5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Scala

Eine Einführung in Scala

Rüdiger Keller

MATHEMA Software GmbH

Scala – Überblick

- ***Sca***lable ***La***nguage
- Statisch typisierte Sprache für die JVM
- „Ein besseres Java“, vereint Konzepte objektorientierter und funktionaler Programmierung
- Entwickelt unter der Regie von Martin Odersky am Ecole Polytechnique Fédérale de Lausanne
- Designbeginn 2001, aktuelle Version 2.9.1
- Kommerzieller Support durch Firma Typesafe

Ein Vorgeschmack

```
object Fakultaet {
  def main(args: Array[String]): Unit  = println(fakultaet(20))
  def fakultaet(n: BigInt): BigInt = if(n == 0) 1 else n * fakultaet(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, …)
- Statisches Typsystem
- Vollständig objektorientiert, keine primitiven Typen
- Funktional: unterstützt Funktionen und Closures
- Keine Checked-Exceptions

Syntax - Definitionen

Java

```
int i;
final int i;

void method() { … }
int num() { return res; }
int add(int a) { ... }

class Foo<T>
```

Scala

```
var i: Int
val i: Int

def method { … }
def num: Int = res
def add(a: Int): Int = { … }

class Foo[T]
```

Typinferenz

- Typinferenz erlaubt das Weglassen von Typangaben

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

Syntax - Ausdrücke

- Methodenaufrufe

  obj.method(arg)
  obj method arg

- Operatoren sind normale Methoden

  1.0 + 2.0 ist äquivalent zu 1.0.+(2.0)

- Bedingte Ausdrücke

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

- Schleifen

  while(cond) { expr }
  for(x <- expr1) { expr2 }

Funktionen

- Funktionen kennen wir aus der Schulmathematik

  f(x) = x + 3
  g(x, y) = 2xy

- In Scala

```
val f = (x: Int) => x + 3
val g = (x: Int, y: Int) => 2 * x * y
```

- Aufruf wie eine Methode

```
val y = f(39)
val z = g(3, 7)
```

Collections

- Scala hat eigene Collections mit Listen, Maps, Sets, ...
- Mutable und immutable Varianten, immutable ist Standard
- Sehr mächtig durch eine Vielzahl an Operationen

Collections - Beispiele

- Gehaltserhöhung für alle

```
angestellte.foreach(_.gehalt *= 1.1)
```

- Die zehn Bestverdiener

```
val topTen = angestellte.sortBy(_.gehalt).takeRight(10)
```

- Minderjährige und Erwachsene unterscheiden

```
val (minors, adults) = persons.partition(_.age < 18)
```

- Summe aller Gehälter

```
val summe = angestellte.map(_.gehalt).sum
```

Parallel Collections

- Neu seit Scala 2.9
- Operationen werden parallel ausgeführt
- Lastverteilung per Work-Stealing mit Hilfe des Fork-Join-Frameworks (JSR-166, auch in Java 7)

```
val topTen = angestellte.par.sortBy(_.gehalt).takeRight(10)

val (minors, adults) = persons.par.partition(_.age < 18)

val summe = angestellte.par.map(_.gehalt).sum
```

Case Klassen

- Value-Klassen mit Gettern, Settern, equals, hashCode und toString

```
case class Angesteller(
  name: String,
  abteilung: String,
  gehalt: BigDecimal
)

val hans = Angesteller("Hans Schmidt", "Personal", 50000)
val petra = Angesteller("Petra Huber", "Personal", 50000)
val hansBefördert = hans.copy(gehalt = hans.gehalt * 2)
val name = hans.name
```

Pattern-Matching

- Match-Ausdruck: switch on steroids

```
def asString(x: Any) = x match {
  case null => "null"
  case "" => "Der leere String"
  case d: Date => dateFormat.format(d)
  case i: Int if(i % 2 == 0) => "Eine gerade Zahl"
  case Angestellter(_, "Personal", _) => "Personaler"
  case Angestellter(_, _, gehalt) if(gehalt > 1e6) => "Millionär"
  case _ => x.toString
}
```

For-Comprehensions

- Ähnlich for-each in Java, aber 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")

Option – ein typsicheres null

- Kann entweder Some(x) sein, oder None

```
val opt: Option[String] = ...
opt match {
  case Some(string) => println(string)
  case None => println("None - da war nichts")
}
opt foreach println
val string = opt.getOrElse("None - da war nichts")
```

- Verkettung von Methoden die Option liefern mit for

```
for {
  ufo <- getUfo()
  alien <- ufo.getCrewMember()
  weapon <- alien.getWeapon()
} weapon.fire()
```

Actors

- Ein nebenläufiges Programmiermodell basierend auf Nachrichtenaustausch
- Ein Actor ist „single-threaded“

```
class Pinger extends Actor {
  var pings = 0
  def act = loop {
    react {
      case Ping => pings += 1; sender ! Pong(pings)
      case Quit => exit()
    }
  }
}
```

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/
- "Programming in Scala, Second Edition", M. Odersky, L. Spoon, B. Venners, Artima Inc, 2010

Vielen Dank!

ruediger.keller@mathema.de

MATHEMA Software GmbH

Backup-Folien

Backup-Folien

Implicit-Conversions

- Implicit-Conversion bei unpassendem Typ, ermöglicht interne DSLs und Erweiterung gegebener Typen

```
(0 until 10).toList
val hello = "Hello World" dropRight 6
val m = Map(1 -> "a", 2 -> "b")
```

- Do it yourself

```
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
```

Interne DSL – ScalaTest™

```
class StackSpec extends FlatSpec with ShouldMatchers {

  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should equal (2)
    stack.pop() should equal (1)
  }

  it should "throw NoSuchElementException if an empty stack is popped" in {
    val emptyStack = new Stack[String]
    evaluating { emptyStack.pop() } should produce [NoSuchElementException]
  }
}
```

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

val a = new A with B with C
```

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 , _) }
    }
}
```

Java vs Scala – 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)
```