14 coole Scala-Features (Teil 3 von 3)

5. Oktober 2016 (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-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!

Willkommen zurück zum dritten und damit letzten Teil meiner Scala-Reihe! In den letzten zwei Posts habe ich bereits gezeigt, welche erstaunlichen Features Scala bietet, und was dadurch alles möglich wird. Okay, ich gebe zu, ich bin schon ziemlich **von Scala überzeugt** – Aber daran wird sich auch dieses Mal nichts ändern 🙂

Wir haben bis jetzt schon einiges gesehen: Die Macht von **Kontrollstrukturen**, **implizite Konvertierung** und **Typinferenz** und die Mischung von Objektorientierung und funktionaler Programmierung. Das meiste was ich bis jetzt vorgestellt habe, hat aber direkt mit der Sprache Scala selbst zu tun. In diesem letzten Kapitel möchte ich mich etwas davon entfernen, der Fokus liegt eher auf Features, die sich aus der mitgelieferten **Scala Bibliothek** ergeben.

Heute geht es um Scalas **XML-Support**, um **Actors** für Parallel-Programmierung, um intuitives **Parsen** und um **Build-Kontrolle**. Und ganz zum Schluss habe ich auch noch einige Nachteile und Problemfelder von Scala gesammelt. **Los geht's!**

11. XML-Support

Was war nochmal **XML**? Sieht das nicht so ähnlich aus wie HTML? Okay, vielleicht sollten wir von vorne anfangen. XML steht für ***eXtensible Markup Language*** und ist eine semi-strukturierte **Auszeichnungssprache**. Sie wird zur **Kommunikation** von Prozessen eingesetzt, Beispielweise über das Internet. Hierbei ist die Verwendung von XML allerdings implementierungsunabhängig; was bedeutet, dass XML innerhalb von anderen Sprachen verwendet werden kann, beispielweise um ein Objekt zu ***serialisieren*** (Fachwörter sind super, oder? Mit serialisieren meine ich natürlich die Persistierung einer textuellen Repräsentation. Was, das versteht immer noch kein Schwein? Okay, man wandelt eben ein Objekt im Speicher in Text um, den man speichern oder jemand anderem schicken kann).

XML kann z.B. so aussehen:

```
1 <skateboard>
2   <deck size="7.75" brand="Jart">
3     <Griptape color="black"></griptape>
4   </deck>
5   <trucks stolenByACat="False"> ... </trucks>
6 </skateboard>
```

Wenn du jetzt XML nutzen möchtest, um Daten zu verpacken bzw. um empfangene Daten zu lesen, kannst du dir mühsam etwas dafür selbst programmieren. Oder du nutzt den **XML-Support** den die meisten Sprachen von Haus aus mitbringen. Ich hatte z.B. für meine Bachelor-Arbeit das Vergnügen mit dem XmlParser von C# zu spielen. Und hab ihn erstmal überhaupt nicht verstanden und totalen Mist programmiert; und das, obwohl der eigentlich ziemlich intuitiv ist. Aber das wird dir mit den meisten Parsern passieren, man muss sich eben erstmal einlernen. Optimal wäre es natürlich, wenn es gar keinen Parser benötigen würde, weil XML schon **nativ unterstützt** wird... ***oh, hallo Scala!***

Folgender Code ist absolut korrektes Scala:

```
1 val myXML = <wheel size="50mm"><bearing abec="7"></bearing></wheel>
```

Auffällig ist hier, dass XML einfach so, mit anderem Scala-Code ***in einer Zeile*** steht. Das funktioniert, weil der Compiler automatisch XML erkennt und ihm den Typ ***xml.NodeSeq*** zuweist. Mit XML-Code arbeiten geht dann übrigens genauso flott. Hier ein paar einfache Beispiele:

```
1 val myXML = <wheel size="50mm"><bearing abec="7"><ball type="steel" /></bearing>
2
3 println(myXML \ "bearing") // Extrahiert den bearing-tag
4 println(myXML \\ "ball") // Extrahiert den ball-tag (Deep-Search)
5 println(myXML \\ "ball" \ "@type") // Extrahiert das Attribut type aus balls
```

Aber damit ist noch nicht genug. XML wird nicht nur als Auszeichnungsform in Scala akzeptiert, sondern profitiert auch von den **Kontrollstrukturen** von Scala:

```
1  val color = "orange"
2  val moreXML = <color>{if (color != "lila") color else xml.NodeSeq.Empty}</color
3
4  println(moreXML) // <color>orange</color>
5
6  val printMe = moreXML match {
7    case <color>{foundColor}</color> => foundColor.text + " ist eine tolle Farbe!
8    case _ => "Igit. LILA!"
9    // Unerreichbar
10   case <color>{contents @ _*}</color> => "Mehr Inhalt!"
11 }
```

Im ersten Teil wird **dynamisch im Code** XML erzeugt. Dabei steht ein If-Statement in geschweiften Klammern mitten im XML-Code und erzeugt dort zur Laufzeit den Inhalt. Überhaupt kein Problem. Der zweite Teil ist sogar noch cooler: Hier wird das XML mit **Pattern-Matching** wieder auseinander genommen.

> Die letzte Zeile kann übrigens wegen dem Catch-All-Unterstrich nicht erreicht werden und ist nur der ***Vollständigkeit halber*** bzw. zur Demonstration da: Dieser Syntax wird benutzt, wenn man mehr Inhalt als nur eine einfach Node erwartet.

12. Actors

Ein weiteres cooles Feature der Scala-Bibliothek sind Actors, die eine **Alternative zur Verwendung von Threads** für Parallelisierung darstellen. Hand aufs Herz: Wer hat schon mal versucht etwas mit Hilfe von Threads zu Programmieren und dabei

furchtbar gescheitert? Eben. Versucht man Synchronisierungs-Probleme zu vermeiden, erhöht man das Risiko von **Deadlocks**. Obwohl Threads klug eingesetzt und im Pool verwaltet durchaus funktionieren können, geht Scala einen anderen Weg.

Der Grundgedanke ist einfach: Das **Problem bei der Verwendung von Threads** ist häufig die **Verwaltung des Zugriffs** auf geteilte Variablen und Objekte. Anstatt diesen jetzt besonders klug zu regeln, schafft man ihn einfach komplett ab. **Actors teilen kein Wissen** und arbeiten nur auf Zuruf: Jeder Actor ist komplett unabhängig, bekommt alle Informationen per Nachricht geschickt und antwortet, sobald er fertig ist. Die Wahrscheinlichkeit einer über mehrere Actor bzw. Threads hinausgehende Abhängigkeit wird so minimiert.

Actor sind gut in die Sprache integriert und schon mit wenigen Zeilen Code einsatzbereit. Hier ein kurzes Beispiel:

```
1  object EchoActor extends Actor {
2    override def act(): Unit = {
3      loop {
4        receive {
5          case msg => println("Got message: " + msg)
6  } } } }
7
8  def main(args: Array[String]): Unit = {
9
10   val loopingActor = actor {
11     for (i <- 1 to 3) {
12       println("Loop!")
13       Thread.sleep(100)
14   } }
15
16   EchoActor.start()
17   EchoActor ! "Echo!”
18   println("Hello World")
19 }
```

Eine mögliche Ausgabe wäre:

```
1 Hello World
2 Loop!
3 Got message: Echo!
4 Loop!
5 Loop!
```

Um den Code kurz zu erklären: Das Objekt Actor erbt von der **Klasse Actor** und implementiert die Methode act(). Das sieht analog zum Erstellen eines Threads aus. In der Methode act wird in einer while(true) bzw. loop-Schleife der ***„Posteingang“*** des

Actors abgefragt. Sobald eine Nachricht empfangen wird, wird der Inhalt über ***Pattern Matching*** analysiert. Wird ein passendes Pattern gefunden wird die Nachricht weiter verarbeitet, sonst wird sie ignoriert und verworfen.

In der Main-Methode wird ein weiterer Actor erstellt. Dieser wird mit der **Actor-Eigenen Kurzschreibweise** aufgebaut und startet deswegen automatisch, während das Objekt EchoActor erst noch auf den Start-Befehl wartet. Durch den Aufruf der ***Ausrufezeichen-Methode*** wird eine Nachricht an diesen gesendet. Interessant an dieser Stelle: Die Ausgabe ist ***nicht deterministisch***, auch andere Reihenfolgen sind möglich!

Damit Actor optimal funktionieren, gibt es noch einige Regeln, die man einhalten sollte: Actors sollten auch bei der Verarbeitung einer Nachricht **nicht blockieren** und sich ausschließlich über Nachrichten unterhalten. Außerdem sollten die Nachrichten im Idealfall ***immutable*** sein. Dann bietet das Actor-Modell aber eine gelungene **Alternative** zu herkömmlichen Threads an!

13. Combinator Parsing

Kommen wir zu **meinem persönlichen Highlight von Scala**. Was kann so cool sein, dass ich es mir bis ganz zum Schluss aufgehoben habe? Noch spannender als Pattern Matching, eigene Kontrollstrukturen oder Typinferenz? Naja, wie wäre es mit der Möglichkeit seine **eigene** Sprache in der Sprache zu definieren?

Ich rede hierbei nicht von Methoden, die als Operatoren dienen in Zusammenspiel mit impliziter Konvertierung. Diese Techniken können verwendet werden, damit ***innerhalb*** von Scala DSL-artiger Code geschrieben werden kann, der nicht mehr wirklich nach Scala aussieht (Ein Beispiel ist SBT, siehe nächstes Kapitel).
Nein, ich rede davon, **externen Code** der z.B. als Textdatei vorliegt und selbst definierten Regeln folgt, einzulesen und automatisch richtig umzusetzen.

Das geht stark in Richtung **Compilerbau** (Compiler sind die Programme, die Quellcode wie Java oder Scala einlesen und ihn in ausführbare Programme übersetzen). Die klassischen ersten Schritte hierfür sind **Parsing**, Lexing und Aufbauen einer Objekthierarchie die anschließend weiter verarbeitet werden kann. Um es einfach zu halten: Der gelesene Code muss verstanden, geprüft und dann in vorbereitete Objekte im Speicher umgewandelt werden.

Das ist ein ziemlich **komplexer Schritt**, weswegen meistens fertige Software-Tools verwendet werden, die eine ***Grammatik*** annehmen und Quellcode ausspucken, mit dem die eigene Sprache gelesen werden kann. Unter einer Grammatik versteht man in der Informatik in diesem Kontext einfach eine Definition, welche Worte bzw. Zeichen **aufeinander folgen** können. Wenn wir z.B. eine Variable definieren, dann beginnen wir mit ***val*** oder ***var***, darauf folgt der Name der Variablen (für den natürlich auch gewisse Regeln gelten), gefolgt von einer optionalen Zuweisung, die wiederum mit einem Ist-Gleich-Zeichen beginnt, usw. Das ist eine **Grammatik**.

Wahrscheinlich könnt ihr euch schon denken auf was das hinaus läuft. Richtig: Scala bietet eine **komplette Bibliothek zum Einlesen und Umwandeln eigener Sprachen**. Und das coole dabei: Es ist super einfach dieses Sprachen zu definieren, weil der Syntax hierfür sehr stark an die Art erinnert, wie man **kontextfreie Grammatiken** sowieso aufschreiben würde. Das Ergebnis ist extrem einfach erweiterbar, wartbar und funktioniert ohne externe Tools, wird also direkt in Scala hingeschrieben. Willkommen in der Welt der **Parser Combinator**!

Obwohl Parser Combinators genial sind und vieles vereinfachen, handelt sich bei der Definition von eigenen Sprachen durch kontextfreie Grammatiken um ein **komplexes Themengebiet**. Wer z.B. an meiner Uni Informatik studiert, lernt diese erst im 3. Semester kennen. Mit den Basics des Compilerbaus beschäftigt man sich erst gegen Ende es Bachelor-Studiengangs und in der Tiefe wird dieses Thema erst im Master besprochen. Also auf jeden Fall ein Gebiet, das den Umfang dieses Blog-Posts etwas übersteigt. Ich möchte trotzdem ein paar **syntaktische Basics** festhalten:

```scala
trait Greeting {
  val name: String
}

object GreetingsParser extends JavaTokenParsers {

  case class SimpleHello(override val name: String) extends Greeting

  case class NormalHello(override val name: String, content: String) extends Gr

  case class Bye(override val name: String) extends Greeting

  def helloGreeting: Parser[Greeting] = "Hello|Hi".r ~> "[a-zA-Z]+".r ~ opt("[^
    case (x ~ Some(y)) => NormalHello(x, y)
    case (x ~ None) => SimpleHello(x)
  }

  def byeGreeting: Parser[Greeting] = "Bye" ~> "[a-zA-Z]+".r ^^ {
    case (x) => Bye(x)
  }

  def multiGreeting: Parser[List[Greeting]] = repsep(helloGreeting | byeGreetir

  def main(args: Array[String]): Unit = {

    parse(helloGreeting, "Hello Seb whats up?") match {
      case Success(matched, _) => println("MATCHED: " + matched)
      case Failure(msg, _) => println("FAILURE: " + msg)
      case Error(msg, _) => println("ERROR: " + msg)
    }

    parse(multiGreeting, "Hello Seb ; Bye Tom") match {
      case Success(matched, _) => println("MATCHED: " + matched)
      case Failure(msg, _) => println("FAILURE: " + msg)
      case Error(msg, _) => println("ERROR: " + msg)
    }

    parse(byeGreeting, "ASDFMovie") match {
      case Success(matched, _) => println("MATCHED: " + matched)
      case Failure(msg, _) => println("FAILURE: " + msg)
      case Error(msg, _) => println("ERROR: " + msg)
    }

  }

}
```

Dieser Code kann einfache Begrüßungen einlesen und wandelt sie automatisch in Objekte um. Dabei werden eine Menge von Scala-Features benutzt, auf die ich hier aus Platzgründen nicht eingehen will, z.B. **Case Klassen** und **Regex**. Ich versuche mich kurz zu halten:

SimpleHello, ***NormalHello*** und Bye sind meine Objektrepräsentationen einer Begrüßung, alle erben vom ***Trait Greeting*** (der vorsieht, dass es mindestens einen Namen des Gegrüßten gibt). In den Methoden ***helloGreeting***, ***byeGreeting*** und ***multiGreeting*** passiert nun die eigentliche Magie. Hier ist jedes einzelne Zeichen entscheidend:

Die String-Literale mit dem r dahinter sind **Regex-Ausdrücke**. ***„Hello|Hi“*** bedeutet z.B. dass eine Begrüßung mit Hello oder Hi beginnen kann. Hiernach folgt der Name (der nur aus Buchstaben bestehen muss und nicht leer sein darf). Der Pfeil dazwischen bedeutet, dass es für die nachfolgende Auswertung irrelevant ist, ob eine Begrüßung jetzt mit ***„Hello“*** der ***„Hi“*** begonnen hat. Nicht irrelevant sondern **optional** ist der Teil danach. So kann nach dem eigentlich Namen optional ein Teilsatz wie ***„was geht“*** folgen.

Das danach ist kein Smiley, sondern leitet die **Verarbeitung** des geparsten Strings ein: In den nächsten beiden Zeilen wird der gefundene Satz mit Hilfe von **Extractors** auseinander genommen. Entweder gibt es den zusätzlichen Teilsatz, dann wird ein ***NormalHello*** zurückgegeben, sonst ist es ein ***SimpleHello***. Analog funktioniert die Erkennung von ***byeGreeting***.

Die Erkennung in ***multiGreeting*** ist auch nicht wesentlich komplexer. ***repsep*** steht für ***„Repitition Seperated“***, also eine Wiederholung eines Syntaxes, getrennt durch ein explizites Zeichen. Das Ergebnis hiervon ist eine Liste. Der (unnötig komplizierte) Code in der ***main()-Methode*** sollte selbsterklärend sein, das Ergebnis der Ausführung lautet wie folgt:

```
1 MATCHED: NormalHello(Seb,whats up?)
2 MATCHED: List(SimpleHello(Seb), Bye(Tom))
3 FAILURE: `Bye' expected but `A' found
```

Wer schon mal mit anderen Sprachen einen Compiler gebaut hat oder schon mal versucht einen eignen Parser zu schreiben, wird jetzt wahrscheinlich jubeln. Scala ermöglicht es, dass die Komplexität des Parsens **nicht mehr im Algorithmus** selbst liegt und geht dabei sogar über LL(1)-Grammatiken (~!) hinaus. Somit bleibt nur noch die Aufgabe, die Grammatik selbst zu entwerfen. Nur das ist meistens so ein ***ganz anderes Problem…***

14. SBT

Okay, das war's. Wir sind quasi durch. Besser als Combinator Parsing wird es nun wirklich nicht mehr. Als letztes Feature möchte ich noch schnell SBT (**Scala Build Tool**) vorstellen. Aber um ehrlich zu sein, habe ich von Build-Tools nicht viel Ahnung. SBT ist eine **interne DSL**, also auch „nur" Scala Code. Außerdem ist SBT **Maven-kompatibel**. Welche Software man am Schluss verwenden möchte, bzw. ob man überhaupt ein Build-Tool für kleinere Projekte verwenden will ist jedem selbst überlassen. Hier ein kurzes Code-Beispiel eines kleinen Neben-Projekts von mir:

```
1  name := "MyLittleTool"
2  version := "1.0"
3  scalaVersion := "2.11.8"
4
5  libraryDependencies += "com.github.scopt" %% "scopt" % "3.5.0"
6  libraryDependencies += "net.lingala.zip4j" % "zip4j" % "1.3.2"
7  resolvers += Resolver.sonatypeRepo("public")
8
9  libraryDependencies := {
10   CrossVersion.partialVersion(scalaVersion.value) match {
11     case Some((2, scalaMajor)) if scalaMajor >= 11 =>
12       libraryDependencies.value ++ Seq(
13         "org.scala-lang.modules" %% "scala-xml" % "1.0.4")
14     case _ =>
15       libraryDependencies.value
16   }
17 }
```

In diesem Beispiel werden zunächst allgemeine Informationen des Projekts gesetzt, dann einige **Abhängigkeiten** hinzugefügt, die SBT automatisch versucht aufzulösen. Im unteren Teil wird abhängig von der verwendeten Scala-Version der XML-Support von Scala separat hinzugefügt (was bei neueren Versionen notwendig ist).

Nachteile von Scala

Das waren meine persönlichen Top-Features von Scala. Objektivität… ist so eine Sache. Ich denke man hat des Öfteren bemerkt, dass ich von der Sprache ein ***kleines bisschen begeistert*** bin. Nichts desto trotz wäre es falsch zu behaupten, das die Verwendung von Scala nicht auch **Nachteile** mit sich bringt. Ein paar davon möchte ich hier zum Schluss aufzählen.

Scala ist eine Mehrparadigmen-Sprache und damit recht **umfangreich**. Natürlich bringt die Kombination von parallelem, funktionalem, objektorientiertem und imperativen Paradigma viele Vorteile mit sich, aber auch nur, wenn man von diesen schon mal etwas gehört hat und diese nutzen kann. Das ist auch der Grund, ***warum***

ich Scala Anfängern nicht empfehlen würde. Ich hatte Scala erst kennen gelernt, als ich in all diesen Paradigmen schon etwas Erfahrung gesammelt hatte, was mir den Einstieg erleichtert hat. ***Ohne Vorwissen dürften viele Konzepte der Sprache schwerer zu verstehen und anzuwenden sein.*** (Das hat übrigens auch **James Gosling** (Erfinder von Java) über Scala gesagt, als man ihn auf einem Interview darauf angesprochen hat!)

Scala bietet viele Möglichkeiten. Aber wie heißt es so schön: ***„Aus großer Macht folgt große Verantwortung"***. Gerade durch die Mischung von imperativem und funktionalem Code ist es ***noch einfacher unleserlichen und schwachen Code hinzuschreiben***. Das ist tatsächlich ein solch großes Problem, dass die Entwickler von Twitter in ihrer Scala-Leitlinie (http://twitter.github.io/effectivescala/) dieser Fragestellung ein eigenes Kapitel gewidmet haben. Die Einfachheit von Scala kann sehr gut genutzt werden, um besseren und lesbareren Code bei gleicher oder sogar besserer Performance zu erzeugen. Aber es ist leider genauso ***einfach, alles falsch zu machen***.

Wo wir gerade bei der **Performance** sind: Hier kommen beide zuvor genannten Nachteile noch mal zusammen. Natürlich kann funktionaler Code optimiert und performant sein, auch wenn es C-Nazis vielleicht nicht wahrhaben wollen. Aber gerade hier ist es ohne Vorwissen noch einfacher, Fehler einzubauen. Stichwort: ***Tail-Recursion***.
Obwohl Scala viele Erweiterungen und Möglichkeiten bietet, die man sich z.B. in Java wünschen würde, gibt es an anderen Stellen auch **Einschränkungen**. Um ein paar (weitere) Beispiele zu nennen: Scala kennt keine ++-Methode, kein break und kein continue. Der Hintergrund dieser Entscheidung ist das Vermeiden von Seiteneffekten und Code-Sprüngen. Trotzdem ist hier (und bei manch anderer Eigenheit) eine **Umgewöhnung** notwendig.

Und zum Schluss: Scala ist im Gegensatz zu Java kein „Superstar" – ***Die Sprache ist nicht so weit verbreitet***. Auf der Seite githut.info (http://githut.info/) gibt es eine Übersicht der meist verwendeten Sprachen auf GitHub. Java ist dort auf Platz 2, ***Scala nur auf Platz 19***. Das bedeutet, dass die Sprache tendenziell weniger Leute sprechen, was z.B. ein **zusätzliches Hindernis** für Unterstützung bei Open-Source Projekten ist. Außerdem ist somit auch die **Toolunterstützung** geringer. In der IDE IntelliJ von JetBrains spürt man einen merklichen Unterschied, was den Funktionsumfang im

Gegensatz zu Java angeht. Immerhin auf Code-Ebene ist dies aber egal, da Java-Code ohne Probleme auch in Scala genutzt und insbesondere Java-Bibliotheken so weiterverwendet werden können. **Glück gehabt!**

Fazit von Teil 3

Mehrere Wochen habe ich jetzt an dieser Reihe gearbeitet. Wenn du bis hierhin alles gelesen hast, hast du mehr als 8000 Wörter hinter dir. **Glückwunsch!** Ich hoffe, du hast etwas mitgenommen. 🙂

Ich habe mich zu diesem Zeitpunkt ca. seit 6 Monaten mit der Sprache Scala beschäftigt und habe nicht vor in absehbarer Zeit aufzuhören. Denn obwohl ich inzwischen mehrere Bücher hierzu gelesen habe und Scala zu meiner **Entwicklungssprache Nummer Eins** geworden ist, bin ich eigentlich doch noch ziemlich am Anfang.

Hier nochmal die **14 Features auf einen Blick**: Syntax, Erweiterbarkeit, Typinferenz, Objektorientierung, Objekte & Traits, Funktionalität, List-Comprehension, Pattern Matching, eigene Kontrollstrukturen, implizite Konvertierung, XML-Support, Actors, Combinator Parsing und SBT.

Vielleicht entdecke ich in nächster Zeit noch **weitere coole Features**. Ein Gebiet mit dem ich mich z.B. noch gar nicht beschäftigt habe, ist die Skalierbarkeit für große Software-Systeme und Oberflächen-Programmierung mit Swing in Scala oder JavaFX bzw. ScalaFX.

Ich hoffe, ich habe dir bis hier hin einen guten Einstieg oder Überblick geboten. Diese Posts sind auch für mich selbst interessant, da die Code-Bespiele ein gutes **Nachschlagewerk für die Scala-Basics** bieten. 8122 Wörter. Geplant waren 2000-3000. Alles klar. ***Danke fürs Lesen.*** Ladet euch den Scala-Compiler runter. Fangt an zu Schreiben. Und dann lasst mich wissen, was ihr cooles auf die Beine stellt habt. **Bye!**

Haaaalt Stopp. Ich will mehr!

Falls du jetzt **noch mehr über Scala erfahren** willst, kann ich dir zwei Quellen ans Herz legen. Das Buch **Programming in Scala** (http://amzn.to/2cCY2k8) bietet einen super Einstieg in die Sprache, ich habe es bereits mehrmals durchgelesen. Es ist zwar auf englisch, aber sehr verständlich geschrieben. Das meiste Wissen dieser Reihe stammt aus diesem Buch!

Außerdem gibt es eine **Sammlung von Best Practices von Twitter** (http://twitter.github.io/effectivescala/). Dort ist Scala nämlich eine der Hauptentwicklungssprachen. Hier gibt es viele Tipps, Anleitungen und Wissen, das sich mit der Zeit ergeben hat. Definitiv kein Fehler, sich an dieses zu Herzen zu nehmen!

Teilen mit:

E-Mail (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/?share=email&nb=1)

Facebook 1 (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/?share=facebook&nb=1)

Twitter (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/?share=twitter&nb=1)

Gefällt mir:

Lade ...

coding (http://sebinside.de/tag/coding/) funktional (http://sebinside.de/tag/funktional/)

funktionale programmierung (http://sebinside.de/tag/funktionale-programmierung/)

haskell (http://sebinside.de/tag/haskell/)

imperative programmierung (http://sebinside.de/tag/imperative-programmierung/)

Programmieren (http://sebinside.de/tag/programmieren/)

Programmierparadigmen (http://sebinside.de/tag/programmierparadigmen/)

Programmiersprachen (http://sebinside.de/tag/programmiersprachen/)

programmierung (http://sebinside.de/tag/programmierung/) scala (http://sebinside.de/tag/scala/)

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

SUPRALP (HTTP://73.NO-IP.BIZ/SUPRA)
5. Oktober 2016 um 18:11 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-72)

haha, stolenByACat… 😀

stolenBySupralp=“true“ 😛

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=72#RESPOND)

LANIE987
6. Oktober 2016 um 13:16 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-73)

Ist das tatsächlich eine Anspielung auf ‚lila ist keine Farbe‘? 😀

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=73#RESPOND)

BUBA DABU
25. Oktober 2016 um 23:17 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-74)

Kann man sich deine Bachelorarbeit ansehen?

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=74#RESPOND)

TRIDDEL02
9. November 2016 um 21:54 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-78)

Hei Skate, vllt liest du das hier ja 🙂

Hat nichts mit diesem Beitrag zutun, allgemeines Anliegen!! 🙂 Bin nur zufällig hier xD

Kurze Info, ich bin 15, Gymnasium 9. Klasse und sehr Informatik interessiert, bzw will später definitiv in die Software Entwicklung gehen (nicht sicher, was genau.. spiele, Programme etc.) ist aber auch eig. unwichtig.

Es geht mir darum, ich würde gerne mal erfahren, diese „reinen Sprachen“ , was ist das(?). Ich habe (mehr oder weniger große) Erfahrung mit so einer art Mix aus Hard-/ und Software (bspw. Raspberry pi, Arduino…). Ich teste mich gerade an Java heran und habe mir auch schon einiges zu Imperativen und logischen Sprachen etc durchgelesen (Ja, auch deinen Blog 😛) Nun die eig. Frage, was ist Scala in dem Sinne, ich weiß es ist eine „Erweiterung“ (oder?) einer Sprache, aber.. Was? Wie? Wo? Ich dachte immer, entweder man schreibt in Java oder c++ (beispiel). Könntest du dazu vllt mal etwas

schreiben 🙂 Oder auch, ein Informatik Studium, wie sieht so eine Bachelor arbeit aus? Auch generell mal so, was ist für dich das programmieren? Wie weit warst du mit 15?

LG,
Triddel02

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=78#RESPOND)

SEB (HTTP://SEBINSIDE.DE)
10. November 2016 um 16:32 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-79)

Der Oberbegriff für diese Konzepte (prozedural, objektorientiert, imperativ, etc.) sind Programmierparadigmen. Diese geben vor, wie Sprachen aufgebaut sind. Eine konkrete Programmiersprache übernimmt dann die Konzepte bzw. Paradigmas, z.B. ist C imperativ oder Java objektorientiert. Bei Scala handelt es sich um eine Hybrid-Sprache: Sie übernimmt sowohl Konzepte der Objektorientierung als auch aus der funktionalen Programmierung.

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=79#RESPOND)

MAXIHUHE04 (@MAXIHUHE04) (HTTP://TWITTER.COM/MAXIHUHE04)
11. Februar 2017 um 10:18 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-84)

Hab mir jetzt alle 8000 Wörter durchgelesen 😉 – WIE geil ist Scala eigentlich? 😀 muss jetzt auf jeden Fall mehr mit Scala machen (war sehr verständlich und gut geschrieben, ich mag den Blog :D)

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=84#RESPOND)

TJARDF
27. Februar 2017 um 17:16 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-85)

hi skate danke für den tipp werde mich mit der Sprache mal ein bisschen in Informatik in der Schule beschäftigen
Tschau

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=85#RESPOND)

JAKOB
28. März 2017 um 22:50 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-86)

Hey Seb!
Du hast mir echt Laune auf diese Sprache gemacht! Hast du ein paar Bücher, Kurse etc die du empfehlen kannst? Oder machst du evtl. sogar selber eine Tutorial Reihe? Würde mich freuen!

Gruß Jakob

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=86#RESPOND)

FABIANFENCING (HTTPS://MYFENCINGBLOGBLOG.WORDPRESS.COM)
23. November 2017 um 22:39 Uhr (http://sebinside.de/2016/10/05/14-coole-scala-features-teil-3-von-3/#comment-128)

Hallo Seb hast du eigentlich schon mal etwas in Swift geschrieben? Und wenn ja was hältst du von Swift?

ANTWORTEN (HTTP://SEBINSIDE.DE/2016/10/05/14-COOLE-SCALA-FEATURES-TEIL-3-VON-3/?REPLYTOCOM=128#RESPOND)

KOMMENTAR VERFASSEN

Gib hier deinen Kommentar ein ...

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

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

SOCIAL MEDIA

(htt p://t witt er.c om/ skat e70 2)

(htt p://f aceb ook. com /seb insid e) 2)

(htt p://y outu be.c om/ skat e70 2)

(htt p://g ithu b.co m/s ebin side)

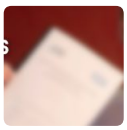
BLOG VIA E-MAIL ABONNIEREN

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

E-Mail-Adresse

ABONNIEREN

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/)

Suche …

NEUESTE KOMMENTARE

- Bowser bei Hallo. äh.. Welt! (http://sebinside.de/2016/03/22/hallo-aeh-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

(htt	(htt	(htt	
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)

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!

Impressum (http://skate702.de/impressum/) © 2016 skate702 – Theme von Colorlib (http://colorlib.com/) Powered by WordPress (http://wordpress.org/)