β — Scala

Better Scala, constant work in progress

Markus Klink

29. Juli 2014

Inhaltsverzeichnis

1. REPL

> Ever tried. Ever failed. No matter. Try Again. Fail again. Fail better.
>
> *(Samuell Beckett)*

Nach der Installation von Scala können wir die `REPL` aufrufen:

```
~$ scala
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit
  Server VM, Java 1.7.0_15).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

`REPL` steht für „read-eval-print-loop“ uns ist für uns Programmierer ein nettes Sketchbook um Ideen auszuprobieren, ohne durch langsame Compilezeiten gehindert zu werden. In der `REPL` werden evaluierten Ausdrücken automatisch Variablen zugewiesen, auf die man in künftigen Ausdrücken wieder zurückgreifen kann. Dadurch ist es unnötig, stets eigene Variablen oder Values zu deklarieren.

```
scala> 5
res0: Int = 5

scala> 6
res1: Int = 6

scala> res0 + res1
res2: Int = 11
```

Wenn es nötig ist, Audrücke zu formulieren, die mehrere Zeilen umschliessen, antwortet die `REPL` stehst mit dem „|“-Symbol und man fährt einfach fort, die Anweisung einzugeben.

```
scala> """This is a String
     | spanning multiple lines."""
```

```
res3: String =
This is a String
spanning multiple lines.
```

Möchte man Code aus einen Programm kopieren, empfiehlt es sich den Code in die Zwischenablage zu kopieren, und in der `REPL :paste` einzugeben, den Text zu kopieren, und das Kopieren mit `Ctrl+D` abzuschliessen.

Der einzige Schönheitsfehler der `REPL` besteht darin, dass es innerhalb der `REPL` nicht möglich ist, Pakete zu definieren (allerdings können Pakete verwendet werden). Das tut dem Gedanken des Rapid Prototyping innerhalb der `REPL` allerdings keinen Abbruch.

Tabelle 1.1.: Übersicht der wichtigsten `REPL` Befehle

:help	Hilfe einblenden
:quit	`REPL` beenden
:load	scala Datei laden
:implicits	listet die implicit Deklarationen im Scope auf
:cp <path>	jar Datei laden
ctrl + a	An den Anfang der Zeile springen
ctrl + e	An das Ende der Zeile springen
:paste	Text aus der Zwischenablage einfügen
ctrl + d	Pastemodus beenden
↑	Vorherige Zeile wiederholen
↓	

Teil I.

Scala

2. Expressions

Expression Any bit of Scala code that yields a result. You can also say that an expression evaluates to a result or results in a value.

In Scala ist zunächst alles ein Ausdruck, also eine Expression, die in der Regel auch einen Rückgabewert liefert. Das ist nicht unähnlich zu anderen Programmiersprachen, wo man dieses Konzept auch kennt, geht in funktionalen Sprachen aber weiter. Zum Beispiel liefert die if/else Abfrage auch einen Wert. Im Beispiel unten wäre es die Rückgabe des Strings „SCALA_HOME is (not) set“. Weder muss hierfür explizit eine Funktion definiert werden, noch sind return statements erforderlich. In Scala sind diese sogar meist unerwünscht, und werden nur ausnahmsweise verwendet. In der Regel ist die Rückgabe eines Ausdrucks einfach die letzte Expression, die ausgewertet wurde, und es ist ein leichtes, den Kontrollfluss seiner eigenen Funktionen entsprechend zu gestalten.

```
scala> "hello"
res0: String = hello

scala> 3 + 5
res1: Int = 8

scala> if (System.getenv("SCALA_HOME") == null)
  |   "SCALA_HOME not set"
  | else
  |   "SCALA_HOME is set"
res5: String = SCALA_HOME is set

scala> Option(System.getenv("SCALA_HOME")) match {
     | case Some(home) => "SCALA_HOME is set"
     | case None => "SCALA_HOME is not set"
     | }
res6: String = SCALA_HOME is set

scala> res6 == "SCALA_HOME is set"
res7: Boolean = true
```

Im obigen Beispiel sieht man in der if/else Abfrage noch eine Abfrage nach „null“. Das sieht man in Scala Code sehr selten, und wird hier nur verwendet, weil eine Methode (getenv) aus der java-Welt aufgerufen wurde. Scala kennt ein Konstrukt aus der Standardbibliothek namens scaladoc: scala.Option[1], welches im nächsten Beispiel herangezogen worden ist. Das benutzte Sprachkonstrukt des Pattern Matching mit „match/case“ werden wir noch behandeln.

Und zuguterletzt sehen wir in der letzen Zeile noch den Vergleich mit Hilfe von „==“. Java würde hier ein „false“ liefern, da die beiden Strings eine unterschiedliche Objektidentität liefern. Scala vergleicht standardmässig über die Gleichheit mit equals, da Daten dort nicht veränderlich sind, und ihnen somit in der Regel auch keine eigene Objektidentität gewährt wird.

[1] http://www.scala-lang.org/api/current/index.html#scala.Option

3. Immutability

In der funktionalen Programmierung sind Objekte und "Variablenïn der Regel immutable, dass heist, wenn der Wert einmal zugewiesen worden ist, können diese Objekte nicht mehr verändert werden. In der objektorientierten Programmierung wird oftmals aber auch gefordert, dass Objekte auch einen Zustand halten, das heisst, sie sind über die Zeit veränderbar. In Scala ist dies (mit Einschränkungen) nicht so. Ein Objekt (bzw. der Bauplan Klasse) ist eher wie ein Namensraum zu betrachten, der Werte und Funktionen mit gemeinsamer Verantwortung kapselt.

Es gibt in Scala (leider?) aber auch die Möglichkeit Variable zu definieren. Der Unterschied ist im REPL Ausschnitt ersichtlich:

```
scala> val x = 5
x: Int = 5

scala> x = 6
<console>:8: error: reassignment to val
       x = 6
         ^

scala> var y = 6
y: Int = 6

scala> y = 5
y: Int = 5
```

Unveränderliche Objekte haben in verteilten, nebenläufigen Systemen den sehr großen Vorteil, dass der Zustand auch durch nebenläufigen Zugriff nicht zerstört werden kann. Des weiteren ist es häufig einfacher über Programme nachzudenken, die diese Eigenschaft besitzen.

3.1. Referential transparency

Referential transparency A property of functions that are independent of temporal context and have no side effects. For a particular input, an invoca-

> tion of a referentially transparent function can be replaced by its result without changing the program semantics.

Verkürzt besagt die referentielle Transparenz, dass der Platzhalter eines Wertes in jeden Ausdruck auch durch die Berechnung des Wertes ausgetauscht werden kann.

```
object Main extends App {

  import java.util.Random

  val x = new Random().nextInt(10)
  println(s"x = $x")
  println(s"x + x + x = ${x + x + x}")

  print("Aber mit new Random().nextInt(10) + ... = ")
  println(new Random().nextInt(10) +
    new Random().nextInt(10) +
    new Random().nextInt(10))
}
```

4. Functions

> Haskell: If I want a gang of elderly math professors to bind me up and yell at me, they better do it in person, not with a compiler.
>
> *(@zedshaw)*

Method A method is a function that is a member of some class, trait, or singleton object.

Function A function can be invoked with a list of arguments to produce a result. A function has a parameter list, a body, and a result type. Functions that are members of a class, trait, or singleton object are called methods. Functions defined inside other functions are called local functions. Functions with the result type of Unit are called procedures. Anonymous functions in source code are called function literals. At run time, function literals are instantiated into objects called function values.

4.1. Pure Functions

Sprechen wir als erstes über Funktionen, schliesslich ist Scala eine funktionale Programmiersprache, erwartungsgemäß nehmen Funktionen dort also einen außerordentlichen Rang ein. Die Vorstellung über Funktionen können dabei sehr unterschiedlich sein, wie der obige Tweet belegt. In der Tat sind funktionale Programmiersprachen zum Teil relativ mathematisch veranlagt. Ein Umstand der ihnen sehr zum Vorteil gereicht, denn einige dieser Eigenschaften ermöglichen es Programmierern hervorragend Schlüsse über den vorliegenden Code zu ziehen. Pure Funktionen sind das Nesthäkchen der funktionalen Programmierer, Funktionen mit Seiteneffekten die Nestbeschmutzer, und Funktionen, die gar nichts zurückgeben, ebenso.

Definieren wir zunächst eine uns bekannte Funktion, die der Geradengleichung entspricht. Eine geometrische Gerade kann über ihre Steigung m und einen Schnittpunkt mit der y-Achse y_0 definiert werden:

$$f(x) = mx + y_0 \tag{4.1}$$

beziehungsweise in Scala:

```
def gerade( m: Double, y0: Double, x: Double) : Double = {
  m*x + y0
}
```

gerade ist eine Funktion, hat 3 Parameter vom Typ Double, liefert ein Double zurück und ist eine pure Funktion . Das heisst, wir können sie immer und jeder Zeit aufrufen, sie wird für die selben Eingabeparameter auch immer den selben Ausgabewert liefern. *gerade* ist eine Funktion, die keinerlei Seiteneffekte ausübt.

TODO: Exkurs: Seiteneffekte

Der Aufruf der Funktion ist selbstverständlich recht mühselig, bei jedem Aufruf ist es notwendig wieder und wieder die Steigung m und den Schnittpunkt mit der y-Achse zu übergeben. Viel besser wäre es, wenn wir eine Funktion bauen könnten, die uns einfach wiederum eine Funktion zurückliefert.

4.2. Higher order functions

Funktionen, die als Parameter wiederum Funktionen erhalten, werden Funktionen höherer Ordnung genannt, und werden sehr oft angewandt. Dabei ist es gleichgültig, ob sie Funktionen zurückgeben, oder empfangen. In Scala werden sie direkt von der Sprachsyntax unterstützt.

```
def gerade(m: Double, y0:Double) : Double => Double = {
  x => m*x + y0
}
```

Ein kurzer Test in der `REPL`:

```
scala> def gerade(m: Double, y0: Double)
     | : Double => Double  = {
     | x => m * x + y0
     | }
gerade: (m: Double, y0: Double)Double => Double

scala> val f = gerade(1,0)
f: Double => Double = <function1>

scala> f(1)
```

```
res0: Double = 1.0

scala> f(2)
res1: Double = 2.0

scala> f(3)
res2: Double = 3.0
```

Eine syntaktische Variante zur Definition der Funktion gerade sieht so aus:

```
def gerade(m: Double, y0 : Double) = {
     x: Double => m * x + y0
 }
```

Hier wurde auf die explizite Angabe des Rückgabewertes verzichtet, was in Scala mit Ausnahme von rekursiven Funktionen, und wenn der Compiler mal nicht weiter weiss, immer möglich ist. Dafür kann nun der Typ des Parameters x nicht ermittelt werden, so dass er explizit angegeben werden muss. Beide Varianten sieht man in Code sehr häufig, und es ist empfehlenswert sich damit vertraut zu machen.

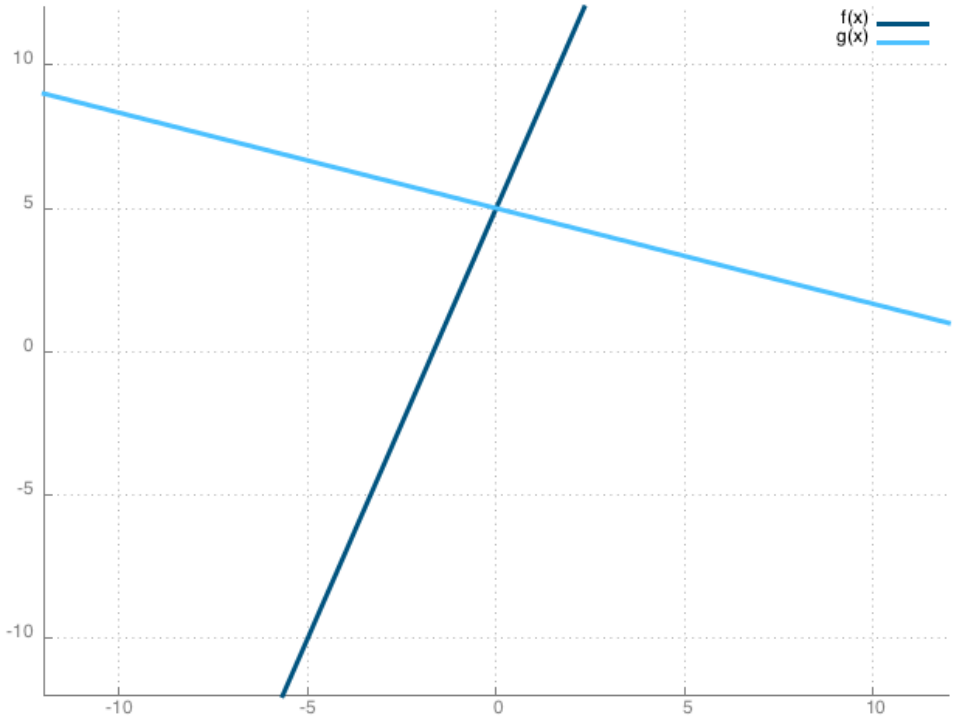
Wie im Listing ersichtlich, ist es möglich die Funktion, die zurückgegeben wurde, wiederum zu speichern, und sie über den Namen f wiederum zu benutzen. Damit ist es ein Leichtes, eine neue Funktion zu bauen, die uns eine orthogonale Gerade zu einer bereits bestehenden Geraden zurückgibt. Die Signatur entspricht also einer Funktion, die eine Funktion entgegennimmt, und eine Funktion zurückgibt. Weiterhin benötigen wir zur Berechnung der Orthogonalen die Steigung der Eingangsgeraden:

$$m_f = \frac{\Delta y}{\Delta x} = \frac{f(1) - f(0)}{1 - 0} = f(1) - f(0) \tag{4.2}$$

Die Steigung der Orthogonalen m_0 ist dann der negative Kehrwert der Steigung m_f der Ursprungsgeraden:

$$m_o = -\frac{1}{m_f} \tag{4.3}$$

Ausserdem vereinfachen wir das Problem und bestehen darauf, dass wir stets die Orthogonale haben möchten, die durch den Punkt $x = 0$ geht.



In der REPL:

```
scala> def gerade(m: Double, y0: Double) : Double => Double =
{
     |   x => m * x + y0
     | }
gerade: (m: Double, y0: Double)Double => Double

scala> def orthogonal(f: Double => Double) : Double => Double = {
     |   val mf = f(1) - f(0)
     |   x => - (1/mf) * x + f(0)
     | }
orthogonal: (f: Double => Double)Double => Double

scala> def orthogonal2(f: Double => Double) : Double => Double = {
     |   val mf = f(1) - f(0)
     |   gerade(-1/mf, f(0))
     | }
```

In diesem Beispiel wurden 2 Alternativen implementiert. in der ersten Varianten von *orthogonal* wird die Geradengleichung verwendet, um den notwendigen

Rückgabewert zu liefern, in der Variante *orthogonal2* benutzen wir einfach unsere bereits vorhandene Funktion gerade.

Eine weitere Variante ist denkbar, und wird extrem häufig eingesetzt, wenn die Funktion, die *orthogonal* übergeben werden soll, erst ad hoc definiert wird:

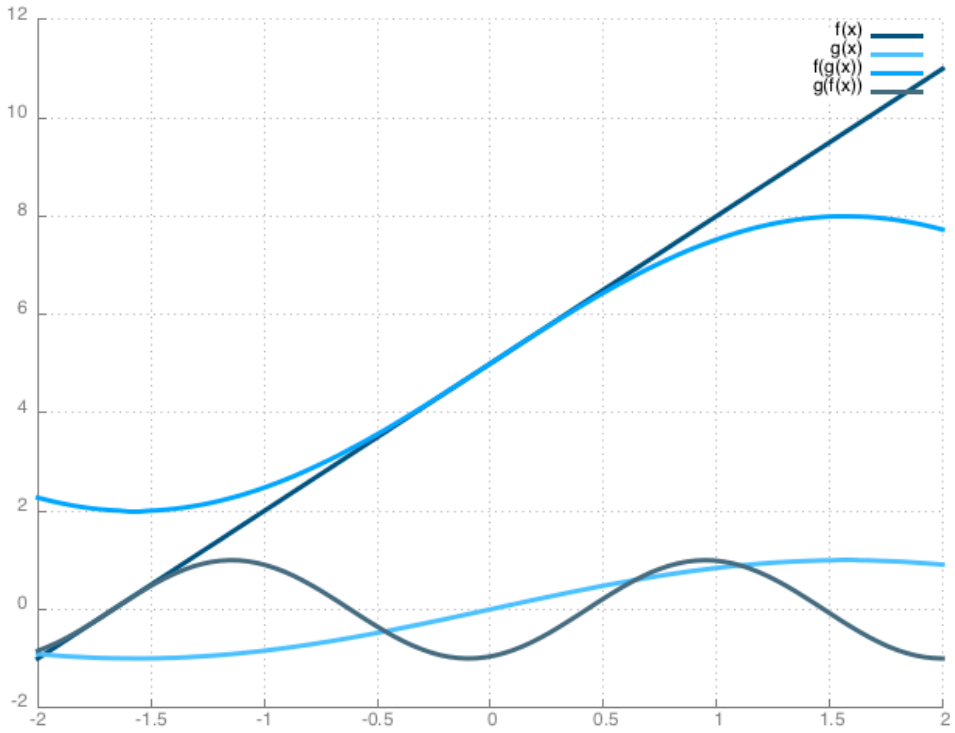
```
scala> def gerade(m : Double, y0 : Double) = {
     | x:Double => m * x + y0
     | }
gerade: (m: Double, y0: Double)Double => Double

scala> def orthogonal(f: Double => Double) = {
     | val m = f(1) - f(0)
     | gerade(-1/m, f(0))
     | }
orthogonal: (f: Double => Double)Double => Double

scala> orthogonal { x: Double => 3*x + 5}
res0: Double => Double = <function1>
```

4.3. Function composition

Betrachtet man kurz scaladoc: scala.Function1[1], sieht dort einige vordefinierte Funktionen, wie *compose* oder *andThen*, die man auf Funktionen anwenden kann.

$$(f \text{ compose } g)(x) = f(g(x)) = (f \circ g)(x) \tag{4.4}$$

[1] http://www.scala-lang.org/api/current/index.html#scala.Function1



Die letztere Schreibweise $(f \circ g)(x)$ begegnet einem öfters, sie liest sich „f folgt auf g“.

4.4. Currying

In Scala können wir ohne weiteres Funktionen mit mehreren Parametern schreiben:

```
scala> def add(x: Int, y:Int) = x + y
add: (x: Int, y: Int)Int

scala> add _
res3: (Int, Int) => Int = <function2>
```

Es gibt einige funktionale Programmiersprachen, die diese Fähigkeit nicht ohne weiteres haben. Das liegt letztendlich an der strikten Umsetzung des sogenannten λ - Kalkül (wikipedia: Simply_typed_lambda_calculus[2]). Funktionen mit mehreren Parametern kann man allerdings sehr einfach umschreiben,

[2] http://en.wikipedia.org/wiki/Simply_typed_lambda_calculus

indem man sie durch Funktionen ersetzt, die eine Funktion zurückgeben:

```
scala> def add(x: Int) = (y: Int) => x + y
add: (x: Int)Int => Int

scala> add _
res0: Int => (Int => Int) = <function1>
```

Der Aufruf erfolgt dann wie in diesen Beispielen:

```
scala> add(3)
res1: Int => Int = <function1>

scala> res1(4)
res2: Int = 7

scala> add(3)(7)
res3: Int = 10
```

Für das Durchführen des Currying gibt es sogar Hilfsfunktion in der Scala Standard Bibliothek:

```
scala> def add(x: Int, y: Int) = x + y
add: (x: Int, y: Int)Int

scala> add _
res0: (Int, Int) => Int = <function2>

scala> val curried = (add _).curried
curried: Int => (Int => Int) = <function1>

scala> curried(3)(4)
res6: Int = 7
```

4.5. Partial functions

Liest man sich scaladoc: scala.Function1[3] nochmals durch, so stösst man dort auf folgenden Satz:

> Note that Function1 does not define a total function, as might be suggested by the existence of scaladoc: scala.PartialFunction[4]. The

[3] http://www.scala-lang.org/api/current/index.html#scala.Function1
[4] http://www.scala-lang.org/api/current/index.html#scala.PartialFunction

> only distinction between Function1 and PartialFunction is that the latter can specify inputs which it will not handle.

Huh? Ok - eine totale Funktion, ist so etwas wie unsere Funktion *gerade*. Für jeden beliebigen ihrer Eingabewerte erzeugt sie eine entsprechende Ausgabe. Das muss nicht immer so sein. Zum Beispiel ist es nicht erlaubt, aus einer negativen, reellen Zahl die Wurzel zu ziehen. In Scala könnten wir das Problem wie folgt lösen:

```
scala> def wurzel : PartialFunction[Double, Double] = {
 | case x: Double if (x>=0) => math.sqrt(x)
 | }
wurzel: PartialFunction[Double,Double]

scala> wurzel(4)
res1: Double = 2.0

scala> wurzel.isDefinedAt(-4)
res2: Boolean = false

scala> wurzel.isDefinedAt(4)
res3: Boolean = true

scala> math.sqrt(-4)
res4: Double = NaN
```

Partielle Funktionen spielen eine größere Rolle bei der Verwendung des Collection Frameworks und im Pattern Matching.

4.6. Parameter

Parameter können normal als Werte übergeben werden, oder aber wie im zweiten Fall auch als Expression:

```
scala> def f(i : Int) = {
     | println("in function f")
     | println(i)
     | }
f: (i: Int)Unit

scala> f(3)
in function f
```

```
3

scala> f { println("In Parameter")
     | 3
     | }
In Parameter
in function f
3
```

Wie man im zweiten Beispiel sieht, wird der Ausdruck bereits vorm Aufruf der Funktion ausgewertet.

Im Unterschied dazu, kann man auch Parameter definierbaren, die erst ausgewertet werden, wenn innerhalb der Operation verwendet werden. Das ist insbesondere dann nützlich, wenn Parameter zum Beispiel nur für den seltenen Fall einer Fehlerbehandlung Verwendung finden.

```
scala> def g(i : => Int) = {
     | println("in function g")
     | println(i)
     | }
g: (i: => Int)Unit

scala> g { println("in parameter")
     | 3
     | }
in function g
in parameter
3
```

Mit Hilfe dieses Sprachkonstrukts lassen sich bereits sehr elegante Lösungen erzielen, zum Beispiel kann man eine Funktion definieren, die als Wrapper um irgendetwas dient, dessen Zeit man messen möchte [5]:

TODO: Verweis auf Generics

```
scala> def timer[A](x: => A) = {
     | val start = compat.Platform.currentTime
     | val result = x
     | val end = compat.Platform.currentTime
     | println(end - start)
     | result
     | }
```

[5] Diese Implementierung beinhaltet einen Haken. Durch Aufruf von println wird ein Seiteneffekt erzielt, außerdem ist die Auflösung von currentTime die Einheit milliSekunde, aber als Demonstration reicht es.

```
timer: [A](x: => A)A

scala> timer {
     | orthogonal{ x => 3*x + 5 }
     | }
1
res1: Double => Double = <function1>
```

4.7. Uniform Access Principle

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class Person(name: String) {
 def age = 32
}

// Exiting paste mode, now interpreting.

defined class Person

scala> Person("Klink")
res0: Person = Person(Klink)

scala> res0.age()
<console>:11: error: Int does not take parameters
              res0.age()
                     ^

scala> res0.age
res2: Int = 32

scala> Array[String]("foo","bar").length
res3: Int = 2
```

Das Uniform Access Principle besagt, dass Funktionen durch Val-Ausdrücke ersetzbar sein können. Werden parameterlose Funktionen, wie age oder length ohne Klammern definiert, so ist ihr Aufruf auch nur ohne Klammern möglich. Dadurch sind die Scala Programme syntaktisch äquivalent zu Programmen, die

val's anstelle von Funktionen verwenden.

4.8. Lifting

In Diskussionen um funktionale Programmierung begegnet einen immer wieder der Ausdruck lifting einer Funktion, lift a function, lifted function etc.

Der Sachverhalt ist einfach erklärt. Hat man zum Beispiel eine Klasse

```
case class Pair[A](a: A, b: A)
```

und man hat immer wieder die Aufgabe, dass man Pair-Instanzen hat, mit deren Wert man arbeitet, um das Ergebnis wiederum in einer neuen PAir-Instanz abzulegen, so kann man die Funktionen in den Kontext von Pair "heben", also liften.

```
def lift1[A, B](f: A => B): Pair[A] => Pair[B] = {
      p: Pair[A] =>
        Pair(f(p.a), f(p.a))
    }
```

4.9. Closures

Bisher hatten wir nur Funktionen, die ihre Rückgabewerte aus den Eingangsparametern berechnen. Im Idealfall sind diese Funktionen ohne Seiteneffekte. Allerdings ist auch auch möglich Funktionen zu definieren, die mit Werten arbeiten, die nicht aus den Eingangsparametern bezogen werden, sondern sich lediglich im Scope befinden. Diese Werte sind dann nicht an Eingangsparameter gebunden, sondern "frei". Eine Closure, die freie Werte verwendet umschliesst diese.

Einerseits ist dieses Konzept sehr mächtig, da es dabei helfen kann unnötige Parameter zu umgehen, weil die Werte aus dem Kontext bezogen werden können, andererseits stellen sich oftmals ungewollte Effekte ein, die nicht so leicht zu debuggen sind (Akka!)

```
class Demo(var x: Int) {
    val add: Int => Int = { y: Int => x + y }
    def getX() = x
  }
```

() => Notizen

5. Classes & Objects

5.1. Classes

Klassen sind, wie in anderen objektorientierten Programmiersprachen auch, zumächst einmal einfach Baupläne für Objekte. Klassen kapseln Informationen und bieten über Methoden einen definierten Zugang zu diesen Informationen an. Somit verwaltet ein Objekt auch einen veränderlichen Zustand.

Genau diese Denkweise kann man auch auf Klassen in Scala übertragen. Die Sprachfeatures stehen diesen Vorgehen nicht im Weg. Andererseits sollten man als Scala Programmierer etwas anders denken, und sich bewusst einschränken. Wenn es nicht gewichtige andere Gründe gibt, stellt für uns eine Klasse einfach nur einen Ort der Verantwortung dar (quasi ein Namensraum für Methoden, die mit Informationen arbeiten), aber kein Ort um den Zustand eines Objektes zu verwalten. Objekte von Klassen sollten in aller Regel also auch unveränderlich sein.

Die Definition einer Klasse in Scala ist wesentlich kompakter als in Java und mit Hilfe von „Case Classes“ ist es nochmals möglich sehr viel Boilerplatecode einzusparen.

```
scala> class Line(m: Double, y0: Double) {
     |   def fn(x: Double) = { m * x + y0 }
     |   def apply(x: Double) = fn(x)
     | }
defined class Line

scala> val f = new Line(3,0)
f: Line = Line@5c22251d

scala> f(0) // f.apply(0)
res0: Double = 0.0

scala> f(1)
res1: Double = 3.0

scala> f.fn(2)
```

```
res3: Double = 6.0
```

TODO: Konstruktoren und KonstruktorParameter erklären

In der obigen Definition der Klasse Line ist die apply Funktion besonders interessant. Sie ermöglicht anhand der Scala Language Specification eine Abkürzung, denn es gilt, dass f.apply(2) identisch ist mit f(2). Weiterhin wird ein Objekt einer Klasse einfach mit Hilfe von „new“ erzeugt, wie wir es von java bereits kennen.

Übung: Die obige Funktion soll um eine Funktion def add(l: Line): Line ergänzt werden. Das funktioniert nicht wie erwartet.

5.2. Case Classes

Case Classes sind Scala Klassen, die automatisch vom Compiler um bestimmte Methoden und Eigenschaften ergänzt werden.

So bekommt jede Klasse automatisch:

- öffentliche Felder für jeden Konstruktorparameter
- eine equals Methode, die die Klassenattribute vergleicht
- eine copy Methode
- eine hashCode Methode
- eine sinnvolle toString Methode

und desweiteren ein Companion Object mit:

- apply und
- unapply Methode

```
scala> case class Line(m: Double, y0: Double) {
     | def apply(x: Double) = m * x + y0
     | def unapply() : Option[(Double, Double)] = Some((m,y0))
     | }
defined class Line

scala> val f = Line(6,0)
f: Line = Line(6.0,0.0)

scala> val g = Line(6,0)
g: Line = Line(6.0,0.0)
```

```
scala> f == g
res0: Boolean = true

scala> val h = f.copy(y0 = 1)
h: Line = Line(6.0,1.0)

scala> f == h
res1: Boolean = false

scala> val Line(m,x) = h
m: Double = 6.0
x: Double = 1.0

scala> h(1)
res2: Double = 7.0
```

5.3. Objects

Neben Klassen können in Scala auch direkt Objekte beschrieben werden, die dann nicht mehr instanziiert werden müssen, sondern direkt zur Verfügung stehen. Sie ähneln in gewisser Weise Singletonklassen in Java [1]

```
scala> case class Line(m: Double, y0: Double) {
     | def apply(x: Double) = m * x + y0
     | }
defined class Line

scala> object LineUtil {
     |   def plot(f: Line) = {
     |     (-10 to 10).map(x => (x,f(x)))
     |   }
     | }
defined module LineUtil

scala> val f = Line(3,0)
f: Line = Line(3.0,0.0)
```

[1] Der Vergleich zu Singletons hinkt aber. Ein Singleton ist eine künstliche singuläre Instanz einer Klasse, ein Object hingegen einfach nur singuläre Instanz im Scope. Ist ein Object innerhalb einer Klasse definiert, hat jedes Klassenobjekt wiederum eine eingebettet Objektinstanz - welche Mitglieder der Klasse zugreifen kann.

```
scala> LineUtil.plot(f)
res0: scala.collection.immutable.IndexedSeq[(Int, Double)] =
  Vector((-10,-30.0), (-9,-27.0), (-8,-24.0), (-7,-21.0),
  (-6,-18.0), (-5,-15.0), (-4,-12.0), (-3,-9.0), (-2,-6.0),
  (-1,-3.0), (0,0.0), (1,3.0), (2,6.0), (3,9.0), (4,12.0),
  (5,15.0), (6,18.0), (7,21.0), (8,24.0), (9,27.0), (10,30.0))
```

5.4. Companion Objects

Compagnion Objekts sind Objekte, die eine Klasse begleiten. Das heisst, sie müssen in der selben Quelldatei angelegt werden und den gleichen Namen haben, wie die zugehörige Klasse. Compagnion Objekte dürfen dann auch auf private Eigenschaften der Klasse zugreifen.

Compagnion Objekte eignen sich oftmals dafür notwendige Konstanten zu definieren, Extraktoren zu schreiben, oder implizite Umwandlungen in andere Datentypen zu schreiben.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class Line(m : Double, y0 : Double) {
def apply(x: Double) = { m * x + y0 }
}

object Line {
def apply(x1: Double, y1: Double,
                x2: Double, y2: Double) : Line = {
  val m = (y2-y1)/(x2-x1)
  val y0 = y1 - m * x1
  new Line(m, y0)
}
}

// Exiting paste mode, now interpreting.

defined class Line
defined module Line

scala> val f = Line(2,0,4,0)
f: Line = Line(0.0,0.0)

scala> f(3)
```

```
res0: Double = 0.0
```

Die Case Class Line hat zwei Konstruktorparameter, und wir könnten Sie wie gewöhnlich mit Hilfe von new erzeugen:

```
scala> val f = new Line(3,1)
f: Line = Line(3.0,1.0)
```

5.5. apply und unapply

Das bei Case Classes das new weggelassen werden kann, liegt nicht an irgendeiner syntaktischen Ausnahme von Scala, sondern daran, dass für die Case Class Line automatisch eine apply Methode im Kompanionobjekt angelegt wird.

Diese Definition sieht in etwa wie folgt aus:

```
def apply(m: Double, y0: Double) = new Line(m,y0)
```

Der Aufruf von **val** f = Line(3,1) ist also eigentlich ein Aufruf von **val** f = Line.apply(3,1), nur dass der Name der apply Methode unterschlagen werden darf. Dies ist eine Konvention für die Funktion apply.

Die Definition einer Methode apply ist nicht auf das Companion Object beschränkt, sondern kann überall Verwendung finden. Häufig wird so ein schneller und eingängiger Accessor definiert (apply(index : Int) statt get(index : Int)):

```
scala> val l = List(1,2,3,4,5)
l: List[Int] = List(1, 2, 3, 4, 5)

scala> l(0)
res0: Int = 1

scala> l(3)
res1: Int = 4

scala> l(10)
java.lang.IndexOutOfBoundsException: 10
```

Es gibt auch eine unapply Methode, die Elemente aus einer Klasse herausextrahiert, und deswegen auch ein Extraktor genannt wird:

```
object Main extends App {
```

```
  val pers = Person("Klink", "Markus")
  val Person(n, vn) = pers

  println(s"Name␣$n")
  println(s"Vorname␣$vn")
  val pers2: Person = null
  pers2 match {
    case Person(xn, xvn) ⇒ println(s"$xn␣,␣$xvn")
    case _ ⇒ println("null")
  }

  val x = Person.unapply(pers2)
  println(x)
}

class Person(val name: String, val vorname: String)
object Person {
  def apply(name: String, vorname: String) = new Person(name, vorname)
  def unapply(p: Person) = if (p eq null) None else Some((p.name, p.vorname
}
```

5.6. Value classes

Wert Klassen sind einfache Wrapper um einen Typ, die von AnyVal abgeleitet sind. Ihr Vorteil besteht darin, dass man Typsicherheit erlangt, ohne dass explizit ein Objekt vom Typ Meter alloziert werden muss, welche Laufzeitvorteile bringt.

```
class Meter(val value: Double) extends AnyVal {
  def +(m: Meter): Meter = new Meter(value + m.value)
}

case class Schiff(laenge: Meter, breite: Meter)
```

Wertklassen werden häufig zusammen mit impliziten Klassen (siehe Abschnitt **??**, Seite **??**) verwendet. Wichtige Eigenschaften der Wertklassen:

- leiten sich von AnyVal ab
- ein public val Konstruktor Argument
- beinhaltet nur def's, und keine weiteren var's, val's , Klassen, Objekte etc.

6. Packages & Scope

6.1. Packages

```
package demo.pack

class ScopeDemo {}

package demo {
  package pack {

    class ScopeDemo2 {
      val x = new ScopeDemo()
      val io = new java.io.FileOutputStream("test.dat")
      val y = new sub.ScopeDemo4()
    }

  }
}

package demo.pack.sub {
  class ScopeDemo3 {
   val x = new ScopeDemo
  }
  class ScopeDemo4
}

//package demo.pack.java {
// the definition of the package would cause a syntax error
//}
```

In Scala ist die Paket und Dateistruktur im Gegensatz zu java vollkommen frei. Sie korrelieren nicht. Aus Gründen der Übersicht, ist es aber empfehlenswert sich an einer ähnlichen Struktur zu orientieren.

Pakete können mit geringfügig unterschiedlicher Semantik wie folgt definiert werden:

```
package org.oose.stuff
definitions

package org.oose
package stuff
definitions

package org {
  package oose {
    package stuff {
    definitions
    }
  }
}
```

Eine Paketdeklararion *package p* $\{definitions\}$ weist alle Typedefinitionen dem Bereich des Pakets p zu. Hat p die Form eines Pfades $x.y.z$, so sind die Mitglieder von x und $x.y$ im Pakete z nicht sichtbar. Wählt man die 3. Form, so sind die Mitglieder von x und $x.y$ im Paket z ohne weitere Imports sichtbar.

6.2. Imports

```
class ScopeDemo2 {
      import java.io._
      val x = new ScopeDemo()
      val io = new FileOutputStream("test.dat")
      val y = {
        import sub._
        import ScopeDemo2.foo
        new ScopeDemo4()
        foo()
      }
    }

    object ScopeDemo2 {
      def foo() = "foo"
    }
```

Imports können wie in Java nach den Paketdeklarationen erfolgen, oder aber auch in jeder Form von Expression verwendet werden, um den Scope des Imports einzuschränken.

Weitere Importausdrücke:

```
// wildcard
import scala.collection._
// scala.collection is already in scope
import mutable._
// alias
import java.util.{Random => RandomGenerator}
// multiple single imports
import java.util.{List, Vector, Set}
```

() => Notizen

7. Inheritance, Traits & Generics

7.1. Vererbung

```
scala> class Person(name: String)
defined class Person

scala> class Manager(name:String) extends Person(name)
defined class Manager

scala> def foo(l : Array[Person]) = l.foreach(println _)
foo: (l: Array[Person])Unit

scala> foo(Array[Person](new Person("p1"))
     | )
$line3.$read$$iw$$iw$Person@543b0d43

scala> foo(Array[Person](new Person("p1")))
$line3.$read$$iw$$iw$Person@795a2ae9

scala> foo(Array[Manager](new Manager("p1")))
<console>:11: error: type mismatch;
 found   : Array[Manager]
 required: Array[Person]
Note: Manager <: Person, but class Array is invariant in type T.
You may wish to investigate a wildcard type such as `_ <: Person`. (SLS 3.2.10)
              foo(Array[Manager](new Manager("p1")))
```

Array ist invariant (oder auch nonvariant).

```
class Array[T]
```

nonvariant A type parameter of a class or trait is by default nonvariant. The class or trait then does not subtype when that parameter changes. For example, because class Array is nonvariant in its type parameter, Array[String] is neither a subtype nor a supertype of Array[Any].

Dasselbe nochmal mit List:

```
scala> def bar(l: List[Person]) = l.foreach(println _)
bar: (l: List[Person])Unit

scala> bar(List[Person](new Person("p1")))
$line3.$read$$iw$$iw$Person@66e35ba7

scala> bar(List[Manager](new Manager("p1")))
$line5.$read$$iw$$iw$Manager@7b8538d7
```

```
class List[+A]
```

covariant A covariant annotation can be applied to a type parameter of a class or trait by putting a plus sign (+) before the type parameter. The class or trait then subtypes covariantly with—in the same direction as—the type annotated parameter. For example, List is covariant in its type parameter, so List[String] is a subtype of List[Any].

7.2. Traits

7.2.1. ...als reine Interfaces

```
package demotraits

object SimpleTraits extends App{

  val dobermann = new Dog("Hasso")
  dobermann.move()
  dobermann.eat()
  println(dobermann.name)

}

trait Animal {
  def classificationName() : String
}

trait Mammal {
  def sleep()
}

trait Legs {
```

```
  self : Mammal =>
  def move() : Unit
}

trait Mouth {
  def eat() : Unit
}

class Dog(val name: String) extends Mammal with Legs with Mouth {
  val classificationName = "Dog"
  def move() = println("running")
  def eat() = println("eating")
  def sleep() = println("sleeping")
}
```

Klassen können durch beliebig viele Traits erweitert werden. Haben diese Traits keinerlei Implementierung, so funktioniert die Vererbung ähnlich wie bei Java, und die Klasse hat die Aufgabe den Vertrag der Traits zu realisieren. Traits können mit *extends* $TraitName1$ *with* $TraitName2$*...with* $TraitNameX$ einer Klasse zugewiesen werden.

7.3. Mixin

Traits können allerdings auch eine Implementierung beinhalten (jedoch keinen Konstruktor). Werden mehrere Traits einer Klasse zugewiesen, so kann es vorkommen, dass es Konflikte in der Namensgebung gibt. Diese werden durch das Verfahren der Class Linearization aufgelöst.

7.4. Type Bounds

ACHTUNG: View Bounds <% sind deprecated.

Tabelle 7.1.: Type Bounds

$[A]$	generischer Typ , invariant
$[+A]$	covarianter Typ
$[-A]$	contravarianter Typ
$[A <: B]$	A muss eine Unterklasse von B sein
$[A >: B]$	A muss ein Supertyp von B sein
$[A <\% I]$	View Bound: für A gibt es eine implizite Konvertierung nach I.
$[A : M]$	Context Bound: für a gibt es eine implizite Konvertierung nach M[A]

TODO: Erläuterung erstellen

8. Pattern Matching

Mit Hilfe von Pattern Matching wird bestimmt, ob ein bestimmter Wert (oder eine Sequenz von Werten) erfolgreich gegen ein Muster testet. Muster können vielfältige Formen annehmen. Einige Beispiele:

ex: IOException passt auf alle Instanzen von IOException

Some(x) passt auf Werte der Form Some(v), und bindet x auf den Wert von v

(x, _) passt auf Paare von Werten und bindet x auf den ersten Wert des Paares

head :: tail matched gegen head und tail einer Liste

x :: y :: xs passt auf Listen mit einer Länge größer 2

addr @ Address("Grundstr", _ , _) passt auf alle Adressen in der Grundstrasse, aber die Inhalte sind danach belanglos, und der Alias addr für das gesamte Addressobjekt wird verwendet.

Simples Beispiel für den Einsatz in match / case Strukturen:

```
object Main extends App {

  val x : Option[Int] = Some(1)

  x match {
    case _ =>
      println("Matched immer / wildcard")
  }

  x match {
    case identifier =>
      println(s"Matched immer $identifier")
  }

  x match {
    case typ : Option[Int] =>
      println(s"Matched immer gegen den Typ Option[Int] $typ")
```

```
  }

  x match {
    case Some(identifier) =>
      println(s"extrahiert: $identifier")
    case _ => println("matcht nicht")
  }

  x match {
    case None =>
      println(s"x ist nicht None")
    case _ =>
      println("match weil nicht none")
  }

}
```

Übung Funktioniert das?

```
object PatternPartial extends App {

  case class Address(street: String, nr: Int)

  val addresses = List(Address("Grundstrasse", 29),
    Address("Rotherbaum", 38),
    Address("Elbchausee", 1))

  // ???
  addresses.filter {
    case Address(str, nr) => nr > 1
  }

  addresses.filter { a: Address => a.nr > 1}

}
```

9. Implicits

9.1. Implizite Funktionen

Implizite Funktionen oder Werte befinden sich im Scope und können vom Compiler automatisch aufgelöst werden. Implizite Funktionen werden beispielsweise häufig zur automatischen Typkonvertierung verwendet. Im folgenden Beispiel wird eine Klasse FacInt erzeugt, die ein Int kapselt und die Funktion ! zur Berechnung der Fakultät zur Verfügung stellt. Im Compagnion Object folgen dann die beiden impliziten Funktionen, die ein Int in ein FacInt wandeln, und wieder zurück. So ist es möglich - wie in der Repl gezeigt - auf einen Int Wert die Funktion ! aufzurufen, sofern sich das Compagnion Object im Scope befindet und vom Compiler gefunden werden kann.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class FacInt(value: Int) {
  def !() : Int = {
    (1 to value).foldLeft(1)(_ * _)
  }
}

object FacInt {
  implicit def intToFacInt(value: Int) : FacInt= FacInt(value)
  implicit def facIntToInt(x: FacInt)  : Int= x.value
}

// Exiting paste mode, now interpreting.


scala> 3!
<console>:8: error: value ! is not a member of Int
              3!
               ^
scala> import FacInt._
import FacInt._
```

```
scala> 3!
res1: Int = 6

scala>
```

Dieses Muster ist insbesondere deswegen so nützlich, weil man mit impliziten Funktionen sehr einfach bestehende Klassen (Double) um Funktionen erweitern kann, ohne deren Sourcecode zu modifizieren (wenn man ihn überhaupt hat).

9.2. Implizite Werte

Implizite Werte können verwendet werden, wenn Aufrufe von Funktionen parametrisiert werden sollen, die konfigurierbare Standardwerte benötigen.

```
package demoimplicit

object Main extends App {
  import Helper._

  implicit val e = 0.00001
  println("The result is: " + (3.0 =~= 3.0000001))
  println("The result is: " + (3.0 =~= 3.0001))
  println("The result is: " + (3.0.=~=(3.0001)(0.1)))
}

object Helper {

  implicit class DoubleWrapper(val underlying: Double)
    extends AnyVal {
    def =~=(that: Double)(implicit epsilon: Double) = {
      ((underlying - that).abs <= epsilon)
    }
  }
}
```

9.3. Implizite Klassen und Value Classes

Wichtige Eigenschaften impliziter Klassen:

- müssen sich in einen Scope befinden

- Hauptkontraktor wird zu einer impliziten Funktion, die die Wandlung von unterliegenden Typ in den Typ der impliziten Klasse vornimmt.
- der Konstruktor hat genau ein nicht implizites Argument
- es darf keinen gleichnamigen Namen im Scope geben => keine case class (aufgrund des impliziten Compagnion Object)

```
object Helper {

  implicit class DoubleWrapper(val underlying: Double)
    extends AnyVal {
    def =~=(that: Double)(implicit epsilon: Double) = {
      ((underlying - that).abs <= epsilon)
    }
  }
}
```

TODO: Erklärung beenden

() => Notizen

10. Collections

Collections kommen in 2 Varianten: Mutable, das heisst veränderlich und ummutable, das heisst, die Collection kann nach Anlage, nicht mehr verändert werden. Zugunsten des Denkmusters funktionaler Programmierung, werden in Scala immutable Collections bevorzugt.

10.1. Basistraits

10.1.1. foreach

Tabelle 10.1.: foreach

foreach	führt eine Funktion auf jedem Element aus und gibt Unit zurück

An der Spitze der Collection API steht der Trait „Traversable“. Für uns wichtig ist, dass alle Collections ein Traversable sind, und damit die abstrakte Funktion foreach anbieten:

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

foreach bekommt eine Funktion übergeben, die ein Element aus der Collection als Parameter akzeptiert und einen Wert zurückliefest. foreach selbst ist aber vom Typ Unit.

```
scala> val coll : Traversable[Integer] = List(1,2,3)
coll: Traversable[Integer] = List(1, 2, 3)

scala> coll.foreach(e => println(e))
1
2
3
```

10.1.2. Addition

Traversables können addiert werden und liefern (bei ummutable Traversables) ein neues Traversable

Tabelle 10.2.: Addition operations

++	addiert 2 Collections und liefert als Ergebnis eine neue Collection zurück

```
scala> val coll2:Traversable[Integer] = List(4,5,6)
coll2: Traversable[Integer] = List(4, 5, 6)

scala> coll ++ coll2
res4: Traversable[Integer] = List(1, 2, 3, 4, 5, 6)
```

10.1.3. Map operations

Die Mapoperationen iterieren durch eine Collection und liefern eine neue Collection zurück.

Tabelle 10.3.: Map operations

map	konvertiert Elemente anhand einer Funktion
flatMap	konvertiert Elemente anhand einer Funktion
flatten	„plättet“ das Ergebnis
collect	

Vereinfacht betrachtet hat map die Signatur

```
def map[A](f: A => B) : Traversable[B]
```

map nimmt also einzelnen die Elemente aus der einen Liste, wendet die Funktion f auf das Element an, und steckt das Ergebnis in die Ausgangsliste. Das Wort Liste ist in diesen Kontext vielleicht hilfreich, aber eigentlich ist das Traversable über welches wir hier sprechen ganz allgemein betrachtet einfach irgendeine Art von Container, oder wie man häufig auch sagt: ein „Computational Context“.

Und ein Beispiel:

```
scala> coll.map(e => e.toString)
```

```
res5: Traversable[String] = List(1, 2, 3)

scala> coll.map(e => List(-e, e))
res6: Traversable[List[Any]] =
    List(List(-1, 1), List(-2, 2), List(-3, 3))
```

Im letzten Beispiel wurde als Ergebnis eine Liste zurückgegeben, die wiederum Listenelemente enthält, oder allgemeiner, wir haben einen Container, der wiederum Containerelemente enthält. Diese allgemeine Situation tritt recht häufig auf, wann immer wir mit Containern von irgendetwas arbeiten (Collections, Futures, Options etc.). Will man die Container nicht immer wieder in Container packen, kann man das Ergebnis entweder flachdrücken oder das Mapping und flachdrücken gleich in einen Schritt durchführen:

```
scala> res6.flatten
res9: Traversable[Any] = List(-1, 1, -2, 2, -3, 3)

scala> coll.flatMap(e => List(-e, e))
res10: Traversable[Any] = List(-1, 1, -2, 2, -3, 3)
```

Es lohnt sich an dieser Stelle noch einmal genauer auf die Signatur von flatMap zu achten (vereinfacht):

```
def flatMap[B](f: A => Traversable[B]) : Traversable[B]
```

TODO: collect erklären

10.1.4. Element retrieval operations

Tabelle 10.4.: Element retrieval operations

head	ermittelt das erste Element
last	ermittelt das letzte Element
headOption	das erste Some(element) oder None bei einer leeren Collection
lastOption	dito
find	findet ein Element anhand einer Prädikatfunktion f: A => Boolean

```
scala> val someEmptyColl : Traversable[Integer] =
  Traversable.empty
someEmptyColl: Traversable[Integer] = List()
```

```
scala> coll.head
res13: Integer = 1

scala> someEmptyColl.head
java.util.NoSuchElementException: head of empty list
        at scala.collection.immutable.Nil$.head(List.scala:337)
        ...

scala> someEmptyColl.headOption
res15: Option[Integer] = None

scala> coll.headOption
res16: Option[Integer] = Some(1)

scala> val list = List(1,-1,2,-2)
list: List[Int] = List(1, -1, 2, -2)

scala> list.find(e => e<0)
res19: Option[Int] = Some(-1)
```

10.1.5. Folds

Es gibt weiterhin einige Funktionen, die eine Collection auf einen singulären Wert reduzieren.

Tabelle 10.5.: Folds

reduce	$reduce[A1 >: A](op : (A1, A1) => A1) : A1$ wendet die binäre Operation op auf die Werte von coll an.
foldLeft	$foldLeft[B](z : B)(op : (B, A) => B) : B$ wendet die binäre Operation op auf den Startwert z und die Elemente der Collection von links nach rechts an.
foldRight	$foldRight[B](z : B)(op : (A, B) => B) : B$ wie foldLeft aber von rechts nach links

10.2. for comprehension

for comprehensions sind eine syntaktische Kurzschreibweise für die Verkettung von flatMap und Map Operationen.

Eine for comprehension hat die Form:

```
for {
```

```
seq
} yield { expr }
```

wobei die angedeutete Sequenz eine Sequenz von sogenannten Generatoren der Form a <- b ist. Eine for comprehension wird immer in entsprechende Aufrufe von flatMap und map aufgerufen, deswegen müssen diese Operation für die Werte auf der rechten Seite definiert sein.

Die Übersetzung erfolgt dabei nach folgenden Schema:

10.2.1. 1 Generator

Die Anwendung eines Generators ist äquivalent zur Verwendung von map.

```
def add1(x: Option[Int]) : Option[Int] = {
    val debug = x.map(a => a+1)
    for {
      a <- x
    } yield a + 1
}
```

10.2.2. 2 Generatoren und mehr Generatoren

Die Anwendung von 2 Generatoren erzeugt eine Sequenz von flatMap und map.

```
def add(x: Option[Int], y: Option[Int]) : Option[Int] = {
    val debug = x.flatMap(a => for (b <- y) yield a + b)
    val debug2 = x.flatMap(a => y.map(b => a + b))

    for {
        a <- x
        b <- y
    } yield a + b
}
```

Als Beispiel hier noch eine Auflösung für mehrere Generatoren

```
def add(x:Option[Int], y: Option[Int], z: Option[Int]) : Option[Int] = {
    val debug =
      x.flatMap(a => y.flatMap(b => z.map(c => a + b + c)))
    for {
      a <- x
      b <- y
      c <- z
```

```
    } yield a + b + c
}
```

10.2.3. Verständnis

Die for comprehension erlaubt es durch Strukturen zu navigieren, in denen Dinge stecken. Zum Beispiel gibt es List[A], Option[A], Future[A] – alles Strukturen die wiederum Dinge enthalten. Die enthaltenen Dinge können geändert werden und ggf. auch durch ihren Typ wandeln. Es ist aber nicht möglich, die Struktur zu ändern. Beginnt man mit einer Liste kommt am Ende der for comprehension auch wieder eine Liste heraus. Strukturen wie die oben genannten verhalten sich im funktionalen Terminus monadisch, und monadische Strukturen können mit map/flatMap bearbeitet werden. Die for comprehension stellt hierfür einfach eine besser lesbare syntaktische Alternative dar.

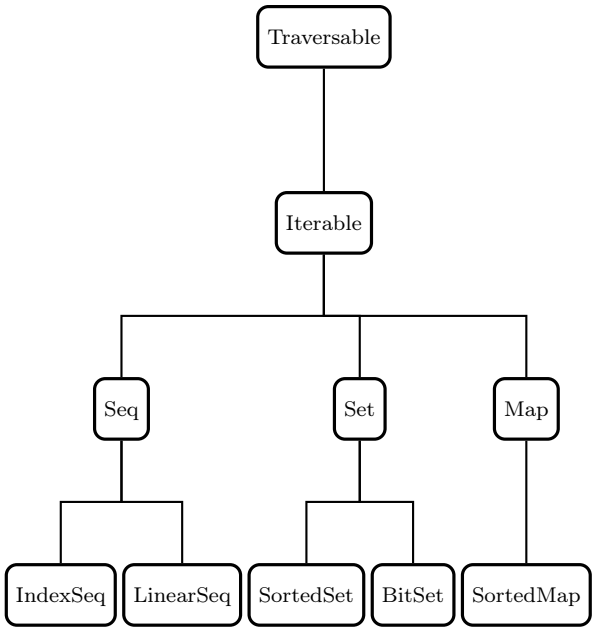


Abbildung 10.1.: Basistraits

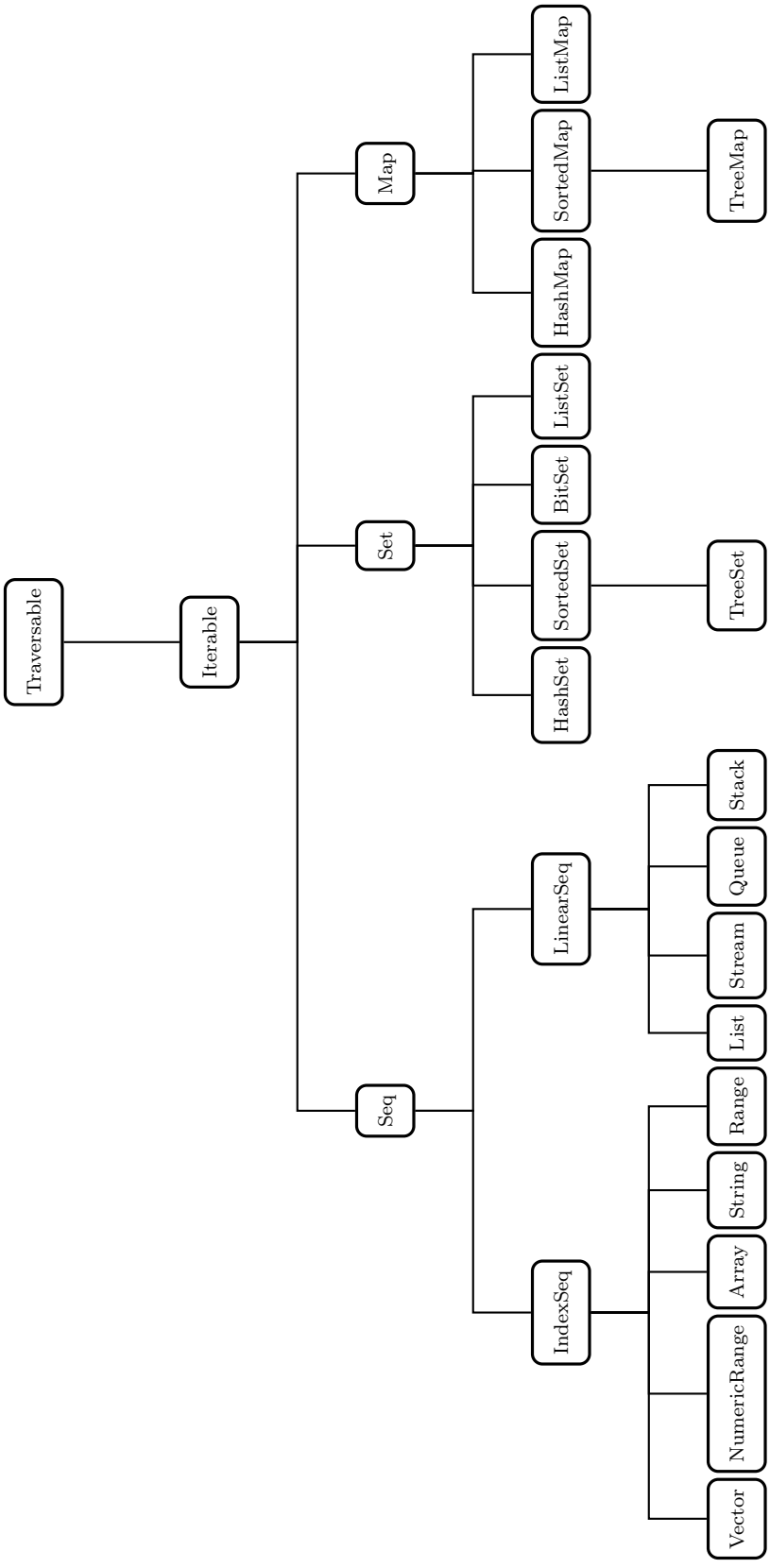


Abbildung 10.2.: Collection Hierachy

11. Option, Try, Either

11.1. Option

Eine Option repräsentiert einen optionalen Wert eines bestimmten Typs A. Optionen können verwendet werden, um auf null - Werte zu verzichten, und lassen sich in for comprehensions einsetzen. Durch Verwendung des Typs Option wird den Programmier bewusst signalisiert, dass es sich hier um einen optionalen Wert handelt (im Gegensatz zur Verwendung von null).

```
sealed abstract class Option[+A]
final case class Some[+A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

11.2. Try

Try repräsentiert eine Berechnung, die entweder einen Erfolg (Success) oder einen Fehler liefert (Failure). Das Scheitern einer Berechnung wird stets durch eine Exception/Throwable ausgedrückt.

```
sealed abstract class Try[+T]
final case class Failure[+T](val exception: Throwable) extends Try[T]
final case class Success[+T](value: T) extends Try[T]

object Try {
  def apply[T](r: => T): Try[T] =
    try Success(r) catch {
      case NonFatal(e) => Failure(e)
    }
}
```

11.3. Either

Ein Either hält entweder einen linken oder einen rechten Wert. Traditionell beinhaltet der linke Wert einen Fehlerwert, und der rechte Wert den richtigen

Wert. Im Unterschied zu Option, kann ein fehlender Wert also zum Beispiel durch eine Fehlermeldung unterstützt werden. Und im Unterscheid zu Try muss die fehlerhafte Berechnung nicht durch eine Exception symbolisiert werden.

Either ist nicht monadisch, und kann ohne weiteres nicht in for comprehensions eingesetzt werden. Das liegt daran, dass ohne weiteres hinzutun nicht bekannt ist, ob eine Sequenz durch die linke oder rechte Seite gewählt werden soll.

```
sealed abstract class Either[+A, +B]
final case class Left[+A, +B](a: A) extends Either[A, B]
final case class Right[+A, +B](b: B) extends Either[A, B]
```

12. Futures

Dr. Emmett Brown: Then tell me, future boy, who's President of the United States in 1985?
Marty McFly: Ronald Reagan.
Dr. Emmett Brown: Ronald Reagan? The actor?

(Back to the future, 1985)

12.1. Basics

```
trait Future[+T] extends Awaitable[T] {
  def onComplete[U](func: Try[T] => U): Unit
  def recover[U >: T](pf: PartialFunction[Throwable, U]): Future[U]
  def recoverWith[U >: T](pf: PartialFunction[Throwable, Future[U]]) : Future[U]
}
object Future {
  def apply[T](body: =>T) : Future[T]
}
```

Ein Future repräsentiert eine langandauernde Berechnung, die scheitern kann. Mit Hilfe von Futures lassen sich Berechnungen in den Hintergrund verschieben, das heisst, sie werden ggf. auf einen eigenen Thread ausgeführt und stehen erst später als Ergebnis zur Verfügung. Derweil kann die Applikation schon weitere Berechnungen tätigen, zum Beispiel einen neuen Webrequest bearbeiten. Futures werden besonders gerne verwendet, um langwierige IO Arbeiten zu tätigen (Datei- oder Datenbankzugriffe), da diese ansonsten den Prozessor blockieren, der nur auf eine Antwort wartet, während das IO System damit beschädigt ist, Daten zu verarbeiten. Ein anderes denkbares Szenario besteht darin, den Aufruf eines Webservice „in die Zukunft“ zu verlagern, und anstelle davon, dass die Antwort des Fremdsystems, unseren Prozessor blockiert, können in der Zwischenzeit weitere Tätigkeiten auf unseren Server ausgeführt werden.

Es ist so gedacht, dass ein Future im Hintergrund ausgeführt wird. Dieser Ausführungskontext muss konfiguriert werden, und nennt sich der ExecutionContext. Im ExecutionContext wird prinzipiell ein ThreadPool konfiguriert, der zur Ausführung von Futures zur Verfügung steht. Die verschiedenen Frameworks und auch Scala selbst stellen meist einen vorkonfigurierten ExecutionContext zur Verfügung, der als impliziter Parameter den verschiedenen Funktionen zur Verfügung steht.

```
scala> import scala.concurrent._
import scala.concurrent._

scala> import ExecutionContext.Implicits.global
import ExecutionContext.Implicits.global

scala> import scala.util._
import scala.util._

scala> val myFuture = Future {
     | // langwierige Operation
     | 3
     | }
myFuture: Future[Int] =
  scala.concurrent.impl.Promise$DefaultPromise@556edb6b

scala> myFuture.onComplete {
     | // callback
     | case Success(e) => println(e)
     | case Failure(_) => println("D'oh!")
     | }

scala> 3
```

12.2. map & flatMap

Definieren wir uns nun einfach mal eine einfache Funktion add, die allerdings nicht sofort einen Wert liefert, sondern das Ergebnis in ein Future packt:

```
scala> def addiere(x: Int, y:Int) = Future { x + y }
addiere: (x: Int, y: Int)Future[Int]
```

Über diese Funktion können wir eine Addition durchführen, deren Wert uns irgendwann zur Verfügung steht.

```
scala> val result = addiere(1,1)
result: Future[Int] =
  scala.concurrent.impl.Promise$DefaultPromise@29cfaebf
```

Wenn wir mit diesen Ergebnis weiterarbeiten wollen – um zum Beispiel das Ergebnis mit 3 zu multiplizieren, dann könnten wir folgende Dinge tun. Entweder wir lassen uns das Ergebnis liefern, verlieren unser Future und rechnen einfach in der Gegenwart weiter, oder man rechnet einfach in der Zukunft. Und dafür benutzen wir map. Der Inhalt eines Containers (Future) kann erfasst werden, und mit Hilfe einer Funktion auf ein neues Ergebnis gekappt werden.

```
scala> result.map(value => value * 3)
res3: Future[Int] =
   scala.concurrent.impl.Promise$DefaultPromise@43f57b4c
```

Wir wir sehen bekommen wir wieder ein Ergebnis vom Typ Future[Int]. Was aber, wenn das Ergebnis unserer Berechnung selbst wieder ein Future wäre? Also wenn wir nicht multiplizieren, sondern mit Hilfe der addiere Funktion den Wert 2 auf result addieren möchten?

```
scala> result.map(value => addiere(value,2))
res4: Future[Future[Int]] =
  scala.concurrent.impl.Promise$DefaultPromise@520aa4b8
```

Das Ergebnis ist ein Future[Future[Int]]. Bäh! Und genau hier können wir wieder an Stelle von map die Funktion flatMap verwenden:

```
scala> result.flatMap(value => addiere(value,2))
res5: Future[Int] =
  scala.concurrent.impl.Promise$DefaultPromise@218ccd2
```

Et voilà! Das Ergebnis ist wiederum ein normales Future[Int].

Wenn Futures map und flatMap anbieten können wir den sequentiellen Ablauf der Berechnung auch einfach mit Hilfe einer for comprehension darstellen:

```
scala> val calc1 = Future { true }

scala> val calc2 = Future { false }

scala> val calc3 : Future[Boolean] =
       |  Future { throw new Exception("Server Down") }

scala> for {
```

```
     | a <- calc1
     | b <- calc2
     | c <- calc3
     | } yield { a && b && c }
res0: scala.concurrent.Future[Boolean] =
   |  scala.concurrent.impl.Promise$DefaultPromise@572a9eef

scala> Await.result(res0, 5 seconds)
java.lang.Exception: Server Down
```

12.3. Callbacks

```
def onComplete[U](func: Try[T] => U): Unit
```

An einen Future können Callbacks (auch mehrere) registriert werden, die ausgeführt werden, sobald der Future zu einen Ergebnis gelangt ist. Dabei ist der Rückgabewert Unit zu beachten! Callbacks eignen sich also nicht per se zum Future-chaining.

Desöfteren hat man aber das Problem, dass man einen Programmfluss auf der Basis von Futures gestalten möchte, der in der Form von for comprehensions nicht mehr elegant lösbar ist. for comprehensions hangeln sich ja immer nur durch den "guten"Programmfluss, und reichen in der Sequenz aufgetretene Fehler nach unten weiter, auf die dann nicht mehr richtig reagiert werden kann.

12.4. Promise

12.5. async / await

12.6. Nützliches

Nehmen wir nun mal an, wir haben eine Liste der Zahlen von 1 bis 10 und wollen auf jede Zahl 1 addieren:

```
scala> (1 to 10).map(i => addiere(i,1))
res6: scala.collection.immutable.IndexedSeq[Future[Int]] =
  Vector(scala.concurrent.impl.Promise$DefaultPromise@63ccd69e, ...)
```

Das Ergebnis ist ein recht hässliche Liste, nämlich in diesem Fall ein Vector[Future[Int]]. Wir müssten, wenn wir die Ergebnisse abfragen wollen, also stets auf jedes einzelne dieser Elemente eine Callbackfunktion registrieren, was

sehr mühselig sein kann. Viel schöner wäre es, wenn wir die Liste der Future in einen Future einer Liste Future[List[Int]] konvertieren könnten:

```
scala> Future.sequence(res6)
res7: Future[scala.collection.immutable.IndexedSeq[Int]] =
  scala.concurrent.impl.Promise$DefaultPromise@5e9b626a
```

Dafür kann man die Funktion sequence des Future Compagnion Objekts verwenden.

() => Notizen

13. Actors

Actor Akteure kapseln einen änderbaren Zustand, besitzen Verhalten und reagieren auf Nachrichten, die sie über eine Mailbox empfangen. Akteure bilden eine Hierarchie (Baum).

Akteure kommunizieren über Nachrichten. Die wichtigste Eigenschaft von Akteuren besteht darin, dass diese Nachrichten stets nacheinander abgearbeitet werden, es kann also (idealerweise) niemals ein nebenläufiger Zugriff auf den Zustand eines Akteurs erfolgen. Akteure können Nachrichten empfangen und senden. Nachrichten sind immutable (oder sollten es zumindest sein). Das Bearbeiten von Nachrichten erfolgt über die Akteure hinweg asynchron und parallel. Es gibt keine Garantie für das Zustellen einer Nachricht. Nachrichten können also verloren gehen. Diese fehlende Garantie ist ein wichtiger Aspekt bei der Arbeit mit Akteuren, da Akteure sehr häufig dazu genutzt werden fehlertolerante Systeme zu bauen, und Teile des Systems können ausfallen.

```
package demoactor

import akka.actor._
import akka.pattern._
import akka.util._
import scala.concurrent.duration._
import scala.concurrent.Await
import scala.concurrent.ExecutionContext.Implicits._

object Simple extends App {

  val actorSystem = ActorSystem("MySystem")
  val actor = actorSystem.actorOf(Props[Responder])
  implicit val timeout : Timeout = 5.seconds

  actor ! SetName("Markus")
  actor ! SetName("Johannes")

  val msg = Await.result(
      (actor ? Hello)
```

```
      .mapTo[Reply], 5.seconds)
  println(msg.reply)

}

class Responder extends Actor {
  var name : Option[String] = None

  def receive = {
    case SetName(n) => name = Some(n)
    case Hello => {
      val answer = name getOrElse("I haven't got a name yet")
      sender ! Reply(answer)
    }
  }
}

case class SetName(name : String)
case object Hello
case class Reply(reply: String)
```

14. Testen mit specs2

Would you please tell me
when my light turns green?

(Dexy's Midnight Runners)

14.1. sbt config

Es gibt eine Vielzahl von Testframeworks, die mit Scala und Play2 arbeiten. Am besten gefällt mir specs2 (see Torreborre).

`http://etorreborre.github.io/specs2/guide/org.specs2.guide.UserGuide.html`

Als Abhängigkeiten benötigt man

```
libraryDependencies ++= Seq(
"org.specs2" %% "specs2" % "2.2.2" %"test",
"junit" % "junit" % "4.11" % "test"
)
```

ACHTUNG: Wenn mit Play gearbeitet wird, sollte specs2 nicht mit in die Dependencies aufgenommen werden, da es sonst zu unscheinbaren Konflikten kommt.

Die Abhängigkeit zu JUnit wird nur benötigt, sofern man die Test in eclipse laufen lassen möchte, ansonsten reicht der einfache sbt Task `sbt test` oder `testOnly common.config.ConfigSpec`, um eine einzelne Testspezifikation auszuführen.

14.2. Specifications, Fragmente und Example

Specs2 kennt 2 verschiedene Arten Tests zu spezifizieren: Acceptance Tests und Unit Tests. Wir betrachten hier nur Unit Tests. Beide Arten von Test verhalten sich geringfügig anders.

```
package demotest

import org.junit.runner.RunWith
import org.specs2.mutable._
import org.specs2.runner.JUnitRunner
```

```
@RunWith(classOf[JUnitRunner])
class DemoSpec extends Specification {

  val numbers = Vector(1,2,3,4,5,6)

  "Numbers " should {
    "contain 6 numbers" in {
      numbers must have size(6)
    }

    "contain the numbers 1-6" in {
      numbers.diff(List(1,2,3,4,5,6)) must have size(0)
    }
  }
}
```

Das wirklich wichtige ist der Trait org.specs2.mutable.Specification. Dieser wird benötigt, um Unit Test artige Spezifikationen zu schreiben. Es gibt noch ein Paralleluniversum in `org.specs2`, welches wir ignorieren.

Jeder Block der Gestalt

```
"contain 6 numbers" in {
      numbers must have size(6)
    }
```

nennt sich ein Example. Examples werden mit

```
"Numbers" should {
...
}
```

gruppiert. Solche Gruppen heissen Fragmente. Jedes Fragment kann beliebig viele Examples beinhalten, und jede Spezifikation wiederum beliebig viele Fragmente. Jedes Examples liefert als Rückgabe ein `Result`. Aufgrund einiger weniger Zaubereien können verschiedene Dinge in Result konvertiert werden. Die Funktion `in` ist wie folgt definiert: **def** in[T : AsResult](r: =>T): Example. Irgendein Ergebnis vom Typ T muss also in den Container AsResult gepackt werden können, was uns in der Context Bound [T : AsResult] verrät.

Statt mit should can ein Fragment auch über can bzw. » definiert werden, wenn gar kein Text im Output angefügt werden soll.

Sind die Tests geschrieben worden, können sie in sbt ausgeführt werden, bzw. insofern die Specification mit der Annotation `@RunWith(classOf[JUnitRunner])`

versehen worden ist, auch direkt in eclipse. Dafür wird auch junit im Classpath verlangt.

```
> test
[info] DemoSpec
[info]
[info] Numbers  should
[info] + contain 6 numbers
[info] + contain the numbers 1-6
[info]
[info]
[info] Total for specification DemoSpec
[info] Finished in 19 ms
[info] 2 examples, 0 failure, 0 error
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 1 s, completed 31.10.2013 15:11:06
```

14.3. Matcher

`http://etorreborre.github.io/specs2/guide/org.specs2.guide.Matchers.html#Matchers`

TODO: Matcher beschreiben

Jedes Example liefert ein Result, bzw. etwas das als Result interpretiert werden kann.

```
def in[T : AsResult](r: =>T): Example = {
      val example = exampleFactory.newExample(s, r)
      example
    }
```

Sprich: wenn wir die Funktion „in“ aufrufen übergeben wir einen Codeblock der etwas vom Typ T zurückliefert r: =>T, und „in“ verlässt sich aufgrund des Context Bound [T : AsResult], dass dieses T später mal als Result interpretiert werden kann.

Die StandardResults sind die folgenden (recht selbsterklärenden):

```
trait StandardResults {
  def done = Success("DONE")
  def wontdo = Success("WONT_DO")
  def todo = Pending("TODO")
  def pending = Pending("PENDING")
  def anError = Error("error")
```

```
  def success = Success("success")
  def failure = Failure("failure")
  def skipped = Skipped("skipped")
}
```

Must und Should sind wiederum spezielle Funktionen, die ausdrücken, dass wir eine Erwartungshaltung gegenüber einen Ergebnis haben, welches durch einen Matcher ausgedrückt wird.

```
val numbers = Vector(1, 2, 3, 4, 5, 6)

  "Numbers " should {
    "pass these tests" in {
      numbers must have size (6)
      numbers must be size (6)
      numbers must size(6)
      numbers should have size (6)
      numbers must not be empty
      numbers must not be empty and have size (6)
      numbers must contain(5)
      Some(numbers) must beAnInstanceOf[Some[Vector[Int]]]
      numbers(2) must beEqualTo[Int](2)
    }
  }
```

Beispielhaft hier die Definition von must (should ist genau äquivalent):

```
 class MustExpectable[T] private[specs2] (tm: () => T)
   extends Expectable[T](tm) { outer =>
  def must(m: =>Matcher[T])       =
    applyMatcher(m)
  def mustEqual(other: =>Any)     =
    applyMatcher(new BeEqualTo(other))
  def mustNotEqual(other: =>Any) =
    applyMatcher(new BeEqualTo(other).not)
  def must_==(other: =>Any)       =
    applyMatcher(new BeEqualTo(other))
  def must_!=(other: =>Any)       =
    applyMatcher(new BeEqualTo(other).not)
}
object MustExpectable {
  def apply[T](t: =>T) = new MustExpectable(() => t)
}
```

Be und have sind wiederum äquivalent, und können weggelassen werden. Sie reichen eigentlich nur einen erwarteten Matcher durch, um die Sprache des Tests natürlicher klingen zu lassen.

size, empty, contain etc. sind dann wiederum nur Beispiele für eine Vielzahl von Funktionen, mit denen die Tests geschrieben werden können. `http://etorreborre.github.io/specs2/guide/org.specs2.guide.Matchers.html#Matchers`

() => Notizen

Teil II.

Pattern

15. Cake Pattern

15.1. Verwendungszweck

Statisch konfigurierbares Dependency Injection Pattern.

15.2. Aufbau

1. Definiere Traits für jedes Stück Kuchen / Komponente
   a) Deklariere über den self Typ, ob in diese Komponente andere Komponenten injiziert werden sollen
   b) Definiere innerhalb der Komponente Traits oder Klassen, die als Service angeboten werden sollen.
   c) Definiere innerhalb der Komponente ein abstraktes Feld vom Typ des Service.
2. Füge die Komponenten zusammen
   a) Erstelle ein Objekt mit den Mixen Traits der benötigten Komponenten.
   b) Überschreibe jedes geerbte Feld mit der konkreten Implementierung des Service der benötigt wird.

15.3. Beispiel

```
package cakepattern

object Main extends App with ConcreteCake1Component with Cake2Component {
  val cake2Service = new Cake2Service {}

  println(cake2Service.cake2Function())
}
```

```
trait Cake1Component {
  self =>
  val cake1Service: Cake1Service
  trait Cake1Service {
    def cake1Function(): String
  }
}

trait ConcreteCake1Component extends Cake1Component {
  self =>
  val cake1Service = new ConcreteCake1Service

  class ConcreteCake1Service extends Cake1Service {
    def cake1Function(): String = "Hello"
  }
}

trait Cake2Component {
  self: Cake1Component =>

  val cake2Service: Cake2Service
  trait Cake2Service {
    def cake2Function() = {
      cake1Service.cake1Function() + ", world"
    }
  }

}
```

16. Signaturen

Früher oder später wird man sich mit diesen merkwürdigen Konzepten wie Functor, Monad, Applicative und der Scalabibliothek scalaz beschäftigen, und sich so den funktionalen „Design Pattern“ annähern. Die folgenden Abschnitte geben hierzu einen kleinen Überblick **??**

$$(A => B) => (F[A] => F[B]) \qquad \text{(Functor/map)}$$

$$(A => F[B]) => (F[A] => F[B]) \qquad \text{(Monad/flatMap)}$$

$$(F[A => B]) => (F[A] => F[B]) \qquad \text{(Applicative/apply)}$$

Betrachten wir die „Gleichung“ für Functor:

$$(A => B) => (F[A] => F[B]) \tag{16.1}$$

Was sie ausdrückt, ist dass wir eine Funktion von $A => B$ haben, und eine Funktion von $F[A] => F[B]$ erhalten. Lassen wir das zweite Klammerpaar weg:

$$(A => B) => F[A] => F[B] \tag{16.2}$$

und formulieren die Funktion um (decurrying):

$$(A => B), F[A]) => F[B] \tag{16.3}$$

was wiederum dieser Funktion entspricht (umdrehen der Parameter):

$$(F[A], A => B)) => F[B] \tag{16.4}$$

Das entspricht nun schon sehr unserer Verwendung von map:

```
scala> val intOpt= Some(1)
intOpt: Some[Int] = Some(1)

scala> intOpt map { (x: Int) => (x * x).toString()}
```

```
res0: Option[String] = Some(1)
```

intOpt entspricht unseren F[A], $A => B$ der Funktion (x: Int) => (x * x).toString() und das Ergebnis vom Typ Option[String] wiederum F[B]. Mit Hilfe von map ist es also möglich Funktionen auf Werte innerhalb von Container anzuwenden.

Diese Signaturen sind an und für sich noch nicht nützlich. Andererseits wird man sie immer wieder sehen, genauso wie man als Java Programmierer darauf achtet, ob man irgendwo eine statische INSTANCE Methode sieht, und dann weiss, dass man es mit einen Singleton Pattern zu tun hat. Oder Klassen die create Methoden besitzen, und wir denken uns: „Aha, Factory“.

17. Typen

17.1. Werteebene

Betrachten wir zunächst einfache Funktionen, die wir bereits kennengelernt haben, und die hier in ihrer curried Fassung hingeschrieben sind:

```
scala> val x  = 3
x: Int = 3

scala> val f = { x: Int => x.toString() }
f: Int => String = <function1>

scala> val g = { x: Int => y:Int => x.toString() + y.toString()}
g: Int => (Int => String) = <function1>

scala> def h(f: Int => String) = f(3)
h: (f: Int => String)String

scala> h _
res0: (Int => String) => String = <function1>
```

Wir haben zunächst Werte, die wir direkt instantiieren können. Also so etwas wie Int, String, Person, und viele weitere. Dann haben wir Funktionen mit Parametern, die uns wiederum einen Wert zurückliefen (f,g). Man könnte sagen, dass sie aus den Eingangsparametern einen Wert konstruieren. Und wir haben Funktionen höherer Ordnung, die als Parameter eine Funktion erwarten, und ggf. sogar wiederum Funktionen zurückliefen (h).

17.2. Typebene

Auf der Typebene haben wir diese Stufigkeit in ganz ähnlicher Weise.

```
scala> kind[Int]
res0: String = Int's kind is *. This is a proper type.
```

```
scala> kind[List[_]]
res2: String = List's kind is * -> *.
This is a type constructor: a 1st-order-kinded type.

scala> kind[Either[_,_]]
res3: String = Either's kind is *->*->*.
This is a type constructor: a 1st-order-kinded type.

scala> kind[Monad[List]]
res6: String = Monad's kind is (* -> *) -> *.
This is a type constructor that takes type constructor(s):
a higher-kinded type.
```

Wir haben Typen Int, String, Person... Diese Typen können wir direkt instanzieren. Dann haben wir sogenannte First Order Kinder Types, wie zum Beispiel List. Zum Erzeugen von List müssen wir, wie bei einer Funktion Parameter übergeben. Mal einen (List), oder mehrere (Either). Solche Typen nennt man First Order Kinder Types. Und zuguterletzt haben wir auf der Typebene das was die Funktionen höherer Ordnung sind: Type Constructors, wie Monad, welcher als Parameter wiederum einen First Order Kinder Type erwartet.

18. Basisklassen funktionaler Programmierung

18.1. Typeclasses

Aufgrund des mächtigen Typsystems von Scala verwendet man zum Teil andere Idiome als es in objektorientierten Programmiersprachen üblich ist. Zum Beispiel spielt Vererbung als Mittel zur Wiederwendung von Funktionalitäten eine wesentlich untergeordnetere Rolle. Stattdessen werden sehr häufig Traits definiert, die mit Hilfen von impliziten Klassen, die Funktionalität zur Verfügung stellen.

Dabei probiert man sehr häufig Funktionalitäten auf Typbasis ein für alle mal zu lösen. Das führt teilweise zu einer recht mathematischen Betrachtungsweise, über die man allerdings exzellent nachdenken kann. Gemeinsam mit dem Postulat der Freiheit von Seiteneffekten, ergeben sich dann erstaunliche Wiederverwendungsmöglichkeiten dieser Typklassen im Kontext unterschiedlichster Algorithmen.

18.2. Monoid

Der Kern eines Monoid ist ganz einfach erklärt:

```
trait Monoid[A] {
    def append(a: A, b: A): A
    def zero: A
  }
```

Ein Monoid für den Typ A hat also genau eine Funktion *append* mit zwei Parametern, und liefert wieder ein A zurück. Außerdem gibt es ein neutrales A in der Rolle der "Null".

Zwei Beispiele:

```
implicit val intMonoid = new Monoid[Int] {
        def append(a: Int, b: Int) = a + b
```

```
    def zero = 0
}

implicit def listInstances[A] = new Monoid[List[A]] {
    def append(a: List[A], b: List[A]) = a ++ b
    def zero : List[A] = List.empty
}
```

Aber über die Operation minus mit zero 0 wäre Int kein Monoid, denn Typklassen müssen nicht nur ihre Typsignaturen einhalten, sondern auch geforderte Gesetzmässigkeiten. Für einen Monoid sind diese ebenfalls einfach zu verstehen.

Es gilt:

$$\forall a \in A \quad a = append(a, zero) \qquad \text{(Left Indentity)}$$

$$\forall a \in A \quad a = append(zero, a) \qquad \text{(Right Identity)}$$

$$\forall a, b, c \in A \quad (a \text{ append } b) \text{ append } c = a \text{ append } (b \text{ append } c) \qquad \text{(Associativity)}$$

An diesen Gesetzen gibt es auch nichts zu diskutieren. Sie gelten immer, ohne Ausnahme! [1]

18.3. Functor

Obwohl er in den Theorien meist an erster Stelle steht, folgt nun erst der Funktor, da seine Implementierung syntaktisch ein klein wenig komplizierter ist.

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B) : F[B]
}
```

Ein Funktor beschreibt eine Function $f : A => B$, welche auf einen Funktor vom Typ F[A] angewendet werden kann, und ein F[B] liefert. F ist dabei ein Typkonstruktor, also so etwas wie Option, List, Future etc.

Auch hier gibt es wieder Gesetze, nämlich das der Komposition:

$$map(f) \circ map(g) = map(f \circ g) \qquad \text{(Composition)}$$

[1] Gibt es kein *zero* Element, aber die binäre Funktion *append*, so spricht man von einer Semigroup.

18.4. Applicative

Der nächste Baustein in unseren Sortiment ist der Applicative. Während ein Funktor es erlaubt, eine Funktion auf ein Element innerhalb eines Kontext auszuführen, und wiederum einen Wert im Kontext zurückliefert, arbeitet der Applicative(-functor), so dass eine Funktion die sich im Kontext befindet auf einen Wert im Kontext angewandt wird, und einen Wert im Kontext zurückliefert.

Oder in konkreten Beispielen:

- eine Liste mit Funktionswerten kann auf eine Liste von Funktionen angewandt werden, um eine Liste von Ergebnissen zu erhalten.
- ein optionaler Wert kann auf eine optionale Funktion anwendet werden, um ein optionales Ergebnis zu erhalten.

```
trait Applicative[F[_]] {
    def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
    // alias
    def <*>[A,B](fa: F[A])(f: F[A=>B]) : F[B] = ap(fa)(f)
  }
```

18.5. Monad

Monaden haben wir nun schon einige kenngelernt, ohne im Detail darüber gesprochen zu haben:

Option liefert einen optionalen Wert

Try signalisiert, dass eine Berechnung fehlschlagen kann

Future signalisiert, dass eine Berechnung zu einem späteren Zeitpunkt ausgeführt werden kann.

All diesen Typen ist gemein, dass sie die Funktionen *map* und $flatMap$ implementieren, und wie wir gesehen haben, lassen sich mit diesen Funktionen und for comprehensions Berechnungen durchführen, die uns vergessen lassen, mit welchen Sonderfällen wir in diesen Berechnungen umgehen müssen.

$$(A => B) => (F[A] => F[B]) \qquad \text{(Functor/map)}$$

$$(A => F[B]) => (F[A] => F[B]) \qquad \text{(Monad/flatMap)}$$

> A monadic for comprehension is an embedded programming language with the semantics defined by the monad. [2]

Wir hatten auch gesehen, dass es ggf. wünschenswert ist, möglichst lange in unserer Monate zu bleiben, weil das Äuspacken"des darin enthaltenen Wertes potentiell gefährlich ist (Option liefert None, das Future ist fehlgeschlagen, im Try lauert eine Exception, ...)

[2] `http://de.slideshare.net/StackMob/monad-transformers-in-the-wild`

Literaturverzeichnis

Reactive manifesto. URL http://www.reactivemanifesto.org/.

Scala downloads. URL http://www.scala-lang.org/downloads.

2012. URL http://thedet.wordpress.com/2012/05/04/functors-monads-applicatives-different-implementations/.

Scala glossary, November 2013. URL http://docs.scala-lang.org/glossary/.

Jamie Allen. *Effective Akka*. O'Reilly, 2013.

Pavlo Baron. *Erlang/OTP*. Open Source Press, 2012.

Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala*. Manning Publications.

Peter Eisentraut and Bernd Helmle. *PostgreSQL*. O'Reilly, 2013.

Cay S. Horstmann. *Scala for the impatient*. Addison-Wesley, 2012.

F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics*. Cambridge Press, 2nd edition, 2009.

Miran Lipovaca. *Learn you a haskell for great good*. No Starch Press, 2011.

Martin Odersky and Lex Spoon. Scala collections. URL http://docs.scala-lang.org/overviews/collections/introduction.html.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 1st edition, 2008.

Stefan Tilkov. *REST und HTTP*. dpunkt.Verlag, 2nd edition, 2011.

Eric Torreborre. Specs2 test framework. URL http://etorreborre.github.io/specs2/.

Eugene Yokota. Learning scalaz. URL `http://eed3si9n.com/learning-scalaz/`.

Index