Scala Einführung

Konzepte objektorientierter Programmiersprachen

Ewelina Wasilewski
www.ewelinawasilewski.com

Inhaltsverzeichnis

Scala

“Scala ist eine ausgereifte, objekt-funktionale, statisch typisierte, leichgewichtige, ausdruckstarke, pragmatiche und skalierbare Sprache für die Java Virtual Machine, die vollständig kompatibel zu Java ist” Heiko Seeberger, Jan Blankenhorn

Skalierbarkeit / Scalable language

Von Skripten zu großen Projekten.

Von sequentiell bis massiv parallel.

*unveränderliche Variable und Datenstrukturen erlauben es, auf das Synchronisieren zu verzichten

*erleichtert asynchrones Programmieren (durch das von Erlang inspirierte Actor - Modell)

Um eigene Sprachkonstrukte erweiterbar.

Scala läuft auf der JVM

Lässt sich gut mit Java integrieren.

Alle Klassen des Paketes `java.lang` stehen automatisch zur Verfügung, andere müssen explizit importiert werden.

Scala ist eine High-Level Sprache

In Java muss man sich im Detail darum kümmern, wie die Anforderung umzusetzen ist.

```
boolean nameHasUpperCase = false;
for(int i=0; i < name.length(); i++)
{
    if(Character.isUpperCase(name.charAt(i)))
        nameHasUpperCase = true;
        break;
}
```

Dank funktionalen Fähigkeiten von Scala ist es möglich Komplexität besser zu verbergen.

Scala ist statisch typisiert

Zur Übersetzungszeit muss der Datentyp einer Variablen bekannt sein.

Programme in Scala lassen sich oft wie eine dynamische Skriptsprache schreiben.

Dank Typinferenz muss den Datentyp nicht immer explizit angeben werden.

(Ausnahmen: **Rekursion**, **return** als Schlüsselwort, **formale Parameter**, guter Stil bei öffentlicher Schnittstelle)

Entwicklungsumgebungen

IDEs, die Scala unterstützen

- Scala-Plugins für die IDEs z.B. IntelliJ, Eclipse oder NetBeans.

Komandozeilen-Werkzeuge

- Interaktive Konsole REPL (Read Evaluate Print Loop).

Von Java zu Scala (Schlüsselworte)

Java	Scala
class	class
interface	x
x	trait
abstract	abstract
static	x
x	object
public	x (public ist der Standard)
Die anderen Modifier gibt	es in Scala ebenfalls sind aber flexibler
x	case (Syntaktischer Zucker)

Von Java zu Scala (Schlüsselworte)

Java	**Scala**
final	final
x	sealed (versiegelt)
if then else	if then else (evaluiert)
while	while
do while	x
for (...;...;...)	x
for (<Typ> arg: Coll)	for (arg <- Coll)
(a < b) ? a : b	x

Variablen

Es gibt zwei unterschiedliche Arten von Variablen:

`var` definiert eine veränderliche Variable.

`val` definiert eine unveränderliche Variable.

Wann immer es geht: `val`s benutzen

Methoden

```
def max(x: Int, y: Int): Int = {
   if(x > y)
      x
   else
      y
}

max(3,5)   // => 5
```

Die Rückgabe einer Methode ist das Ergebnis des zuletzt ausgeführten Ausdrucks.

Unit Methoden

In Scala gibt es auch Methoden, die nichts zurückgeben. Dafür wird der Typ `Unit` verwendet. Der Rückgabewert ist `()`.

Er signalisiert, dass der Wert nicht von weiterer Bedeutung ist.

Skripte in Scala schreiben (CLI)

In Scala können auch Skripte geschreiben werden

hallo.scala

```
println("Hallo " + args(0))
```

und in der Console aufrufen und ausgeben werden

```
$ scala hallo.scala Argument1
```

Schleifen (imperativ)

In Scala gibt es zwei Schleifen, die aus Java bekannt sind

```
var i = 0
while (i < args.length){
    println(args(i))
    i += 1
}

for(arg<-args)
    println(arg)
```

Beide haben Rückgabetyp `Unit`.

Schleife (funktional)

Schleife evaluiert zu einer Collection.

for expression / for comprehension

```
val liste = List(1, 2, 3)
val a = for (x <- liste) yield x+1
//a: List[Int] = List(2, 3, 4)
```

Funktionale Programmierung

Zustand der Objekten und Variablen ist nicht änderbar, daher möglich Korrektheit einer Methode zu prüfen.

Funktionen sind First-Class-Objekte.

Konzept der Funktionen höherer Ordnung vorhanden.

Funktionen höherer Ordnung

Bezeichnet Funktionen, die andere Funktionen als Argument erwarten oder als Rückgabewert haben.

```
var name = "hallo Welt"
var nameHasUpperCase = name.exists{x => x.isUpper}
```

Objektorientierte Programmierung

Scala ist eine objektorientierte Sprache, in dem Sinne, dass alles ein Objekt ist (es gibt in Scala keine primitiven Typen).

Zahlen und Funktionen sind auch Objekte.

Zahlen sind Objekte

Es gibt keine Operatoren wie +, −, /, * in Java, das sind Namen von Methoden.

1 + 1 => (1).+(1)

Präzedenz:

1 + 2 * 3 / x

1 + (2).*(3) / x

1 + ((2).*(3))./(x)

(1).+(((2).*(3))./(x))

Vorteile?

Nachteile?

Funktionen sind Objekte

```
val add = {(x: Int, y: Int) => x + y}
val mult = {(x: Int, y: Int) => x * y}

def calc(myFunction: (Int, Int) => Int,
   a: Int, b: Int) = {
      myFunction(a,b)
}

calc(add, 2, 5) // => 7
calc(mult, 2, 5) // => 10
```

Typparametrisierung

Eine Instanz kann zusätzlich ein Typparameter in eckigen Klammern übergeben:

```
val greetStrings: Array[String] = new Array[String](3)
```

Zuweisung von Werten eines veränderlichen Array:

```
greetStrings(0)=”Hallo”
greetStrings.update(0, ”Hallo”)
```

Werte des Arrays ausgeben:

```
for(i <-0 to 2)
    println(greetStrings(i))
```

oder `println(greetStrings.apply(i))`

Array Erstellen und Initialisieren

Andere Alternative:

```
val nummer = Array("null", "eins", "zwei")
```

Frage: warum schreibt man kein `new` vor `Array`?

Array Erstellen und Initialisieren

Andere Alternative:

```
val nummer = Array("null", "eins", "zwei")
```

Frage: warum schreibt man kein `new` vor `Array`?

```
val nummer = Array.apply("null", "eins",
"zwei")
```

Ist apply eine statische Methode der Klasse Array?

Listen

Eine Liste ist in Scala eine unveränderliche Datenstruktur, das heißt, die enthaltene Elemente können nie geändert werden.

1.Erzeugen einer neuen Liste

```
val leereListe = List()
5 +: leereListe
val einsZweiDrei = List(1, 2, 3)
```

2. Erzeugen einer neuen Liste

```
4 :: 5 :: Nil        //  (((Nil).::(5)).::(4)
4 :: 5 :: nichtleereListe
```

Listen / Elementenzugriff

Da List unveränderlich ist, ist das Überschreiben eines Wertes nicht möglich

```
val liste = List(1, 2, 3)
liste(0) = 4 // Fehler! value update is not a member of List[Int]
```

Elementzugriff

```
val myList = List(2,11,33,22)
myList(1) // => 11
```

Methoden der Klasse List

++ Verbindet eine Liste mit einer Collection und erzeugt daraus neue Liste.

reverse liefert eine Liste mit allen Elementen der Ursprungsliste aber in umgekehrter Reheinfolge.

exists Überprüft, ob zumindest ein Element der Liste die Anforderung der übergebenen Funktion (Predikat) entspricht.

filter Erzeugt eine Liste bestehend aus Elementen der Ausgangsliste, die eine zu definierende Bedingung erfüllen.

Methoden der Klasse List

foreach Iteriert über alle Elemente der Liste, welche in einer zu übergebenen Funktion verarbeitet werden können.

map Erzeugt eine neue Liste durch Anwendung einer Funktion auf alle Elemente der Ausgangsliste

Tupel

Tupel fassen eine feste Anzahl von Werten bzw. Objekten zusammen, wobei die Werte unterschiedliche Typen haben dürfen.

```
val pair = (1, "a")
```

Zugriff auf einzelne Werte des Tupels

```
pair._1
pair._2
```

Unveränderliche Collection

Beispiel: `Set`

Erstellen und Initialisieren:

```
var arbeitswoche = Set("Montag", "Dienstag")
arbeitswoche += "Mittwoch"
arbeitswoche.contains(“Freitag”) // => false
```

Bei unveränderlichen Set's gibt es keine += Methode

```
arbeitswoche += "Mittwoch" => arbeitswoche = arbeitswoche + “Mittwoch”.
```

Veränderliche Collection

Beispiel: `Map`

Wenn man eine veränderliche Map benutzt, muss man das Package entsprechend importieren:

```
import scala.collection.mutable.Map

val personAlter = Map[String, Int]()

personAlter += ("Alexander" -> 25)
personAlter += ("Thomas" -> 27)
personAlter += ("Anika" -> 23)
personAlter("Thomas")  // => 27
```

Beispiele: Endrekursion

Eine Funktion f ist endrekursiv, wenn der rekursive Aufruf von f die letzte Operation in f ist.

Der Scala Compiler kann meistens selbst erkennt, dass es sich um eine Endrekursion handelt, dann optimiert der das zu einer Schleife (ohne den Stack zu überlasten)

Ergebnis

Normale Rekursion

```
factorial(3)
=  if (3 == 0) 1 else 3 * factorial(3 - 1)
=  3 * factorial(3 - 1)
=  3 * factorial(2)
=  3 * {if (2 == 0) 1 else 2 * factorial(2 - 1)}
=  3 * (2 * factorial(2 - 1))
=  3 * (2 * factorial(1))
=  3 * (2 * {if (1 == 0) 1 else 1 * factorial(1 - 1)} )
=  3 * (2 * (1 * factorial(0)))
=  3 * (2 * (1 * {if (0 == 0) 1 else 0 * factorial(0 - 1)} ))
=  3 * (2 * (1 * 1))
=  3 * (2 * 1)
=  3 * 2
=  6
```

Ergebnis

Endrekursion

```
factorial(3)
=  fac(3, 1)
=  if (3 == 0) 1 else fac(3 - 1, 3 * 1)
=  fac(3 - 1, 3 * 1)
=  fac(2, 3)
=  if (2 == 0) 3 else fac(2 - 1, 2 * 3)
=  fac(2 - 1, 2 * 3)
=  fac(1, 6)
=  if (1 == 0) 6 else fac(1 - 1, 1 * 6)
=  fac(1 - 1, 1 * 6)
=  fac(0, 6)
=  if (0 == 0) 6 else fac(0 - 1, 0 * 6)
=  6
```

Scala Ökosystem

Play (Webframework)

Lift (Webframework)

Akka (Aktorensystem)

Slick (Datenbankabstraktion)

Specs2 (Testframework)

ScalaTest (Testframework)

scalaz (pure functional library)

sbt (BuildTool)

Quellen

Bücher:

"Programming in Scala", Martin Odersky

"Durchstarten mit Scala", Heiko Seeberger und Roman Roelofsen

Links:

http://www.magjs.de/2012-01/strehl/strehl.html

http://www.gm.fh-koeln.de/ehses/paradigmen/folien/scala.pdf

http://timpt.de/topic153.html

http://www2.weiglewilczek.com/fileadmin/Publications/Einstieg_in_Scala_2__JS_02_01.pdf

http://flex.winfxpro.info/download/?noderef=workspace://SpacesStore/33d19cd2-2b2b-41be-a211-415c5eaacf59

Vielen Dank!

source code: https://github.com/ewasilewski/scala-presentation/blob/master/examples.md