Skalierbare Sprache

Scala – Teil 1: Einstieg

Heiko Seeberger, Jan Blankenhorn

Scala hat derzeit kräftigen Rückenwind. Und das zurecht, denn Scala 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. Höchste Zeit für eine Erkundungstour durch Scala-Land, zu der wir Sie im Rahmen dieser Serie einladen.

„Jeder sollte Scala lernen", sagt zumindest Ted Neward. Warum eigentlich nicht? Als Entwickler sollten wir ja nach einem ungeschriebenen Gesetz ohnehin jedes Jahr eine neue Programmiersprache lernen. Aber wieso gerade Scala, wo es doch so viele andere Möglichkeiten gibt?

Ganz einfach: Scala ist eine ausgereifte, objekt-funktionale, statisch typisierte, leichtgewichtige, ausdrucksstarke, pragmatische und skalierbare Sprache für die Java Virtual Machine, die 100 % „abwärtskompatibel" zu Java ist.

Zu viel auf einmal? Keine Sorge, wir werden in dieser Serie auf alle Argumente näher eingehen. In diesem ersten Teil verschaffen wir uns einen groben Überblick und in den beiden weiteren Teilen werden wir dann ausgewählte Themen vertiefen.

Warum Scala?

Die Java-Plattform erfreut sich schon seit vielen Jahren enormer Beliebtheit und es sieht derzeit nicht so aus, als würde ihr Stern bald untergehen. Allerdings hat die Sprache Java Konkurrenz bekommen: Ursprünglich einziger Bewohner der Java-Welt, tummeln sich dort heute etliche neue und zum Teil auch alte Sprachen. Davon tun sich aktuell insbesondere JRuby, Groovy, Clojure und eben Scala hervor.

Natürlich ist ein Blick in die Glaskugel immer eine riskante Sache und daher werden wir hier nicht die Alleinherrschaft von Scala oder den Niedergang einer anderen Sprache weissagen. Das ist wohl auch nicht nötig, denn mit hoher Wahrscheinlichkeit können wir davon ausgehen, dass wir in Zukunft polyglott programmieren werden, d. h. ein Miteinander von verschiedenen Programmiersprachen erleben werden. Wir sind überzeugt, dass Scala dabei eine Rolle spielen wird, und wagen die Prognose, dass diese eher eine Haupt- als eine Nebenrolle sein wird. Das sehen interessanterweise auch die Schöpfer von Groovy und JRuby so [Stra09].

Scala ist – wie Java – eine statisch typisierte Sprache, d. h. der Compiler prüft, ob im Code „alles mit rechten Dingen zugeht". Aus unserer Sicht ist das für viele Einsatzszenarien von Vorteil, insbesondere im Zusammenspiel mit einem mächtigen Typsystem. Denn so lassen sich Fehler schon frühzeitig – nämlich bei der Entwicklung – erkennen und vermeiden, wohingegen bei dynamisch typisierten Sprachen die eine oder andere „Überraschung" im Betrieb zu erwarten ist. Häufig wird gegen statisch typisierte Sprachen vorgebracht, deren Code sei schwergewichtig, z. B. wegen der erforderlichen Typdeklarationen. Scala räumt mit diesem Argument gründlich auf: Typinferenz – der Compiler „errät" die Typen – und eine enorme Flexibilität der Sprache selbst lassen Scala-Code ebenso leichtgewichtig erscheinen wie zum Beispiel Groovy-Code.

Ein weiterer Pluspunkt für Scala ist ihr hybrider Charakter: Scala ist objekt-funktional. Das bedeutet, dass Scala das Beste aus den beiden Welten der Objektorientierung (OO) und der funktionalen Programmierung [WikiPedia] vereint. Manche Probleme lassen sich gut mit OO-Konzepten behandeln, für andere sind funktionale Abstraktionen besser geeignet, z. B. Closures. Mit Scala steht uns jederzeit die beste Variante zur Verfügung. Darüber hinaus wird durch den hybriden Charakter ein sanfter Einstieg ermöglicht, indem z. B. für Java-Entwickler die bestehenden OO-Kenntnisse weiterhin gültig und anwendbar bleiben. Nach und nach können funktionale Ansätze aufgegriffen und dadurch das volle Potenzial von Scala ausgeschöpft werden.

Weitere Argumente für Scala sind z. B. die kompakte und ausdrucksstarke Notation, die fortschrittlichen Möglichkeiten zur Schaffung von internen oder externen DLSs, die herrlich einfache Handhabung von XML oder die volle Interoperabilität mit Java. Kurzum, es gibt jede Menge Gründe, dem Aufruf von Ted Neward zu folgen und Scala-Land zu erkunden.

Hello World à la Scala

Natürlich beginnen wir unsere Erkundungstour mit dem Klassiker schlechthin:

```
object Hello {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

Zunächst entdecken wir viele Parallelen zu Java, aber auch einige Unterschiede, z. B. die Schlüsselwörter **`object`** zur Definition von Singleton Objects und **`def`** zur Definition von Methoden sowie das Fehlen von Semikolons. Auf diese Details und viele weitere gehen wir noch ein, aber zunächst sorgen wir dafür, dass unser Klassiker übersetzt und ausgeführt werden kann.

Mit Version 2.8 gibt es endlich ordentliche Scala-Plug-Ins für populäre Entwicklungsumgebungen wie Eclipse, Netbeans oder IntelliJ IDEA. Allerdings werden wir uns im Folgenden auf die Kommandozeile bzw. auf die REPL (Read Evaluate Print Loop) – die interaktive Scala-Konsole – beschränken. Das gibt uns zum einen ein „unverfälschtes Gefühl" dafür, was hinter den Kulissen abläuft. Zum anderen ist interaktives Scripting in der REPL auch in „echten" Projekten oft sehr nützlich und sollte daher zum Handwerkszeug eines Scala-Entwicklers gehören.

Für obiges „Hello World" sowie die weiteren Beispiele benötigen wir also nur die Scala-Distribution in Version 2.8 [ScalaRC]. Nachdem wir das Archiv entpackt haben, finden wir im **`bin`**-Verzeichnis u. a. die Programme **`scalac`** und **`fsc`** (fast scala

compiler) zum Kompilieren sowie `scala` zum Ausführen von Scala-Programmen bzw. Starten der REPL. Wenn nicht anders vermerkt, dann werden wir für die Code-Beispiele stets die REPL verwenden: Starten Sie einfach `scala`, um die Beispiele nachzuvollziehen.

Schreiben wir nun also unser „Hello World" mit einem beliebigen Texteditor, von denen viele bereits Syntax-Highlighting für Scala beherrschen, in eine Datei namens `Hello.scala`, rufen wir `fsc Hello.scala` auf und anschließend `scala Hello`. Dann sollten wir den bekannten Gruß entgegen nehmen dürfen:

```
demo$ vi Hello.scala
demo$ fsc Hello.scala
demo$ scala Hello
Hello World!
```

OO ganz einfach

Wir nähern uns Scala zunächst aus der OO-Perspektive, für viele Java-Entwickler ein vertrautes Terrain. In Sachen Objektorientierung ist Scala einfacher als Java, denn in Scala ist alles, wirklich alles, ein Objekt. Somit gibt es in Scala auch keine primitiven Datentypen. Zum Beispiel steht anstelle von `int` in Java die Klasse `Int` in Scala:

```
scala> val x = 1
x: Int = 1
```

Hier weisen wir der Variablen `x` ein `Int`-Objekt zu, wobei wir das Schlüsselwort `val` verwenden, mit dem wir eine unveränderliche Variable erhalten. Zunächst fällt auf, dass wir kein Semikolon benötigen: Dank der Semikolon-Inferenz ist das nicht nötig, außer wenn wir z. B. mehrere Anweisungen in eine Zeile schreiben. Dann fällt auf, dass wir keinen Typ angeben: Dank der Typinferenz kann der Compiler den Typ automatisch bestimmen. Wie wir anhand der Antwort der REPL sehen, wird in unserem Beispiel `Int` als Typ verwendet. Natürlich könnten wir auch explizit einen Typ vorgeben, indem wir nach dem Variablennamen einen Doppelpunkt und den Typ schreiben:

```
scala> val x: Double = 1
x: Double = 1.0
```

In der Praxis erweist sich die Typinferenz als sehr probates Mittel, um knappen und lesbaren Code zu erhalten. Denken wir zum Beispiel an eine Liste von ganzen Zahlen. In Java müssten wir auf der „linken Seite" `List<Integer>` und auf der „rechten Seite" `ArrayList<Integer>` schreiben. In Scala genügt `List` auf der „rechten Seite", denn auch die Parametrisierung der `List` mit `Int` wird automatisch erkannt:

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

Im Vorgriff auf die nächste Folge: Hier geben die eckigen Klassen den Typparameter an, so ähnlich wie die spitzen Klammern bei den Java Generics.

Singleton Objects statt static

Wie in aller Welt funktioniert eigentlich `List(1, 3, 5)`? Ganz einfach: In Scala gibt es neben Klassen, die wir mit dem Schlüsselwort `class` definieren, auch sogenannte Singleton Objects, die wir mit dem Schlüsselwort `object` erzeugen. Diese sind echte Objekte, die auch als Parameter übergeben werden können. Dafür gibt es – ganz im Sinne der reinen Objektorientierung – keine static Members.

Neben der Klasse `List` gibt es in der Scala-Bibliothek auch ein gleichnamiges Singleton Object, ein sogenanntes Companion Object, welches die Methode `apply` enthält, die eine neue `List`-Instanz erzeugt. Wann immer wir auf einem Objekt einen Aufruf versuchen, d. h. dahinter Klammern mit Argumenten schreiben, übersetzt der Compiler das im Hintergrund in einen Aufruf der Methode `apply`:

```
scala> List.apply(1, 3, 5)
res0: List[Int] = List(1, 3, 5)
```

Klassen und Felder

Das funktioniert natürlich auch mit unseren eigenen Objekten, als Beispiel dient eine Person mit einem Namen. Zunächst schreiben wir die Klasse, wobei wir nach dem Klassennamen in Klammern die sogenannten Klassenparameter schreiben, die wir quasi als private Felder betrachten können:

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

Wenn wir keinen Zugriffsmodifizierer verwenden, dann können wir Klassen, Felder und Methoden öffentlich nutzen. Es gibt also im Gegensatz zu Java in Scala kein `public`, wohl jedoch `protected` und `private`.

Aus der Klassendefinition ergibt sich implizit der sogenannte Primary Constructor, den wir mit allen Klassenparametern aufrufen:

```
scala> val angie = new Person("Angela Merkel")
angie: Person = Person@6d0cecb2
```

Damit wir unsere Person auch sinnvoll nutzen können, müssen wir auf den Namen lesend zugreifen können. Dazu machen wir den Klassenparameter zu einem unveränderlichen Feld, indem wir das Schlüsselwort `val` voranstellen:

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

Nun können wir auf das Attribut mit der von Java gewohnten Punktnotation zugreifen:

```
scala> angie.name
res0: String = Angela Merkel
```

Bevor wir uns dem Companion Object zuwenden, halten wir kurz inne, um einen Vergleich zu Java zu ziehen: Mit einer Zeile Scala-Code haben wir eine Klasse definiert, die ein lesbares Feld enthält. Das entspricht einer Java-Klasse mit einem privaten Feld, einem Konstruktor, der dieses Feld initialisiert, und einem Getter. Je nachdem, wie wir Codezeilen zählen wollen, eine drastische Einsparung!

Methoden

Kommen wir nun zum Companion Object, das die Methode `apply` implementieren soll:

```
scala> object Person {
  | def apply(name: String) = new Person(name)
  | }
defined module Person
```

Die REPL erkennt, wenn unsere Eingaben trotz Zeilenumbruch noch nicht vollständig sind, und signalisiert das, indem sie einen vertikalen Strich voranstellt. Dieser ist eine Besonderheit der REPL und nicht Bestandteil des Scala-Codes.

In Scala definieren wir Methoden mit dem Schlüsselwort `def`, gefolgt vom Namen der Methode und der Parameterliste. Anschließend könnten wir optional nach einem Doppelpunkt den Rückgabetyp angeben, aber hier setzen wir wieder auf Typinferenz. Die Implementierung der Methode wird durch

das Gleichheitszeichen eingeleitet. Für Einzeiler können die geschweiften Klammern entfallen. Als Rückgabewert wird in Scala immer der letzte Ausdruck verwendet, in unserem Fall also eine neue Instanz von `Person`.

Nun können wir neue Personen auch ohne das Schlüsselwort `new` erzeugen, weil die Methode `apply` auf dem Companion Object aufgerufen wird:

```
scala> val angie = Person("Angela Merkel")
angie: Person = Person@7a15b555
```

Damit unser kleines Beispiel so richtig rund ist, wollen wir noch die `toString`-Methode überschreiben, sodass der Name ausgegeben wird:

```
scala> class Person(val name: String) {
  | override def toString = name
  | }
defined class Person
```

Um Methoden zu überschreiben, müssen wir das Schlüsselwort `override` verwenden. Da `toString` keine Parameter hat, lassen wir gleich die komplette Parameterliste, d. h. auch die Klammern, weg. Nachdem wir unser Companion Object nochmals neu definiert haben, gibt die REPL beim Anlegen einer neuen Person deren Namen aus:

```
scala> val angie = Person("Angela Merkel")
angie: Person = Angela Merkel
```

Operatoren

Da in Scala alles ein Objekt ist, gibt es natürlich keine Operatoren, sondern nur Methoden. Allerdings gibt es viel weniger Einschränkungen als in Java, welche Zeichen verwendet werden dürfen. Deshalb können wir für die Methodennamen auch „Operator"-Zeichen verwenden. Zum Beispiel ist `+` eine Methode der Klasse `Int`:

```
scala> x.+(1)
res0: Int = 2
```

Das sieht natürlich komisch aus, wenn die Addition als „klassischer" Methodenaufruf geschrieben wird. Daher erlaubt Scala auch die sogenannte Infix-Operator-Notation, bei der Punkt und Klammern entfallen:

```
scala> x + 1
res1: Int = 2
```

Bitte beachten Sie, dass `x.+(y)` und `x + y` nur unterschiedliche Schreibweisen für den Aufruf der Methode `+` auf dem Objekt `x` mit dem Parameter `y` sind. Die Infix-Operator-Notation funktioniert natürlich nicht nur für `Int`s, sondern kann ganz allgemein immer dann verwendet werden, wenn eine Methode genau einen Parameter erwartet.

Warum auf Java Closures warten?

Damit beenden wir unseren Ausflug zu den OO-Eigenschaften von Scala und wenden uns der funktionalen Programmierung (FP) zu. Ohne hier zu tief einzusteigen, wollen wir ganz kurz die wichtigsten FP-Grundlagen erläutern:

Zunächst einmal besteht ein FP-Programm aus Funktionen, die Aufrufparameter und ein Resultat haben, also „so etwas" wie Methoden sind. Nach der reinen Lehre sind Funktionen frei von Seiteneffekten, also unabhängig vom Programmzustand, sodass wiederholte Aufrufe immer dasselbe Resultat ergeben. Darüber hinaus gibt es nur unveränderliche Werte. Dadurch lassen sich FP-Programme sehr gut testen, ja wir können sogar deren Korrektheit beweisen.

Soweit könnten wir prinzipiell auch mit Java funktional programmieren. Jedoch kennt die funktionale Programmierung sogenannte Funktionen höherer Ordnung, d. h. Funktionen, denen andere Funktionen als Aufrufparameter übergeben werden können. Auf diese Weise können FP-Programme durch Komposition von Funktionen erstellt werden. Spätestens hier enden die Java-Möglichkeiten, wenngleich Closures für Java 7 in der Planung sind.

So viel zur Theorie. Was bedeutet das für Scala in der Praxis? Betrachten wir dazu als Beispiel einer Liste mit ungeraden Zahlen:

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

Wie machen wir daraus gerade Zahlen? Indem wir zu jedem Element eins addieren:

```
scala> val evens = odds map { x => x + 1 }
evens: List[Int] = List(2, 4, 6)
```

Zunächst weisen wir auf die Verwendung der Infix-Operator-Notation hin, d. h. wir könnten auch `odds.map(x => x + 1)` schreiben. Weiter fällt auf, dass wir geschweifte Klammern statt runder verwenden dürfen, um zu verdeutlichen, dass das Argument kein „normaler" Wert, sondern ein Funktionsliteral ist.

Wir sehen hier also den Aufruf der Methode `map` auf einem `List`-Objekt. Diese Methode erwartet einen Parameter vom Typ `Function1`, d. h. eine Funktion mit einem Parameter. Und der Ausdruck `x => x + 1` stellt einen Wert für so eine Funktion dar. Links vom Pfeil steht die Parameterliste, hier nur ein Parameter mit dem Namen x. Dank Typinferenz verzichten wir wieder auf die Angabe des Typs, aber natürlich könnten wir auch `x: Int => x + 1` schreiben. Rechts vom Pfeil steht die Implementierung der Funktion, hier nur ein einfacher Einzeiler.

Wenn in Scala alles ein Objekt ist, dann müssten doch auch Funktionen Objekte sein, oder? Richtig! Wir können eine Funktion definieren und diese einer Variable zuweisen:

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

Die so definierte Funktion vom Typ `Function2` können wir aufrufen, indem wir in Klammern die beiden Aufrufparameter hinter den Variablennamen schreiben. Auch hier wird im Hintergrund der Compiler die Methode `apply` aufrufen, die von jedem Funktionstyp implementiert wird:

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

OK, weitere FP-Details werden in den kommenden Teilen folgen. Hier wollen wir abschließend den konzeptionellen Unterschied zu Java betrachten: Wir haben durch das Funktionsliteral ausgedrückt, was wir tun wollen. In Java hätten wir auf imperative Weise angegeben, wie es getan werden soll, nämlich durch eine Iteration.

Das Wie ist in den meisten Fällen jedoch ein „low-level" Detail, um das wir uns nicht kümmern möchten. Mit funktionaler Programmierung erreichen wir eine höhere Abstraktionsebene und somit gesteigerte Produktivität. Und das gilt nicht nur für das Programmieren, sondern auch für das Lesen und Verstehen von Code.

Fazit

In diesem ersten Überblick über Scala haben wir gesehen, dass Scala im Vergleich zu Java einfacher und zugleich mächtiger

ist. Zunächst fallen die leichtgewichtige Syntax auf und die Möglichkeiten, Code einzusparen. Dann gibt es keine „Ausnahmen“ wie primitive Datentypen oder Operatoren, sondern Scala ist Objektorientierung pur. Und schließlich ermöglicht die funktionale Programmierung höhere Abstraktionen und kompakteren Code. In den kommenden Folgen werden wir die begonnenen Themen vertiefen und neue kennenlernen.

Literatur und Links

[New08] T. Neward, The busy Java developer's guide to Scala: Functional programming for the object oriented, `http://www.ibm.com/developerworks/java/library/j-scala01228.html`

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

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

[ScalaRC] Scala Downloads, Version 2.8, `www.scala-lang.org/`

[Stra09] James Strachan's Blog, Scala as a long term replacement for java/javac, 6.7.2009, `macstrac.blogspot.com/2009/04/scala-as-long-term-replacement-for.html`

[WikiPedia] Funktionale Programmierung, `de.wikipedia.org/wiki/Funktionale_Programmierung`

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