12.–15.09.2010
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Eine Einführung in Scala

Rüdiger Keller / Michael Wiedeking

MATHEMA Software GmbH

Scala - Kurzübersicht

- **Sca**lable **La**nguage
- Statisch typisierte Sprache für die JVM (und .Net)
- „Ein besseres Java“, vereint Objektorientierung und funktionale Programmierung
- Entwickelt am Ecole Polytechnique Fédérale de Lausanne unter der Regie von Martin Odersky
- Design ab 2001, Version 1.0 veröffentlicht 2003, Version 2.0 veröffentlicht 2006, aktuelle Version 2.8

Scala – Ein Vorgeschmack

```
object Fakultät {

   def main(args : Array[String]): Unit  = println(fakultät(20))

   def fakultät(n : BigInt): BigInt =
        if (n == 0) 1 else n * fakultät(n - 1)

}
```

Ähnlich Java, aber

- keine statischen Methoden, stattdessen Objekte
- andere Definitionssyntax
- public ist Standard
- Semikolons sind optional

Allgemeines

- Scala und Java sind voll interoperabel
  Gegenseitiger Zugriff auf Methoden, Felder, Klassen, …
- Alles ist ein Objekt
  Keine primitiven Typen
- Mächtiges statisches Typsystem
- Funktional:
  Funktionen und Closures sind Objekte
- Prägnante Syntax:
  optionale Semikolons und Klammern,
  Typinferenz,
  und vieles mehr
- Keine Checked-Exceptions

Definitionen

Definitionen (Java vs. Scala)

int *i*;	**var** *i* : Int
final int *i*;	**val** *i* : Int
void method() { … }	**def** method { … }
int num() { **return** *res*; }	**def** num : Int = *res*
int add(int *a*) { ... }	**def** add(*a* : Int): Int = { … }
class Foo<*T*>	**class** Foo[*T*]

Typinferenz

Typinferenz erlaubt das Weglassen von Typangaben

val test = "Test"	String
val map = Map("a" -> 1, "b" -> 2)	Map[String, Int]
def fun(*x* : Any) = *x*.toString	Rückgabewert: String

Ausdrücke

- Methodenaufrufe

  obj.method(arg)
  obj method arg

- Operatoren sind normale Methoden

  1 + 2 ist äquivalent zu 1.+(2)

- Bedingte Ausdrücke

  if(cond) expr1 **else** expr2
  val v = **if**(cond) a **else** b

- Schleifen

  while(cond) { expr }

Klassen

Java

```
public class Beispiel {
  private final int a;
  private final int b;
  public Beispiel(int a, int b) {
    this.a = a;
    this.b = b;
  }
  public int getA() {
    return a;
  }
  public int method(int x) {
    return b + x;
  }
}
```

Scala

```
class Beispiel(val a: Int, b: Int) {
  def method(x: Int) = b + x
}
```

Verwendung

```
val bsp = new Beispiel(3, 7)
val a = bsp.a
val b = bsp.method(42)
```

Objekte

- Scala kennt keine statischen Felder oder Methoden
- Stattdessen Objekte: Singleton-Klassen

```
object Counter {
    private var counter = 0
    def next() = {
        counter += 1
        counter
    }
}
```

```
Counter.next()
```

Traits

- Traits ähneln Javas Interfaces
- Können auch Implementierungen enthalten
- Können selbst von Klassen oder Traits erben

```
trait Trait {
    def abstractMethod(s: String): Int
    def concreteMethod(s: String) = field + s
    var field = "fields work too"
}

class Class extends Super with TraitA with TraitB
```

Case Klassen

- Wie normale Klassen, aber
  - Können ohne **new** erzeugt werden
  - Konstruktorparameter sind implizit **val**
  - Implementieren equals, hashCode, toString und copy-Methode
  - Können in Pattern-Matches verwendet werden

```
case class Name(vorname: String, nachname: String)
case class Person(name: Name, alter: Int)

val arthur = Person(Name("Arthur", "Dent"), 25)

val olderArthur = arthur.copy(alter = 42)
```

Pattern-Matching

- Ähnlich wie switch in Java, aber viel mächtiger
  - Wert- und Typ-Patterns
  - Guards (if-Ausdrücke)
  - Beliebige Kombinationen davon

```
def tester(exp: Any): String = exp match {
    case null => "Nur null"
    case "Servus" => "Moin moin"
    case i: Int => (i * i).toString
    case d: Double if(d > 0) => Math.sqrt(d).toString
    case Person(Name(vorname, _), alter) => vorname + " ist " + alter
    case _ => "Etwas anderes"
}
```

Collections

- Scala hat eigene Collections mit Listen, Maps, Sets, …
- Mutable und immutable Varianten, immutable ist Standard
- Collection-Literale

  Map.empty
  List(1, 2, 3)

- Seq ist eine geordnete Sequenz, wie Javas List
- Range als spezielle Sequenz von Integern

```
val einsZwei = 1 until 3
val einsZweiDrei = 1 to 3
val nullZweiVier = 0 until 5 by 2
```

Collections – Standardoperationen

- foreach

  `List(1, 2, 3).foreach(x => print(x))` *Ausgabe*: 123

- filter

  `List(-1, 0, 1).filter(_ > 0)` *Ergebnis*: List(1)

- map

  `List("a", "bb", "ccc").map(_.length)` *Ergebnis*: List(1, 2, 3)

- und viele mehr

for-Comprehensions

- Syntax ähnlich der for-each-Schleife von Java
- Wesentlich mächtiger

```
for(x <- 0 until 10) print(x)
```
Ausgabe: 0123456789

```
for(x <- List(1, 2, 3, 4, 5) if(x % 2 != 0)) yield x
```
Ergebnis: List(1, 3, 5)

```
for(x <- List("a", "b"); y <- (1 to 3)) yield (x + y)
```
Ergebnis: List("a1", "a2", "a3", "b1", "b2", "b3")

```
for(x <- List(1, 2, 3); val y = x * x; z <- 1 to y by x) yield z
```
Ergebnis: List(0, 1, 0, 2, 4, 0, 3, 6, 9)

Option

- Option - Das bessere null
  - Kann entweder Some(x) sein, oder None()
  - Zeigt explizit, dass der Wert auch "nichts" sein kann
  - Verhält sich wie eine Collection-Klasse und unterstützt bspw. foreach und map

```
def optional(): Option[String] = …

optional() match {
    case Some(x) => println(x)
    case _ => println("Nix")
}
```

```
optional() foreach { println(_) }

val o = optional().getOrElse("Nix")
```

Implicit-Conversions – „Pimp my Library“

- Implicit-Conversion bei nicht passendem Typ
- Ermöglicht interne DSLs und Erweiterung externer Typen

```
class PimpedString(s: String) {
    def *(n: Int) = { var tmp = ""; for(0 until n) tmp += s; tmp }
}

implicit def toPimpedString(s: String) = new PimpedString(s)
```

```
"Hallo" * 3                  // Ergebnis: "HalloHalloHallo"
"Hallo Welt".dropRight(5)    // Ergebnis: "Hallo"
0 until 10                   // Ergebnis: Range(0, …, 9)
```

Beispiel – Indexerzeugung

Java

```
public Map<Character, List<String>> createIndex(List<String> keywords) {
  Map<Character, List<String>> result = new HashMap<Character, List<String>>();
  for(String word : keywords) {
    char firstChar = word.charAt(0);
    if(!result.containsKey(firstChar)) {
      result.put(firstChar, new  ArrayList<String>());
    }
    result.get(firstChar).add(word);
  }
  for(List<String> list : result.values()) {
    Collections.sort(list);
  }
  return result;
}
```

Scala

```
def createIndex(keywords: Seq[String]) = keywords groupBy (_.head) mapValues (_.sorted)
```

Referenzen

- Die Website von Scala

  http://www.scala-lang.org/
- Ein Blog mit vielen Tips und Beispielen

  http://daily-scala.blogspot.com/
- Ein Buch über Scala, online verfügbar

  http://programming-scala.labs.oreilly.com/
- Die Scala Mailing-Listen, hier über Nabble

  http://old.nabble.com/Scala-Programming-Language-f20934.html
- "Programming in Scala", M. Odersky, L. Spoon, B. Venners, Artima Inc, 2008

12.–15.09.2010
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Vielen Dank!

Rüdiger Keller

MATHEMA Software GmbH

Backup-Folien

Pattern-Matching

- Eine weitere Form von Pattern-Matching

```
def pair(): Pair[Int, Int] = …
val (a, b) = pair()
```

```
def getName(): Name = …
val Name(vorname, nachname) = getName()
```

Traits - Ableitung

- Traits können auch von Klassen ableiten

```
class Rnd {
    def supply = scala.util.Random.nextInt(10)
}

trait Doubler extends Rnd {
    override def supply = 2 * super.supply
}

trait AddTen extends Rnd {
    override def supply = 10 + super.supply
}
```

Traits - Linearisierung

- Keine Mehrfachvererbung, sondern Linearisierung

```
class DoubleAndAddTenRnd extends Rnd with Doubler with AddTen
class AddTenAndDoubleRnd extends Rnd with AddTen with Doubler
```

- Wird linearisiert zu:

DoubleAndAddTenRnd -> AddTen -> Doubler -> Rnd -> AnyRef -> Any

AddTenAndDoubleRnd -> Doubler -> AddTen -> Rnd -> AnyRef -> Any

for-Comprehensions - Transformation

- Werden vom Compiler zu Ausdrücken aus foreach, map, flatMap und filter umgewandelt

```
for(x <- 0 until 10) print(x)
(0 until 10).foreach(x => print(x))

for(x <- Seq(1, 2, 3); y <- (-3 to 3); if y != 0) println(x * y)
Seq(1, 2, 3).foreach(x => (-3 to 3).filter(_ != 0).foreach(y => println(x * y)))

val listOfPairs = for(x <- List(-1, 0, 1); y <- List(2, 3, 4)) yield (x, y)
val listOfPairs = List(-1, 0, 1).flatMap(x => List(2, 3, 4).map(y => (x, y)))
```

Loan Pattern

```
def use[T <: { def close(): Unit }](resource: T)(block: T => Unit) = {
    try {
        block(resource)
    } finally {
        if(resource != null) resource.close()
    }
}
```

```
use(new FileInputStream("source")) { in =>
    use(new FileOutputStream("target")) { out =>
        val buffer = new Array[Byte](1024)
        val iterator = Iterator.continually(in.read(buffer))
        iterator takeWhile (_ != -1) foreach { out.write(buffer, 0 , _) }
    }
}
```