ISA
Institut für
SoftwareArchitektur

TECHNISCHE HOCHSCHULE MITTELHESSEN

Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Asynchrone Verarbeitung: Promises und Futures

- Promise und Future in Scala
- Callback-Bridge
- Abbruch synchroner Berechnungen

Promise und Future

Future

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

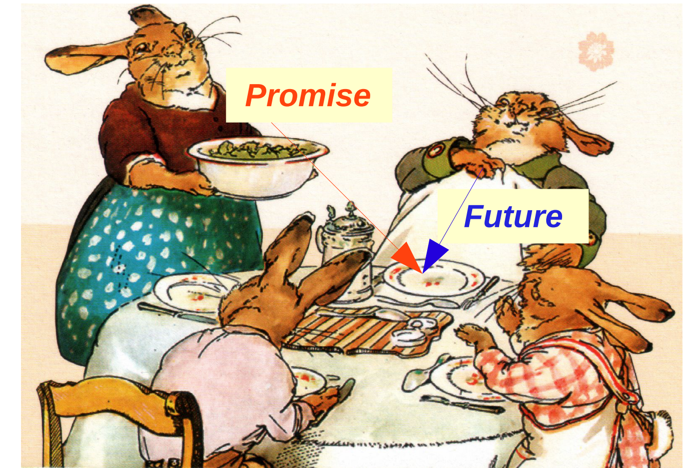
Promise

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



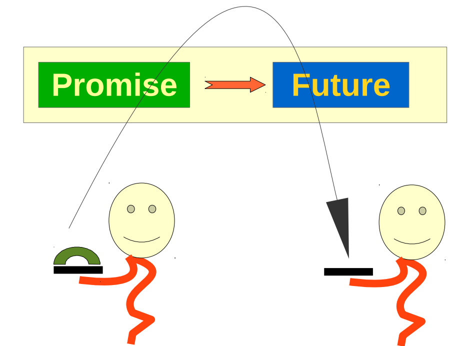
Des einen Versprechen ist des anderen (Hoffnung auf die) Zukunft

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

Promise und Future

Beispiel

```
import scala.concurrent.{ Promise, Future, ExecutionContext, Await }
import scala.concurrent.TimeoutException
import scala.concurrent.duration._

object Promise_Future_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 the future
  new 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
  new Thread ( () => {
    promise.success(Factorizer.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]]

  new 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 ?: Number too large!"))
        }
      } 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)

}
```

Callback-Future-Bridge mit Promise

Callback-Stil für Asynchrone Systemaufrufe

I/O-Operationen werden vom Betriebssystem ausgeführt

Oft kann die Operation asynchron zur gesamten Anwendung ausgeführt werden

d.h. sie wird nicht in einem Thread / Prozess der Anwendung

sondern von einem Thread / Prozess des Systems ausgeführt

Diese Operationen können gelegentlich mit Callbacks aufgerufen werden

Die Callbacks

- **sind gelegentlich die einzige Möglichkeit eigenen Code asynchron auszuführen, oder**
- **sie erlauben es eigenen Code in einer besonders effizienten Art asynchron auszuführen**

Beispiel JavaScript*

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Lies eine Datei und verarbeite sie asynchron mit dem (System-) Thread der sie gelesen hat

* Quelle: https://nodejs.org/api/fs.html

Callback-Future-Bridge mit Promise

Callback-Stil für Asynchrone Systemaufrufe

Beispiel CB-Stil in Java (Effiziente IO mit NIO-Features)

```
import java.nio.ByteBuffer
import java.nio.channels.AsynchronousFileChannel
import java.nio.channels.CompletionHandler
import java.nio.charset.Charset
import java.nio.file.{Paths, StandardOpenOption}
```

```
object AsyncFileRead_CallbackStile extends App {

  def readFile(fileName: String, cb: (Either[Throwable, String]) => Unit): Unit = try {
    val buffer = ByteBuffer.allocate(1024);
    val filePath = Paths.get(fileName);
    val asynchronousFileChannel = AsynchronousFileChannel.open(filePath, StandardOpenOption.READ)
    asynchronousFileChannel.read(
      buffer, 0, buffer,
      new CompletionHandler[Integer, ByteBuffer] { // callback: what to do after a read
        override def completed(result: Integer, attachment: ByteBuffer): Unit = {
          buffer.flip();
          val fileContent: String = Charset.forName(System.getProperty("file.encoding")).decode(attachment).toString;
          buffer.clear();
          cb(Right(fileContent))
        }
        override def failed(err: Throwable, attachment: ByteBuffer): Unit = {
          cb(Left(err))
        }
      })
  } catch {
    case err: Throwable =>
      cb(Left(err))
  }
}
```

API-Funktion im CB-Stil

Asynchrones Lesen mit Hilfe der API-Funktion `asynchronousFileChannel.read` die im Callback-Stil gehalten ist.

(`readFile` ist ein nur ein Wrapper zur Verschönerung)

```
object AsyncFileRead_App {
  def main(args: Array[String]):Unit = {
    AsyncFileRead_CallbackStile.readFile(
      "/Users/someOne/someWhere/some.txt",
      { case Right(str)  => println("Successfully read:\n" + str)
        case Left(err)   => println("Error because of: " + err.toString)
      })

    Thread.sleep(50000)
  }
}
```

Aufruf im CB-Stil

Callback-Future-Bridge mit Promise

Callback-Bridge: Callback-Stil => Future-Stil

Hülle zur Transformation des Callback-Stils in den Future-Stil

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

```
object AsyncFileRead_FutureStile extends App {
  def readFile(fileName: String): Future[String] = {
    val promise = Promise[String]
    try {
      val buffer = ByteBuffer.allocate(1024);
      val filePath = Paths.get(fileName);
      val asynchronousFileChannel = AsynchronousFileChannel.open(filePath, StandardOpenOption.READ)
      asynchronousFileChannel.read(
        buffer, 0, buffer,
        new CompletionHandler[Integer, ByteBuffer] {
          override def completed(result: Integer, attachment: ByteBuffer): Unit = {
            buffer.flip();
            val fileContent: String = Charset.forName(System.getProperty("file.encoding")).decode(attachment).toString;
            buffer.clear();
            promise.success(fileContent)
          }
          override def failed(exc: Throwable, attachment: ByteBuffer): Unit = {
            promise.failure(exc)
          }
        })
      promise.future
    } catch {
      case e: java.nio.file.NoSuchFileException => promise.failure(e)
        promise.future
    }
  }
}
```

API-Funktion im CB-Stil

Aufruf im Future-Stil

```
object AsyncFileRead_App {
  def main(args: Array[String]):Unit = {
    val future = AsyncFileRead_FutureStile.readFile("/Users/someOne/someWhere/some.txt")
    future.onComplete {
        case Success(str)  => println("Successfully read:\n" + str)
        case Failure(err)   => println("Error because of: " + err.toString)
      }
    Thread.sleep(50000)
  }
}
```

Futures abbrechen

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

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

    val promise = Promise[List[Long]]

    new Thread( () => {
      try {
        val x = s.toLong
        var res: List[Long] = null
        val t = new 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
}
```

nach 1000ms: 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)

   // execution context for onComplete
  import scala.concurrent.ExecutionContext.Implicits.global

  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)

}
```

Promise und Future

Futures abbrechen

Der Konsument kann ein Future nicht (wirklich) abbrechen
Er kann seine Wartezeit mit firstCompletedOf beschränken

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

object Promise_Future_App extends App {

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

  val futureLst: Future[List[Long]] = Factorizer.factorsFuture(str)
  val timeOut = Future { Thread.sleep(1000); "Timeout!" }

  val lstOrTimeOut = Future.firstCompletedOf( List(
    timeOut,
    futureLst.map( (lst: List[Long]) => lst.mkString(","))))

  lstOrTimeOut.onComplete {
    case Success(s) => println(s)
    case Failure(e) => println(e)
  }

  Thread.sleep(10000)

}
```

nach 1000ms:
Abbruch durch den Konsumenten.
Allerdings: die asynchrone Aktion läuft weiter. Ihr Ergebnis wird aber ignoriert werden.

Futures abbrechen

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 _ =>
    }
  }
```

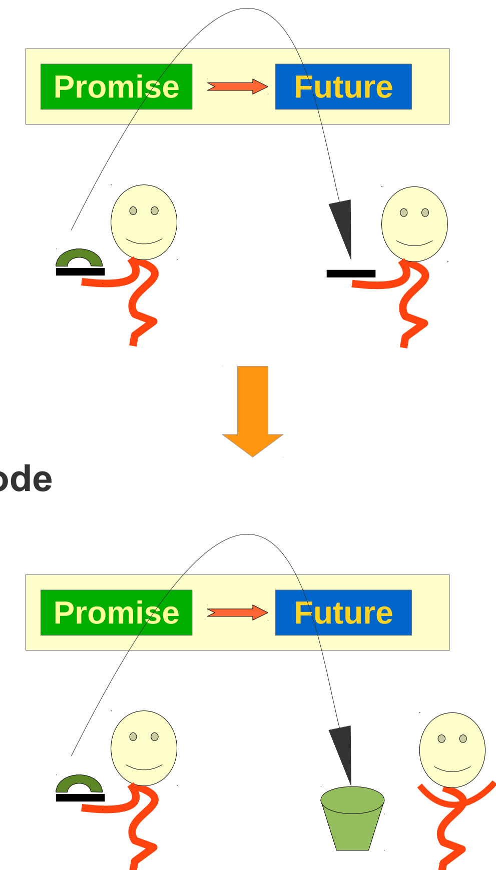
Abbruch durch den Produzenten: Future wird mit Exception gefüllt, aber die asynchrone Berechnung wird <u>nicht</u> abgebrochen. Sie läuft weiter (ins Leere, denn das Future kann nicht zweimal gefüllt werden).

Die Asynchrone Aktion wird i.A. nicht unterbrochen

Achtung Futures in Scala:

Designentscheidung Scala

- **Kein Abbruch einer asynchronen Berechnung in einem Future durch den Konsumenten**
- **Abbruch via Promise durch Zugriff auf die Thread-Ebene**



JUC-Futures

- **können mit *cancel* abgebrochen werden**
- **der Mechanismus beruht**
  - **auf korrekter interrupt-Beachtung und -Behandlung im Anwender-Code**
  - **angemessener Weiterleitung im Systemcode (des Thread-Pools)**

Die Beschränkungen von Scala

- **machen den Code robuster**
- **und führen normalerweise nicht zu Problemen:**

  asynchrone Aktionen sind sehr oft reines Warten auf I/O-Operationen, ohne Ressourcen-Verbrauch eine sinnlose asynchrone Aktion, deren Ergebnis nicht beachtet werden wird, ist oft völlig unproblematisch
- **Sollen asynchrone Aktionen wirklich abgebrochen werden, z.B. weil sie CPU-intensiv sind, dann muss dies explizit organisiert werden (via Promise)**

Die Asynchrone Aktion durch Kunden unterbrechen

Promise und Future

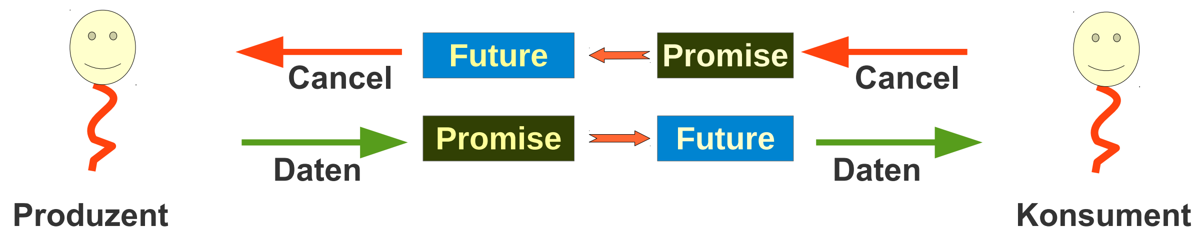
- sind die Enden eines gerichteten Kommunikationswegs

Soll der Konsument die Aktion abbrechen können

- dann muss ein umgekehrter Kommunikationsweg eingerichtet werden

Es ist naheliegend (aber nicht die einzige Möglichkeit)

- ein Paar Promise-Future für den umgekehrten Kommunikationsweg zu verwenden



Die Asynchrone Aktion durch Kunden unterbrechen

Beispiel:

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

abstract class CancelableFuture[T] {

  private val cancelPromise = Promise[Unit]
  private val cancelFuture = cancelPromise.future
  private val valueFuture = Future { computation }

  protected final def isCanceled: Boolean = cancelFuture.isCompleted

  protected def computation: T

  final def cancel = cancelPromise.trySuccess(())

  def onComplete[U](f: (Try[T]) ⇒ U): Unit = valueFuture.onComplete(f)

}
```

Future das durch den Kunde unterbrechbar ist

```
val futureString = new CancelableFuture[String]{
  override def computation: String = {
    var i = 0
    while (i < 5) {
      if (isCanceled) throw new CancellationException()
      Thread.sleep(500)
      println(s"$i working")
      i = i + 1
    }
    "42"
  }
}
```

durch Kunde unterbrechbare Berechnung der Antwort auf alle Fragen.

```
futureString onComplete {
  case Success(s) =>
      println("Computation succeeded with: " + s)
  case Failure(e) =>
      println("Computation failed with: " + e)
}
Thread.sleep(1500)
println("Going to cancel")
futureString.cancel
Thread.sleep(1500)
```

Test