15.–18. 09. 2008
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Varietékünstler

Die Programmiersprache Scala

Jan Hermanns

www.sushi3003.de

Wer hat`s erfunden?

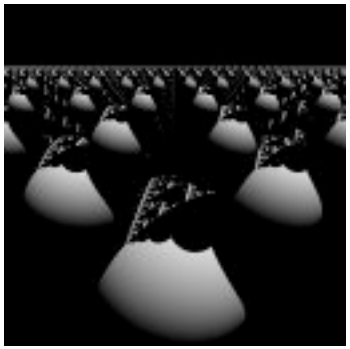
- Entwickelt von Martin Odersky an der EPFL
- Erstes Release 2003

Scala Überblick

- Interoperabilität mit Java
  - Java-Bytecode Compiler
  - aber es gibt auch einen Scripting-Interpreter
- Vereint OOP und FP
  - einheitliches Objektmodell
  - higher-order functions
- Statisches Typsystem (mit Type Inference)
- Objektdekomposition mittels Pattern Matching
- XML Support

Scala Performance

- Raytracer Implementierungen im Vergleich



Java-Version	*12s*
Scala-Version	*12s*
Groovy-Version	*2h 31m 42s*

Hello World

```
object HelloWorld {
    def main(args:Array[String]) =
        println("Hello World!")
}
```

Ein etwas aufwendigeres Beispiel

```
package swingdemo

import javax.swing.{JFrame, JLabel}

object Main extends Application {
    import JFrame._

    def getTitle() = "Scala can Swing"

    val frm = new JFrame(getTitle)
    frm.getContentPane.add(new JLabel("Hello"))
    frm.setDefaultCloseOperation(EXIT_ON_CLOSE)
    frm.pack
    frm setVisible true
}
```

Interoperabilität

- Scala kann
  - Instanzen von Java-Klassen erzeugen
  - von Java-Klassen erben und Java-Interfaces implementieren
  - auf Methoden und Felder von Java-Komponenten zugreifen
- Scala-Klassen können von Java aus instanziiert und verwendet werden
  - im Zweifel das Kompilat mit javap anschauen

Konstruktoren

```
class ImmutableFoo(val bar:Int)
class MutableFoo(var bar:Int)
```

- Für Konstruktorparameter werden Properties angelegt, deren Zugriff durch die Modiefier **val** und **var** geregelt wird
  - val bedeutet read-only
  - var bedeutet read-write

```
val mutable = new MutableFoo(32)
println(mutable.bar)
mutable.bar = 35
```

Java vs. Scala

- Java

```
public class Person {
   private String firstName;
   private String lastName;
   private int    age;

   public Person(String firstName, String lastName, int age) {
      this.firstName  = firstName;
      this.lastName  = lastName;
      this.age          = age;
   }

   public void setFirstName(String firstName) { this.firstName = firstName; }
   public void String getFirstName() { return this.firstName; }
   public void setLastName(String lastName) { this.lastName = lastName; }
   public void String getLastName() { return this.lastName; }
   public void setAge(int age) { this.age = age; }
   public void int getAge() { return this.age; }
}
```

- Scala

```
class Person(var firstName: String, var lastName: String, var age: Int)
```

Uniform Access Principle

- Implementierung eigener “virtueller” Properties

```
class AbsInt(num:Int) {
    private var v = Math.abs(num)

    def value = v
    def value_=(num:Int) = {
        v = Math.abs(num)
    }
}
val n = new AbsInt(-5)
println(n.value)
n.value = 6  // n.value_=(6)
```

Methoden als Operatoren verwenden

```
class MyInt(val n:Int) {
   def max(that:MyInt) =
     if (n > that.n) this else that
}
```

- Jede Methode, die genau eine Parameter erwartet kann als Infix-Operator verwendet werden

```
var i = new MyInt(1)
var j = new MyInt(2)
println(i max j)
```

Operatoren definieren

```
class MyInt(val n:Int) {
   def +(that:MyInt) = new MyInt(n + that.n)
   def unary_-() = new MyInt(-1 * n)
   def ++() = new MyInt(n +1)
}
```

- Verwendung von Infix-, Prefix- und Postfix-Operatoren

```
var i = new MyInt(1)
var j = new MyInt(2)
println(i + j)
println(-i)
println(i++)
```

Traits

- Vergleichbar mit Java-Interfaces, allerdings können Traits auch Implementierungen beinhalten

```
trait Foo {
    def foo():Int
    def bar() = foo + 1
}

class Test extends Foo {
    def foo() = 4
}
```

Traits

- Objekte können auch ad hoc mit Traits erweitert werden

```
trait Logging {
    def log = println(this)
}
class Test {
    override def toString = "Test"
}

val t = new Test with Logging
t.log
```

Traits “Mehrfach Vererbung”

```
abstract class Hello {
    def hello:String
}
trait Foo extends Hello {
    override def hello = "Foo"
}
trait Bar extends Hello {
    override def hello = "Bar"
}
class Test extends Foo with Bar {
    override def toString = hello
}
println(new Test)
```

Pattern Matching

- Das `match/case` Konstrukt dient zur Dekomposition von Objekten

```
def doMatch(a:Any) = a match {
   case i:Int   => "Integer:" + i
   case f:Float => "Float:" + f
   case _       => "don't know"
}

println(doMatch(3))
println(doMatch(2.1F))
println(doMatch("hello"))
```

Pattern Matching

- Syntactic Sugar für Listen und Tuppel
- if-Guards

```
def doMatch(a:Any) = a match {
   case x::y::10::rest => "1"
   case (x:Int, y:Int) if (x+y == 10) => "2"
   case (x,_)          => "3"
}

println(doMatch(List(1,2,10,4,5)))
println(doMatch((4, 6)))
println(doMatch(("hello", 1)))
```

Pattern Matching

- Mittels "**case**" können eigene Klassen erstellt werden die mittels Pattern Matching zerlegt werden können

```
case class Person(var name:String, var age:Int)

def doMatch(p:Person) = p match {
  case Person(n, a) if (a < 18) => "young:"+n
  case Person(n, _)             => "adult:"+n
}

println(doMatch(Person("Tim", 5)))
println(doMatch(Person("Jan", 33)))
```

Pattern Matching

```
abstract class Expr
case class Const(v:Int)           extends Expr
case class Mul(e1:Expr, e2:Expr) extends Expr
case class Add(e1:Expr, e2:Expr) extends Expr

def interp(e:Expr):Int = e match {
   case Const(v)     => v
   case Mul(e1, e2) => interp(e1) * interp(e2)
   case Add(e1, e2) => interp(e1) + interp(e2)
}

val e = Add(Mul(Const(2),Const(6)),Add(Const(3),Const(4)))
println(interp(e))
```

Pattern Matching

```
def opt(e:Expr):Expr = e match {
   case Mul(e1,e2)
      if e1==Const(0) || e2==Const(0) => Const(0)
   case Mul(Const(1), x) => x
   case Mul(x, Const(1)) => x
   case Mul(e1, e2)      => Mul(opt(e1), opt(e2))
   case Add(e1, e2)      => Add(opt(e1), opt(e2))
   case x:Const          => x
}

val e1 = Add(Mul(Const(2), Const(1)), Add(Const(3), Const(4)))
val e2 = Add(Mul(Const(2), Const(0)), Add(Const(3), Const(4)))

println(opt(e1)) // Add(Const(2),Add(Const(3),Const(4)))
println(opt(e2)) // Add(Const(0),Add(Const(3),Const(4)))
```

Die “apply” Methode

- Impliziter Aufruf der “apply” Methode, wenn ein Ausdruck der Form: `obj(param)` bzw. `obj(p1,p2,...)` gefunden wurde

```
val a = Array("hello", "world")
println(a(0))
```

entspricht

```
val a = Array.apply("hello", "world")
println(a.apply(0))
```

First Class Functions

- Funktionen sind auch nur Objekte
  - und können daher wie ganz normale Werte herumgereicht werden

```
val mul = (x:Int, y:Int) => x * y
val multiply = mul
println(multiply(3, 4))
```

Higher-Order Functions

- Funktionen die andere Funktionen als Parameter entgegennehmen oder Funktionen als Rückgabewerte zurückliefern

```
def sumF(f:Int=>Int, x:Int, y:Int):Int =
  if   (x > y) 0
  else f(x) + sumF(f, x+1, y)

def square(x:Int) = x * x

println(sumF(identity, 1, 3))
println(sumF(square, 1, 3))
println(sumF(x => x * x, 1, 3))
```

Tail-Rekursion

- Die JVM bietet leider keine Unterstützung für Tail-Calls
  - der Scala Compiler kann jedoch einfache Tail-Rekursive Aufrufe zu Schleifen optimieren

```
def sumF(f:Int=>Int,x:Int,y:Int,a:Int):Int =
   if(x > y) a
   else sumF(f, x+1, y, a + f(x))

println(sumF(identity, 1, 3, 0))
```

Currying

- Zurückliefern einer inneren Funktion

```
def sumF(f:Int=>Int):(Int,Int)=>Int = {
   def anoSumF(x:Int, y:Int):Int =
      if (x>y) 0 else f(x) + anoSumF(x+1,y)
   anoSumF
}

println((sumF(identity)) (1, 3))
```

- Funktions-Applikation ist linksassoziativ

```
println(sumF(identity) (1, 3))
```

Currying

- Spezielle Syntax erspart uns die innere Funktion

```
def sumF(f:Int=>Int)(x:Int, y:Int):Int =
   if (x>y) 0 else f(x) + sumF(f)(x+1, y)

println(sumF(identity)(1,3))
```

- Underscore Notation

```
def sumInts = sumF(identity) _
println(sumInts(1,3))
```

Call-by-Value vs Call-by-Name

- Call-by-Value

```
def eagerFoo(bar:Int) = {
   println("eager"); bar + 1
}
println(eagerFoo({println("arg"); 1}))
```

- Call-by-Name

```
def lazyFoo(bar: =>Int) = {
   println("lazy"); bar + 1
}
println(lazyFoo({println("arg"); 1}))
```

Eigene Kontrollstrukturen

```
def whileLoop(cond: =>Boolean)(body: =>Unit):Unit =
   if (cond) {
      body
      whileLoop(cond)(body)
   }


var i = 10
whileLoop (i > 0) {
    println(i)
    i -= 1
}
```

Eigene Kontrollstrukturen 2

```
class LoopUnlessCond(body: => Unit) {
   def unless(cond: => Boolean):Unit = {
      body
      if (!cond) unless(cond)
   }
}

def loop(body: => Unit): LoopUnlessCond =
   new LoopUnlessCond(body)

var i = 10
loop {
    println(i)
    i -= 1
} unless (i == 0)
```

For Comprehensions

- Funktionieren ähnlich wie eine Query-Language
  - Iteration über eine oder mehrere Sequenzen
  - Anwendung von Filtern
  - Rückgabe einer neuen Sequenz

```
for (i <- 1 to 10 if i % 2 == 0) yield i + 1
```

  - intern werden die Higher-Order Functions map, flatMap, und filter

```
for (i <- 1 until 10 if i % 2 != 0)
  println(i)
```

  - bzw. foreach aufgerufen

For Comprehensions

```
case class Team(val name:String, val score:Int)

val teams = List(Team("Wookies", 7),
                 Team("Ewoks", 4),
                 Team("Sith", 6))

for (t1 <- teams;
     t2 <- teams if t1 != t2 && t1.score > t2.score)
   println(t1.name + " defeated " + t2.name)
```

- Output

```
Wookies defeated Ewoks
Wookies defeated Sith
Sith defeated Ewoks
```

Vielen Dank!

Jan Hermanns
www.sushi3003.de