



14 coole Scala-Features (Teil 1 von 3)

📅 21. September 2016 (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-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!

"If I were to pick a language to use today other than Java, it would be Scala."

James Gosling, Erfinder von Java

Scala, Scala, Scala. In meinen letzten beiden Blog-Posts über Programmierparadigmen und funktionale Programmierung habe ich schon angedeutet, dass ich mich in letzter Zeit viel mit dieser Sprache auseinander gesetzt

habe. Ja, sogar ein Großteil meiner Bachelorarbeit habe ich in Scala geschrieben. Bevor ich euch hier ein weiteres Mal von dieser Sprache überzeugen will, kurz meine eigene Geschichte.

Ich habe vor 10 Jahren mit Programmieren angefangen, zunächst mit Visual Basic.NET und Java. Seit 4 Jahren studiere ich jetzt Informatik und Mitte letzten Jahres habe ich die funktionale Programmiersprache Haskell kennengelernt. Anfang dieses Jahres musste ich mich für meine letzte Pflichtklausur im Studium dann intensiv mit dieser und vielen anderen Sprachen verschiedenster Paradigmen – unter anderem auch Scala beschäftigen.

10 Minuten nach der Klausur habe ich mir dann mein erstes Scala-Buch ausgeliehen. Das ist jetzt 5 Monate her. In dieser Zeit habe ich mich intensiv mit der Sprache auseinander gesetzt und sie lieben gelernt.

Was ist Scala?

Der Name Scala steht für **Skalierbarkeit**. Beim Design der Sprache wurde viel Wert darauf gelegt, dass die Sprache in alle Richtungen erweiterbar ist. Und ja, es ist in Scala sogar möglich, eine eigene Sprache (sog. DSL – Domain Specific Languages) zu definieren. Während in diesem Bereich die Freiheiten nahezu unbegrenzt sind, ist Scala in anderen Bereichen, grade bei der Objektorientierung wesentlich restriktiver als z.B. Java und räumt an vielen Stellen mit historisch bedingten Halbwahrheiten auf.

Wo wir es gerade von Objektorientierung haben: Scala ist eine **Multiparadigmen-Sprache**. Das heißt, sie kombiniert verschiedene Sprachkonzepte, ohne den Programmierer zu zwingen, diese auch zu nutzen. Vorhanden sind zum einen imperativer Code und Objektorientierung (es gibt viele Gemeinsamkeiten zu Sprachen wie Java, C# oder C++) und auf der anderen Seite funktionale Programmierung. Ja, ich weiß, seit Version 8 kann Java auch Lambdas, wow. Scala geht hier aber wesentlich weiter und erinnert an vielen Stellen an Haskell.

Außerdem läuft Scala überall dort, wo auch Java läuft. Der Scala-Compiler erzeugt nämlich **Java-Byte-Code**, der mit einer zusätzlichen Library auch von jeder stinknormalen Java-Umgebung ausgeführt werden kann. Prinzipiell ist auch eine .NET-Übersetzung möglich – diese hat sich aber nie durchgesetzt. Somit ist es möglich, Java-Code in Scala zu nutzen; oder Scala-Code in Java. Scala-Entwickler profitieren deshalb insbesondere von der großen Vielfalt offener Java-Libraries. Doch so viel zur Übersicht. Gehen wir jetzt etwas weiter ins Detail. Im Folgenden gehe ich auf 14 coole Features und Vorteile von Scala ein. Viel Spaß!

1. Die Syntax ist genial einfach.

Das erste und offensichtlichste Feature von Scala ist der **Syntax**, als die Art, wie Code hingeschrieben wird. Zu Syntax gehört die Reihenfolge von Anweisungen und die Bedeutung von Komma, Semikolon, Klammern usw.

Vergleichen wir als Einstieg mal zwei Zeilen Java mit zwei Zeilen Scala-Code:

```
1 // Java
2 List<Integer> numbers = Arrays.asList(new Integer[] {1,2,3,4});
3 System.out.println(numbers);
4
5 // Scala
6 val numbers = List(1,2,3,4)
7 println(numbers)
```

Okay, ich gebe zu: Dieses Beispiel ist natürlich furchtbar konstruiert. Es zeigt nicht nur, dass der Scala-Syntax einfacher ist, sondern auch, wie umständlich es ist, in Java 8 eine Liste mit Initial-Werten zu erzeugen.

Aber schauen wir uns mal die Unterschiede an. Zunächst: Scala benötigt kein **Semikolon** am Zeilenende. Außerdem lässt sich println direkt aufrufen, ohne erst den Weg über System.out zu gehen. Als nächstes fällt auf, dass die Variable Numbers keine **Typangabe** (z.B. List) hat. Wie das funktionieren kann, erkläre ich in einem späteren Kapitel. Der Rest sind dann nur noch kluge Hilfsfunktionen, die einem das Leben erleichtern. Und davon hat Scala eine Menge!

Das war aber erst der Anfang. Scala kommt mit vielen Kurzschreibweisen daher, sowohl der Typ als auch Klammern und Punkte sind oft optional. Maps lassen sich per Pfeil-Schreibweise erzeugen, **Kontrollstrukturen** wie for und match sind super

mächtig, return-Statements überflüssig. Der Vergleich `==` testet tatsächlich auf den Inhalt eines Objekts und nicht dessen Referenz und mit `val` und `var` kann direkt festgelegt werden, ob ein Wert Read-Only (oder in Java: `final`) ist.

Ein paar Beispiele:

```
1 val someString = "abcde"
2 val shortString = someString.substring 2
3
4 for(c <- shortString)
5   println(c)
6
7 println(someString == "ab" + shortString) // True
8
9 val myMap = Map(1 -> "value1", 4 -> "value2")
```

Das ermöglicht es natürlich auch, richtig, richtig unleserlichen Code zu schreiben. Genauer gehe ich hierauf im letzten Kapitel ein, aber hier schon mal ein Negativ-Beispiel aus meinem ersten Scala-Programm. Ihr könnt ja mal raten, was hier eigentlich passiert.

```
1 content = (for (i <- 0 until size * 2 by 2) yield groups(i) -> groups(i + 1)).toB
```

2. Alles kann überschrieben werden.

Einer der größten Vorteile von Scala ist es, dass die meiste Funktionalität **nicht** in die Sprachdefinition „hineingebrannt“ ist, sondern durch die Scala-Bibliothek zur Verfügung gestellt wird. Was bedeutet das?

Sprachen wie C# oder C++ machen es vor: In diesen ist es im Gegensatz zu Java möglich, **Operatoren** wie `+` oder `*` zu **überschreiben**. Du hast deinen eignen Zahlentyp `RationalNumber` definiert? Super, dann kannst du beide mit einem einfachen `+` addieren, anstatt erst die `add`-Methode aufzurufen.

Scala geht hier ein paar Schritte weiter. **Jede Methode kann auch ein Operator sein**. Ein Beispiel habe ich bereits oben mit `someString.substring 2` gezeigt. Obwohl `substring` eine Methode der Klasse `String` ist, wird sie hier wie ein Operator eingesetzt. Und wenn du jetzt der Meinung bist, statt `substring` sollte diese Methode besser `<|` heißen, dann kannst du das in Scala einfach definieren!

```
1 def <|-- (beginIndex: Integer) {  
2     //...  
3 }
```

Mit ein bisschen Extra-Magie wäre dann auch `someString <| - 2` legaler Scala-Code. Das ist auch die Grundlage für die in der Einleitung angesprochene Domain Specific Languages. ***Dir gefällt der Scala-Syntax nicht? Dann definier doch deinen Eigenen!***

Hier noch ein paar Beispiele, was alles in Scala wie festgebrannter Syntax aussieht und in Wirklichkeit auch nur ein Methodenaufruf (also überschreibbar!) ist:

```
1 // a : Int, b : List, c: Array, d: Double  
2  
3 a = 3 + 4  
4 a = 3.+(4)  
5  
6 b = List(1,2,3)  
7 b = List.apply(1,2,3)  
8  
9 c(2) = 4  
10 c.update(2) = 4  
11  
12 d = -2.0  
13 d = (2.0).unary_-
```

3. Die Typinferenz

Um Typinferenz verstehen und schätzen zu können, muss man den Unterschied zwischen statischer und dynamischer Typisierung kennen. Deswegen zunächst ein kurzer Exkurs.

Stellt man sich Variablen als Schublade für einen Inhalt (einen Wert) vor, dann gibt der Typ die Form der Schublade an. So wird festgesetzt ob eine Variable eine Ganzzahl, eine Kommazahl, ein Zeichen oder einen ganzen Satz in sich tragen kann. Viele Sprachen (Java, C++, ...) sind **statisch typisiert**, das bedeutet, dass bereits zu Compile-Zeit feststeht, welcher Typ eine Variable hat, weil der Programmierer es selbst hingeschrieben hat. Das bringt eine bessere Performance und eine hohe Sicherheit mit sich, da viele Fehler bereits vor der eigentlichen Ausführung des Programms gefunden werden können. Allerdings schränkt es den Entwickler auch ein und ist meistens nervige Schreiarbeit.

Einen anderen Ansatz bringt **dynamische Typisierung** (z.B. JavaScript) mit. Hier steht der Typ erst bei der eigentlichen Verwendung fest. Das ist natürlich ultra-flexibel. Aber was passiert, wenn du einer Zahl mitten im Programm ein Wort zuweist? Richtig, Crash. Bei einer statisch typisierten Sprache wäre dieser Fehler schon viel früher aufgefallen, das Programm hätte gar nicht kompiliert werden können. *Exkurs, Ende.*

Scala ist statisch typisiert. ABER: Der Typ muss in den meisten trotzdem nicht hingeschrieben werden. Diese Technik, dass der Compiler selbst feststellen kann, was der beste Typ für eine Variable ist, heißt Typinferenz. Und dieser Vorgang ist extrem kompliziert. Ernsthaft, von Hand diesen Prozess durchzurechnen gehört zum Schwersten was ich in diesem Jahr so gemacht habe 😊

Typinferenz ermöglicht es, den Typ einfach wegzulassen. Weniger Schreibarbeit, erst recht wenn es mal über einen Integer oder String hinausgeht. Das ist auch der Grund, warum Variablendefinitionen in Scala mit `val` oder `var` beginnen. Muss ein Typ angegeben werden, wird dieser in einer UML-ähnlichen Syntax dahinter geschrieben:

```
1 // Gleiches Ergebnis
2 val a = 3
3 val b : Int = 3
```

Übrigens: Andere Sprachen bringen ähnliche Konstrukte mit. C# hat z.B. das Schlüsselwort `var`, C++ den Typ-Platzhalter `auto`.

4. Umgang mit Klassen vereinfacht.

Alles klar, steigen wir jetzt mal in die **Objektorientierung** ein. Kurz zur Erinnerung: Objektorientierung bedeutet, Quellcode anhand von Klassen und Objekten zu unterteilen. Das Objekt *Auto* hätte dann z.B. eine *Farbe*, *Hubraum* und verschiedene Methoden wie *Hupe()* oder *StarteMotor()*. Wie das genau funktioniert und welche Vorteile sich daraus ergeben – googelt es einfach, wenn ihr noch nie davon gehört haben solltet. Hier nur so viel: Es ist der **Defacto-Standard heutiger Softwareentwicklung**.

Je nach Größe des Softwaresystems sind Klassen, Attribute und Methoden eh generiert; zum Beispiel aus einem (UML)-Modell. Wenn man Klassen aber mal von Hand schreiben muss, wird man in einigen Programmiersprachen schnell mal verrückt. Nehmen wir mal als Beispiel wieder Java: Warum muss ich mich selbst um **korrekte Sichtbarkeiten, Getter, Setter und Konstruktoren** kümmern? C# bietet an dieser Stelle immerhin eine syntaktische Verbesserung – aber trotzdem, zwischen den Schritten „*Entwerfen/Denken*“ und „*Verwenden*“ steht noch der Schritt: „*Unnötig viel Code hinschreiben/generieren*“. Scala verkürzt diese Schreibarbeit wesentlich.

```
1 class Auto(val hubraum: Int, var farbe: String) {  
2     def hupe() = {}  
3     def starteMotor() = {}  
4 }
```

Diese vier Zeilen erzeugen eine Klasse mit Konstruktor, Sichtbarkeiten, Attributen mit Zugriffsrechten und zwei Methoden. Eine äquivalente Java-Implementierung mit identischer Funktionalität wäre **18 Zeilen** – und damit mehr als 4x so lang. Von der Übersichtlichkeit müssen wir gar nicht erst reden.

Was passiert in diesen vier Zeilen? Eine Klasse „Auto“ wird erstellt. Diese Klasse hat zwei Attribute: Hubraum und Farbe. Hubraum kann im Nachhinein nicht neu gesetzt werden (Java-Äquivalent: Kein Setter), Farbe schon. Scala verwaltet im Hintergrund **automatisch** den Zugriff auf die Variablen, ohne dass Getter und Setter explizit hingeschrieben werden müssen. Ebenfalls spendiert und Scala einen Konstruktor, in dem beide Attribute automatisch gesetzt bzw. übergeben werden können. Funktionalität von 16 Zeilen, reduziert auf eine einzige. Der Rest sind die Methodendefinitionen. Um diese kommen wir natürlich nicht herum.

Wo ich grade **Konstruktoren** angesprochen habe: Hier sind die Regeln in Scala etwas **strenger** als z.B. in Java. Will man weitere Konstruktoren erstellen, müssen diese immer den Haupt-Konstruktor aufrufen. Ähnliche Einschränkungen gibt es bei der Verwendung von Javas super-Schlüsselwort. Und zum Schluss noch ein weiterer Tipp: Der Code, der innerhalb einer Klasse nicht in einer eigenen Methode steht, wird **automatisch zum Haupt-Konstruktor dazugezählt**. Kann manchmal durchaus verwirrend aussehen 😊

5. Klassen, Objekte und Traits.

Wir bleiben noch etwas bei der **Objektorientierung**. Denn hier gibt es grade für Java-Programmierer einiges zu lernen. Scala ist nicht nur bei Konstruktoren strenger, sondern hat auch an anderen Stellen Einschränkungen – zu Gunsten des Paradigmas, versteht sich. Dafür ist aber eine (vereinfachte) Form der **Mehrfachvererbung** möglich, die sonst vor allem aus C++ bekannt ist. Also, fangen wir an!

In Scala gibt es **Klassen**, **Objekte** und **Traits**. Klassen verhalten sich wie z.B. aus Java gewohnt. Allerdings können weder Methoden noch Attribute als static definiert werden. Zur Erinnerung: Der Modifier static ermöglicht es, auf Elemente einer Klasse auch ohne Instanziierung zuzugreifen. Bekanntestes Beispiel: Die Klasse Math. Um mathematische Hilfsfunktionen zu verwenden, muss nicht erst ein Math-Objekt erstellt werden; das ist der Sinn von statischen Methoden bzw. Klassenmethoden.

Üblicherweise wird **static** für öffentliche Konstanten und Hilfsmethoden verwendet. Allerdings lädt static auch wunderbar zum Schummeln ein, da hiermit das eigentlichen Klassenkonzept aufgebrochen werden kann. Das ist einer der Gründe, warum static nicht in Scala übernommen wurde.

Wer in Scala statische Methoden – bestes Beispiel die main()-Methode als Einstiegspunkt – definieren will, benötigt dafür ein **Objekt**. Die Syntax stimmt exakt mit dem einer Klasse überein, nur wird statt dem Schlüsselwort class eben object verwendet. Bei Objekten handelt es sich um sog. Singletons, also um einmalig instanziierte Klassen. Das ermöglicht insbesondere die Verwendung von statischen Methoden, bringt aber auch eine **klarere Trennung** mit sich.

Wer sowohl die Funktionalität einer Klasse, als auch eines Objekts benötigt, kann einfach in derselben Quelldatei beides mit dem gleichen Namen erstellen. Man spricht hier von **Companion Objects**. Etwas umständlicher, aber dafür konsequenter und lesbarer!


```
1 class myColor(val red: Double, val green: Double, val blue: Double) {  
2   def getHTMLCode: String = { ... }  
3 }  
4  
5 // Hier steht der statische, von den Attributen unabhängige Code  
6 object myColor {  
7   val perfectGreen = new myColor(0.0, 1.0, 0.0)  
8   def isWebColor(color: myColor): Boolean = { ... }  
9   def isWebColor(red: Double, green: Double, blue: Double): Boolean = { ... }  
10 }
```

Bleiben wir noch kurz bei den **Schlüsselwörtern**: Auch private, public und protected funktionieren in Scala anders als in Java. Der erste Unterschied: Wird kein Modifier angegeben, ist ein Element automatisch public. Der Modifier private verhält sich weitestgehend wie in Java, geändert wurde jedoch die Bedeutung von protected. Geregelt wird hiermit nur noch das Verhalten bei Vererbung, nicht mehr die Sichtbarkeit. Hierfür kommt eine neue Syntax mit eckigen Klammern ins Spiel, mit dem die Sichtbarkeit wesentlich genauer eingestellt werden kann.

```
1 package myTest  
2 class myClass {  
3   private[this] helpingMethod = {} // Nur in derselben Instanz sichtbar  
4   private[myTest] packageMethod = {} // Innerhalb desselben Packages sichtbar  
5 }
```

Kommen wir zum Schluss noch zu **Traits**. Diese kommen den Java Interfaces am Nächsten. In Traits ist es allerdings auch möglich Attribute und Methoden zu definieren, anstatt sie nur zu deklarieren (Was in Java 8 mit Default-Implementierung jetzt auch funktioniert, aber eigentlich zu etwas anderem dient). Ebenfalls ist es möglich, mehr als einen Trait zu implementieren und somit sogar **stapelbare Funktionalität** zu erreichen. Im Gegensatz zu C++ können hier typische Probleme der Mehrfachvererbung wie das **Diamantenproblem** (<https://de.wikipedia.org/wiki/Diamond-Problem>) allerdings nicht auftreten, da im Zweifelsfall der zuletzt angegebene Trait das Sagen hat. Man spricht deshalb auch von „mixins“ – was es meiner Meinung nach ziemlich gut auf den Punkt bringt.

```
1 trait Parent {  
2   def both() = 1  
3 }  
4  
5 trait A extends Parent {  
6   override def both() = 3  
7   def printThis() = { println("This") }  
8 }  
9  
10 trait B extends Parent {  
11   override def both() = 5  
12   def printThat() = { println("That") }  
13 }  
14  
15 object mainObject extends A with B {  
16   def main(args: Array[String]) = {  
17     println(both()) // Gibt 5 aus  
18     printThis()  
19     printThat()  
20   }  
21 }
```

Fazit von Teil 1

Und das waren sie, die ersten fünf „coolen Features“ von Scala. Eigentlich war gedacht, nur ein paar Vorteile der Sprache zu nennen und nicht so weit ins Detail zu gehen... aber gut, hoffentlich bleiben so nach über 2000 Wörter für Teil 1 wenigstens weniger Fragen offen.

Besprochen habe ich im ersten Teil neben allgemeinen Grundlagen und **Vorteilen die Syntax, Erweiterbarkeit und Skalierbarkeit**. Ebenfalls habe ich erklärt wie in Scala **Typinferenz** und **Objektorientierung** funktioniert.

Im nächsten Teil werden wir uns weitere Vorteile der **funktionalen Programmierung** in Objekten ansehen und insbesondere die **erweiterten Kontrollstrukturen** von Scala untersuchen. Wer bis jetzt schon der Meinung ist, dass Scala schon das ein oder andere coole Feature mitbringt und manches vielleicht ganz elegant erweitert... glaubt mir, *das war noch gar nichts!*

Teilen mit:

 E-Mail (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/?share=email&nb=1>)

 Facebook 1 (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/?share=facebook&nb=1>)

 Twitter (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/?share=twitter&nb=1>)

Gefällt mir:

Lade ...

funktional (<http://sebinside.de/tag/funktional/>) haskell (<http://sebinside.de/tag/haskell/>)java (<http://sebinside.de/tag/java/>)Programmierparadigmen (<http://sebinside.de/tag/programmierparadigmen/>)Programmiersprachen (<http://sebinside.de/tag/programmiersprachen/>)programmierung (<http://sebinside.de/tag/programmierung/>) scala (<http://sebinside.de/tag/scala/>)

3 Gedanken zu „14 coole Scala-Features (Teil 1 von 3)“

"VALLE"

22. September 2016 um 16:38 Uhr (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/#comment-62>)

Also wirklich! Zwei Programmiersprachen vergleichen und im selben Zug dass mit das verwechseln. Ich bin zutiefst enttäuscht!

Zitat:“ Als nächstes fällt auf, das die Variable Numbers keine Typangabe (z.B. List) hat. “

War nicht ganz ernst gemeint, Skate. Ich wollte dir eigentlich nur den Rechtschreibfehler zeigen!

Also mach weiter so! Ich finde den blog echt interessant und schön gestaltet.

Ursprünglich bin ich über YouTube auf dich gestoßen.

ANTWORTEN ([HTTP://SEBINSIDE.DE/2016/09/21/14-COOLE-SCALA-FEATURES-TEIL-1-VON-3/?REPLYTOCOM=62#RESPOND](http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/?replytocom=62#respond))

[SABLE] (@OSLPYT) ([HTTP://TWITTER.COM/OSLPYT](http://twitter.com/OSLPYT))

23. September 2016 um 16:08 Uhr (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/#comment-64>)

Scala: Kein Problem

Java: Kein Problem

Deutsch: Hä?

ANTWORTEN ([HTTP://SEBINSIDE.DE/2016/09/21/14-COOLE-SCALA-FEATURES-TEIL-1-VON-3/?REPLYTOCOM=64#RESPOND](http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/?replytocom=64#respond))

MINNA ([HTTP://WWW.MINNA-SERVER.DE](http://www.minna-server.de))

16. Juli 2017 um 9:06 Uhr (<http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/#comment-123>)

Das ist eine wirklich super gute Einführung zu Scala. Den Inhalt in der Kürze und nebenbei auch noch leidenschaftlich rüber zu bringen, wiegt alle Scheibfehler der Welt auf.

ANTWORTEN ([HTTP://SEBINSIDE.DE/2016/09/21/14-COOLE-SCALA-FEATURES-TEIL-1-VON-3/?REPLYTOCOM=123#RESPOND](http://sebinside.de/2016/09/21/14-coole-scala-features-teil-1-von-3/?REPLYTOCOM=123#RESPOND))

KOMMENTAR VERFASSEN


Gib hier deinen Kommentar ein ...

◀ Von imperativ zu funktional – Ein Scala-Beispiel (<http://sebinside.de/2016/08/26/von-imperativ-zu-funktional-ein-scala-beispiel/>)

14 coole Scala-Features (Teil 2 von 3) ▶ (<http://sebinside.de/2016/09/28/14-coole-scala-features-teil-2-von-3/>)

SOCIAL MEDIA

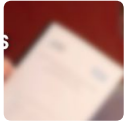


(http (http (http 
p://t p://f p://y (htt
witt aceb outu p://g
er.c ook. be.c ithu
om/ com om/ b.co
skat /seb skat m/s
e70 insid e70 ebin
2) e) 2) side)

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-ah-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



FEED

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



Ich heie 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!