ergebnis verwenden:

- Await.result
- Callback registrieren: onComplete ...
- For-Comprehension
- Kombinatoren

Abbruch von aussen

- nicht möglich / vorgesehen

Promise und Future

Future

Ein Container mit einem Wert über den ich irgendwann in der Zukunft verfügen kann

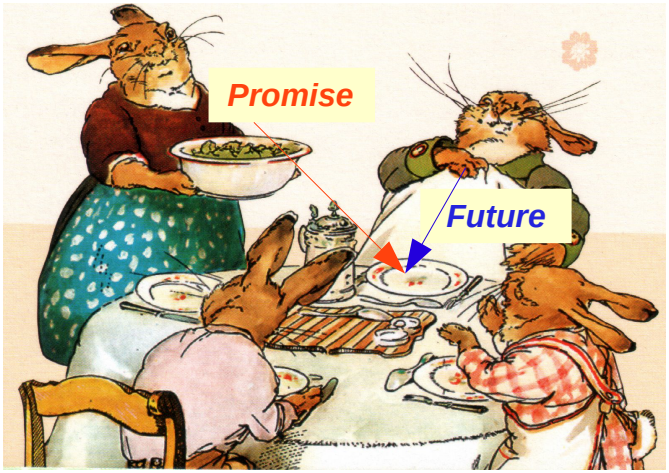
Promise

Ein Container mit einem Wert den ich irgendwann in der Zukunft liefern werde

Promise und Future

Zwei Seiten der gleichen Sache

- **Promise: Lieferantenseite**
- **Future: Konsumentenseite**



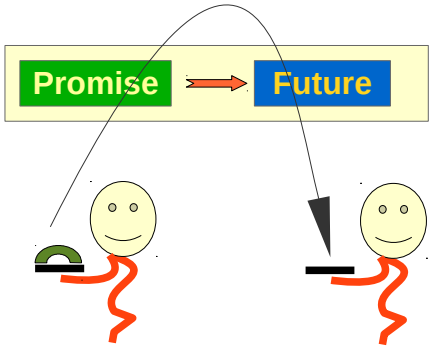
*Des einen **Versprechen** ist des anderen (Hoffnung auf die) **Zukunft***

Bildquelle: F. *Koch-Gotha,* A. Sixtus: *Die Häschenschule*, A.Hahn-Verlag, 1924

Promise und Future

Beispiel

```
import scala.concurrent.{ Promise, Future, ExecutionContext, Await }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure, Try }
import scala.concurrent.TimeoutException
import scala.concurrent.duration._

object PromiseFutureEx1_Main extends App {

  val promise = Promise[List[Long]] //a promise of a list of Longs

  val future = promise.future //a promise may create hope for the future

  // Consumer deals with futures: trusts in things that will come in future
  thread({
     try {
      Await.result(future, 30.second).foreach { println(_) }
    } catch {
      case e: TimeoutException => println("That takes too long")
    }
  }) start


  // Producer deals with promises: will make promises come true
  thread({
     promise.success(factors(954290000))
  }) start

}
```



Promise-Future:

Ein Interaktions-(Synchronisations-) Muster für eine spezielle Produzenten-Konsumenten-Beziehungen:

Der Produzent schreibt einmal(!), der Konsument kann den einen Wert beliebig oft lesen.

Gebrochene Versprechen

Ein Promise-Objekt mit einem Fehler füllen
Beispiel – 1

```
def factorsFuture(s: String) : Future[List[Long]] = {

  val promise = Promise[List[Long]]

  thread({
      try {
        val x = s.toLong
        if (x < 1000000000L) {
          try {
            promise.success(factors(x))
          } catch {
            case NonFatal(e) => promise.failure(e)
          }
        } else {
          promise.failure(new IllegalArgumentException(s"factors of $x: Are you kidding?"))
        }
      } catch {
        case e: NumberFormatException => promise.failure(new IllegalArgumentException("Numbers please"))
      }
  }) start

  promise.future

}
```

Gebrochene Versprechen:

Ein Promise-Objekt mit einem Fehler füllen
Beispiel – 2

```
import scala.concurrent.{ Promise, Future, ExecutionContext, Await }
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }
import scala.util.control.NonFatal

object PromiseFutureEx2_Main extends App {

  def factorsFuture(s: String) : Future[List[Long]] = { … }

  val str = scala.io.StdIn.readLine()
  val future = factorsFuture(str)

  future onComplete {
    case Success(lst) => println(s"factors of $str = $lst")
    case Failure(t)   => println(s"factors of $str failed because of :" + t)
  }

  Thread.sleep(10000)

}
```

Futures abbrechen

Der Konsument kann die Berechnung über Future.firstCompletedOf abbrechen
(Siehe weiter oben)

Der Produzent kann über ein *Promise*-Objekt das assozierte Future abbrechen – 1:

```
def factorsFuture(s: String) : Future[List[Long]] = {

    val promise = Promise[List[Long]]

    thread({
      try {
        val x = s.toLong
        var res: List[Long] = null
        val t = thread({
           res = factors(x)
        })
        t.start
        Thread.sleep(1000)
        if (t.isAlive()) {
          t.interrupt() //factors should take care of interrupts
          promise.failure(new java.util.concurrent.TimeoutException() )
        } else {
          promise.success(res)
        }
      } catch {
        case e: NumberFormatException => promise.failure(new IllegalArgumentException("Numbers please"))
      }
    }) start

    promise.future
}
```

Abbruch durch Produzent

Futures abbrechen

Der Produzent kann über ein *Promise*-Objekt das assozierte Future abbrechen – 2

```
object CancelPromise_Main extends App {

  val str = scala.io.StdIn.readLine()
  val future: Future[List[Long]] = Factorizer.factorsFuture(str)

  implicit val execContext = ExecutionContext.fromExecutorService(Executors.newWorkStealingPool())

  future.onComplete {
    case Success(lst) => println(s"factors of $str = $lst")
    case Failure(t)   => println(s"factors of $str failed because of :" + t)
  }

  Thread.sleep(10000)

}
```

Futures abbrechen ohne die Berechnung zu stoppen

Achtung:

- Eine synchronen Berechnung, die mit einem Future verbunden ist, kann in Scala nicht auf der Ebene des Futures oder des Promises abgebrochen werden

  (Im Gegensatz zu JUC-Futures)

- Man muss
  - wie im vorherigen Beispiel, auf die Ebene der Threads herabsteigen und die asynchrone Berechnung direkt mit Threads realisieren
  - oder JUC-Mittel verwenden
  - oder die asynchrone Berechnung einfach weiterlaufen lassen

    dann aber darauf achten, dass das Promise nicht zweimal gefüllt wird.

Futures abbrechen ohne die Berechnung zu stoppen

Beispiel: Ein cachender Faktorisierer

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.util.{ Success, Failure }

object FactorizationFutureCached {

  // cached promises
  val cache = new Cache[Long, Promise[List[Long]]](5)

  def factors(n: Long): Future[List[Long]] = cache.get(n) match {
      case Some(promise) => promise.future
      case None => {
        val promise = Promise[List[Long]]
        Future {
            Factorization.factors(n)
        } onComplete { // promise may already be completed
          case Success(lst) => if (!promise.isCompleted) promise.success(lst)
          case Failure(t) => if (!promise.isCompleted) promise.failure(t)
        }
        cache.put(n, promise)
        promise.future
      }
  }


}
```

```
// break a promise
def cancel(n: Long) {
  cache.get(n) match {
    case Some(promise) => // provide failure value (computation is not stopped!)
      promise.failure(new CancellationException)
      cache.invalidate(n)
    case _ =>
  }
}
```

Blockaden in asynchronen Aktionen

Asynchrone Aktionen sollten keine blockierenden Aktionen enthalten

Blockierende Aktionen sind gelegentlich unvermeidlich

Achtung:

- Blockierende Aktionen legen einen Thread im Threadpool lahm
- Deadlockgefahr

scala.concurrent.**blocking**

- weist auf eine Blockade hin
- die Zahl der Threads im Threadpool wird u.U. erhöht
  (ohne Wirkung bei Threadpools mit fixer Anzahl von Threads)

```
val futures2 = for (i <- 1 to 16) yield Future[Int] {
      blocking { Thread.sleep(10000) }
      i
  }

for (future <- futures2) {
     println(Await.result(future, 30 . second))
}
```

Blockaden in asynchronen Aktionen

Asynchrone Aktionen sollten keine blockierenden Aktionen enthalten

Blockierende Aktionen sind gelegentlich unvermeidlich

Achtung:

- Blockierende Aktionen legen einen Thread im Threadpool lahm
- Deadlockgefahr

scala.concurrent.**blocking**

- weist auf eine Blockade hin
- die Zahl der Threads im Threadpool wird u.U. erhöht
  (ohne Wirkung bei Threadpools mit fixer Anzahl von Threads)

```
val futures2 = for (i <- 1 to 16) yield Future[Int] {
      blocking { Thread.sleep(10000) }
      i
  }

for (future <- futures2) {
     println(Await.result(future, 30 . second))
}
```