Scala

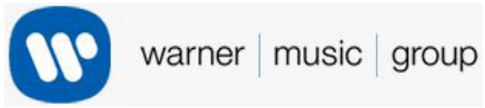
Jan Kriesten

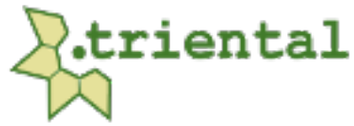
AK Objekttechnologie Norddeutschland

12. April 2010

Vorstellung

- Internetagentur mit Standort Hamburg
- Web business seit 1995
- Consulting, Training, Softwareentwicklung
- Systemadministration
- Kunden





Agenda

- Warum Scala?
- Spracheigenschaften
- Scala programmieren
- Ausblick und Resümee

Warum Scala?

- OOP - Gestern und heute
- Dynamische Sprachen
- Auftritt: Scala

OOP - Gestern und heute

- Die letzten 10 Jahre:
  - Java als „Main Stream OOP“
  - Smalltalk als „Pure OOP“

- Aktuell:
  - Java als OOP
  - Dynamische Sprachen (Groovy, Ruby, Python, ...)

Dynamische Sprachen

- Pragmatischer Ansatz
- Einfache Syntax: leicht zu erlernen + programmieren
- Übernahmen aus der Funktionalen Programmierung, z.B.:
  - Closures
  - Listen-Verarbeitung
- Scriptfähig
- Keine statische Typisierung

➠ falscher Ansatz!

Auftritt Scala (1)

- Idee:
  - Allgemeine Sprache
  - Konzepte sollen sowohl im Kleinen wie in Großen greifen
  - Symbiose und Generalisierung von funktionalen und objektorientierten Konzepten
- Scala:
  - OOP + FP vollständig kompatibel mit Java (.NET soll folgen)
  - Open Source seit Januar 2004

Auftritt Scala (2)

- Scala entwickelt vom Team um Martin Odersky an der École Polytechnique Fédérale de Lausanne (EPFL)
- Scala bietet
  - ein einheitliches Objekt-Modell,
  - Pattern Matching,
  - Higher-order Functions,
  - neue Wege zu abstrahieren und Programme zu schreiben.

Spracheigenschaften

- Kompatibel
- Funktional
- Kompakt
- Klar und zielgerichtet
- Erweiterbar
- Performant

Kompatibel

- Nahtlose Java-Integration:
  - Methoden-Aufruf
  - Field-Zugriff
  - Class-Vererbung
  - Interface-Implementierung
- JVM Bytecode
- Code ähnelt Java mit einigen Änderungen

```
object Test1 {
  def main( args: Array[String] ) {
    val b=new StringBuilder()
    for( i<-0 until args.length ) {
      if( i>0 ) b.append( " " )
      b.append( args(i).toUpperCase )
    }
    println( b.toString )
  }
}
```

Funktional

- Programm im funktionalen Stil:
  - Abstraktion: Array als Instanz einer Sequenz
  - Higher-order Functions statt Schleifen

```
object Test2 {
  def main(args: Array[String]){
    println( args
               .map( _.toUpperCase )
               .mkString( " " ) )
  }
}
```

Kompakt (1)

- Unterstützung:
  - Folgerung von Typen („type inference“)
  - kompakte Class-Syntax
  - erweiterbare APIs
  - Closures als Kontroll-Abstraktionen

```
var capital = Map( "US" -> "Washington",
                   "France" -> "paris",
                   "Japan" -> "tokyo" )

capital += ( "Russia" -> "Moscow" )

for( (country,city) <- capital )
    capital += (country -> city.capitalize)

assert( capital("Japan") == "Tokyo" )
```

LOC Scala : Java: ≤50%

Kompakt (2)

Java:

```
List<Integer> nums = Arrays.asList(1,2,2,3,4,6,6,7,8,8);
List<Integer> dups = new ArrayList<Integer>();

int prev = nums.get(0);
for( Integer i: nums.subList(1) ) {
    if( prev==i ) dups.add(i);
    prev=i;
}
```

Scala:

```
val list = List(1,2,2,3,4,6,6,7,8,8)
val dups = for( (x,y)<-list.zip( list.tail ); if x==y ) yield x
```

Klar und zielgerichtet

- Code nutzt Abstraktion der Bibliothek - keine spezielle Syntax
- Vorteil: Bibliotheken sind erweiterbar und geben Kontrolle
- Fehler werden durch die statische Typisierung früh erkannt

```
import scala.collection.mutable._

val capital = new HashMap[String,String]
              with SynchronizedMap[String,String] {
                override def default(key: String) = "?"
              }

capital += ( "US" -> "Washington",
             "France" -> "Paris",
             "Japan" -> "Tokyo" )

assert( capital("Russia") == "?" )
```

Erweiterbar (1)

- Scala ist „schlank“ in seinen Sprachkonstrukten
- Typen lassen sich über Bibliotheken hinzufügen, als wären sie native Sprachelemente
- Beispiel: Komplexe Zahlen

```
scala> import Complex._
import Complex._

scala> val x = 1 + 1 * i
x: Complex = 1.0+1.0i

scala> val y = x * i
y: Complex = -1.0+1.0i

scala> val z = y + 1
z: Complex = 0.0+1.0i
```

Erweiterbar (2)

```
object Complex {
  val i = new Complex(0,1)
  implicit def int2complex(x: Int): Complex = new Complex(x,0)
  ...
}
class Complex(val re: Double, val im: Double) {
  def + (that: Complex): Complex = new Complex( this.re + that.re, this.im + that.im )
  def - (that: Complex): Complex = new Complex( this.re - that.re, this.im - that.im )
  def * (that: Complex): Complex = new Complex( this.re * that.re - this.im * that.im,
                                                this.re * that.im + this.im * that.re )
  def / (that: Complex): Complex = {
    val denom = that.re * that.re + that.im * that.im
    new Complex( (this.re * that.re + this.im * that.im) / denom,
                 (this.im * that.im - this.re * that.im) / denom )
  }
  override def toString = re+(if(im<0) "-" + (-im) else "+" + im) + "i"
  ...
}
```

Performant

- Scala generiert viele zusätzliche Konstrukte:
  - Weiterleitung von Methodenaufrufen
  - Hilfsobjekte
  - Innere anonyme Klassen
- Startzeiten sind z.T. länger aufgrund vieler class-Dateien
- JIT optimiert diese häufig weg, Laufzeit vergleichbar mit Java
- Scala ist z.T. schneller in der Ausführung, da die generierten class-Dateien gut zu optimieren sind

Scala programmieren

- Definitionen
- Functions / Closures
- Currying
- Neue Kontrollstrukturen
- Case Classes / Pattern Matching
- Traits
- Actors
- Tool Support

Definitionen

- class - entspricht Java class; Besonderheit: "case class"
- object - Singleton class (kein static wie in Java)
- trait - entspricht Java Interface (kann Implementierungen enthalten)
- var

```
var mutable = "String reference"
mutable = "String reference changed"
```

- val

```
val immutable = "fixed String reference"
```

- lazy val

```
lazy val lval = {
  println( "init" )
  "lazy String reference"
}
```

- def

```
def function( arg: String ): String = {
  "Hello " + arg
}
```

Functions / Closures

- First-class Functions:
  - Funktionen sind Objekte, können übergeben und Variablen zugewiesen werden

```
val increase = (x: Int) => x + 1
```

- Higher-order Functions: Funktionen als Parameter

```
val list = List(-15, -8, -5, 0, 5, 8, 15)
list.filter( _ < 0 )
```

- Closures binden ‚freie Variablen' zur Laufzeit

```
var more = 1
val addMore = (x: Int) => x + more
```

Currying

- Funktionen mit mehreren Argument-Listen

```
def curriedSum(x: Int)(y: Int) = x + y
curriedSum(1)(2)
```

- Illustration des Prozesses

```
def first(x: Int) = (y: Int) => x + y
val second = first(1)
second(2)
```

- Referenz auf die 2. Funktion über Platzhalter (partially applied functions):

```
val onePlus = curriedSum(1) _
onePlus(2)
```

Neue Kontrollstrukturen

- First-class functions + Currying bieten neue Möglichkeiten, Kontrollstrukturen zu etablieren
- Beispiel: break / continue

```
object BreakLoop {
  object Break extends RuntimeException
  object Continue extends Exception
  def break { throw Break }
  def continue { throw Continue }
  def whileTrue( condition: => Boolean )(block: => Unit) {
    try {
      while( condition ) try { block } catch { case Continue => }
    } catch { case Break => }
  }
}
```

```
import BreakLoop._
var i = 0
whileTrue( i<10 ) {
  i += 1
  if( i<2 ) continue
  if( i>=8 ) break
  println( i )
}
```

Case Classes / Pattern Matching

- Case Classes

```
case class Number( value: Int )
val n = Number(1)
```

- Case Classes geben Einblick in die Objektkonstruktion

- Pattern Matching

```
def isBig( n: Number ) = {
  n match {
    case Number(i) if i<5 => false
    case _ => true
  }
}
```

- Traits können Implementierungen enthalten

```
abstract class IntQueue {
  def get: Int
  def put( x: Int )
}
trait Doubling extends IntQueue {
  abstract override def put( x: Int ) = super.put( 2 * x )
}
```

- Traits können Klassen hinzugefügt werden

```
class StandardQueue extends IntQueue {
  private val buf = new scala.collection.mutable.ArrayBuffer[Int]
  def get = buf.remove( 0 )
  def put( x: Int ) = buf += x
}
val queue = new StandardQueue with Doubling
```

Actors

- Actors sind bessere Concurrency-Konstrukte
- Message-Passing Concurrency

```
class Ping(count: Int, pong: Actor) extends Actor {
  def act = { var pingsLeft = count
    loop { react {
      case "Start" | "Pong" => println( "Pong" )
        if( pingsLeft>0 ) { pong ! "Ping"; pingsLeft -= 1 }
        else { pong ! "Stop"; exit }
    } }
  }
}
class Pong extends Actor {
  def act = loop { react {
    case "Ping" => println( "Ping" ); sender ! "Pong"
    case "Stop" => exit } }
}
```

Tool Support

- Plugins sind derzeit für alle gängigen IDEs in Entwicklung
- Eclipse-Plugin
  - http://www.scala-lang.org/downloads/distrib/files/nightly/scala.update/
- Netbeans:
  - http://wiki.netbeans.org/Scala
- IDEA (v9)
  - http://plugins.intellij.net/plugin/?id=1347
- Maven
  - http://scala-tools.org/mvnsites/maven-scala-plugin/

Ausblick und Resümee

- Weitere Entwicklungen
- Resümee
- Literatur
- Interaktiv

Weitere Entwicklungen

- Scala 2.8
  - Named/default parameters
  - Nested annotations (wichtig für z.B. JPA)
  - Continuations
  - Interactive Interpreter (REPL)
  - Verbesserung der Standardbibliothek

Resümee (1)

Scala adds	Scala removes
reines Objekt-System	statische Member
Überladen von Operatoren	spezielle Behandlung primitiver Datentypen
Closures als Kontrollabstraktionen	break, continue
Mixin-Komposition mit Traits	spezielle Behandlung von Interfaces
Abstrakte Membertypen	Wildcards
Pattern Matching	

Resümee 2: Scala - Das konsequentere Java

- Structural typing
  - Java

```
String hello = new Object() {
  public String getHello() {
    return "Hello";
  }
}.getHello();

Object obj = new Object() {
  public String getHello() {
    return "Hello";
  }
}
obj.getHello();  // Fehler!
```

  - Scala

```
val obj = new Object {
  def getHello = "Hello"
}

obj.getHello
```

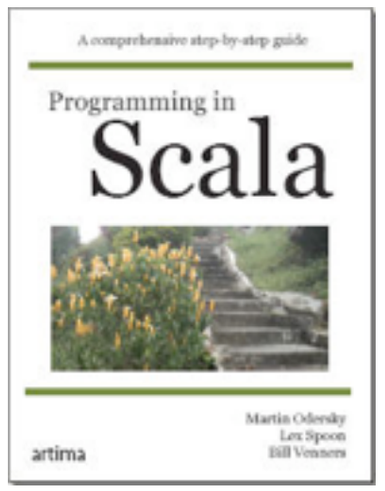
Resümee (3)

- Scala ist eine „typsichere dynamische“ Sprache
- Scala ist sehr effizient zu programmieren
- Scala bietet hohe Produktivität
- Scala macht Spaß!
- Mit Scala 2.8 für den produktiven Einsatz bereit

Literatur

- Homepage: http://www.scala-lang.org/
- Scala for Java Refugees:

  http://www.codecommit.com/blog/scala/scala-for-java-refugees-part-1
- Scalaz: http://code.google.com/p/scalaz/
  - Arrows, Functors, Monads - functional fun
- "Programming in Scala" (Artima, ISBN 0981531601)
- "Programming Scala" (Pragmatic Programmers, ISBN 193435631X)
- "Beginning Scala" (APress, ISBN 1430219890)



Interaktiv

- Mailinglisten

  http://www.scala-lang.org/community/index.html

- IRC: #scala@irc.freenode.net

  aktiv, Hilfe in allen Lagen

- Blog-Aggregation

  http://planetscala.com/

Vielen Dank!