



14 coole Scala-Features (Teil 2 von 3)

📅 28. September 2016 (<http://sebinside.de/2016/09/28/14-coole-scala-features-teil-2-von-3/>) 👤 seb (<http://sebinside.de/author/seb/>) 📁 Coding (<http://sebinside.de/category/coding/>), Studium (<http://sebinside.de/category/studium/>)

Scala ist meine aktuelle Lieblingsprogrammiersprache. Ich liebe es, super effizienten High-Level-Code zu schreiben und Scala als Multiparadigmen-Sprache bietet hierfür alles, was man sich wünschen kann – und Java-Kompatibilität noch kostenlos dazu. Ich habe mal die coolsten Features gesammelt und stelle sie in dieser Reihe vor!

Herzlich willkommen zurück zum zweiten Teil der Scala-Feature-Reihe! Im **ersten Teil** habe ich bereits in die Scala Syntax eingeführt und Features wie die Erweiterbarkeit und Typinferenz vorgestellt. Außerdem habe ich das Thema Objektorientierung weitestgehend abgehandelt.

In diesem Teil starten wir erstmal erneut mit **funktionaler Programmierung** bzw. mit den Vorteilen, die sich daraus in Scala ergeben. Vor allem steht dieser Post aber unter dem Zeichen der **Kontrollstrukturen**: For-Schleifen in Scala und **Pattern Matching** sind extrem cool. Außerdem ist es sogar möglich eigene Kontrollstrukturen zu definieren und durch implizite Konvertierung Funktionalität von anderen APIs zu erweitern. Los geht's!

6. Der funktionale Ansatz

Funktionale Programmierung in Scala habe ich bereits in einem anderen Blog-Post (<http://sebinside.de/2016/08/26/von-imperativ-zu-funktional-ein-scala-beispiel/>) beschrieben. Durch den richtigen Einsatz von **High-Level-Funktionen** ist es so möglich Code wesentlich kompakter zu schreiben, bei identischer Funktionalität und eventuell sogar besserer Performance! Hier nochmal das Code-Beispiel zur Erinnerung:

```
1 // Imperative
2 def printArgsImperative(args: Array[String]): Unit = {
3     var i = 0
4     while (i < args.length) {
5         println(args(i))
6         i+1
7     }
8 }
9
10 // Functional
11 def printArgsFunctional(args: Array[String]): Unit = args.foreach(println)
```

Foreach ist aber nicht die einzige High-Level-Funktion die in Verbindung mit funktionaler Programmierung einiges erleichtern kann. Viele zum Beispiel auch in Haskell vorhandene Funktionen, die mit Version 8 auch teilweise den Einzug in Java erhalten haben, existieren in Scala ebenfalls: **Filter**, **Map**, **Reduce**, **Zip**, **ForAll**, **TakeWhile**, **DropWhile**, uvm.

Hier ein kurzes Beispiel mit dem verbreiteten **Filter-Map-Reduce-Verfahren**. Zunächst werden im Filter alle Zahlen größer als 10 entfernt, dann zu allen verbleibenden Zahlen um 1 erhöht und zum Schluss alle Zahlen aufaddiert. Auch wenn der Sinn dieses Beispiels eine andere Frage ist, überlegt euch mal, wie viele Zeilen man hierfür mit imperativen Code benötigt hätte... und ob das noch übersichtlich gewesen wäre.

```
1 // myList ist eine Liste mit Werten vom Typ Int
2 println(myList.filter(_ < 10).map(_ + 1).sum)
```

Ebenfalls interessant ist der Umgang mit **immutable** und mutable Objects in Scala (übersetzt in etwa: veränderbar). Wem das grade überhaupt nichts sagt: Ein String ist z.B. immutable. Wenn ich in Java oder Scala auf einem String Funktionen wie substring oder replace aufrufe, verändert das nie den String selbst, sondern liefert

ein neues Objekt mit den Änderungen zurück. Das Gegenstück hierzu ist der `StringBuilder`: Hier wird nicht für jede Änderung ein neues Objekt erzeugt, was eine bessere Laufzeit zur Folge haben kann.

Unveränderbare Objekte können aber durchaus Vorteile haben, da ihr Verhalten grade in **parallelen Systemen** einfacher vorhersagbar ist. Aus diesem Grund gibt es in Scala von fast allem eine mutable und eine immutable Version: `List`, `Set`, `String` usw. Was man davon nun nutzt, muss man für sich selbst entscheiden. Ich hatte auch einige Bedenken bei unveränderbaren Objekten, aber sogar Twitter empfiehlt in den offiziellen Best Practices (<http://twitter.github.io/effectivescala/#Collections>) deren Nutzung.

7. Die For-Kontrollstruktur und List-Comprehension

In seiner einfachsten Form erinnert Scalas For-Kontrollstruktur an For-Each aus Java oder Python:

```
1 for(myListElement <- myList) { ... }
```

Aber in Wirklich ist diese Kontrollstruktur erneut an Haskell, dieses Mal an der **List Comprehension** orientiert. So kann man über Zahlenräume iterieren, Bedingungen durch Guards überprüfen und die Ausführung verschachteln. Ich habe die Möglichkeiten im folgenden Code-Beispiel mal kurz zusammengefasst:

```
1 // Gibt die Zahlen 1 bis 10 aus
2 for (i <- 1 to 10) { println(i) }
3
4 // Gibt die Zahlen 1 bis 9 in 2er-Schritten aus
5 for (i <- 1 until 10 by 2) { println(i) }
6
7 // Gibt das kleine 1x1 aus
8 for(i <- 1 to 10; j <- 1 to 10; x = i * j) { println(x) }
9
10 // Gibt nur gerade Produkte aus dem 1x1 aus
11 for (i <- 1 to 10; j <- 1 to 10; x = i * j if x % 2 == 0) { println(x) }
```

Außerdem kann man aus diesen List Comprehensions auch gleich neue **Listen** (bzw. Sequenzen) **erzeugen**! Das geht entweder im einfachen Fall mit Zahlen (bzw. Ranges) oder auch mit allen anderen Objekten. Das Zauberwort heißt **yield**. Kürzer geht echt

nicht mehr!

```
1 // Speichert das kleine 1x1
2 val my1x1 = for (i <- 1 to 10; j <- 1 to 10; x = i * j) yield x
```

8. Pattern Matching

Wir bleiben bei Scala-Features, die Haskell ähneln. In seiner einfachsten Form funktioniert Pattern Matching ähnlich wie **switch in Java**: Statt mehrere ifs zu verschachteln, werden für eine Variable verschiedene Zustände und Reaktionen darauf definiert. Nur kann match etwas mehr erkennen als switch. Viel mehr.

Hundert Mal mehr! Aber beginnen wir mit einem einfachen Java-switch-Beispiel:

```
1 // Java (i ist ein Integer)
2 switch (i) {
3     case 1:
4         // ...
5         break;
6     case 2:
7         // ...
8     default:
9         // ...
10        break;
11 }
```

Entweder ist die Variable 1, 2 oder der Default-Fall wird aufgerufen. Oh, und ich habe bei Fall 2 den Break vergessen. Das heißt, im Falle von $i=2$ wird automatisch zusätzlich der Default-Zweig ausgeführt. **Blöd gelaufen...** In Scala sieht das auf jeden Fall so aus:

```
1 i match {
2     case 1 => ...
3     case 2 => ...
4     case _ => ...
5 }
```

Okay, sieht noch relativ ähnlich aus. Break wird hier weggelassen, der Funktionsaufruf an sich nicht mit einem Doppelpunkt sondern mit einem Lambda-ähnlichen Pfeil eingeleitet. Und statt default wird ein Unterstrich verwendet. Die Macht von match liegt allerdings in der **Erkennungsvielfalt**:

```
1 i match {  
2   case 1 => ...  
3   case Math.PI => ...  
4   case i: String => ...  
5   case List(0, 1, _*) => ...  
6   case head :: second :: tail => ...  
7   case (a, _) => ...  
8   case i: Int if i % 2 == 0 => ...  
9 }
```

Gehen wir der Reihe nach durch:

- In der **ersten Zeile** wird untersucht, ob i den Wert 1 hat.
- In der **zweiten Zeile** wird i mit Pi, einer Konstanten verglichen.
- In der **dritten Zeile** ist der Wert egal, hier wird nur der Typ untersucht.
- In der **vierten Zeile** wird untersucht, ob i eine Liste mit beliebiger Länge ist, die mit den Werten 0 und 1 beginnt.
- In der **fünften Zeile** wird ebenfalls geguckt, ob es sich um eine Liste handelt – hier werden die ersten beiden Elemente extrahiert um sie später noch zu verwenden.
- In der **zweitletzten Zeile** wird versucht die Variable mit einem Tupel zu vergleichen. Schon durch die Schreibweise wird hier angegeben, dass uns im Falle eines Matches nur der erste Wert interessiert (der zweite wird durch den Unterstrich verworfen).
- In der **letzten Zeile** wird zunächst der Typ von i untersucht, dann noch ihr Wert durch ein zusätzliches if getestet.

Wie man in diesem künstlichen Beispiel schon sehen kann: Ich habe nicht zu viel versprochen. Match kann auf **sehr vieles untersuchen** und ist deswegen überall in Scala wiederzufinden: In **Catch-Blöcken** werden Exceptions mit match untersucht, statt null-Werten kann **Option** verwendet und mit match untersucht werden, ganze XML-Dateien können **mit match gelesen** werden (Siehe Teil 3 der Reihe), und noch so, so, so vieles mehr.

Abschließen möchte ich noch mit einem kleinen Praxis-Beispiel. Hier wird ein String mit Hilfe von **Regex im Match-Block** auf seine syntaktische Gestalt untersucht und anschließend anhand der Gruppen zerlegt und die Inhalte extrahiert. Man stelle sich das jetzt noch mit if-Guards vor. Jedem, der sich schon mal mit schon mal mit der Auswertung von größeren regulären Ausdrücken rumschlagen musste, dürfte gerade das Herz aufgehen! 😊

```
1 val myPattern = "([ab]+)([c]+)d".r
2 "aabbcccd" match {
3   case myPattern(abs,cs) => println(abs + ", " + cs)
4 }
5 // Output: aabb, cccc
```

9. Eigene Kontrollstrukturen

Kann es noch besser werden als List-Comprehension und Pattern-Matching? Klar, wie wär's denn mit **selbst definierten Kontrollstrukturen**! Schon in Teil 1 habe ich gezeigt, wie in Scala fast alles überschrieben werden kann und was mit der Syntax alles möglich ist (Stichwort: Methoden als Operatoren).

Eigentlich ist es naheliegend, dass sich auch eigene Kontrollstrukturen erzeugen lassen. Der Grund dafür ist einfach: Durch eigene Kontrollstrukturen lassen sich **Redundanzen** im Code noch besser vermeiden, das Endprodukt ist noch erweiterbarer und einfacher zu verstehen.

Die Grundlage hierfür ist zunächst wieder ein **syntaktischer Trick**. Funktionen mit nur einem Parameter lassen sich nicht nur mit normalen Klammern aufrufen, sondern auch mit geschweiften Klammern. Sieht schon ein bisschen nach Kontrollstruktur aus.

```
1 println {
2   „Hello World“
3 }
```

Der wesentliche Unterschied zu einer Kontrollstruktur wie if oder while ist hier allerdings die Reihenfolge der **Anweisungsauswertung**. Bei println() handelt es sich um eine herkömmliche Methode, was bedeutet, dass beim Aufruf erst alle Parameter ausgewertet werden. Beim Aufruf von println(5 + 3) wird zunächst 5 + 3 = 8 ausgewertet und anschließend println(8) aufgerufen. Man nennt dieses Verhalten **Call-By-Value** und es ist die normale Vorgehensweise in imperativen Sprachen.

Stellen wir uns jetzt mal den folgenden Code als Call-By-Value vor:

```
1 if (someString != null) {  
2   someString.length()  
3 }
```

Call-By-Value in diesem Fall bedeutet: Sowohl die Bedingung im Kopf als auch die Methode im Körper werden ausgewertet. Und was ist, wenn someString null ist? Genau, **Programmabsturz**.

Um echtes „Kontrollstruktur-Verhalten“ zu erhalten, benötigen wir also öfters mal mehr als Call-By-Value. Quasi ein Aufruf, der den Code „*weitergibt*“ und die Auswertung so weit *nach hinten verschiebt*, wie nur irgend möglich. Die Rede ist von **Call-By-Name**. Versuchen wir es mal mit einem Praxis-Beispiel: Der folgende Code ist eine Kurzschreibweise der berühmten while(true)-Schleife!

```
1 def loop(x: => Unit): Unit = {  
2   while (true)  
3     x  
4 }
```

Das war's schon. Dieser Code lässt sich als **Kontrollstruktur** verwenden. Eine etwas eigenartige Schreibweise ist beim Parameter zu beobachten: x ist eine Funktion von etwas nicht definiertem zum Typ Unit, also etwas nicht Definiertes ohne Rückgabeparameter. Genau dieser Syntax erzeugt den **Call-By-Name-Aufruf**, der es ermöglicht, dass der Inhalt erst in der while(true)-Schleife ausgewertet wird. Der Aufruf ist dann sehr gemütlich (Die Ausgabe in diesem Fall: „Hello World“. Und zwar unendlich oft):

```
1 loop {  
2   println(„Hello World“)  
3 }
```

Damit geht aber noch mehr. In der Einleitung dieses Kapitels hatte ich erwähnt, dass die Geschweifte-Klammer-Syntax für Methoden nur funktioniert, wenn diese nur einen Parameter haben. Im gerade eben gezeigten Loop-Beispiel wäre das der Code, der immer wieder ausgeführt werden soll und deswegen by-Name übergeben wird.

Wie definiere ich jetzt aber eine Kontrollstruktur, die Code nur eine bestimmte Anzahl male wiederholt? Der Aufruf könnte wie folgt aussehen:

```
1 repeat(3) {  
2   println(„Hello World 3 times“)  
3 }
```

Auffällig sollte hier auf jeden Fall der zusätzliche Parameter 3 sein. Der Trick an dieser Stelle heißt **Currying**; und hat nichts mit Fastfood zu tun, der Name stammt von Haskell Brooks Curry, einem US-amerikanischen Mathematiker, der Namensgeber für die Sprache Haskell war und viel Grundlagenarbeit im Bereich von funktionalen Sprachen geleistet hat.

Ohne die Technik im Detail erklären zu wollen: Es ist möglich eine Funktion so zu definieren, dass sie auch weniger als die angegebenen Parameter annimmt. Wird sie mit zu wenig Parametern aufgerufen, liefert sie dann nicht das gewünschte Ergebnis, sondern *eine weitere Funktion*, die erst auf die restlichen Parameter wartet. Dieses Verhalten heißt Currying und ermöglicht noch viel mehr, als nur ein paar Kontrollstrukturen zu bauen.

Der konkrete Code der Kontrollstruktur sieht dann so aus. Man sieht sowohl den Call-By-Name-Teil als auch die durch Klammern getrennten Parameter des Curryings.

```
1 def repeat(times: Int)(x: => Unit): Unit = {  
2   for (i <- 0 until times)  
3     x  
4 }
```

Eigene Kontrollstrukturen zu erstellen ist wirklich cool, insbesondere durch die Möglichkeiten von Call-by-Name. Allerdings ist das natürlich **kein harmloses Feature**, denn genau wie auch bei „normalen“ Methoden ist es hier sehr einfach durch missverständliche Definition viel kaputt zu machen.

Dennoch: Ein weiterer Beweis dafür, dass einen die Sprache Scala echt **einiges** im Code hinschreiben lässt!

10. Implizite Konvertierung

Das Finale des zweiten Teils meiner Scala-Reihe. Das letzte wie auch dieses Mal hatten wir schon viele verrückte Features, die es ermöglichen, sehr kompakten und einfach lesbaren und trotzdem hinreichend performanten Code zu schreiben.

Implizite Konvertierung treibt das ganz auf die Spitze. Und mit steigenden Möglichkeiten, steigt eben auch das Risiko was kaputt zu machen. Deswegen: **Vorsicht!**

Bei der Programmierung kann man Quellcode grob in zwei Kategorien einteilen: Code, der von euch selbst stammt und den ihr deswegen nach Belieben erweitern und an eure Bedürfnisse anpassen könnt und **Code aus fremden Bibliotheken**, der nur über eine zuvor definierte Schnittstelle ansprechbar ist. Insbesondere gehört hierzu natürlich auch die Scala bzw. Java Bibliothek.

Wenn ich jetzt eine Methode benötige, die mit Elementen einer fremden Bibliothek arbeitet (z.B. einen String verarbeitet), dann bleibt mir nichts anderes übrig, als dieses Element als Parameter anzunehmen. Egal, wie viel Sinn diese Methode auch im fremden Code ergeben würde, ich kann ja nicht einfach deren Code umschreiben ohne dass Chaos vorprogrammiert wäre. Und dennoch ist es möglich, es so aussehen zu lassen, als könnte ich: Mit **implizierter Konvertierung**.

Starten wir mit einem Anwendungsfall: Ich möchte die Klasse String um eine Methode erweitern, die einzelne Zeichen durch Leerzeichen getrennt ausgibt. Aufgerufen werden soll diese so:

```
1 "abcd".printWithSpaces() // Ausgabe: a b c d
```

Jetzt hat die Klasse String natürlich keine solche Methode. Also definiere ich mir kurzerhand meinen eigenen String-**Wrapper**:

```
1 class MyRichString(s: String) {  
2   def printWithSpaces(): Unit = println(s.mkString(" "))  
3 }
```

Natürlich etwas unschön, aber immerhin kann ich jetzt den String richtig verarbeiten:

```
1 new MyRichString("abcd").printWithSpaces()
```

Aber immer explizit den Konstruktor mitschleppen? Das ist **zu umständlich**. Und wahrscheinlich haben es einige schon gerochen: Genau hier kommt implizite Konvertierung ins Spiel! Ich definiere zusätzlich zu meiner eigenen Wrapper-Klasse noch eine **implizite Konvertierungs-Funktion** für die Umwandlung von String zu MyRichString:

```
1 implicit def string2myRichString(s: String): MyRichString = new MyRichString(s)  
2  
3 // Und somit:  
4 "abcd".printWithSpaces() // Ausgabe: a b c d
```

Und schon ist der fertige Aufruf dieses Kapitels möglich. Ein Aufruf einer Methode auf einem Objekt, das in dessen Klasse eigentlich gar nicht definiert wurde. ***Doch was passiert hier in Wirklichkeit?***

Bei der **Kompilierung** prüft Scala, ob die Klasse String eine Methode `printWithSpaces()` besitzt. Ist dies der Fall, ist die Sache klar. In unserem Beispiel kommt die Methode aber in einer anderen Klasse vor, der Klasse `MyRichString`. Das entdeckt der Compiler auch, als er nach der entsprechenden Funktion sucht. Nun geht er den **Index aller implizierten Konvertierungen** durch und sucht nach einer Funktion mit der **korrekten Signatur** (in diesem Beispiel: Von String nach `MyRichString`). Dann wird der Aufruf im Code einfach durch die Anwendung dieser Funktion erweitert.

Klingt kompliziert, und ist tatsächlich auch mit einigen Einschränkungen verbunden. So muss die Definition der implizierten Konvertierung mit dem Schlüsselwort **implicit** markiert und beim potenziellen Aufruf **sichtbar** sein, also im Scope liegen. Ebenfalls ist die implizierte Konvertierung eines Wertes immer der letzte Ausweg des Compilers und dieser probiert auch gar nicht erst, mehr als eine Konvertierung gleichzeitig anzuwenden. Die **Komplexität** einer solchen Aufgabe könnte sehr schnell erheblich wachsen.

Die Scala-Bibliothek selbst nutzt implizierte Konvertierung tatsächlich überall. Habt ihr euch schon gewundert, warum ein String in Scala **viel mehr Funktionen** hat als der in Java und trotzdem zu diesem kompatibel ist? Natürlich, implizierte Konvertierung.

Oder was passiert, wenn man in anderen Sprachen einer Double-Variablen einen Integer-Wert zuweist? Richtig, das Typsystem erkennt, dass das von der Container-Größe her funktioniert und wandelt den Wert einfach um. ***In Scala kommt das Typsystem hier gar nicht erst zum Einsatz***, stattdessen implizierte Konvertierung.

Und es gibt noch ein Beispiel, und dieses hat tatsächlich den Vogel abgeschossen. Als ich hiervon zum ersten Mal gelesen habe, musste ich erstmal nur den Kopf schütteln. Es ist das Parade-Beispiel für implizierte Konvertierung. Beim Erstellen einer `HashMap` kommt gerne mal der **Pfeil-Syntax** zum Einsatz, um aus Schlüssel und Wert automatisch ein Tupel zu machen. Ich rede von so etwas:

```
1 val countriesAndCapitals = Map("Germany" -> "Berlin", "France" -> "Paris")
```

Super coole Kurzschreibweise, oder? Da hat Scala aber eine tolle Funktion in den Syntax integriert. Und dann funktioniert das auch noch automatisch mit jedem Typ. Toll. Von wegen: Das ist implizierte Konvertierung. Der Code aus der Predef-Library sieht in etwa so aus:

```
1 class ArrowAssoc[A](x: A) {  
2   def -> [B](y: B): Tuple2[A,B] = Tuple2(x,y)  
3 }  
4  
5 implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)
```

Nichts ist in Scala das, wonach es aussieht. Gewöhnt euch dran 😊

Fazit von Teil 2

Alles klar – oder total verwirrt? Ich denke es ist keine Schande, wenn du dir einige Absätze oder Kapitel mehrfach durchlesen musstest. Ich meine, ich habe einige Kapitel in der **Scala-Bibel** (<http://amzn.to/2cJrjPI>) bis zu sieben Mal durchgelesen um das hier zusammenfassen zu können; und die haben ein kleines bisschen mehr Erfahrung im Schreiben als ich.

Dennoch hoffe ich, dass ich dich erneut von den vielen **Vorzügen und Möglichkeiten** der Sprache Scala überzeugen konnte – oder dich zumindest völlig erschlagen habe. Wir haben dieses Mal einige coole Erweiterungen der Sprache wie **List-Comprehension** und **Pattern Matching** besprochen, und ich habe fortgeschrittene Features wie **eigene Kontrollstrukturen** und **implizite Konvertierung** vorgestellt. Solltest du in der Zwischenzeit Interesse an der Sprache gewonnen haben, sei mit diesen Techniken auf jeden Fall vorsichtig!

Im letzten Kapitel werden wir die Sprache an sich hinter uns lassen und stattdessen einige coole Teile der **Scala-Bibliothek** anschauen. Konkret handelt Teil 3 vom **nativen XML-Support** von Scala, der vereinfachten Nebenläufigkeit durch **Actors** und der Möglichkeit, eigene Compiler in Scala mit Hilfe von **Combinator Parnern** zu schreiben. Neben so vielen Vorteilen möchte ich aber ganz zum Schluss auch noch auf die Nachteile und Risiken der Sprache eingehen.

Bis zu diesem Punkt habe ich übrigens an dieser Reihe schon ***mehr als 10 Stunden*** gearbeitet. Ich hoffe der Aufwand lohnt sich. Wir sehen uns in Teil 3!

Teilen mit:

 E-Mail (<http://sebinside.de/2016/09/28/14-coole-scala-features-teil-2-von-3/?share=email&nb=1>)

 Facebook 1 (<http://sebinside.de/2016/09/28/14-coole-scala-features-teil-2-von-3/?share=facebook&nb=1>)

 Twitter (<http://sebinside.de/2016/09/28/14-coole-scala-features-teil-2-von-3/?share=twitter&nb=1>)

Gefällt mir:

Lade ...

[coding](http://sebinside.de/tag/coding/) (<http://sebinside.de/tag/coding/>)

[funktionale programmierung](http://sebinside.de/tag/funktionale-programmierung/) (<http://sebinside.de/tag/funktionale-programmierung/>)

[haskell](http://sebinside.de/tag/haskell/) (<http://sebinside.de/tag/haskell/>)

[imperative programmierung](http://sebinside.de/tag/imperative-programmierung/) (<http://sebinside.de/tag/imperative-programmierung/>)

[java](http://sebinside.de/tag/java/) (<http://sebinside.de/tag/java/>) [Programmieren](http://sebinside.de/tag/programmieren/) (<http://sebinside.de/tag/programmieren/>)

[Programmierparadigmen](http://sebinside.de/tag/programmierparadigmen/) (<http://sebinside.de/tag/programmierparadigmen/>)

[Programmiersprachen](http://sebinside.de/tag/programmiersprachen/) (<http://sebinside.de/tag/programmiersprachen/>)

[programmierung](http://sebinside.de/tag/programmierung/) (<http://sebinside.de/tag/programmierung/>) [scala](http://sebinside.de/tag/scala/) (<http://sebinside.de/tag/scala/>)

KOMMENTAR VERFASSEN

Gib hier deinen Kommentar ein ...

◀ [14 coole Scala-Features \(Teil 1 von 3\)](http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/) (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/>)

[14 coole Scala-Features \(Teil 3 von 3\)](http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/) ▶ (<http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/>)

SOCIAL MEDIA



(http://twitter.com/sebinside) (http://facebook.com/sebinside) (http://youtube.com/sebinside) (http://github.com/sebinside)

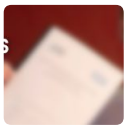
Wittmann, Sebastian (http://www.wittmann-sebastian.de)

er.cookbooks.beautifulmatters.com/skateboard/e70insider/2017/04/13/rueckblick-auf-den-bachelor/

BLOG VIA E-MAIL ABONNIEREN

Gib Deine E-Mail-Adresse an, um diesen Blog zu abonnieren und Benachrichtigungen über neue Beiträge via E-Mail zu erhalten.

BELIEBTE BEITRÄGE



Rückblick auf den Bachelor (<http://sebinside.de/2017/04/13/rueckblick-auf-den-bachelor/>)

13 Apr, 2017



Programmierparadigmen (<http://sebinside.de/2016/07/02/programmierparadigmen/>)

02 Jul, 2016



Webseiten kommen und gehen (<http://sebinside.de/2016/06/15/webseiten-kommen-und-gehen/>)

15 Jun, 2016

KATEGORIEN

📁 Allgemein (<http://sebinside.de/category/allgemein/>)

📁 Coding (<http://sebinside.de/category/coding/>)

📁 Minecraft (<http://sebinside.de/category/minecraft/>)

📁 Studium (<http://sebinside.de/category/studium/>)



NEUESTE KOMMENTARE

💬 Bowser bei Hallo. äh.. Welt! (<http://sebinside.de/2016/03/22/hallo-ae-h-welt/#comment-138>)

💬 daeaevaeaed bei Rückblick auf den Bachelor (<http://sebinside.de/2017/04/13/rueckblick-auf-den-bachelor/#comment-137>)

💬 coernerbrot bei Programmierparadigmen
(<http://sebinside.de/2016/07/02/programmierparadigmen/#comment-136>)

💬 seb (<http://sebinside.de>) bei Eine neue Plugin-Architektur für Code Overflow
(<http://sebinside.de/2018/01/16/eine-neue-plugin-architektur-fuer-code-overflow/#comment-135>)

💬 Sven (<https://usacookie.de>) bei Rückblick auf den Bachelor
(<http://sebinside.de/2017/04/13/rueckblick-auf-den-bachelor/#comment-134>)

ARCHIV

Januar 2018 (<http://sebinside.de/2018/01/>)

Juni 2017 (<http://sebinside.de/2017/06/>)

Mai 2017 (<http://sebinside.de/2017/05/>)

April 2017 (<http://sebinside.de/2017/04/>)

Oktober 2016 (<http://sebinside.de/2016/10/>)

September 2016 (<http://sebinside.de/2016/09/>)

August 2016 (<http://sebinside.de/2016/08/>)

Juli 2016 (<http://sebinside.de/2016/07/>)

Juni 2016 (<http://sebinside.de/2016/06/>)

April 2016 (<http://sebinside.de/2016/04/>)


März 2016 (<http://sebinside.de/2016/03/>)

SOCIAL MEDIA



(<http://twitter.com/skate702>) (<http://facebook.com/skate702>) (<http://youtube.com/skate702>) (<http://googleplus.com/skate702>)
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>
<http://twitter.com/skate702> <http://facebook.com/skate702> <http://youtube.com/skate702> <http://googleplus.com/skate702>

FEED

 (<http://sebinside.de/feed/>) RSS - Beiträge (<http://sebinside.de/feed/>)

 (<http://sebinside.de/comments/feed/>) RSS - Kommentare (<http://sebinside.de/comments/feed/>)



Ich heiße Sebastian, studiere Informatik am KIT in Karlsruhe und betreibe den YouTube Kanal skate702 (<http://youtube.com/skate702>). Auf diesem Blog geht es um Technik, Software und Minecraft!