Seminararbeit

Scala

Sarah Zeitlhofer
Sarah.Zeitlhofer@student.uibk.ac.at

15. Februar 2011

Seminarleiter: Dr. Georg Moser, Dr. René Thiemann
Betreuer: Dr. Christian Sternagel

Vorwort

This paper covers the 2004 released programming language Scala, which combines object-oriented programming with functional programming. The article contains various aspects, initiating the creation of the language, a short information about the creator Martin Odersky, as well as an outlook of Scala's future. Furthermore the basic concepts like objects, constructors, traits, classes, pattern matching, generics, loops and the implicit keyword are covered. Afterwards, an introduction to the usage of the Scala compiler, as well as an explanation how to execute Scala programs, is given. Moreover opinions of professional programmers are mentioned. Within the third chapter of this paper the features of Scala are pointed out, plus a comparison between Scala and the object-oriented programming language Java, as well as the functional programming language OCaml is drawn. Finally, last questions like "Why was Scala created?", "Which language would be the best Scala replacement?" and "Which usually intended goals have been achieved?" are answered.

Inhaltsverzeichnis

1. Geschichte und Ausblick

Der Name Scala steht für skalierbare Sprache (**sca**lable **la**nguage). Scala ist eine Programmiersprache die Objektorientierung und funktionale Programmierung miteinander vereint und wurde mit dem Ziel entworfen, mit den Wünschen der Benutzer mit zu wachsen [1]. In Scala ist jeder Wert ein Objekt und jede Funktion ein Wert. Objekte sind definiert durch Klassen und Funktionen können ineinander verschachtelt sein. Diese zwei Eigenschaften verdeutlichen die Verbindung zwischen Objektorientierung und funktionaler Programmierung [2]. In einem Interview mit Martin Odersky, dem Schöpfer von Scala, erklärte dieser, dass Scala zuerst eine industrielle Sprache sei, da sie von vielen Firmen genutzt, an Universitäten jedoch kaum gelehrt werde [3]. Durch den effizienten Programmierstil, die Flexibilität, sowie die Kompatibilität mit Java, die es ermöglicht Java's Bibliotheken und Frameworks ohne Probleme zu nutzen, ist Scala mächtiger als Java [27]. Scala adaptiert Features von Java und C#, beinhaltet Closures und verwendet Pattern Matching und Traits [4]. Closures sind ein Konzept der funktionalen Programmierung und bezeichnen Programmfunktionen, welche beim Aufruf einen Teil ihres Erstellungskontexts reproduzieren [5]. Ein Trait ist eine Sammlung von Methoden, ähnelt Mixins, und wird als simples, konzeptuelles Model für strukturierte objektorientierte Programmierung verwendet [6]. Die Idee des einheitlichen Objektmodels wurde von Smalltalk übernommen, Scala's abstrakte Typen ähneln OCaml's abstrakten Typen und Views wurden von Haskell's Typklassen beeinflusst [7]. OCaml steht für Objective Caml und eine der bekanntesten ML-Sprachen.

1.1. Geschichte

Ein Mann, eine Vision – Scala entwickelte sich 2001 aus der Programmiersprache Funnel und wurde 2003 erstmals an der École Polytechnique Fédérale de Laussane in der Schweiz verwendet. Funnel ist eine Programmiersprache basierend auf funktionalen Netzen. Sie vereint funktionale Programmierung mit Petri Netzen, um eine generelle und leichte Notation zu ermöglichen [8]. Ein Jahr später wurde Scala veröffentlicht. Entworfen wurde Scala vom Deutschen Martin Odersky. Laut Odersky sind die Schlüsselkonzepte seiner Sprache Skalierbarkeit sowie Vereinigung von Objektorientierung und funktionaler Programmierung [4].

1.2. Martin Odersky

Abbildung 1: Martin Odersky

Martin Odersky wurde am 3. September 1958 in Deutschland geboren. Er ist Professor für Programmiermethodik an der École Polytechnique Fédérale de Laussane. Seine Spezialgebiete sind die Code Analyse sowie Programmiersprachen. 1989 erhielt Odersky seinen "Doctor of Philosophy" von der Eidgenössischen Technischen Hochschule in Zürich. 2008 wurde er zum Fellow der ACM (Association for Computing Machinery). Seine Forschungen werden von der "Swiss National Science Foundation," der Europäischen Kommission und der Hasler Stiftung gesponsert [28].

1.3. Ausblick

Nebenläufige Programmierung beschäftigt jedes Entwicklerteam, da zu keinem Zeitpunkt mit Sicherheit gesagt werden kann, was im Programm gerade vor sich geht. Scala stellt einen sinnvollen und flexiblen Lösungsansatz zur Verfügung, die sogenannten "Actors." Actors sind nebenläufige Prozesse, die durch den Austausch von asynchronen bzw. synchronen Nachrichten miteinander kommunizieren. Jedoch sind Actors nicht der einzige Weg um das Problem der Nebenläufigkeit in den Griff zu bekommen [9]. Weitere Konzepte sind die Benutzung von Transaktionen sowie Join Pattern, welche die nebenläufigen Funktionen des Join-Calculus sind [10]. Da Scala kompatibel zu Java ist, können auch alle in Java bekannten Lösungskonzepte verwendet werden.

1.3.1. Ist Scala eine esoterische Sprache?

Esoterische Sprachen wurden entworfen um die Grenzen der Programmierung zu testen und kennzeichnen sich dadurch, dass sie in der Programmierwelt keinen Gebrauch finden. Beispiele für bekannte esoterische Sprachen sind Brainfuck oder LOLCODE [11]. Da Scala im industriellen Raum bereits weit verbreitet ist, gehört die Sprache nicht zu den esoterischen Sprachen.

1.3.2. Gehören Features von Scala zum Programmierstandard?

Scala übernimmt den Großteil seiner Features von anderen Programmiersprachen wie Java, C#, OCaml, Haskell sowie Smalltalk. Dadurch sind die meisten Features bereits im Programmierstandard enthalten. Allgemein versteht man unter einem Programmierstandard bzw. Programmierstil ein Set von Regeln und Richtlinien, die beim Schreiben von Code verwendet werden sollen. Ein guter Stil erleichtert die Lesbarkeit sowie Verständlichkeit fremden Codes. Programmierstandards helfen somit die Fehleranfälligkeit zu reduzieren. Jede Sprache

hat ihre eigenen Regeln, welche bei der Entwicklung von der Firma oder Organisation festgelegt werden. Programmierstandards werden meistens für eine spezifische Programmiersprache entwickelt, hin und wieder auch für eine Familie von Sprachen [12].

1.3.3. Schwierigkeiten bei der Programmierung mit Scala

Aus meiner Sicht liegen die Schwierigkeiten für geübte Java-Programmierer im funktionalen Aspekt der Sprache. Funktionale Programmierung arbeitet mit Rekursionen, d.h. Funktionen rufen sich selbst rekursiv auf. Für mich liegt eine weitere Herausforderung in der Reichhaltigkeit der Sprache. Dadurch das Scala objektorientierte und funktionale Programmierung vereint, stehen dem Programmierer eine Vielfalt von Konzepten zur Verfügung, welche bei richtiger Anwendung die Effizienz des Programms steigern.

2. Wie wird Scala verwendet?

2.1. Basiskonzepte

2.1.1. Übersetzung & Ausführung

Scala Programme werden vom *Scala Compiler* übersetzt. Als Input wird eine *.scala* Datei benötigt, die durch den Aufruf `scalac filename.scala` in Bytecode übersetzt wird. Der Dateiname des Bytecodes trägt den Namen jener `object` Klasseninstanz, welche die main Methode beinhaltet. Die lexikalische Analyse ist der erste Schritt zur Übersetzung des Quellcodes in Bytecode. Scala's Scanner verwendet zur Analyse von Tokens das "longest matching rule" Verfahren, welches die längstmögliche Zeichenkette im Input konsumiert. Mit Hilfe des Befehls `scala bytecode_filename` wird aus dem Bytecode lauffähiger Code generiert [29].

Beispiel: Es existiert eine Datei mit dem Namen *HelloWorld.scala*. Um dieses Programm kompilieren zu können werden folgende Aufrufe getätigt:

- `$ scalac HelloWorld.scala`
- `$ scala HelloWorld`

2.1.2. Interaktion mit Java

Eine von Scala's Stärken ist die vollständige Kompatibilität mit Java, welche alle `java.lang` Klassen dem Scala Code automatisch zur Verfügung stellt. Andere Java Klassen müssen explizit, mittels `import`-Befehl, importiert werden. Der `import`-Befehl in Scala ist mächtiger als jener in Java und benutzt anstelle des Sterns `*` für den Mehrfachimport den Unterstrich `_`. Außerdem können mehrere bestimmte Klassen mittels geschwungener Klammern in der selben Zeile angegeben werden. Weiters ist anzumerken, dass in Scala direkt von Java Klassen geerbt werden kann und dass Java Interfaces direkt implementiert werden können [29].

```
/*importiert Iterator Klasse*/
import java.util.Iterator

/*importiert List und Map Klasse*/
import java.util.{List,Map}

/*importiert alle Klassen im java.util Packet*/
import java.util._
```

Listing 1: import Befehle

2.1.3. Objekte

Im Gegensatz zu Java, das zwischen primitiven Datentypen und Referenztypen unterscheidet, ist in Scala jeder Wert, sowie jede Funktion ein Objekt. Scala erlaubt es Funktionen als Argument zu übergeben, in Variablen zu speichern und als Rückgabetyp anderer Funktionen zu verwenden. Diese Fähigkeit Funktionen zu manipulieren ist ein Grundbaustein der funktionalen Programmierung. Ein weiteres Feature von Scala sind anonyme Funktionen. Diese ermöglichen es Funktionen ohne Namen zu definieren, womit die Lesbarkeit des Programms erleichtert wird. Anonyme Funktionen werden mittels Rechtspfeil => gekennzeichnet [29]. Definitionen sowie Deklarationen von Feldern beginnen in Scala mit einem reservierten Schlüsselwort. Variable Definitionen verwenden `var`, Wert-Definitionen beginnen mit `val` und Funktionen mit `def`. Das Schlüsselwort `val` kennzeichnet, dass auf die Variable nur Lesezugriffe verfügbar sind, `var` hingegen erlaubt zusätzlich Schreibzugriffe.

2.1.4. Konstruktoren

Manchmal benötigt eine Klasse mehrere Konstruktoren, welche in Scala als Hilfskonstruktoren bezeichnet werden. Diese zusätzlichen Konstruktoren kennzeichnen sich mittels den Schlüsselwörtern `def this (...)`. Im Rumpf eines Hilfskonstruktors muss zuerst entweder der Hauptkonstruktor oder ein anderer Hilfskonstruktor mittels `this(...)` aufgerufen werden. Durch diesen Netzeffekt ruft jeder Hilfskonstruktor schlussendlich den Hauptkonstruktor auf. In Scala ist es nur dem Hauptkonstruktor, der den einzigen Einstiegspunkt in die Klasse bereitstellt, möglich, den Superklassenkonstruktor aufzurufen [28]. In Scala können beliebig viele Konstruktoren definiert werden, insofern diese sich in ihrer Signatur unterscheiden.

```
class Pair(name: String, age: Int) {          /*Hauptkonstruktor*/
  def this(age: Int) = this("unknown", age)   /*Hilfskonstruktor*/
  def namePair = name
  def agePair = age
}

object PairTest {
  def main(args: Array[String]) {

    /*Aufruf des Hauptkonstruktors*/
    val p0 = new Pair("Sarah",21)
    println("Name: " + p0.namePair + " || Alter: " + p0.agePair)

    /*Aufruf des Hilfskonstruktors*/
    val p1 = new Pair(21)
    println("Name: " + p1.namePair + " || Alter: " + p1.agePair)
  }
}
```

Listing 2: Übersetzung mit: $ `scalac PairTest.scala` || $ `scala PairTest`

2.1.5. Traits

Traits können als Interfaces betrachtet werden, die nicht nur Methodenspezifikationen enthalten können, sondern auch deren Implementierung. Eine Scala Klasse kann den Code von mehreren Traits mittels des Schlüsselwortes `with` importieren. Am Besten sind Traits anhand eines Beispiels zu verstehen:

Beispiel: Der Dichter James Whitcomb Riley widmete im 19. Jahrhundert dem Ententest ein Gedicht: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck [13]." Eine Ente gehört zur Familie der Vögel, kann jedoch schwimmen wie ein Fisch. Wenn wir nun eine Klasse Ente definieren, soll diese sowohl die Eigenschaften eines Vogels, als auch eines Fisches aufweisen. In Java ist dies nicht möglich, da es keine Mehrfachvererbung gibt. Auch in Scala gibt es keine Mehrfachvererbung, jedoch erzielen Traits den Effekt der Mehrfachvererbung.

```
/*Superklasse Tier*/
class Tier(name: String)

/* Die Klasse Fisch erbt von der Superklasse Tier; *
 * jede Instanz dieser Klasse kann schwimmen       */
class Fisch(name: String) extends Tier(name) with swim

/* Die Klasse Vogel erbt von der Superklasse Tier; *
 * jede Instanz dieser Klasse kann fliege          */
class Vogel(name: String) extends Tier(name) with fly

/*Traits*/
trait swim { def swim = "schwimmt" }
trait fly { def fly = "fliegt" }

/**
 * Klasse Ente erbt von Vogel, weiters kann jede Instanz der
 * Klasse Vogel fliegen, sowie durch den mit dem Schluesselwort
 * with inkludierten Trait schwimmen
 */
class Ente(name: String) extends Vogel(name) with swim {
    override
    def toString = name
}

/*Testklasse*/
object Test {
    def main(args: Array[String]) {
        val e = new Ente("Ente")
        println(e.toString() + "␣" + e.swim + "␣und␣" + e.fly)
    }
}
```

Listing 3: Übersetzung mit: $ `scalac Traits.scala` || $ `scala Test`

2.1.6. Klassen

Klassen in Scala gleichen der Syntax von Klassen in Java. Ein grundlegender Unterschied ist jedoch, dass Klassen in Scala Parameter haben können. Diese Parameter müssen beim Erzeugen einer neuen Klasseninstanz übergeben werden. Ein weiterer Unterschied liegt darin, dass die main Methode sich in einer mit `object` gekennzeichneten Klasseninstanz befinden muss. Der Begriff `object` kennzeichnet die Klasseninstanz als Singleton Objekt (auch Modul genannt), d.h. es ist nur möglich eine Instanz dieser Klasse zu erzeugen. Rückgabewerte werden automatisch vom Compiler eruiert [29].

```
class Pair(name: String, age: Int) {
  def namePair() = name /*Funktionen muessen deklariert werden,  */
  def agePair() = age   /*um spaeter aufgerufen werden zu koennen*/
}

/**
 * Jene Klasse die die main-Methode enthaelt muss als object
 * gekennzeichnet sein. Die object Deklaration kennzeichnet,
 * dass die Klasse als Singleton Object deklariert ist.
 */
object PairTest {
  def main(args: Array[String]) { /*main Methode wird benoetigt,*/
                                  /*um das Programm ausfuehren  */
                                  /*zu koennen                  */
    val p = new Pair("Sarah",21)
    println("Name: " + p.namePair())
  }
}
```

Listing 4: Übersetzung mit: $ `scalac PairTest.scala` || $ `scala PairTest`

Scala ermöglicht es Methoden als Felder zu nutzen. Der oben abgebildete Code schaut nun folgendermaßen aus:

```
class Pair(name: String, age: Int) {
  /*Methoden werden nun als Felder deklariert*/
  def namePair = name
  def agePair = age
}

object PairTest {
  def main(args: Array[String]) {
    val p = new Pair("Sarah",21)
    println("Name: " + p.namePair) /*Aufruf des Feldes*/
  }
}
```

Listing 5: Übersetzung mit: $ `scalac PairTest.scala` || $ `scala PairTest`

Wie auch in Java erbt jede Klasse von einer Superklasse. Ist diese Superklasse nicht explizit deklariert, wird die Default-Klasse `scala.AnyRef` als Superklasse verwendet. Methoden der Superklasse dürfen von den Subklassen überschrieben werden, indem diese explizit durch das Schlüsselwort `override` gekennzeichnet werden.

```
class Pair(name: String, age: Int) {
 def namePair = name
 def agePair = age

 /**
  * Methode der Superklasse wird ueberschrieben,
  * gekennzeichnet durch das Schluesselwort override
  */
 override
 def toString() = "Name: " + name + " || Age: " + age
}

object PairTest {
 def main(args: Array[String]) {
   val p = new Pair("Sarah",21)
   println(p.toString())/*Ueberschriebene Methode wird aufgerufen*/
 }
}
```

Listing 6: Übersetzung mit: $ `scalac PairTest.scala` || $ `scala PairTest`

2.1.7. Pattern Matching

Scala hat einen eingebauten Pattern Matching Mechanismus, welcher erlaubt, jegliche Art von Daten mit Hilfe des Schlüsselwortes `match` und der "First Match" Strategie zu matchen [14]. Ein Pattern Match beinhaltet eine Sequenz von Alternativen, ähnlich wie bei Java's `switch`-Statement. Jede dieser Alternativen beginnt mit dem `case`-Schlüsselwort.

```
def guessWhat(anything: Any) = anything match {
   case name: String  => "String"
   case age: Int      => "Integer"
   case _             => "Something else"
}
```

Listing 7: simples Pattern Matching Beispiel

Case Classes

Pattern Matching wird vor allem im Zusammenhang mit Case Classes, die durch die Schlüsselwörter `case class` identifiziert werden, verwendet. Scala erlaubt jedoch auch Pattern Matching ohne die Verwendung von Case Classes. Case Classes sind reguläre Klassen, die zusätzliche Funktionalitäten zur Verfügung

stellen. Im Gegensatz zu normalen Klassen sind alle dem Konstruktor übergebenen Argumente von außen zugänglich. Im folgenden Code-Beispiel hat die Subklasse *Fisch* einen `case` Modifikator.

```
scala> abstract class Tier
/*defined class Tier*/

scala> case class Fisch(name:String) extends Tier
/*defined class Fisch*/
```

Ein Vorteil von Case Classes ist die Bereitstellung der Factory Methode. Diese ermöglicht es Instanzen einer Klasse ohne dem Schlüsselwort `new` anzulegen.

```
scala> val forelle = Fisch("Forelle")
/*forelle: Fisch = Fisch(Forelle)*/
```

Ein weiterer Vorteil liegt in der Parametrisierung der Argumente. Alle Argumente der Parameterliste einer Case Class werden implizit mit dem Schlüsselwort `val` versehen. Diese Argumente werden als Felder betrachtet.

```
scala> forelle.name
/*res0: String = Forelle*/
```

Zusätzlich stellt der Compiler die vordefinierten Methoden `toString`, `hashCode` sowie `equals` der Case Class zur Verfügung. `toString` ermöglicht den Namen der Klasse und deren Argumente auszugeben, `equals` dient zum strukturellen Vergleich von zwei Instanzen der selben Klasse und `hashCode` bietet die Verwendung der Hash-Codes von Konstruktor-Argumenten an [15].

```
scala> println(forelle)
/*Fisch(Forelle)*/

scala> forelle.name == Fisch("Forelle")
/*res1: Boolean = false*/
```

Verschiedene Arten von Pattern

Wildcard Pattern

Ein Feature von Pattern Matching ist der durch einen Unterstrich dargestellte Wildcard. Dieser erlaubt es jeden beliebigen Wert zu matchen und wird dazu benötigt, alle möglichen Fälle abzufangen. Ansonsten kann es zu einem `scala.MatchError` kommen [29].

```
case _ => "irgendein␣Ausdruck"
```

Variable Pattern

Variable Pattern gleichen dem Wildcard Pattern, da sie ebenso jeden beliebigen Wert matchen. Der Unterschied zum Wildcard liegt jedoch darin, dass Scala die Variable zum gematchten Objekt binden kann.

```
case someExp => "irgendein Ausdruck => exp: " + someExp
```

Pattern Matching mit Konstanten

Konstanten matchen nur sich selbst. Jedes Literal, jedes Singleton Objekt und jedes mit `val` gekennzeichnete Feld kann als Konstante verwendet werden. Das Singleton Objekt Nil matcht nur die leere Liste.

```
def konstantenMatching(x: Any) = x match {
   case Nil   => "leere Liste"
   case false => "Boolean Konstante false"
   case 3     => "Zahl 3"
   case _     => "irgendein Ausdruck"
}
```

Listing 8: Pattern Matching mit Konstanten

Sequenz Pattern

Sequenz Pattern ermöglichen es Sequenzen zu matchen. Durch den Wildcard mit nachfolgendem Stern können beliebig viele Argumente in einer Sequenz gematcht werden.

```
def sequenzMatching(x: Any) = x match {
   case List(_,_,_) => "Liste mit 3 Argumenten"
   case Map(_*)     => "Map mit beliebig vielen Argumenten"
   case _           => "irgendein Ausdruck"
}
```

Listing 9: Pattern Matching von Sequenzen

Pattern Guards

Auf jede `case`-Expression kann ein Guard folgen, welcher immer durch eine `if`-Abfrage gekennzeichnet ist. Gibt es einen Guard, so kommt die nachfolgende Match-Regel nur zum Einsatz, wenn die im Guard-Statement definierte Bedingung wahr ist.

```
def matchingMitGuard(x:Int)(y:Int) = x match {
   case x if x > y => x
   case _          => y
}
```

Listing 10: Pattern Matching mit Guards

2.1.8. Generizität

Generische Programmierung ist ein Programmierstil, der es erlaubt, einen Teil des Programms (z.B. eine Klasse, ein Interface oder eine Methode) mit Typen zu parametrisieren. Scala ermöglicht es generische Klassen und Methoden zu definieren. Der Unterschied zu Java liegt darin, dass einer generischen Klasse zusätzlich eine Typvariable in eckigen, anstatt in spitzen Klammern nachgestellt wird. Gleich wie in Java kann diese Typvariable im Rumpf als normaler Typ verwendet werden.

```
class Pair[T](name: T, age: Int) {
  private var namePairTemp: T = name /*generische Variable*/
  def namePair: T = namePairTemp
  def agePair = age
}

object PairTest {
  def main(args: Array[String]) {
    val p = new Pair("Sarah",21)
    println("Name: " + p.namePair)
  }
}
```

Listing 11: Übersetzung mit: $ `scalac PairTest.scala` || $ `scala PairTest`

2.1.9. Schleifen

WHILE

Scala's `while`-Schleife verhält sich gleich wie in anderen Sprachen. Der Methodenrumpf einer `while`-Schleife wird solange ausgeführt, bis die Schleifen-Bedingung, welche in runden Klammern neben dem Schlüsselwort `while` gegeben ist, nicht mehr erfüllt wird.

while (*Bedingung*) { *Anweisungsfolge* }

```
var i = 0;
while ( i < 5 ) { println(i); i += 1 }
```

DO-WHILE

Ebenso wie in Java gibt es auch in Scala die `do-while`-Schleife. Diese verhält sich ähnlich wie die `while`-Schleife, mit dem Unterschied, dass die Bedingung erst nach einem Durchlauf der Schleife getestet wird.

do { *Anweisungsfolge* } **while** (*Bedingung*)

```
var i = 0;
do { println(i); i += 1 } while ( i < 5 )
```

FOR

Die `for`-Schleife zählt in Scala eigentlich nicht zu den Schleifenkonstrukten, sondern wird als Ausdruck bezeichnet. `for`-Ausdrücke werden vorallem zur Iteration über den Inhalt von Collections verwendet und liefern im Gegensatz zu normalen Schleifen "interessante" Ergebnisse zurück.

for (*Bedingung*) { *Anweisungsfolge* }

```
for ( i <- 0 to 4) { println(i) }
```

Die Bedingung dieses `for`-Ausdrucks besagt, dass die Anweisung für alle Elemente von 0 bis 4 ausgeführt werden soll und kann als i in 0 bis inklusive 4 gelesen werden.

FOREACH

Ebenso wie der `for`-Ausdruck wird auch die `foreach`-Schleife als Ausdruck bezeichnet. Der `foreach`-Ausdruck wird ebenfalls zur Iteration über Collections verwendet.

```
val list = 0::1::2::3::4::Nil
list.foreach ( println )
```

2.1.10. `implicit` Schlüsselwort

Das Schlüsselwort `implicit` unterstützt sowohl die implizite Typkonvertierung als auch die Wiederverwendung von gleichen Variablen bei Funktionsaufrufen.

Implizite Typkonvertierung: Im folgenden Code-Beispiel wird ein schreibgeschütztes Feld *name* vom Typ `java.lang.String` definiert. Anschließend wird auf diesen String die vordefinierte Funktion `capitalize` aufgerufen, welche den ersten Buchstaben des Strings großschreibt. Die Funktion `capitalize` ist jedoch nicht für den Typ `java.lang.String` definiert.

```
val name = "sarah"
println(name.capitalize)
```

Listing 12: Implizite Typkonvertierung

Nun stellt sich die Frage, warum funktioniert der Aufruf?
Der Aufruf funktioniert, weil die Scala Bibliothek eine "Wrapper" Klasse beinhaltet, welche den Namen `scala.runtime.RichString` trägt. Durch den Compiler wird der String *name* automatisch durch einen `stringWrapper` in einen `scala.runtime.RichString` umgewandelt.

```
implicit def stringWrapper(x:String)=new runtime.RichString(x)
```

Durch das `implicit`-Schlüsselwort wird dem Compiler mitgeteilt, dass diese Funktion für eine implizite Umwandlung von einem String in einen RichString verwendet werden kann [28].

Implizite Funktionsparameter: Im nachfolgenden Code-Beispiel werden mittels `implicit`-Schlüsselwort Default-Werte unterstützt. Die Funktion *printing* definiert im Methodenkopf zwei voneinander getrennte Übergabeparameter, wobei der Zweite optional ist. Das `implicit`-Schlüsselwort informiert den Compiler, dass dieser den Wert des Feldes *name* im umliegenden Gültigkeitsbereich suchen soll, falls kein expliziter zweiter Übergabeparameter mitgegeben wurde.

```
implicit val name = "unknown"

def printing(age: Int)(implicit name: String) {
   println("Name: " + name + " | | Alter: " + age)
}

printing(21)("Sarah") /* default Aufruf   */
printing(21)          /* implicit Aufruf */
```

Listing 13: Verwendung von impliziten Funktionsparametern

Implizite Funktionsparameter verhalten sich wie Parameter mit einem Default-Wert, die jedoch den Wert im umliegenden Gültigkeitsbereich suchen müssen. Falls der Compiler keinen Default-Wert findet, wird ein `no implicit argument matching parameter` Fehler geworfen [28].

2.2. Wie denken professionelle Programmierer über Scala?

Laut Ted Neward, unabhängiger Softwareentwickler und Autor der Bücher "Effective Enterprise Java (Addison-Wesley)" und "Java Programming (Manning)," sowie Co-Autor von "C# In a Nutshell (O'Reilly)" [16], sollte jeder Scala lernen [27]. Heiko Seeberger, geschäftsführender Gesellschafter der Weigle Wilczek GmbH, sowie Jan Blankenhorn, Softwareentwickler bei Weigle Wilczek GmbH, schließen sich der Meinung von Ted Neward an und meinen, dass jeder Entwickler ohnehin jedes Jahr eine neue Programmiersprache lernen sollte [27]. Der Schöpfer von Scala, Martin Odersky, meinte im Interview mit Bill Venners, dem Präsidenten der Artima Software Firma [28], dass Scala im Moment keine Sprache für den durchschnittlichen Programmierer ist. Experten sollen mit Scala gefordert werden und produktiver damit arbeiten, als mit Java [17].

3. Scala's Features im Vergleich mit ...

Die übergangslose Zusammenarbeit zwischen Scala und den objektorientierten Programmiersprachen Java und C# war eines der Ziele bei der Entwicklung der Sprache. Scala übernimmt eine Vielfalt von Java's Features und adaptiert ein einheitliches Objektmodel in dem Sinn, dass jeder Wert ein Objekt und jede Operation ein Methodenaufruf ist [7]. Auf das Semikolon, das in vielen anderen Sprachen Voraussetzung ist, kann in Scala verzichtet werden. Werden jedoch in einer Zeile mehrere Anweisungen aneinandergereiht, so müssen diese mit einem Semikolon voneinander getrennt werden.

Die bekanntesten Features von Scala sind:

- **Actors:** Wie bereits im Abschnitt 1.3 erklärt wurde, unterstützen Actors die Parallelität von Prozessen. Zwischen den einzelnen Akteuren werden Nachrichten übertragen und reihenfolgeerhaltend abgearbeitet. Die Verarbeitung der Nachrichten von verschiedenen Akteuren kann gleichzeitig stattfinden. Der Vorteil im Actor-Konzept von Scala liegt darin, dass keine Locks verwendet werden müssen [18], wodurch die Fehlerrate verringert wird.
- **Pattern Matching:** Pattern Matching ist ein mächtiges Feature von funktionalen Sprachen. Für nähere Informationen siehe Abschnitt 2.1.7.
- **Endrekursionen:** Endrekursionen sind eine spezielle Art von Rekursionen. Eine Funktion ist dann endrekursiv, wenn der letzte Aufruf im Funktionsrumpf der rekursive Aufruf der Funktion ist [26].
- **XML Modus:** Scala ermöglicht die einfache Erstellung und Verarbeitung von XML Dokumenten. Durch die direkte Sprachunterstützung können XML Fragmente im Scala Code verwendet werden [19]. Wenn eine geöffnete spitze Klammer im Code gefunden wird, muss die lexikalische Analyse vom Scala Modus in den XML Modus wechseln. Der Scanner wechselt vom XML Modus nur dann in den Scala Modus zurück, wenn entweder eine mit einer geöffneten spitzen Klammer beginnende Anweisung erfolgreich geparst wurde, oder der Parser auf einen integrierten Scala Befehl oder ein Scala Pattern trifft [25].

3.1. JAVA

Scala gleicht der Programmiersprache Java in vielerlei Hinsicht und kann nahtlos mit dieser zusammen verwendet werden. Scala ist keine Obermenge von Java, d.h. einige Eigenschaften wurden weggelassen und wieder andere wurden neu interpretiert. Scala betrachtet jede Java Klasse als zwei Entitäten. Die erste Entität enthält alle dynamischen Mitglieder, die zweite enthält ein Singleton-Objekt, dem alle statische Mitglieder zugeteilt sind [7]. Da beide Sprachen statisch typisiert sind, können Fehler früher erkannt werden.

```scala
/**
 * Uebersetzung:
 *    $ scalac PairTest.scala
 * Ausfuehrung:
 *    $ scala PairTest
 *
 * Ausgabe:
 * -----------
 * Name: Sarah
 */

class Pair(name:String,age:Int){
  def namePair() = name
  def agePair() = age
}


object PairTest {
def main(args:Array[String]){
   val p = new Pair("Sarah",21)
   println("Name: " +
            p.namePair())
}
}
```

```java
/**
 * Uebersetzung:
 *    $ javac PairTest.java
 * Ausfuehrung:
 *    $ java PairTest
 *
 * Ausgabe:
 * -----------
 * Name: Sarah
 */

class Pair{
  private String name;
  private int age;

  public Pair(String name, int age){
    this.name = name;
    this.age = age;
  }

  public String namePair(){
    return name;
  }

  public int agePair(){
    return age;
  }
}

public class PairTest{
public static void main(String[]args){
    Pair p = new Pair("Sarah",21);
    System.out.println("Name: " +
                    p.namePair());
}
}
```

Listing 14: Scala - Java Vergleich

Wie im obigen Vergleichsbeispiel deutlich zu erkennen ist, enthält der in Scala geschriebene Code weniger Zeilen, als jener in Java, obwohl beide die gleiche Ausgabe liefern. Die Test Klassen sind sowohl in Java, als auch in Scala größtenteils gleich. Ein wesentlicher Unterschied liegt jedoch in Zeile 21 im Code Beispiel. In Java muss der Programmierer den Typ von p definieren. In Scala hingegen wird dieser automatisch durch Typinferenz ermittelt. Ein weiterer kleiner Unterschied liegt in der Definition der Klassen (siehe Zeile 19). In Java wird eine normale Klasse erzeugt, in Scala jedoch nur ein Singleton-Objekt. Die Klasse Pair weist mehrere Unterschiede auf. In Java ist es nicht möglich einer Klasse Übergabeparameter mitzugeben. Deshalb muss ein Konstruktor definiert werden, welchem die Übergabeparameter mitgegeben werden. In Scala erfolgt dies bereits bei der Klassendeklaration. Weiters müssen in Java explizit die Methoden mit Rückgabetyp und Sichtbarkeit definiert werden. Im Scala Code erfolgt die Definition der Methode in nur einer Zeile (siehe Zeile 2 oder Zeile 3). Der Rückgabetyp der Methode wird automatisch ermittelt.

3.2. OCaml

In Scala ist jede Funktion ein Wert, wodurch sich der funktionale Aspekt der Sprache auszeichnet. Scala stellt eine leichte Syntax für anonyme Funktionen zur Verfügung, unterstützt Higher-Order Funktionen, erlaubt verschachtelte Funktionen und fördert Currying. Currying ist eine Technik zur Transformation von Funktionen mit mehreren Argumenten in Funktionen mit nur einem Argument. Argumente die weiter benötigt werden, werden durch eine andere Funktion aufgerufen [20]. Scala ist hervorragend zur Entwicklung von Web Services geeignet, da XML Code im Scala Code integriert werden kann [21].

- **Anonyme Funktionen** besitzen keine Namen. Aus diesem Grund können sie nicht beim Namen aufgerufen werden, sondern müssen direkt verwendet werden [22].
  - in Scala:

```
scala> List("Sarah",21) filter (_.isInstanceOf[String])
/*res0: List[Any] = List(Sarah)*/
```

  - in OCaml:

```
# let printName = fun x -> "Name: "^x;;
(* val printName : string -> string = <fun> *)

# printName "Sarah";;
(*- : string = "Name: Sarah"*)
```

- **Methodenaufrufe** sind in beiden Sprachen ziemlich ähnlich. Der einzige Unterschied liegt im Aufruf selbst. In Scala müssen Übergabeparameter immer in runden Klammern übergeben werden. In OCaml reicht es aus, wenn Übergabeparameter durch ein Leerzeichen vom Funktionsnamen getrennt werden.
  - in Scala:

```
scala> def printName(x:String) = x
/*printName: (String)String*/

scala> printName("Sarah")
/*res3: String = Sarah*/
```

  - in OCaml:

```
# let printName x = x;;
(*val printName : 'a -> 'a = <fun>*)

# printName "Sarah";;
(*- : string = "Sarah"*)
```

4. Warum wurde Scala kreiert?

Laut Odersky ist Scala eine gute Basis für nebenläufige und parallele Programmierung. Scala läuft ohne Probleme auf der JVM, unterstützt funktionale Programmierung und hat eine große syntaktische Flexibilität [4].

4.1. Geeignete Scala Alternativen

Als Scala Ersatz wäre definitiv Java am Besten geeignet, da Scala auf Java aufbaut. Der Vorteil liegt darin, dass dem Scala Programmierer die Grundkonzepte von Java bekannt sind.

4.2. Wie lautete das Ziel von Scala?

Oderskys Ziele bei der Entwicklung von Scala waren breit gefächert. Er wollte eine in die Java Infrastruktur integrierte Sprache entwerfen, die rein objektbasiert sein sollte. Jeder Wert sollte ein Objekt sein und jede Operation ein Methodenaufruf. Weiters wollte er eine reibungslose Integration von funktionalen Komponenten, wie Generizität oder Pattern Matching, erstklassige Funktionen, Funktionsliterale und Closures[1]. Außerdem wollte Odersky in seiner Sprache keine statischen Komponenten verwenden [23].

4.2.1. Wurde das Ziel erreicht?

Größtenteils konnte Odersky seine Konzepte umsetzen. Allerdings musste er auch hin und wieder Kompromisse eingehen, um die völlige Kompatibilität mit Java gewährleisten zu können. Anstelle der statischen Komponenten werden Singleton-Objekte verwendet. Weiters wird immer wieder kritisiert, dass Scala sowohl Traits als auch Klassen verwendet. Würde man dieses Design anpassen, so wäre es möglich ohne Klassen auszukommen, allerdings würde dies die Kompatibilität mit Java beeinträchtigen. Ein dritter Kompromiss musste im Bezug auf die Konstante *null* gemacht werden. Da *null* viele Probleme bereitet, wollte man die Verwendung dieser Konstante unterbinden. Leider kommt es auch hierbei zum Konflikt mit Java. Java's Bibliotheken haben vielfach *null* als Rückgabewert. Würde nun die *null* Konstante verschwinden, so wäre wiederum die Austauschbarkeit mit Java betroffen.

[1] siehe Abschnitt 1

Zusammenfassung

Die 2004 vom Deutschen Martin Odersky ins Leben gerufene Programmiersprache Scala vereinigt sowohl funktionale als auch objektorientierte Programmierung. Eine der Stärken von Scala ist ihre Skalierbarkeit, sowie die vollständige Kompatibilität mit Java. Zur Übersetzung von Scala Dateien steht der Scala Compiler zur Verfügung. Dieser produziert den Bytecode, welcher der Java Virtual Maschine übergeben wird.
Da Scala vollständig kompatibel mit Java ist, stehen die meisten `java.lang` Klassen einem Scala Programm zur Verfügung. Andere Java Klassen können an Scala Programme vererbt und Java Interfaces direkt implementiert werden. In Scala ist jeder Wert und jede Funktion ein Objekt, statische Felder bzw. Methoden existieren nicht und Funktionen können als Argumente übergeben werden. Außerdem existieren in Scala Traits, welche zur Definition von Klasseneigenschaften verwendet werden. Klassen gleichen der Syntax von Java Klassen, mit dem Unterschied, dass es in Scala `object` Klasseninstanzen gibt. Diese kennzeichnen die Klasse als Singleton-Objekt. Pattern Matching erlaubt es Daten mit Hilfe der "First Match" Strategie zu matchen und kennzeichnet damit den funktionalen Aspekt der Sprache. Ebenso ist die generische Programmierung in Scala verfügbar. Diese erlaubt es einen Teil des Codes mit Typen zu parametrisieren.
Scala stellt eine gute Basis für nebenläufige Programmierung zur Verfügung. Diese Basis beruht auf Actors, welche durch den Austausch von Nachrichten miteinander kommunizieren. Laut professionellen Programmierern sollte jeder Scala lernen, denn Scala ermöglicht es guten Programmierern Konzepte von funktionaler, sowie objektorientierter Programmierung bestmöglich zu verbinden.

Literatur

[1] Website. http://en.wikipedia.org/wiki/Scala_(programming_language); besucht am 20. Dezember 2010.

[2] Website. http://www.artima.com/weblogs/viewpost.jsp?thread=163733; besucht am 10. Februar 2011.

[3] Website. http://flyingfrogblog.blogspot.com/2010/08/scala-is-foremost-industrial-language.html; besucht am 20. Dezember 2010.

[4] Website. http://www.scala-lang.org/sites/default/files/odersky/scalaliftoff2009.pdf; besucht am 20. Dezember 2010.

[5] Website. http://de.wikipedia.org/wiki/Closure; besucht am 28. Dezember 2010.

[6] Website. http://en.wikipedia.org/wiki/Trait_(computer_programming); besucht am 8. Januar 2011.

[7] Website. http://www.scala-lang.org/docu/files/ScalaOverview.pdf; besucht am 21. Dezember 2010.

[8] Website. http://lamp.epfl.ch/funnel/; besucht am 20. Dezember 2010.

[9] Website. http://programming-scala.labs.oreilly.com/ch09.html; besucht am 28. Dezember 2010.

[10] Website. http://lamp.epfl.ch/~cremet/join_in_scala/; besucht am 28. Dezember 2010.

[11] Website. http://en.wikipedia.org/wiki/Esoteric_programming_language; besucht am 28. Dezember 2010.

[12] Website. http://en.wikipedia.org/wiki/Programming_style; besucht am 6. Januar 2011.

[13] Website. http://de.wikipedia.org/wiki/Ententest; besucht am 2. Januar 2011.

[14] Website. http://www.scala-lang.org/node/120; besucht am 23. Dezember 2010.

[15] Website. http://www.scala-lang.org/node/258; besucht am 13. Januar 2011.

[16] Website. http://www.oreillynet.com/pub/au/845; besucht am 27. Dezember 2010.

[17] Website. http://www.artima.com/forums/flat.jsp?forum=270&thread=306728; besucht am 2. Januar 2011.

[18] Website. http://www.ansorg-it.com/de/scalanews-002.html; besucht am 12. Januar 2011.

[19] Website. http://www.scala-lang.org/node/131; besucht am 11. Januar 2011.

[20] Website. http://en.wikipedia.org/wiki/Currying; besucht am 13. Januar 2011.

[21] Website. http://www.scala-lang.org/node/25; besucht am 13. Januar 2011.

[22] Website. http://www.csc.villanova.edu/~dmatusze/resources/ocaml/ocaml.html#Anonymous%20functions; besucht am 11. Januar 2011.

[23] Website. http://www.artima.com/scalazine/articles/goals_of_scala.html; besucht am 4. Januar 2011.

[24] Website. http://99-bottles-of-beer.net/language-scala-2179.html; besucht am 20. November 2010.

[25] M. Odersky. The Scala Language Specification. Switzerland, 2010. Version 2.8, Kapitel 1.5; verfügbar online unter http://www.scala-lang.org/docu/files/ScalaReference.pdf; besucht am 11. Januar 2011.

[26] C. Sternagel. Functional Programming. Universität Innsbruck, 2009. Foliensatz 8, Folie 12; verfügbar online unter http://cl-informatik.uibk.ac.at/teaching/ws09/fp/ohp/week08-1x4.pdf; besucht am 8. Januar 2011.

[27] H. S. und Jan Blankenhorn. Einstieg, 2010. Scala, Teil 1; verfügbar online unter http://www.weiglewilczek.com/fileadmin/Publications/Einstieg_in_Scala_1__JS_02_10.pdf; besucht am 20. November 2010.

[28] M. O. und Lex Spoon und Bill Venners. Programming in Scala. Artima Press, 2008. Erste Edition, Version 5; A comprehensive step-by-step guide.

[29] M. S. und Philipp Haller. A Scala Tutorial. Switzerland, 2010. For Java Programmers, Version 1.3; verfügbar online unter http://www.scala-lang.org/docu/files/ScalaTutorial.pdf; besucht am 20. November 2010.

A. Implementierung von “99 bottles of beer on the wall”

```
/**
 * Scala implementation of "99 beers on the wall"
 * with a vengeance. Contains examples of Scala
 * features, like tail recursion.
 *
 * @author Eduardo Costa http://extremejava.tpk.com.br
 */

object Beers extends Application {

    def bottles(qty : Int, f : => String) = //higher-order function
      qty match {
        case 0 => "no more bottles of beer" + f
        case 1 => "1 bottle of beer" + f
        case x => x + " bottles of beer" + f
      }

    def beers(qty : Int) = bottles(qty, " on the wall.")

    def sing(qty : Int)(implicit song : String) : String = {
      def takeOne =
        qty match {
          case 0 => "Go to the store and buy some more."
          case x => "Take one down and pass it around."
        }

      def nextQty = //nested functions
        if (qty == 0) 99
        else qty - 1

      def refrain = {
        beers(qty).capitalize + " " + bottles(qty, "") + ".\n"
                      + takeOne + " " + beers(nextQty) + "\n\n"
      }

      if (qty == -1) song
      else sing(qty - 1)(song + refrain) //tail recursion
    }

    implicit val headOfSong : String = ""

    println(sing(99)) //implicit parameter
}
```

Listing 15: Code Beispiel [24]