



Das Beste aus zwei Welten

Scala – Teil 2: FP und OOP

Heiko Seeberger, Jan Blankenhorn

Scala hat derzeit kräftigen Rückenwind, denn diese Sprache ist nicht nur einfacher, sondern auch mächtiger als Java. Ein knapper und prägnanter Programmierstil sowie neue Möglichkeiten durch funktionale Programmierung lassen Entwicklerherzen höher schlagen. Dazu noch viel Flexibilität und volle Interoperabilität mit Java. Wie im ersten Teil geht es auch hier um funktionale Programmierung, aber auch Objektorientierung und Vererbung kommen nicht zu kurz.

Vererbung und Traits

Wenn wir erst einmal nur Klassen betrachten, dann sieht die Vererbung in Scala ganz ähnlich aus wie in Java. Lassen Sie uns zunächst die Klasse **Animal** definieren:

```
scala> class Animal
defined class Animal
```

Und nun, davon abgeleitet, die Klasse **Bird**, indem wir das Schlüsselwort **extends** verwenden:

```
scala> class Bird extends Animal
defined class Bird
```

Analog zu Java können wir in Scala nur eine Superklasse erweitern. Selbstverständlich werden alle **public** und **protected** Members (Felder und Methoden) vererbt.

Auch für abstrakte Klassen, die mit dem Schlüsselwort **abstract** definiert werden, gelten die aus Java bekannten Regeln. Allerdings können bzw. müssen wir bei abstrakten Members auf das Schlüsselwort **abstract** verzichten und einfach die Zuweisung des Feldes oder den Rumpf der Methode weglassen:

```
scala> abstract class Animal {
|   def name: String
|   override def toString = name
| }
defined class Animal
```

Hiermit haben wir die abstrakte Klasse **Animal** mit der abstrakten Methode **name** definiert, die wir gleich verwenden, um **toString** zu überschreiben. Zur Erinnerung (s. auch [SeeBla10]): Die vertikalen Striche gehören nicht zum Scala-Code, sondern sind eine Besonderheit der REPL (Read Evaluate Print Loop) und signalisieren, dass die Eingabe als noch nicht abgeschlossen betrachtet wird.

Warum verwenden wir eine Methode für den Namen? Nun ja, vielleicht können Tiere ja auch wie wir Menschen ihren Namen ändern, z. B. bei einer Vogelhochzeit. In der Regel wird der Name aber konstant bleiben. Daher definieren wir jetzt eine konkrete Basisklasse, welche die Methode **name** mit einem gleichnamigen **val** (unveränderliches Feld) implementiert:

```
scala> class DefaultAnimal(override val name: String) extends Animal
defined class DefaultAnimal
```

Hiermit haben wir den Klassenparameter **name** durch das Schlüsselwort **val** zum unveränderlichen Feld und dadurch zum Parameter für den Primary Constructor gemacht. Die

Verwendung von **override** ist optional, weil wir ein abstraktes Member implementieren und nicht ein konkretes überschreiben, dient jedoch dem Verständnis und der Sicherheit, Schreibfehler zu vermeiden.

Nun wollen wir unseren Vogel fliegen lassen:

```
scala> class Bird(name: String) extends DefaultAnimal(name) {
|   def fly = "I am flying!"
| }
defined class Bird

scala> val bill = new Bird("Bill")
bill: Bird = Bill

scala> bill.fly
res0: java.lang.String = I am flying!
```

Und nun definieren wir noch einen **Fish**, der natürlich schwimmen kann:

```
scala> class Fish(name: String) extends DefaultAnimal(name) {
|   def swim = "I am swimming!"
| }
defined class Fish

scala> val freddy = new Fish("Freddy")
freddy: Fish = Freddy

scala> freddy.swim
res1: java.lang.String = I am swimming!
```

So weit, so gut! Abgesehen von der knapperen Syntax gibt es bisher keinen elementaren Unterschied zu Java. Nun wollen wir eine Ente definieren, die natürlich ein Vogel ist, aber auch schwimmen kann, und zwar genauso wie ein Fisch. Das ist zwar zugegebenermaßen etwas konstruiert, denn Fische schwimmen bekanntlich anders als Enten. Aber „im echten Leben“ möchten wir schon oft eine gemeinsame Basisimplementierung verwenden, denn das reduziert Code und spart Zeit und damit Kosten. Das ist in Java jedoch nicht möglich, denn dort können wir nur von einer Klasse erben und Interfaces sind ausschließlich abstrakt.

Hier kommen in Scala Traits (auf Deutsch etwa Merkmale) ins Spiel, sozusagen Interfaces mit optionaler Implementierung. Lassen Sie uns die Methode **swim** in den Trait **Swimmer** auslagern, wobei wir das Schlüsselwort **trait** verwenden:

```
scala> trait Swimmer {
|   def swim = "I am swimming!"
| }
defined trait Swimmer
```

Damit können wir zunächst unseren **Fish** neu definieren, indem wir mit dem Schlüsselwort **with** den Trait **Swimmer** hinein mixen:

```
scala> class Fish(name: String) extends DefaultAnimal(name) with Swimmer
defined class Fish
```

Dann können wir uns der Klasse **Duck** zuwenden, die ein **Bird** mit hinein gemixtem **Swimmer** erweitert:

```
scala> class Duck(name: String) extends Bird(name) with Swimmer
defined class Duck

scala> val donald = new Duck("Donald")
donald: Duck = Donald

scala> donald.fly
res2: java.lang.String = I am flying!

scala> donald.swim
res3: java.lang.String = I am swimming!
```

Wie wir sehen, kann Donald sowohl fliegen als auch schwimmen. Schön, aber ist das nicht Mehrfachvererbung? Ja und nein. Mit Traits erzielen wir denselben Effekt wie bei klassischer Mehrfachvererbung und umgehen typische Probleme, z. B. das Diamond-Problem [Wikipedia]. Ohne jetzt zu tief einzusteigen, heißt die Scala-Lösung Linearisierung: Beim Instantiieren eines Objekts werden alle vererbten und hinein gemixten Typen in eine lineare Reihenfolge gebracht, bei der das Objekt selbst am Anfang steht. Dadurch wird sichergestellt, dass jederzeit klar ist, welcher Typ aufzurufen ist, was mit `super` gemeint ist und dass das betreffende Objekt das Verhalten der Supertypen modifiziert und nicht umgekehrt. Für Details sei auf die einschlägige Literatur verwiesen [OdSpVe08].

Neben dem statischen Hinein-Mixen von Traits, das wir bei der Definition von neuen Typen einsetzen, können wir auch dynamische Mixins anwenden. Dabei ergänzen wir das Anlegen eines neuen Objekts einfach mit dem Schlüsselwort `with` um den Trait, den wir dynamisch hinein mixen wollen:

```
scala> trait MichaelBuble extends Bird {
  |   override def fly = "I feel good and " + super.fly
  | }
defined trait MichaelBuble

scala> val michael = new Bird("Michael") with MichaelBuble
michael: Bird with MichaelBuble = Michael

scala> michael.fly
res4: java.lang.String = I feel good and I am flying!
```

Auf diese Weise erzielen wir einen ähnlichen Effekt wie bei der aspektorientierten Programmierung, d. h. wir können Method-Interception ganz ohne dynamische Proxys oder Compiler-Erweiterungen (z. B. AspectJ) anwenden.

Methoden und Funktionen

Wir sind bereits im ersten Artikel kurz auf funktionale Programmierung in Scala eingegangen. Zur Erinnerung: Den besonderen Unterschied zu rein objektorientierten Sprachen wie Java stellen Funktionen höherer Ordnung dar, d. h. Funktionen, die andere Funktionen als Parameter erwarten.

Was genau ist eigentlich der Unterschied zwischen einer Methode und einer Funktion? Von ihrer Wirkungsweise her sind beide identisch, denn beide werden (mit Parametern) aufgerufen und liefern ein Ergebnis zurück. Aber Methoden sind stets Bestandteil einer Klasse, wohingegen Funktionen selbst Objekte sind. Zur Verdeutlichung ein kurzes Beispiel. Zunächst eine Methode innerhalb eines Singleton Object:

```
scala> object Calculator {
  |   def add(x: Int, y: Int) = x + y
  | }
defined module Calculator

scala> Calculator.add(1, 2)
res0: Int = 3
```

Und nun eine analoge Funktion, definiert durch ein Funktionsliteral, d. h. eine „hingeschriebene Funktion“:

```
scala> val add = (x: Int, y: Int) => x + y
add: (Int, Int) => Int = <function2>

scala> add(1, 2)
res1: Int = 3
```

Die hier gezeigte Verwendung erscheint quasi identisch, denn sowohl die Methode als auch die Funktion werden mit einer

Parameterliste aufgerufen. Der Unterschied tritt erst so richtig zutage, wenn wir eine Funktion als Argument für den Aufruf einer anderen Methode/Funktion verwenden wollen: Argumente müssen Objekte sein, sodass wir nur die Funktion verwenden können. Als Beispiel betrachten wir die Methode `Traversable.reduceLeft`, die eine Funktion vom Typ `Function2` nach und nach auf alle Elemente anwendet:

```
scala> val oneToFour = 1 to 4
oneToFour: ...Range.Inclusive... = Range(1, 2, 3, 4)

scala> oneToFour.reduceLeft add
res3: Int = 10
```

Hier ist `Traversable` der Supertyp aller Collections und `1 to 4` nichts anderes als der Aufruf der Methode `to` auf dem `Int`-Wert 1, der aufgrund einer Implicit Conversion nach `RichInt` möglich wird.

Nun ist die Verwendung von Funktionsliteralen nicht immer praktikabel. Insbesondere definieren wir Funktionalität in der OO-Welt häufig in Methoden und wollen diese gerne als Funktionen verwenden. Zum Glück können wir eine Methode leicht zu einer Funktion machen, indem wir sie durch Ersetzen der Parameterliste durch einen Unterstrich zu einer Partially Applied Function machen. In unserem Beispiel müssen wir also die `add`-Funktion nicht „mühsam“ hinschreiben, sondern können die `add`-Methode wieder verwenden:

```
scala> val add = Calculator.add _
add: (Int, Int) => Int = <function2>
```

Wir könnten natürlich auch ganz auf die Definition der `add`-Funktion verzichten und direkt die Partially Applied Function an `reduceLeft` übergeben:

```
scala> oneToFour.reduceLeft Calculator.add _
res4: Int = 10
```

Der Scala-Compiler bietet uns hier, wo eine Funktion erwartet wird, eine abgekürzte Schreibweise, sodass wir den Unterstrich auch weglassen können. Dann sieht es so aus, als würden wir die `add`-Methode übergeben, aber in Wirklichkeit macht der Compiler daraus ein Funktionsobjekt:

```
scala> oneToFour.reduceLeft Calculator.add
res5: Int = 10
```

Curry und andere Gewürze

Was wir oben durch das Weglassen der gesamten Parameterliste gemacht haben, können wir auch auf einzelne Parameter übertragen:

```
scala> val addOne = Calculator.add(1, _: Int)
addOne: (Int) => Int = <function1>
```

Hier machen wir aus der `add`-Methode, die zwei Parameter hat, die `addOne`-Funktion mit einem Parameter, indem wir den ersten Parameter von `add` fixieren. Diese neue Funktion können wir dann dort verwenden, wo Funktionen mit einem Parameter erwartet werden, z. B. als Argument der Methode `Traversable.map`:

```
scala> oneToFour.map addOne
res6: ...IndexedSeq[Int] = IndexedSeq(2, 3, 4, 5)
```

Hier bietet Scala auch noch eine interessante Alternative: Statt einer Parameterliste mit zwei Parametern können wir auch zwei Parameterlisten mit je einem Parameter schreiben:

```
scala> object Calculator {
  | def add(x: Int)(y: Int) = x + y
  | }
defined module Calculator
```

Dieses Konzept, das natürlich mit beliebigen und beliebig vielen Parameterlisten funktioniert, wird (nach dem Mathematiker Haskell Brooks Curry) Currying genannt. Mit dem Unterstrich lassen wir die letzte Parameterliste weg und können unsere `addOne`-Funktion nun so definieren:

```
scala> val addOne = Calculator.add(1) _
addOne: (Int) => Int = <function1>
```

Schön, aber wozu können wir das eigentlich verwenden? Zum Beispiel, um mit „ganz normalen“ Sprachmitteln neue Kontrollkonstrukte zu definieren, die sich wie Bestandteile der Sprache anfühlen. Solche ausdrucksstarken Konstrukte bieten sich an, um wiederkehrende Muster zu ersetzen, denn dadurch wird die Codemenge verringert und die Verständlichkeit erhöht. Ein Klassiker ist das Loan-Muster, bei dem eine Ressource garantiert wieder geschlossen wird. Lassen Sie uns als konkretes Beispiel das Schreiben in eine Datei betrachten:

```
scala> import java.io._
import java.io._

scala> def withPrintWriter(file: String)(write: PrintWriter => Unit) {
  | val out = new PrintWriter(file)
  | try {
  |   write(out)
  | } finally {
  |   out.close()
  | }
  | }
withPrintWriter: (file: String)(write: (java.io.PrintWriter) => Unit)Unit

scala> withPrintWriter("test.txt") { out =>
  | out.println("TEST")
  | }
```

Hier haben wir die Methode `withPrintWriter` mit zwei Parameterlisten definiert. Immer wenn eine Parameterliste nur einen Parameter hat, können wir anstelle der runden auch geschweifte Klammern verwenden. Dadurch erscheint die Anwendung dieser Methode wie ein Kontrollkonstrukt, z. B. ähnlich wie `while`.

Funktionale Collections

Es ist für funktionale Programmiersprachen typisch, mächtige Collection-Bibliotheken mitzubringen. Daher wollen wir diesen Artikel mit einigen Collection-Beispielen beschließen. Seit Scala 2.8 erben alle Collection-Typen von `Traversable`, wo die meisten Methoden definiert werden. Subtypen implementieren diese gelegentlich optimiert und ergänzen spezifische Methoden.

Über Collections können wir mit `foreach` iterieren, um einen Seiteneffekt zu erzielen. Das ist zwar nicht die reine FP-Lehre, aber pragmatisches Scala:

```
scala> oneToFour foreach print
1234
```

Die Methode `filter` liefert eine neue Collection zurück, die nur die Elemente enthält, für welche die übergebene Funktion `true` ergibt:

```
scala> oneToFour filter { x => x % 2 == 0 }
res9: ...IndexedSeq[Int] = IndexedSeq(2, 4)
```

Die Methode `sort` erwartet eine Funktion, die zwei Elemente vergleicht und `true` zurück liefert, wenn das erste vor dem zweiten kommen soll. Also drehen wir unsere Sequenz doch einfach um:

```
scala> oneToFour sortWith { (x, y) => x > y }
res10: ...IndexedSeq[Int] = IndexedSeq(4, 3, 2, 1)
```

Natürlich können wir Funktionen auch kombinieren, indem wir die zweite auf dem Resultat der ersten aufrufen:

```
scala> oneToFour filter { x => x % 2 == 0 } map { x => x + 1 }
res11: ...IndexedSeq[Int] = IndexedSeq(3, 5)
```

Um alle Werte zu addieren, können wir, wie schon gezeigt, die Methode `reduceLeft` verwenden. Eine ähnliche Methode ist `foldLeft`, wobei wir hier einen Startwert angeben müssen, der einen anderen Typ haben darf, als derjenige der Collection:

```
scala> oneToFour.foldLeft("#") { (s, x) => s + x }
res12: java.lang.String = #1234
```

Wir könnten diese Beispiele schier endlos fortsetzen, denn die Scala-Collections bieten eine riesige Fülle an Methoden höherer Ordnung. Dem interessierten Leser sei daher die Scala-Dokumentation [Scala] empfohlen, um sich weiter über Collections zu orientieren.

Fazit

Wir haben in diesem Artikel Vererbung und funktionale Programmierung genauer unter die Lupe genommen. Dabei sollte deutlich geworden sein, dass Traits, Funktionen höherer Ordnung und Currying mächtige Features darstellen, um unseren Code vom Umfang her zu reduzieren und ausdrucksstärker zu machen. Da wir viel Zeit damit verbringen, Code zu lesen, können wir mit Scala einen nennenswerten Produktivitätsgewinn erzielen. In der kommenden Folge werden wir uns dem Scala-Typsystem widmen.

Literatur und Links

[OdSpVe08] M. Odersky, L. Spoon, B. Venners, Programming in Scala: A comprehensive step-by-step guide, artima 2008, Kapitel 12.6

[Scala] The Scala Programming Language, <http://www.scala-lang.org/>

[SeeBl10] H. Seeberger, J. Blankenhorn, Scala: – Teil 1: Einstieg, in: JavaSPEKTRUM, 2/2010

[Wikipedia] Diamond-Problem, <http://de.wikipedia.org/wiki/Diamond-Problem>



Heiko Seeberger ist geschäftsführender Gesellschafter der Weigle Wilczek GmbH und verantwortlich für die technologische Strategie des Unternehmens mit den Schwerpunkten Java, Scala, OSGi, Eclipse RCP und Lift. Zudem ist er aktiver Open Source Committer, Autor zahlreicher Fachartikel und Redner auf einschlägigen Konferenzen.
E-Mail: seeberger@weiglewilczek.com.



Jan Blankenhorn ist Softwareentwickler bei der Weigle Wilczek GmbH. Er entwickelt sowohl dynamische Ajax-Webanwendungen als auch Rich Clients mit Eclipse RCP. Neben Softwareentwicklungs- und -wartungsprojekten ist er als Trainer für Eclipse RCP im Rahmen der Eclipse Training Alliance aktiv.
E-Mail: blankenhorn@weiglewilczek.com.