Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

# Seminararbeit

**Futures and Promises in Scala**

Florian Wilhelm
Matrikelnummer: 36873
Sommersemester 2014

11. Mai 2014

**Betreuer:** Prof. Dr. Martin Sulzmann

# Inhaltsverzeichnis

# 1 Einleitung

Ziel dieser Seminararbeit ist es, die Sprachmittel *Future* und *Promise* am Beispiel der Programmiersprache *Scala* zu untersuchen. Anhand von Anwendungsbeispielen wird verdeutlicht, in welchen Situationen diese Sprachmittel hilfreich sind.

Der Code in diesem Dokument ist als Pseudocode zu verstehen, der für die Sache unwichtige Details abstrahiert.

Die Programmbeispiele (siehe [7]) wurden mit *Scala* in Version 2.10.3 geschrieben und getestet. Diese bietet eine eigene Implementierung von *Futures* und *Promises*. Eine kurze Anleitung zur Verwendung dieser Implementierung findet sich im *Scala Improvement Process*-Dokument 14 [5] und in der Dokumentation zur Programmiersprache [4].

*Futures* und *Promises* sind mit dem Ziel entworfen worden, die nebenläufige Programmierung zu vereinfachen. Nebenläufige Programmierung war bereits vor vielen Jahren sinnvoll, um den Prozessor besser auszulasten, aber seit Mehrkernsysteme Einzug in unseren Alltag gehalten haben ist sie noch wichtiger geworden als zuvor.

Dabei bietet *Scala* mehr Abstraktionsmöglichkeiten um die Arbeit mit nebenläufigen Problemstellungen zu vereinfachen. So hat es zum Beispiel das *Aktorenmodell* aus der funktionalen Programmiersprache *Erlang* übernommen.

Beide dieser Ansätze bieten unterschiedliche Herangehensweisen und Vor- und Nachteile. So erlauben *Aktoren* es, einen internen Zustand zu speichern. *Futures* dagegen sind einfach kombinierbar.

Der Einsatzzweck von *Futures* liegt vor allem darin bestimmte aufwändige Berechnungen einmalig durchführen zu lassen.

Dass es sich bei *Futures* und *Promises* um keine neuen Ideen handelt wird deutlich, wenn man [1] betrachtet. Bereits im Jahre 1977 wird die grundlegende Idee beschrieben.

*Futures* werden als Sprachkonstrukte beschrieben, die asynchron laufen. Sie könnten zum Beispiel in der Form

(ENTWEDER <ausdruck_1> <ausdruck_2> .. <ausdruck_n>)

angegeben werden. Diese Notation sollte alle n Ausdrücke nebenläufig ausführen, die Rückgabe sollte die des ersten Ausdrucks sein, der ein Ergebnis zurückliefert. Die Idee war, jeden Ausdruck auf einem eigenen Prozessor auszuführen, wenigstens aber alle Ausdrücke auf mehrere Prozessoren zu verteilen. Hier ist in einem 1977 veröffentlichten Papier bereits eine zentrale Problemstellung der heutigen Zeit beschrieben: Die effiziente Softwareentwicklung in der Multicore-Ära.

Anzumerken ist, dass nebenläufig nicht gleichbedeutend ist mit parallel. Nebenläufige Programme können auch auf Einkerncomputern ausgeführt werden. Ein positiver Effekt der nebenläufigen Programmierung ist, dass ein Programm ohne Probleme parallel laufen kann; dies ist jedoch nicht ihr alleiniges Ziel.

# 2 Futures und Promises

## 2.1 Motivation

**Blockierende Methodenaufrufe** Methodenaufrufe in üblichen Programmiersprachen sind blockierend. Das bedeutet, dass das Hauptprogramm solange blockiert wird, wie der Methodenaufruf läuft. So wird in Listing 1 die Methode `tuEtwas()` eines Objektes aufgerufen.

```
main() {
   objekt.tuEtwas()
   tuEtwasAnderes()
}
```

Listing 1: Blockierender Methodenaufruf

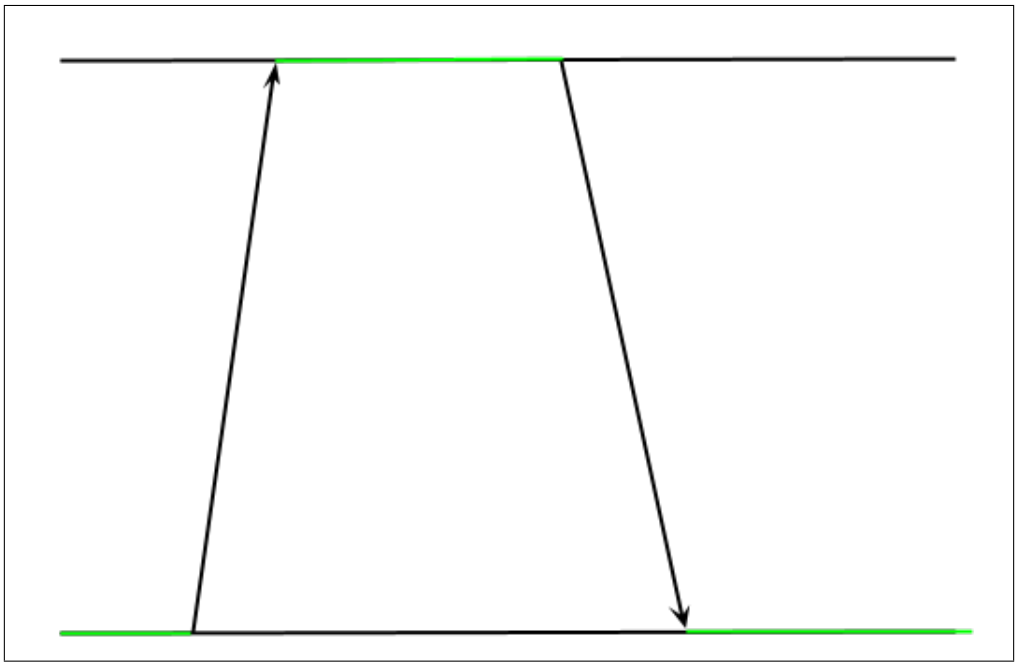

Abbildung 1: Ein blockierender Methodenaufruf

In Abbildung 1 wird dieser Sachverhalt graphisch veranschaulicht: Da der Aufrufer blockiert wird, so lange der Aufgerufene arbeitet kann immer nur ein Thread arbeiten.

Besonders in der heutigen Zeit in der immer mehr Computer mehrere Prozessorkerne haben ist dieses Verhalten nicht erwünscht. Doch auch auf einem Computer mit nur einem Prozessor kann es erforderlich sein, nebenläufig zu programmieren.

**Nicht-blockierende Methodenaufrufe** Führen wir das Schlüsselwort `nebenlaeufig` in unseren Pseudocode ein, können wir ausdrücken, dass die aufgerufene Methode nicht blockieren soll und im Hauptthread weitergearbeitet werden kann.

```
main() {
  nebenlaeufig objekt.tuEtwas()
  tuEtwasAnderes()
}
```

Listing 2: Nebenläufiger Methodenaufruf

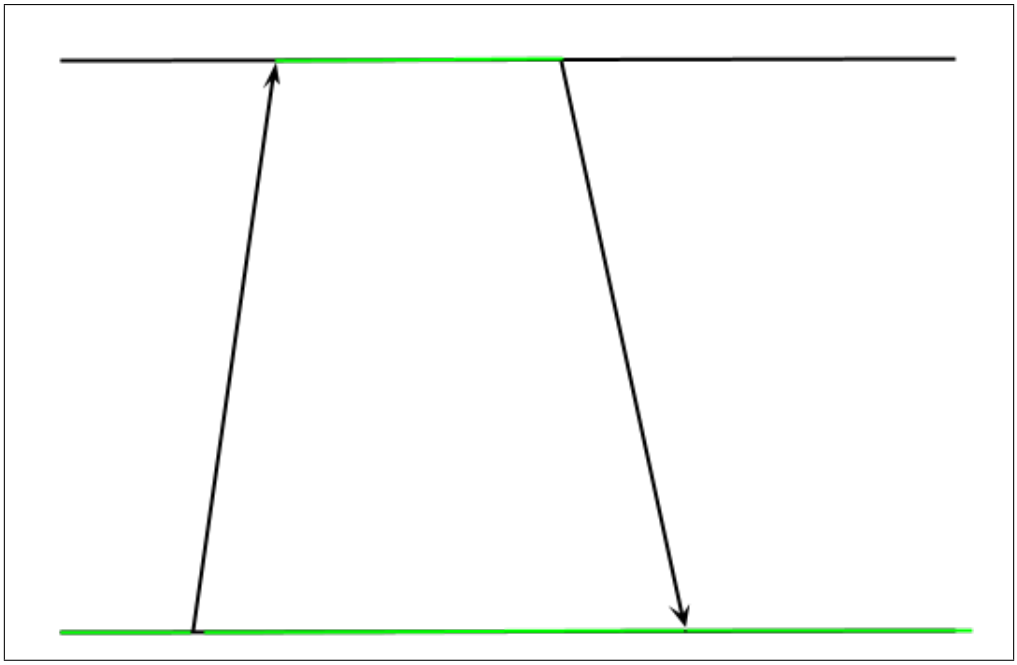

Abbildung 2: Ein nicht blockierender Methodenaufruf

Nun kann `tuEtwasAnderes()` *nebenläufig* ausgeführt werden. Damit ist es bei mehreren Prozessoren möglich, den Code parallel auszuführen.

Abbildung 2 zeigt das gewünschte Verhalten: Es entstehen zwei Threads die gleichzeitig ausgeführt werden können, wenn die Hardware dazu in der Lage ist.

Eine Analogie aus dem IT-Umfeld sind Unix-Shells. Wenn zum Beispiel ein `find`-Befehl ausgeführt wird, blockiert die Shell bis der Befehl abgeschlossen ist. Durch Anhängen eines kaufmännischen „und“-Zeichens ist es möglich, den Befehl im Hintergrund laufen zu lassen. Dadurch kann im selben Terminal weiter gearbeitet werden während der Befehl im Hintergrund läuft.

Klar ist, dass diese Technik keine Wunder bewirken kann und die zur Verfügung stehenden Ressourcen nicht vervielfacht, dennoch können sie so besser ausgenutzt werden da der Prozessor weniger häufig in einen Zustand kommt in dem er keine Arbeit hat.

## 2.2 Futures

Im SIP-14 wird ein *Future* wie folgt beschrieben:

> Futures provide a nice way to reason about performing many operations in parallel – in an efficient and non-blocking way. The idea is simple, a Future is a sort of placeholder object that you can create for a result that doesn't yet exist. Generally, the result of the Future is computed concurrently and can be later collected. Composing concurrent tasks in this way tends to result in faster, asynchronous, non-blocking parallel code.
>
> A Future object either holds a result of a computation or an exception in the case that the computation failed.

Ein *Future* ist eine Abstraktion für einen *Thread*. Dabei werden viele technische Details vor dem Anwendungsentwickler versteckt, um die man sich zum Beispiel bei der Arbeit mit klassischen Threads in Programmiersprachen wie *Java* kümmern muss.

Wichtig hervorzuheben sind die Begriffe *non-blocking* und *placeholder object*.

Auf die Eigenschaften von nicht blockierenden Aufrufen wurde bereits im vorhergehenden Abschnitt eingegangen. *Futures* sind so konzipiert, dass sie nicht blockierend arbeiten, aber wenn es erforderlich ist kann trotzdem blockiert werden bis ein *Future* fertig ist.

Der Begriff *placeholder object* sagt aus, wie das mentale Modell hinter dem Konzept *Future* zu verstehen ist. Ein *Future* ist ein Behältnis, welches irgendwann einen Wert beinhalten wird. Es ist nicht möglich zu sagen wie lange das dauern wird, es ist aber möglich zu erfahren ob dieser Wert bereits verfügbar ist. Wenn er verfügbar ist, dann kann er beliebig oft gelesen werden, aber nie überschrieben werden.

Wichtig zu erwähnen ist noch dass ein *Future* alternativ auch eine Exception beinhalten kann. Dies ist zum Beispiel sinnvoll wenn bei der Berechnung etwas schief geht oder um zu signalisieren dass ein definierter Fehlerzustand erreicht wurde.

Alles was innerhalb eines *Futures* passiert, läuft nicht im Hauptthread.

```
def main() {

  // create a future which will hold an integer value
  val f: Future[Integer] = future {
    // this will run concurrently to the main-thread
    val result = doSomeComputationHere()
    return result  // return the result of the future
  }

  f.onSuccess {
    case value => {
      // will be called when result is available
      doSomethingWithResultHere(value)
    }
  }
```

}

Listing 3: Pseudocode zum Minimalbeispiel eines Futures

In Listing 3 ist der Umgang mit einem *Future* skizziert. Sie sind generische Typen, es ist also möglich *Futures* zu verwenden, die beliebige Typen beinhalten. Der Code des *Futures* wird innerhalb des `future { ... }`-Blocks definiert. Der Rückgabewert muss dem Typen entsprechen, den der *Future* zurückgibt.

Die Verarbeitung des Ergebnisses kann in der `onComplete`-Callback-Methode erfolgen. Um nur auf erfolgreiche oder erfolglose Komplettierungen zu reagieren gibt es die `onSuccess`- bzw. `onFailure`-Methoden, welche das Abprüfen auf Erfolg ersparen.

## 2.3 Promises

Ein *Promise* hingegen wird im SIP-14 folgendermaßen beschrieben:

> A promise can be thought of as a writeable, single-assignment container, which completes a future. That is, a promise can be used to successfully complete a future with a value (by „completing“ the promise) using the `success` method. Conversely, a promise can also be used to complete a future with an exception, by failing the promise, using the `failure` method.

Das Konzept *Promise* baut auf den *Futures* auf. Es bietet einmaligen Schreibzugriff auf ein Objekt, das zum gegebenen Zeitpunkt noch nicht beschrieben werden kann, zum Beispiel weil dieser Wert noch nicht berechnet worden ist.

Er befindet sich immer in einem von zwei Zuständen:

- ***Unerledigt*** Es wurde noch keine Ergebnisse in den Promise geschrieben.
- ***Erledigt*** Der Promise wurde abgeschlossen. Entweder durch ein Ergebnis oder durch eine Exception.

```
def main() {
  // create a promise which will take an integer
  val myPromise = promise[Integer]
  // get this promises future
  val myFuture = myPromise.future

  // complete the promise
  // usually the parameter is not hardcoded
  myPromise.success(23)

  // will be called when the promise is completed
  myFuture.onSuccess {
```

```
    case result: Integer => {
      // do something with result
    }
  }
}
```

Listing 4: Pseudocode zum Minimalbeispiel eines Promises

Im Beispiel in Listing 4 ist ein *Promise* zu sehen. Jedes *Promise* beinhaltet ein *Future*. Im Programm kann so lange gerechnet werden, bis der Wert mit dem das *Promise* vervollständigt werden kann, verfügbar ist. Im Beispiel ist dieser Wert der Einfachheit halber hart codiert, in der Praxis wird er das nicht sein.

**Vergleich** von *Futures* und *Promises*.

| **Eigenschaft** | **Future** | **Promise** |
|---|---|---|
| Teil der Scala 2.10 Standardimplementierung | Ja | Ja |
| Lesbar (mehrfach) | Ja | Nein |
| Schreibbar (einmal) | Nein | Ja |
| Beinhaltet anderen | Nein | Ja |
| Kombinierbar | Ja | Nein |

## 2.4 Kombinatoren

**Motivation** Kombinatoren werden in der offiziellen *Scala*-Dokumentation beschrieben (vgl.: [4]). Sie erlauben es, mehrere *Futures* zu verbinden. Auf diese Weise können komplexe Zusammenhänge relativ einfach im Code formuliert werden.

Eine angenehme Eigenschaft von Kombinatoren ist es, dass man bei der Arbeit damit in der Welt der Futures bleibt. So bildet ein *map* einen Future auf einen anderen ab. Es gibt keinen Bruch innerhalb der Programmiersprache, das mentale Modell der Futures bleibt intakt.

Sie ersparen es dem Programmierer zudem seinen Code mit verschachtelten Callback-Methoden auszustatten. Diese machen Programme nicht nur schwer lesbar, sie vermindern zudem die Sichtbarkeit von *Futures* die innerhalb eines geschachtelten Blocks erzeugt werden, so dass außerhalb dieser Ebene nicht auf den Callback dieses *Futures* reagiert werden kann.

Ein Beispiel für *Futures* mit Kombinatoren ist in 13 gegeben. Ein Äquivalentes Programm mit verschachtelten Callbacks ist in 14 angegeben.

**map** ist einer der grundlegenden Kombinatoren. Er bildet einen *Future* nach bestimmten Regeln auf einen anderen ab.

In 5 ist ein einfaches Beispiel dafür gegeben. In diesem Beispiel wird eine Gurke geerntet. Deren Eigenschaft ist eine Krümmung (in Grad). Da sich nur einigermaßen gerade

Gurken gut verkaufen lassen, wird diese Future per `map`-Funktion auf einen anderen Future abgebildet. Hier findet die Prüfung statt, ob die Gurke verkehrsfähig ist oder entsorgt wird.

Falls sie nicht gerade genug ist, enthält der `harvestFuture` eine Exception.

Möglicherweise schlägt schon der erste *Future* fehl. Im gennanten Beispiel könnte es passieren, dass keine Gurken mehr auf dem Feld sind. In diesem Fall wird der erste *Future* eine Exception beinhalten. Die `map`-Funktion bildet in diesem Fall die Exception des ersten *Futures* auf den zweiten ab.

Im gezeigten Beispiel können daher zwei verschiedene Exceptions auftreten: Im `cucumberFuture` sowie im `harvestFuture` besteht die Möglichkeit eines Fehlers, der dazu führen würde dass das Ziel (die Gruke zu ernten) nicht erreicht wird.

```
val cucumberFuture = harvestCucumber()

val harvestFuture = cucumberFuture.map {
  cucumber =>
    if (isGoodToSell(cucumber))
      loadOnBox(cucumber)
    else
      throw new Exception("Can't sell this one")
}

harvestFuture.onComplete {
  case Success(_) => println("Harvested a Cucumber")
  case Failure(ex) => println(ex)
}
```

Listing 5: Pseudocode zum *map*-Kombinator

**filter** wendet ein Prädikat auf den Inhalt eines Futures an. Ein Beispiel für die Verwendung dieses Kombinators könnte es sein, dass ein *Future* eine Nachricht enthält und die Nachricht nur von Interesse ist wenn sie von einem bestimmten Absender stammt.

Wenn diese Bedinung erfüllt ist, so wird ein neuer Future genau diese Nachricht enthalten, wenn sie nicht erfüllt ist wird er eine `NoSuchElementException` enthalten.

```
val t: Future[Tomato] = harvestTomato()
val tt: Future[Tomato] =
  t.filter(isGood(_.color))

tt.onComplete {
  case Success(theTomato) =>
    println("Harvested " + theTomato)
  case Failure(ex) =>
    println(ex.getMessage)
```

```
}
```

Listing 6: Pseudocode zum *filter*-Kombinator

In 6 ist ein Beispiel für die Verwendung des *filter*-Kombinators gegeben. Dabei fällt eine gewisse Ähnlichkeit zum *map*-Kombinator auf. Aus Anwendungsentwicklersicht ist *filter* ein Spezialfall von *map*. So lassen sich Probleme in denen nach einem Prädikat gefiltert werden muss eleganter lösen als in einer Variante mit *map*.

*map* dagegen erlaubt es beliebige Funktionen aufzurufen. So kann die Gurke direkt innerhalb dieser Funktion verladen werden während die Tomate nach der Prüfung noch nicht verladen ist.

**fallbackTo** ist ein Kombinator der zwei *Futures* verbindet. Dabei wird überprüft, ob der erste einen Wert (und damit keine Exception) enthält. Wenn dies der Fall ist, wird dieser ausgewählt. Enthält der erste Future eine Exception, dann wird der zweite überprüft. Wenn dieser einen Wert enthält, wird er ausgewählt. Wenn er eine Exception enthält, enthält auch der neu erstellte Future eine Exception.

```
val queryGoogle: Future[ResultPage] =
  query(google, serachterm)
val queryBing: Future[ResultPage] =
  query(bing, serachterm)

val query = queryGoogle.fallbackTo(queryBing)
```

Listing 7: Pseudocode zum *fallbackTo*-Kombinator

In 7 ist ein einfaches Beispiel gegeben:

Eine Suchanfrage kann von verschiedenen Anbietern bearbeitet werden. Wenn ein Anbieter ausfällt, steht das Ergebnis eines anderen Anbieters zur Verfügung. Im ungünstigsten Fall stehen keine Anbieter zur Verfügung, dann ist es nicht möglich, ein Ergebnis anzuzeigen.

Dieser Kombinator ist geeignet wenn es eine klare Präferenz gibt, also wenn zum Beispiel von einem Suchanbieter bessere Ergebnisse zu erwarten sind.

Es ist auch möglich mehrere Aufrufe hintereinander zu schreiben, so könnte in diesem Beispiel mit Yahoo noch ein Anbieter für Suchergebnisse aufgenommen werden (Listing 8).

```
val query = queryGoogle.fallbackTo(queryBing).fallbackTo(
    queryYahoo)
```

Listing 8: Pseudocode zum *fallbackTo*-Kombinator mit drei Futures

**firstCompletedOf** wählt den *Future* aus, der als erstes beendet ist. Dies ist nützlich wenn es um Geschwindigkeit geht. Wenn zum Beispiel ein Notfall passiert und über verschiedene Kanäle ein Notruf abgesetzt wird, ist das einzig sinnvolle Kriterium zur Auswahl, über welchen Kanal als erstes Hilfe verfügbar ist.

Wichtig zu beachten ist dabei, dass die anderen *Futures* nicht abgebrochen werden wenn der erste ausgewählt ist. Wenn diese zum Beispiel rechenintensive Operationen durchführen obwohl ein anderer *Future* bereits fertig ist, so werden Ressourcen verschwendet.

```
val doctor1: Future[String] = callDoctor(channel1)
val doctor2: Future[String] = callDoctor(channel2)
val doctor3: Future[String] = callDoctor(channel3)

val doctors = List(doctor1, doctor2, doctor3)

val f = Future.firstCompletedOf(doctors)

f.onSuccess {
  case value => println("The first doctor is here.")
}
```

Listing 9: Pseudocode zum *firstCompletedOf*-Kombinator

Listing 9 skizziert diesen Anwendungsfall. Es werden drei *Futures* erzeugt die jeweils einen Notruf repräsentieren. Auf eine Liste von diesen *Futures* kann die `firstCompletedOf`-Funktion angewandt werden.

Diese wählt den ersten *Future* aus, der einen Wert zurückliefert und erstellt einen neuen *Future* dafür. Nun ist es möglich auf die Callback-Methode dieses *Futures* zu warten.

**andThen** ist geeignet für Operationen die Seiteneffekte haben. Er erlaubt es eine Kette von verarbeitungsschritten bezüglich eines *Futures* zu erzeugen. Dabei ist es garantiert dass die Reihenfolge eingehalten wird, in der die *andThen* Aufrufe im Quelltext stehen.

Jeder Aufruf des *andThen*-Kombinators nimmt einen *Future*, wendet die angegebene Funktion auf diesen an und gibt einen neuen zurück der das Ergebnis dieses *Futures* enthält.

```
val signalFuture = future {
  readSignal()
} andThen {
  case _ => {
    logSignal()
  }
} andThen {
  case _ => {
    checkSignals()
  }
}
```

Listing 10: Pseudocode zum *andThen*-Kombinator

In Listing 10 ist ein Beispiel für die Verwendung angegeben. Es werden Signale empfangen, diese werden im Log gespeichert und danach ausgewertet. Hier kommt es auf die Reihenfolge an, zudem werden Nebeneffekte ausgelöst.

So wird zum Beispiel in `logSignal()` eine Liste von empfangenen Signalen angelegt. Die Funktion `checkSignals()` verändert eine Zustandsvariable in der gespeichert ist, ob die empfangenen Signale den erwarteten ensprechen.

**For Comprehension** ist ein grundlegender Bestandteil der Sprache *Scala*, den es vor *Futures* und *Promises* gab. Dieser erinnert, auch aufgrund des Schlüsselworts *for*, an Schleifen, wie sie aus imperativen und objektorientierten Programmiersprachen bekannt sind.

```
val seq = Seq(1, 2, 3, 4, 5)
for (i <- seq if i % 2 == 0) yield i
```

Listing 11: For Comprehension

In Listing 11 ist ein minimales Beispiel für eine *For Comprehension* gegeben.

Durch den `<-`-Operator wird nach und nach jedes Element der Sequenz an die Variable `i` gebunden.

Für Entwickler die mit imperativen/objektorientierten Sprachen vertraut sind erinnert dies an das *for each*-Konstrukt.

In `yield` kann auf das Ergebnis reagiert werden. In diesem Beispiel passiert nicht mehr als das das Ergebnis auf die Kommandozeile geschrieben wird.

**Beispiel** für die Kombination von *Futures* mittels einer *For Comprehension*

```
def getNewTireAsFuture(): Future[Tire] = future {
  return getNewTire()
}

def replaceTire(tire: Tire): Future[Boolean] = future {
  return replace(tire)
}

def main() {
  // concurrent
  for {
    tire <- getNewTireAsFuture()
    success <- replaceTire(tire)
  } yield println("got response " + success)

  // main thread
  cleanCar()
}
```

Listing 12: Pseudocode zur *For Comprehension*

In diesem Beispiel wird an einem PKW gearbeitet. Es gibt zwei Aufgaben, die erledigt werden müssen: Ein Reifen muss gewechselt werden (dies unterteilt sich in die voneinander abhängigen Teilaufgaben *Neuen Reifen besorgen* und *Neuen Reifen montieren*) und das Fahrzeug muss gewaschen werden. Bevor der neue Reifen also montiert werden kann muss gewartet werden bis er verfügbar ist. Diese Zeit kann genutzt werden um einen Teil zu waschen.

Während die `cleanCar()`-Methode im Hauptthread arbeitet blockiert die andere Aufgabe so lange, bis die erste Teilaufgabe erfüllt ist. Danach kann der neue Reifen montiert werden und eine Erfolgsmeldung ausgegeben werden.

Auf diese Art und Weise ist es Möglich Ketten von *Futures* zu definieren die nacheinander abgearbeitet werden müssen.

# 3 Anwendungsfälle

In diesem Kapitel werden verschiedene Anwendungsfälle geschildert. Sie dienen der Illustration der Verwendung von *Futures* und *Promises*.

## 3.1 Urlaubsplanung

Dieser Anwendungsfall illustriert die Verwendung und Kombination von *Futures*.

**Szenario** Ein Urlaub wird geplant. Dabei gibt es drei Punkte, die zu erledigen sind.

1. Der Wechselkurs der Währung des Reiseziels wird ermittelt.
2. Ein Hotelzimmer wird gebucht.
3. Ein Flug wird gebucht.

Eine von den drei zuvor genannten Schritten unabhängige Tätigkeit ist das Packen der Koffer. Während zum Beispiel auf die Antwort des Servers der Bank gewartet wird, kann an dieser Stelle sinnvolle Arbeit geschehen (mit dem Kofferpacken beginnen).

Für jeden dieser Schritte gibt es eine gewisse Wahrscheinlichkeit, dass er nicht erfolgreich ist. Es kann zum Beispiel sein, dass der Server zum Abfragen der Wechselkurse nicht online ist.

Ein weiteres Risiko im Umgang mit Fremdwährungen ist, dass der Wechselkurs sehr ungünstig ist. Für diesen Fall ist es eine Option, sich ein zweites Reiseziel offen zu halten.

Es besteht zwar nach wie vor die Gefahr, dass beide Währungen zu nicht akzeptablen Kursen erhältlich sind, aber dieser Fall wird hier nicht betrachtet.

Ebenso gibt es bei der Buchung von Hotelzimmer und Flug eine gewisse Wahrscheinlichkeit dass ein technischer Fehler auftritt.

Sollte einer dieser Fehlerzustände eintreten, kann dies mit *Futures* modelliert werden, in dem diese eine Exception statt eines zu erwartenden Wertes enthalten.

**Pseudocode** für diesen Anwendungsfall:

```
val rateDollar: Future[Double] =
  getExchangeRateByFuture("US-Dollar")
val rateFranc: Future[Double] =
  getExchangeRateByFuture("Swiss franc")

val selectedRate = rateDollar.fallbackTo(rateFranc)

val bookAccommodation = selectedRate.map {
  rate => {
    println("Got exchange-rate. It is " + rate)
    if (isExchangeRateAcceptable(rate)) {
      println("Book accommodation")
```

```
      bookAccommodationOnline()
    } else {
      throw new Exception("Rate not acceptable")
    }
  }
}

val bookFlight = bookAccommodation.map {
  successful => {
    println("Book flight")
    bookFlightOnline()
  }
}

bookFlight.onComplete {
  case Success(_) => println("Flight booked")
  case Failure(ex) => println(ex.getMessage)
}

packSuitcase()
```

Listing 13: Code zum Anwendungsfall „Urlaubsplanung“

**Diskussion des Codes** In diesem Programm werden zwei *Futures* erzeugt, die jeweils den Wechselkurs für eine Währung enthalten. Mittels der `fallbackTo`-Methode wird eine Währung ausgewählt. Das *rateDollar*-Objekt genießt Priorität, es wird ausgewählt sofern es keine Exception enthält.

Das so erstellte Objekt *selectedRate* beinhaltet nun entweder den *Future* mit dem Wechselkurs des US-Dollar oder mit dem des Schweizer Franken.

Auf diesem Objekt wird die `map`-Methode aufgerufen. Diese wird, wenn der Wechselkurs akzeptabel ist, das Buchen des Hotelzimmers beauftragen, wenn nicht eine Exception werfen.

Anschließend wird ein *Future* erstellt um den Flug zu buchen. Dies geschieht in Abhänigkeit vom Erfolg des Buchens eines Hotelzimmers.

Wenn auch dieser *Future* erfolgreich abschließt, ist die Liste der drei Aufgaben abgearbeitet.

Zuletzt steht der Methodenaufruf `packSuitcase()`, der blockierend im Main-Thread läuft. Da die Methodenaufrufe der *Futures* alle nicht blockierend sind, laufen diese Aufgaben nebenläufig.

**Alternative: Callback-Verschachtelung** Diese Aufgabe kann komplett ohne die von *Scala* zur Verfügung gestellten Kombinatoren gelöst werden. Dafür ist es notwendig, die Callback-Methoden (`onComplete`, `onSuccess` und `onFailure`) zu verschachteln. Dies führt zu wesentlich unübersichtlicherem Code wie folgendes Beispiel verdeutlichen soll:

```
def doBookAccommodation(rate: Double): Future[Boolean] = {
  println("Got exchange-rate. It is " + rate)
  if (isExchangeRateAcceptable(rate)) {
    println("Book accommodation")
    bookAccommodationOnline()
  } else {
    throw new Exception("Rate not acceptable")
  }
}

def doBookFlight(bookAccommodation: Future[Boolean]) {
  bookAccommodation.onComplete {
    case Success(_) => {
      println("Accommodation booked")
      val bookFlight = future{
        println("Book flight")
        bookFlightOnline()
      }
      bookFlight.onComplete {
        case Success(_) => println("Flight booked")
        case Failure(ex) => println(ex.getMessage)
      }
    }
    case Failure(ex) => println(ex.getMessage)
  }
}

def main() {
  val rateDollar: Future[Double] =
    getExchangeRateByFuture("US-Dollar")

  val rateFranc: Future[Double] =
    getExchangeRateByFuture("Swiss franc")

  rateDollar.onComplete {
    // US-$
    case Success(rate) => {
      val bookAccommodation = doBookAccommodation(rate)
      doBookFlight(bookAccommodation)
    }

    // Swiss franc
    case Failure(ex) => {
      rateFranc.onComplete {
```

```
        case Success(rate) => {
          val bookAccommodation = doBookAccommodation(rate)
          doBookFlight(bookAccommodation)
        }
        case Failure(ex) => println(ex.getMessage)
      }
    }
  }

  // do some unrelated work
  packSuitcase()
}
```

Listing 14: Code zum Anwendungsfall „Urlaubsplanung“ mit verschachtelten Callbacks

## 3.2 Bauarbeiten

Dieser Anwendungsfall demonstriert die Verwendung von *Promises*.

**Szenario** Es finden Bauarbeiten statt. Beteiligt sind zwei handelnde Personen, ein Meister und sein Lehrling. Als sie mit der Arbeit beginnen möchten fällt ihnen auf, dass sie den Werkzeugkasten vergessen haben. Der Lehrling bricht auf, um diesen zu holen. Die Zwischenzeit kann der Meister nutzen, um Messungen vorzunehmen. Wenn der Lehrling mit dem Werkzeugkasten zurückkommt, kann er sich einer anderen Aufgabe (dem Reinigen der Baustelle) widmen, während der Meister das für die anstehende Arbeit passende Werkzeug auswählen kann.

Würden hier blockierende Methodenaufrufe eingesetzt, wäre es nicht möglich, die Zeit in der die Werkzeugkiste geholt wird, sinnvoll zu nutzen.

**Pseudocode** für diesen Anwendungsfall:

```
val myPromise = promise[Integer]
val myFuture = myPromise.future

val apprentice = future {
  val toolbox: Integer = getToolbox()
  // complete the promise
  myPromise.success(toolbox)
  cleanConstructionSite()
}

val craftsman = future {
  takeMeasurement()
  myFuture.onSuccess {
    case toolbox: Integer =>
```

```
      pickTheRightTool(toolbox)
  }
}
```

Listing 15: Code zum Anwendungsfall „Bauarbeiten“

**Diskussion des Codes** Es wird ein *Promise* erzeugt und der dazugehörige *Future* gespeichert. Der Lehrling und der Meister werden als *Future* modelliert. Der Future des Lehrlings wird mit dem Methodenaufruf `getToolbox()` blockiert, da er währenddessen keine weitere Tätigkeit ausführen kann. Der Future des Meisters ist davon nicht betroffen. Sobald der Lehrling das Versprechen einlöst wird die `onSuccess`-Callback-Methode des zugehörigen Futures ausgelöst. An dieser Stelle kann der Meister seine Vorarbeiten beenden und sich an die eigentliche Arbeit machen.

**Mehrmaliges Schreiben** ist bei Promises nicht erlaubt. Nehmen wir an, der Code wird folgendermaßen verändert:

```
val myPromise = promise[Integer]
val myFuture = myPromise.future

val apprentice = future {
  val toolbox: Integer = getToolbox()
  // complete the promise
  myPromise.success(toolbox)
  // this will cause an exception!
  myPromise.success(toolbox)
  cleanConstructionSite()
}

val craftsman = future {
  takeMeasurement()
  myFuture.onSuccess {
    case toolbox: Integer =>
      pickTheRightTool(toolbox)
  }
}
```

Listing 16: Modifizierter Code zum Anwendungsfall „Bauarbeiten“

Der zweite Aufruf der `success()`-Methode des *Promises* wird eine Exception auslösen, unabhängig vom Parameter.

Dieses Verhalten ist gewollt und macht *Promises* berechenbar. Es gibt zwei Möglichkeiten wie diese abgeschlossen werden können: Entweder durch einen Wert des Typs mit dem sie typisiert sind, oder durch eine Exception, falls bei der Berechnung ein Fehler aufgetreten ist. Ein Beispiel in unserem Anwendungsfall dafür könnte sein, dass der Lehrling den Werkzeugkasten im Auto nicht finden kann.

# 4 Futures in anderen Sprachen

Viele Programmiersprachen bieten mittlerweile Implementierungen von *Futures* und zum Teil von *Promises*.

**Java** bietet seit Version 1.5 der Sprache eine `java.util.concurrent.Future`-Klasse an (vgl.: [2]).

Der Zugriff auf das Ergebnis eines *Futures* funktioniert über eine `get()`-Methode, die den Aufrufer blockiert bis das Ergebnis vorliegt.

Ein Nachteil dieser Implementierung besteht in den fehlenden *Callback*-Methoden. So ist es nicht möglich, informiert zu werden wenn ein *Future* ein Ergebnis enthält, stattdessen ist es notwendig `get()` aufzurufen und im Fehlerfall eine Exception zu behandeln.

Es ist zwar möglich eine Callback-Klasse auf Basis des *Interface* `Runnable` zu erstellen, doch dies ist aus Entwicklersicht nicht mit dem Komfort vergleichbar, den *Scala Futures* bieten.

Ein weiterer Nachteil besteht darin dass die aus *Scala* bekannten Kombinatoren nicht verfügbar sind. Es ist zwar möglich deren Funktion nachzuprogrammieren, doch dies sorgt für erheblichen Mehraufwand auf Seiten des Anwendungsentwicklers und macht *Java Futures* damit nicht besonders attraktiv.

Drittanbieter wie zum Beispiel die *Guava Google Libraries for Java* bieten eigene Implementierungen, die einzelne Nachteile ausbessern. So gibt es dort zum Beispiel einen *ListenableFuture* der die bekannten `onSuccess()` und `onFaulure()`-Callback-Methoden kennt. Zusätzlich bietet diese Klasse Methoden die zum Teil den Kombinatoren aus *Scala* ähneln.

**C++** Seit *C++11* kennt der *C++*-Standard *std::future* und *std::promise*(vgl.: [6]).

*Futures* in *C++* funktionieren nicht gleich wie in *Scala*, so sind zum Beispiel die Kombinatoren wie in Java auch hier nicht Verfügbar.

Der Zugriff auf den Wert eines *Futures* erfolgt ebenfalls über die `get()`-Methode, die den Aufrufer so lange blockiert bis das Ergebnis des *Futures* vorliegt.

**Scala** Es gab bereits seit längerem verschiedene Implementierungen dieser Sprachmittel, unter anderem im Akka-Framework, in Skalaz, in den Twitter-Util-Klassen sowie in java.util.concurrent (vgl. [3]).

Um der Fragmentierung im Umfeld der Programmiersprache entgegenzuwirken und diese Techniken für alle Scala-Entwickler zugänglich zu machen, wurde auf Basis des SIP-Dokuments SIP-14 eine Implementierung im scala.concurrent-Package vorgenommen.

# 5 Zusammenfassung und Fazit

## 5.1 Zusammenfassung

Die vorliegende Seminararbeit führt in das Arbeiten mit *Futures* und *Promises* ein. Sie motiviert deren Verwendung anhand der Eigenschaft, nicht blockierend zu sein und arbeitet heraus wie sie zu verwenden sind. Dabei wird insbesondere auf die Möglichkeit zur Kombination mehrerer Objekte eingegangen.

Anhand von aus dem echten Leben gegriffenen Anwendungsbeispielen wird verdeutlicht, in welchen Situationen *Futures* und *Promises* hilfreich sein können.

## 5.2 Fazit

Ich persönlich habe mich durch diese Arbeit eingehend mit Abstraktionsschichten, die der nebenläufigen Programmierung dienen, beschäftigen können. Die Bedeutung der Nebenläufigkeit darf nicht unterschätzt werden. Besonders in der heutigen Welt, in der zunehmend auch kleine Computer mehr als einen Rechenkern zur Verfügung haben muss zur Kenntnis genommen werden, dass die Welt nicht sequenziell arbeitet.

## 5.3 Ausführen der Code-Beispiele

Alle Programme, die in dieser Arbeit diskutiert wurden sind vollständig in dem GitHub-Repository [7] veröffentlicht. Zur Ausführung ist eine Installation von *Scala* in der Version 2.10 oder höher notwendig.

Die Beispiele enthalten alle ein eigenes *Makefile*, welches mit dem Defaulttarget *build* die Quellen übersetzt und mit dem target *run* das Programm ausführt.

Alternativ kann auch die Kommandozeile

```
scalac *.scala && scala de.hska.wifl1011.seminararbeit.Main
```

von Hand ausgeführt werden.

# 6 Quellen- und Literaturverzeichnis

## Literatur


[1] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977.

[2] Oracle. Future (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`, 2014. [Online; zuletzt abgerufen am 11. Mai 2014].

[3] Heather Miller Philipp Haller. Futures and Promises in Scala 2.10. `http://lampwww.epfl.ch/~hmiller/files/Futures-Try-PhillyETE.pdf`, 2013. [Online; zuletzt abgerufen am 11. Mai 2014].

[4] Heather Miller Philipp Haller, Aleksandar Prokopec. Futures and Promises - Scala Documentation. `http://docs.scala-lang.org/overviews/core/futures.html#functional_composition_and_forcomprehensions`, 2013. [Online; zuletzt abgerufen am 11. Mai 2014].

[5] Heather Miller Philipp Haller, Aleksandar Prokopec. SIP-14 - Futures and Promises. `http://docs.scala-lang.org/sips/completed/futures-promises.html`, 2013. [Online; zuletzt abgerufen am 11. Mai 2014].

[6] Bjarne Stroustrup. C++11 FAQ - std::future and std::promise. `http://www.stroustrup.com/C++11FAQ.html#std-future`, 2014. [Online; zuletzt abgerufen am 11. Mai 2014].

[7] Florian Wilhelm. Code der Seminararbeit. `https://github.com/sputnik27/hska-seminararbeit`, 2014. [Online; zuletzt abgerufen am 11. Mai 2014].