



Tiefe statt Breite

Scala – Teil 3: Typen und Pattern Matching

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. Im letzten Teil haben wir Vererbung, Traits und funktionale Programmierung mit Scala unter die Lupe genommen. Hier schließen wir diese Serie mit einem Blick auf das Typsystem sowie weitere wichtige Sprachfeatures ab.

Das Scala-Typsystem

Wegen der leichtgewichtigen Syntax und der Typinferenz mag dem einen oder anderen bereits entfallen sein, dass Scala eine statisch typisierte Sprache ist: Alles besitzt zum Übersetzungszeitpunkt einen Typ. Wenn wir den Typ nicht explizit angeben, dann ermittelt der Compiler diesen für uns:

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

Wenn wir den Typ angeben, dann muss er auch passen:

```
scala> val x: Int = "Fail"
<console>:5: error: type mismatch;
 found   : java.lang.String("Fail")
 required: Int
 val x: Int = "Fail"
```

Was für Typen gibt es nun in Scala? Die wichtigsten finden wir in Abbildung 1. Beginnen wir zunächst einmal „ganz oben“: Dort, an der Wurzel der Vererbungshierarchie, befindet sich die abstrakte Klasse **Any**. Sie definiert eine überschaubare Anzahl an Methoden, unter anderem `equals`, `hashCode` und `toString` sowie `==` als Äquivalent zu `equals` und `!=` als dessen Negation.

Von **Any** leiten sich **AnyVal** und **AnyRef** ab. **AnyVal** ist die finale Basis-Klasse für Wert-Typen wie z. B. **Int**, **Long** und **Unit**. Hier sehen wir noch einmal sehr schön, dass Scala rein objektorientiert ist und keine primitiven Datentypen kennt. **AnyRef** hingegen ist die Basis für alle Referenz-Typen, zu denen sowohl die Typen von Scala als auch die von Java gehören.

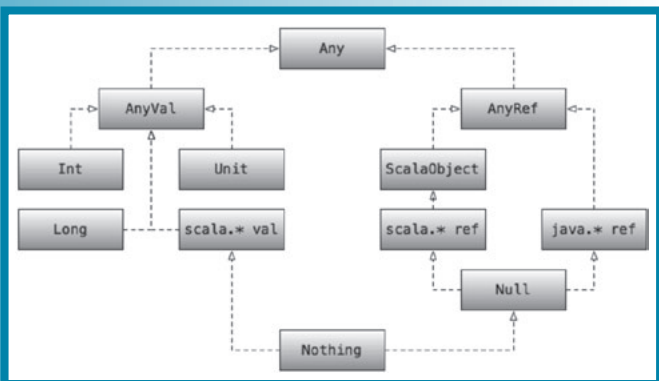


Abb. 1: Scala-Typsystem

Eine Besonderheit und einen Unterschied zu Java stellen die Typen **Null** und **Nothing** dar. Sie stehen in der Vererbungshierarchie „ganz unten“ und sind Subtypen aller Referenz-Typen (**Null**) bzw. aller Typen überhaupt (**Nothing**). **Null** hat genau eine Instanz, nämlich die `null`-Referenz, und **Nothing** kennt überhaupt keine Instanz.

Typparameter

Da die Grundlagen der Java-Generics auf Martin Odersky, den Vater von Scala, zurückgehen, ist es kein Wunder, dass auch Scala Typparameter kennt. Ganz ähnlich wie in Java können wir in Scala einen generischen Typen definieren, indem wir nach dem Namen der Klasse oder des Traits einen oder mehrere Typparameter angeben, jedoch nicht in spitzen, sondern in eckigen Klammern:

```
scala> class Cage[A](val animal: A)
defined class Cage
```

Diese Definition von **Cage** ermöglicht uns, beliebige Typen „einzusperren“. Dabei müssen wir dank Typinferenz den Wert für den Typparameter nicht angeben:

```
scala> new Cage(new Duck("Donald"))
res0: Cage[Duck] = Cage(Donald)

scala> new Cage(new Person("Angela"))
res1: Cage[Person] = Cage(Angela)
```

Unser Beispiel setzt an dieser Stelle voraus, dass wir die Klassen **Animal** und **Person** definiert haben sowie **Duck** als Ableitung von **Animal**.

Natürlich wollen wir unseren Käfig so bauen, dass er nur Tiere aufnehmen kann. Dazu können wir eine Obergrenze für die zulässigen Typen definieren, indem wir den begrenzenden Typen mit vorangestelltem `<`: an den Typparameter anhängen:

```
scala> class Cage[A <: Animal](val animal: A)
defined class Cage

scala> val donaldsCage = new Cage(new Duck("Donald"))
donaldsCage: Cage[Duck] = Cage(Donald)

scala> new Cage(new Person("Angela"))
<console>:10: error: inferred type arguments [Person] do not
conform to method apply's type parameter bounds [A <: Animal]
...

```

Wie wir sehen, können wir unseren **Cage** nun nur noch mit **Animals** parametrisieren, eine **Person** hingegen können wir nicht mehr hinter Gitter bringen.

Wie sieht es eigentlich in Scala mit der Varianz aus, also der Frage, ob aus einer Vererbungsbeziehung der Typparameter eine Vererbungsbeziehung der parametrisierten Typen entsteht? Zur Verdeutlichung folgendes Beispiel: Wir schreiben eine Methode `operationFreedom`, welche einen als Parameter übergebenen **Cage** öffnet und das eingesperrte Tier in die Freiheit entlässt:

```
scala> def operationFreedom(cage: Cage[Animal]) {
|   println(cage.animal + " is free now!")
| }
operationFreedom: (cage: Cage[Animal])Unit
```

Preisfrage: Was passiert nun, wenn wir unseren zuletzt erzeugten `donaldsCage` als Parameter übergeben?

```
scala> operationFreedom(donaldsCage)
<console>:11: error: type mismatch;
 found   : Cage[Duck]
 required: Cage[Animal]
...

```

Oha: Ein `Cage[Duck]` ist offenbar kein `Cage[Animal]`! Obwohl ein `Duck` natürlich ein `Animal` ist. Das heißt, dass in Scala parametrisierte Typen zunächst einmal invariant sind. Das ist in der Regel auch gut so, denn Ko- und Kontravarianz sind knifflige Themen. Zum Beispiel dürfte ein veränderlicher `Cage[Duck]`, d. h. einer mit einem Setter für die Ente, nicht als `Cage[Animal]` interpretiert werden, denn er kann ja nur `Ducks` aufnehmen und keine – sagen wir – Elefanten.

Zum Glück gibt es jedoch in Scala die Möglichkeit, die Varianz zu beeinflussen. In unserem Fall, wo wir einen unveränderlichen `Cage` haben, möchten wir schon, dass ein `Cage[Duck]` ein `Cage[Animal]` ist, d. h. wir wollen Kovarianz für `Cage` erreichen. Dazu stellen wir einfach ein `+` vor den Typparameter:

```
scala> class Cage[+A <: Animal](val animal: A)
defined class Cage
```

Wenn wir nun `donaldsCage` und `operationFreedom` neu erzeugen, dann sind wir am Ziel:

```
scala> operationFreedom(donaldsCage)
Donald is free now!
```

Doch wie geht Scala mit oben beschriebenem Problem um? Ganz einfach: Würden wir unseren `Cage` veränderlich machen, indem wir aus dem `val` ein `var` machen, dann bekommen wir einen Compiler-Fehler:

```
scala> class Cage[+A <: Animal](var animal: A)
<console>:6: error: covariant type A occurs in contravariant position
  in type A of parameter of setter animal_ =
    class Cage[+A <: Animal](var animal: A)
```

Auf diese Weise stellt Scala schon durch das Typsystem und den Compiler sicher, dass wir in Sachen Varianz keine Fehler machen können. Es gäbe noch so einiges über Typparameter zu erzählen, z. B. Kontravarianz und Untergrenzen, aber das würde den Rahmen hier sprengen. Abschließend sei erwähnt, dass Typparameter in Scala zwar durchaus eine komplizierte Sache sind. Aber das gilt nur für diejenigen von uns, die parametrisierte Typen schreiben. Die Nutzung hingegen ist sehr einfach und sicher.

Die Type Erasure umgehen

Ergänzend zum Thema Typparameter wollen wir nun ein wirklich innovatives Scala-Feature vorstellen. Wie in Java werden die Argumente, die im Quellcode für einen Typparameter verwendet werden, vom Compiler entfernt. Dieses Verhalten wird *Type Erasure* genannt und bewirkt, dass zur Laufzeit keine Informationen über die Typparameter vorliegen. Zum Beispiel können wir nicht ohne Weiteres ermitteln, mit welchem Typ eine einfache Methode parametrisiert ist:

```
scala> def create[T] = T.getClass.newInstance
<console>:5: error: not found: value T
def create[T] = T.getClass.newInstance
```

Scala kennt zunächst das Konzept von impliziten Parametern. Wenn wir einer Parameterliste das Schlüsselwort `implicit` voranstellen, dann müssen wir für diese Parameter nicht explizit Argumente übergeben, sofern der Compiler passende Werte findet, die er verwenden kann. Passend bedeutet in diesem Fall nicht nur, dass die Typen passen müssen, sondern die Werte müssen wiederum mit `implicit` definiert worden sein. Auf diese Weise könnte eine Bibliothek zum Beispiel „vernünftige“ Voreinstellungen zur Verfügung stellen, die wir bei Bedarf explizit überschreiben.

Ein ganz spezieller und immer vorhandener impliziter Wert ist das `Manifest`, welches die Information über einen Typparameter über die Methode `erasure` zur Verfügung stellt. Damit können wir obige `create`-Methode folgendermaßen schreiben:

```
scala> def create[T](implicit mf: Manifest[T]) =
  | mf.erasure.newInstance
create: [T](implicit mf: Manifest[T])Any
```

Da wir den impliziten Parameter beim Aufruf gar nicht übergeben müssen, ist die Verwendung dieser Methode denkbar einfach:

```
scala> create[java.util.Date]
res5: Any = Wed May 19 08:03:12 CEST 2010
```

Auf diese Weise können wir also die Type Erasure umgehen und dadurch an manchen Stellen ein einfacheres und besseres API schreiben. In Java hätten wir nämlich für die `create`-Methode einen Parameter vom Typ `Class` benötigt. Anwendungsbeispiele für dieses Feature sind alle Tools, die für uns Instanzen aufgrund von Meta-Informationen erzeugen sollen, z. B. Dependency-Injection-Frameworks oder OR-Mapper.

Case Classes und Pattern Matching

Abschließend gehen wir noch auf ein besonders charakteristisches Scala-Feature ein, das eigentlich aus zwei Teilen besteht. Zunächst einmal können wir durch Voranstellen des Schlüsselwortes `case` eine Klasse zur sogenannten *Case Class* machen:

```
scala> case class Cage[+A <: Animal](animal: A)
defined class Cage
```

Dem aufmerksamen Leser wird auffallen, dass wir nun das Schlüsselwort `val` vor dem Klassenparameter `animal` weglassen. Das liegt daran, dass der Compiler für Case Classes die Klassenparameter automatisch zu `vals` macht. Aber das ist nicht das einzige Geschenk: Zusätzlich bekommen wir „sinnvolle“ Implementierungen von `equals`, `hashCode` und `toString`, allesamt auf Basis der Klassenparameter implementiert. Darüber hinaus bekommen wir auch noch automatisch ein Companion Object, welches die `apply`-Methode implementiert, um eine Case Class ohne das Schlüsselwort `new` zu erzeugen. Das Ganze sieht dann in der Anwendung so aus:

```
scala> val donaldsCage = Cage(new Duck("Donald"))
donaldsCage: Cage[Duck] = Cage(Donald)

scala> donaldsCage.animal
res6: Duck = Donald
```

Wir sehen, dass wir unseren `Cage` nun ohne `new` erzeugen können, die `toString`-Methode den Klassennamen und in Klammern die `toString`-Aufrufe der Klassenparameter zurück gibt und der Zugriff auf `animal` möglich ist, obwohl wir diesen Klassenparameter nicht explizit zum `val` gemacht haben.

Für sich alleine wären Case Classes zwar eine nette Erleichterung für manche Anwendungsfälle, aber so richtig zur Entfaltung kommen sie in Verbindung mit einem weiteren Sprachfeature, dem Pattern Matching (musterbasierter Vergleich).

Zunächst einmal könnte man Pattern Matching als „richtig gemachte“ `switch`-Anweisung betrachten. Die offensichtlichen Unterschiede sind, dass wir beliebige Ausdrücke übergeben dürfen, dass es keinen Fall-Through gibt, d. h. nach einem erfolgreichen Matching ist Schluss, und dass Pattern Matching – ganz im Sinne der funktionalen Programmierung – immer ein Resultat ergibt.



Die Syntax ist denkbar einfach: Wir schreiben einen Ausdruck, gefolgt vom Schlüsselwort `match` und anschließend einen Block mit `case`-Mustern:

```
scala> def whatIsIt(x: Any) = x match {
  | case "Constant" => "It's a string constant."
  | case s: String => "It's the string " + s
  | case Cage(animal) => "It's a cage with a " + animal
  | case _ => "It's something else."
  | }
whatIsIt: (x: Any)java.lang.String

scala> whatIsIt("Hello")
res7: java.lang.String = It's the string Hello

scala> whatIsIt(donaldsCage)
res8: java.lang.String = It's a cage with a Donald
```

Hier ist der Ausdruck der Parameter `x` der Methode `whatIsIt`. Alle `case`-Muster geben einen `String` zurück, sodass der Rückgabotyp der Methode ein `String` ist. Nun zu den einzelnen Mustern:

- ▼ Wir prüfen zunächst mit einem `Constant`-Muster, ob unser Ausdruck eine Konstante darstellt.
- ▼ Danach nutzen wir ein `Variable`-Muster mit Angabe des Typs, um zu prüfen, ob ein `String` vorliegt. Dabei definieren wir die Variable `s`, die wir auf der „rechten Seite“ des `case`-Musters verwenden können.
- ▼ Das dritte `case`-Muster schließlich nutzt unsere `Case Class Cage`. Wir können hier einfach den Konstruktor der `Case Class` mit Variablen für seine Argumente hinschreiben und diese Variablen dann auf der „rechten Seite“ verwenden. Auf diese Weise sparen wir uns eine explizite Prüfung auf den Typ (`instanceOf`) sowie die Dekomposition in die Werte.
- ▼ Abschließend verwenden wir das `Wildcard`-Muster, um sicherzustellen, dass wir alle möglichen Fälle abdecken, denn ansonsten könnte zur Laufzeit ein `MatchError` auftreten.

Der Vollständigkeit halber sei erwähnt, dass `Pattern Matching` nicht nur mit `Case Classes` so elegant funktioniert. Das wäre auch schade, denn wir wollen nicht alles zu `Case Classes` machen und können das bei vorhandenen Klassen, z. B. aus einer Bibliothek, auch gar nicht. Es reicht vollkommen, dass wir einen `Extractor` definieren: Ein `Singleton Object` mit einer `unapply`-Methode, welche die Dekomposition durchführt.

Scala und Java

Zum Abschluss möchten wir noch kurz beleuchten, wie wir Scala in unsere Java-Projekte einbauen können. Das ist natürlich möglich, denn Scala kompiliert zu `Java-Bytecode` und ist sogar hundertprozentig „abwärtskompatibel“ zu Java, sodass wir allen vorhandenen `Java-Code` verwenden können. Der umgekehrte Weg, d. h. Scala aus Java heraus zu nutzen, ist auch möglich, allerdings müssen wir uns dabei beim `API` auf diejenigen `Features` beschränken, die Java kennt.

Daher ist der folgende Weg vermutlich der einfachste: Wir beginnen damit, unsere Tests nicht mehr mit `JUnit` oder `TestNG` zu schreiben, sondern mit `[Specs]` oder `[ScalaTest]`. Das sind zwei `Scala-Frameworks`, die uns ermöglichen, sehr ausdrucksstarke Testfälle zu schreiben, bei denen tatsächlich gilt, dass der Code der Kommentar ist. Dabei sammeln wir wertvolle Praxiserfahrung mit Scala und können nach einer Weile dann dazu übergehen, einzelne Teilprojekte in Scala zu schreiben.

Dabei wäre es geschickt, wenn wir solche Teilprojekte identifizieren könnten, auf welche andere Teilprojekten keine Abhängigkeit haben. Oder mit anderen Worten: Wenn unser Projekt eine Zwiebel ist, bei der die äußeren Schalen von den inneren abhängen, aber nicht umgekehrt, dann sollten wir bei den ganz

äußeren Schalen beginnen. So können wir nach und nach immer mehr Scala einführen, ohne allzu große Risiken einzugehen und ohne mit einem Schlag alles für alle Projektbeteiligten umstellen zu müssen.

Fazit

Es gäbe noch einiges mehr über Scala zu berichten, insbesondere über `Standardbibliotheken` wie z. B. die `Actors-Bibliothek`. Aber was die Sprache selbst angeht, so haben wir im Rahmen dieser dreiteiligen Serie die meisten Aspekte behandelt. Das war möglich, weil Scala keine „breite“ Sprache ist, sondern eine „tiefe“: Die Sprache selbst ist recht schlank, wengleich manche `Features` durchaus einen gewissen Tiefgang haben, z. B. das `Typsystem`.

Dennoch ist es nicht erforderlich, gleich ganz abzutauchen und alles im Detail zu verstehen. Vielmehr ist es sogar dank der Schlankheit recht schnell möglich, Schwimmen zu lernen, d. h. die Sprache so zu beherrschen, dass man schon etliche Vorteile im Vergleich zu Java nutzen kann. Man denke nur an die wesentlich kompaktere Notation und die Mächtigkeit von `functional Collections`.

Wir hoffen, mit dieser Serie nicht nur Wissen über Scala vermittelt, sondern hoffentlich auch Lust auf Scala gemacht zu haben und freuen uns sehr auf Fragen oder Feedback.

Literatur und Links

- [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/>
- [ScalaTest]** Open-Source-Test-Framework, <http://www.scalatest.org/>
- [SeeBla10a]** H. Seeberger, J. Blankenhorn, *Scala: – Teil 1: Einstieg*, in: *JavaSPEKTRUM*, 2/2010
- [SeeBla10b]** H. Seeberger, J. Blankenhorn, *Scala: – Teil 2: FP und OOP*, in: *JavaSPEKTRUM*, 3/2010
- [Specs]** A BDD Library for Scala, Project Hosting on Google Code, <http://code.google.com/p/specs/>



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