Javas Zukunft: Die objektorientiert-funktionale Sprache Scala

Christoph Knabe

Beuth-Hochschule Berlin

http://public.beuth-hochschule.de/~knabe

Gliederung

- Einführung
- Java, aber einfacher
- Steuerkonstrukte, Traits (Interface-Aufrüstung)
- Java-Kompatibilität + Demo
- Pattern Matching
- Funktionale Features
- DSL-Beispiel
- Parallelität + Demo, Skalierbarkeit
- Vergleich zu AOP
- Das Web-Framework Play
- Fazit

Einführung: Zeit für Neues

- Java bei uns seit SS 1998 im Einsatz
- Anwendungsentwicklung noch immer unbequem
  - Oft blutleere Entity-Klassen
  - Oft Logik-Dienste getrennt von den Daten
  - Komplexe UI-Frameworks
  - Parallele Programmierung hochkomplex
  - Threads und Datenbanken nicht skalierbar
- Funktionale Programmierung auf Vormarsch
  - Ruby on Rails
  - C# 3.0 mit funktionalen Erweiterungen (u.a. LINQ), F#
  - Python, JavaScript funktional benutzt
  - Java 8 nahm funktionale Features auf.

Einführung: Wer steht hinter Scala

- Martin Odersky
  - Doktorand bei N. Wirth
  - Autor von pizza, GJ
  - Autor der Java5-Compiler-Implementierung
  - Prof. @ EPFL, Lausanne
- Lightbend
  - Kompentente Köpfe
    http://www.lightbend.com/company/leadership
  - Investoren
- Community
  - Meetups
    http://www.meetup.com/de-DE/topics/scala/
  - Weiteres

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

Einführung: Scala im Überblick

- Scala: "SCAlable LAnguage"
- Voll objektorientiert
- Voll Java-kompatibel
- Voll funktional
- Statisch typsicher ≠ Python, Ruby, JavaScript, …
- Redundanzfreie, orthogonale Syntax
- Actors, Futures, Streams für hohe Parallelität
- Gute Basis für Domain Specific Languages (DSL)

Einführung: Scala als Skriptsprache

- Als Eingabeaufforderung:
  - *scala↵*
    scala> ***val*** *r = 1↵*
    scala> *println(2*math.Pi*r*r)↵*
    6.283185307179586
    scala>
- Als Skriptdatei ohne umgebende Klasse:
  - dir.scala:
    ```
    val dir = new java.io.File(".")
    val files = dir.list
    files foreach println
    ```
  - *scala dir.scala↵*
    hello.scala
    printargs.scala
    pa.scala
    paramwithtypes.scala
    LernVerlauf.odt
    ...

Java, aber einfacher 1

- Jeder Wert ist ein Objekt:
  - scala> *1 .+(2)*↵
    res0: Int = 3
    scala> *1+2*↵
    res1: Int = 3
  - Scala-Werttypen werden effizient durch primitive Java-Typen implementiert.
- Jeder Operator ist eine Methode:
  - In Klasse BigInt:
    def + (that: BigInt): BigInt =
    new BigInt(**this**.bigInteger.add(that.bigInteger))
- Einheitlicher Lesezugriff
  (auf Konstante, Variable, parameterlose Funktion):
  - scala> ***val*** *n = list.size*
    n: Int = 1

Java, aber einfacher 2

- Typ-Inferenz, generische Parameter per [...]:
  - scala> ***val*** *map1 = **new** java.util.HashMap[String,Int]*↵
    map1: java.util.HashMap[String,Int] = {}
  - scala> ***val*** *map2: java.util.Map[String,Int] =* ↵
    | ***new*** *java.util.HashMap*↵
    map2: java.util.Map[String,Int] = {}
- Semikolon-Inferenz:
  - scala> ***val*** *a = 1*↵
    a: Int = 1
    scala> ***val*** *b = 2.0*↵
    b: Double = 2.0
    scala> ***val*** *c = (a*↵
    | *+ b)*
    c: Double = 3.0

Java, aber einfacher 3

- Klassenparameter definieren Konstruktor, Attribute und Zugriffsrechte:
  - **class** MyClass(x: Int, **val** y: Int, **var** z: Int){
    ...
    }
    val a = **new** MyClass(1,2,3)
    a.x //Jeder Zugriff verboten
    val b = a.y //Lesezugriff erlaubt
    a.y = 5 //Schreibzugriff verboten
    val c = a.z //Lesezugriff erlaubt
    a.z = 9 //Schreibzugriff erlaubt
- Zugriffsmethoden selbst definiert:
  - **class** MyClass { **private var** _a = 0
    def a = _a //getter: Default-Impl. Für Attribut a
    def a_=(a: Int){ _a = a } //setter: Default-Impl. Für a
    }
    val m = **new** MyClass; m.a = 9; assertEquals(m.a, 9)

Java, aber einfacher 4

- Java: Der Klasse-Objekt-Bruch:
  - **class** Counter {
    private static int count = 0;
    static void add(**final int** n){ count += n; }
    static void sub(**final int** n){ count -= n; }
    }
    Counter.add(5);
    Counter.sub(2);
  - An wieviel Stellen in Counter muss man ändern für mehrere?
- Scala: **object** definiert eine Singleton-Klasse:
  - **object** Counter {
    private var count = 0
    def add(n: Int){ count += n }
    def sub(n: Int){ count -= n }
    }
    Counter add 5
    Counter sub 2

Java, aber einfacher 5

- Java: Verschiedene Syntax für bedingte Teile:
  - if-Anweisung:
    if(condition)
      thenStmt
    else
      elseStmt
  - if-Ausdruck:
    condition **?** thenExpr **:** elseExpr
- Scala: Einheitliche Syntax für bedingte Teile:
  - **if** (condition) thenExpr **else** elseExpr
  - Bsp.: **if**(a<b){
      println(a)
    }**else**{
      println(b)
    }
  - Bsp.: **val** min = **if**(a<b) a **else** b

Java, aber einfacher 6

- Ein Block ist ein Ausdruck, liefert letzten Wert:
  - ```
    val result = { 1; 2 }
    assert(result === 2)
    ```
- Nützlich z.B. für Default-Ergebnis:
  - ```
    def findOrCreate(id: Long): Person =
    //Blocks and other statements have a result:
    try {
      db.find(classOf[Person], id)
    } catch {
       case ex: NotFoundException => new Person(-1, "")
    }
    ```

Java, aber einfacher 7

- **case** vor Klasse generiert alles für Value-Objekte:

```
case class Person(name: String, geburtsjahr: Int)
```

- Bsp.-Testfall:

```
val hansA = new Person("Hans", 1999) //Objekterzeugung mit new
val hansB = Person("Hans", 1999) // mit Factory-Methode
assertTrue(hansA == hansB) //Vergleich auf Wertgleichheit
assertFalse(hansA eq hansB) //Vergleich auf Objektidentität
assertEquals(hansA.name, hansB.name) //Attribut-Lesezugriffe
assertEquals(hansA.geburtsjahr, hansB.geburtsjahr)
assertEquals(hansA.hashCode, hansB.hashCode) //hashCode-Methode
```

Steuern: Schleifen

- while-Schleifen wie in Java
  - **while**(condition)
    expr
  - **do** expr **while**(condition)
- Keine mittelprüfende Schleife
  - Ohne **break**;
- For-Expression: Einfache Beispiele:
  - **for**(i **<-** 1 to 10) println(i)
  - **for**(i **<-** 1 to 10 by 2) println(i)
  - Letzte Schleife entspricht genau:

```
                                        //implicit conversion:
val one: scala.runtime.RichInt = Predef.intWrapper(1)
val oneToTen: scala.Range.Inclusive = one.to(10)
val oneToTenByTwo: scala.Range = oneToTen.by(2)
for(i <- oneToTenByTwo) println(i)
```

Steuern: Fallunterscheidungen

- match-Expression:
  - Ersetzt switch, viel mächtiger, Bsp.:

```
val firstArg = if (args.isEmpty) "" else args(0)
val friend =
firstArg match {
  case "salt" => "pepper"
  case "chips" => "salsa"
  case "eggs" => "bacon"
  case _ => throw new IllegalArgumentException(firstArg)
}
println(friend)
```

- Ausnahmebehandlung:
  -

```
try {
  doSomething()
} catch {
  case java.io.FileNotFoundException(name) =>
    println("File " + name + " not found")
  case ex: Exception =>
    throw new RuntimeException("Cannot doSomething", ex)
}
```

Traits

- Das Interface-Dilemma in Java:
  - Schmales Interface: Unkomfortable Nutzung
  - Breites Interface: Aufwändige Implementierungen
  - Bsp.: Interface `java.lang.CharSequence` definiert nur 4 Methoden: `charAt`, `length`, `subSequence`, `toString`. Andere, z.B. `trim()` müssten in allen Implementierern definiert werden.
- Ein Trait ist ein teilimplementiertes Interface:
  - **trait** CharSequence { ...
    def subSequence(start: Int, end: Int): CharSequence
    def trim = {
    val start = indexOfFirstNonwhiteChar(**this**)
    val end = indexOfFirstTrailingWhiteChar(**this**)
    subSequence(start, end)
    }
    }

Java-Kompatibilität

Scala kompiliert zu Java-Bytecode.

- Klassen, Methoden, Werttypen, Strings, Ausnahmen:
  Direkte Abbildung auf die entsprechenden Java-Konstrukte
- Ein Scala-Attribut wird zu überschreibbaren set/get-Methoden.
- **object** MyObj {}
  wird abgebildet auf
  final class MyObj$ **implements** scala.ScalaObject {
  static final MyObj$ MODULE$;
  MyObj$();
  }
- Ein Scala-Trait erzeugt ein Java-Interface und Weiteres.
  Ein abstrakter Scala-Trait wird genau zu einem Interface.
- Ein Java-Interface kann durch einen Scala-Trait/Klasse implementiert werden.
- Generizität: Fast 1:1-Entsprechung (~ wildcards)
- Annotations: Java-Annotations in Scala benutzbar
- Siehe Demo `FxScribble` oder `StoreScribble.scala`
  IDEA, Projekt beuth-knabe/ScalaVortrag, View > Enter Presentation Mode

Pattern Matching 1

- Matching extrahiert Teile typsicher:
  - ```
    val zahlen = List[Float](1,2,3,4,5)
    val (summe, mittel) = summeUndMittelwert(zahlen)
    println("Summe: " + summe + ", Mittel: " + mittel)
    ```
- Artikel Scala for Java Refugees: Pattern Matching
  - Color-Hierachie: http://www.codecommit.com/blog/scala/scala-for-java-refugees-part-4
  - Beispiel daraus: `PatternMatching.scala`
  - Erreicht typsichere Fallunterscheidung
  - Ersetzt **`instanceof`** und Type Cast
  - Geeignet für Inspektion eines Syntaxbaums nach dem Interpreter-Muster
    ```

Pattern Matching 2: Option

- Vordefinierte Klasse `Option[T]` deklariert ein optionales Attribut des Typs `T`:
  - ```
    class Person(
      val name: String, val eMail: Option[String]
    )
    ```
- **case**-Klasse `Some[T] extends Option[T]` enthält das Attribut:
  - `new Person("Horst", Some("horst@berlin.de"))`
- **case**-Klasse `None extends Option[T]` enthält kein Attribut:
  - `new Person("Horst", None)`
- Statisch typsichere Extraktion des optionalen Attributs mit match-Ausdruck:
  - ```
    p.name + eMail match { case Some(x) => "," + x
                           case None => ""          }
    ```

Funktional: „Call-by-Name“-Parameter

- Definition **Ausdrucks-Parameter** („call-by-name parameter“)
  - name**: =>** Typ
    Übergebener Ausdruck wird bei jeder Benutzung neu berechnet.
    (vs. Java: Nur Wert- oder Referenzparameter).

- Nützlich für z.B. Logging:
  - **def** debug(msg**: =>** String) = **if**(isDebugEnabled)println(msg)
    ...
    debug("Person " + p.name + " with id " + p.id)
  - Vorteil: Der aktuelle Parameter wird nur berechnet, wenn wirklich nötig.

Funktional: Eigene Steuerkonstrukte definierbar

- Problem: Bis Java 7 kein Rahmen um Anweisungen herum definierbar:
  - ⇒ **try-catch-Schema nicht auslagerbar!**
  - Jedoch nützlich für z.B. Transaktionen.
- Transaktionsschema selbst definiert:

```
def transaction(body: => Unit) = try{
    body;
    db.commit()
}catch{
    case t: Throwable =>db.rollback(); println(t)
}
```

- Transaktion benutzt:

```
def changeName(id: Long, name: String) = transaction {
  val person = db.find(classOf[Person], id)
  person.name = name
  db.save(person)
}
```

Funktionen: Werte erster Klasse

- Java 7: Methoden zweitrangig (vs. Objekte)
  - definieren √ aufrufen √
    Literal × speichern × übergeben × zurückgeben ×
- Funktionsliteral: $(x_1: T_1, x_2: T_2, \ldots) \Rightarrow ausdruck(x_1, x_2, \ldots)$
  - (n: Int) => n%2 == 0 //ist gerade Zahl?
- Funktion speichern:
  - **val** even = (n: Int) => n%2 == 0
- Funktion übergeben, hier an `filter` :
  - **val** numbers = 3 to 12
    val evenNumbers = numbers.`filter`(even)
    assert(evenNumbers === List(4, 6, 8, 10, 12))
- Funktion "höherer Ordnung" liefert eine Funktion:
  - **val** dividableBy3 = (n: Int) => n%3 == 0
    def bothHold(f: Int => Boolean, g: Int => Boolean) =
    (x: Int) => f(x) && g(x)
    val numbers6 = numbers.filter(bothHold(even, dividableBy3))
    assert(numbers6 === List(6, 12))

Funktional: Prägnanter Stil

- Bsp. hasUpperCase in Java:
  - ```
    final String name = …;
    boolean nameHasUpperCase = false;
    for (int i=0; i<name.length(); i++){
        if (Character.isUpperCase(name.charAt(i))){
            nameHasUpperCase = true;
            break;
        }
    }
    ```
  - Imperativer Stil: **`nameHasUpperCase`** und **`i`** sind variabel.
- Bsp. hasUpperCase in Scala:
  - ```
    val nameHasUpperCase = name.exists(_.isUpper)
    ```
  - Funktionaler Stil: Nur unveränderliche Werte
  - Schleife delegiert an Methode `exists` aus `Iterable[A]`:
    ```
    def exists(predicate: A => Boolean): Boolean
    ```

Functional meets Collections

- Viele vordefinierte Funktionen höherer Ordnung:
  - **`.filter(`*prädikat*`)`**: Liefert alle Elemente, die das Prädikat erfüllen.
  - **`.map(`*funktion*`)`**: Liefert *funktion*(x) für alle Elemente x.
  - **`.reduce(`*funktion*`)`**: Fusioniert alle Elemente zu einem Wert durch wiederholte Anwendung von x.*funktion*(y).
- Bsp. Umgang mit Personen → CollectionOperations:

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

val studenten = List(Person("Susi", 5), Person("Hans",
22), Person("Lisa", 20), Person("Franz", 48))

studenten.map(_.alter).filter(_ >= 18).reduce(_ + _)
```

Was berechnet das?

Wirtssprache für DSLs - - Domain Specific Languages

- Parsen: Grammatik für Arithmetic Expr. (EBNF)
  - expr ::= term {'+' term | '-' term}.
    term ::= factor {'*' factor | '/' factor}.
    factor ::= floatingPointNumber | '(' expr ')'.
- In Scala als DSL:
  - ```
    import scala.util.parsing.combinator._
    class Arith extends JavaTokenParsers {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term: Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
    }
    ```
  - Erklärung:

~	Sequenz
|	Alternative
rep	Iteration
opt	Optionalität

  - Siehe Objekt `ArithParse` in IDE
    ```

Bequeme Testsprache als DSL

- Mit ScalaTest lesbare Assertions:

```
@Test def listStoresElements {
  val list = new MutableList[Int]
  list += 1
  list += 2
  list should contain (1)
  list should contain (2)
  list should not contain (3)
}

@Test def getOnEmptyOptionThrows {
  val emptyOption = None
  intercept[NoSuchElementException] {
    emptyOption.get
  }
}
```

Siehe `sample.CollectionTest` im Projekt `scala-lecture`.

Actors für Massive Parallelität

- Java-Threads mit Risiken:
  - Race Conditions: geschwindigkeitsabhängiges undeterministisches Verhalten
  - Deadlocks: Stillstand durch gegenseitiges Warten
  - Zu speicheraufwändig für massive Parallelität (>5000 Threads)
    Java taugt für Millionen Objekte, aber nur Tausende Threads.
  - Zu zeitaufwändig für häufige Thread-Wechsel
- Scala: Actors im Erlang-Stil:
  - No Shared State ⇒ Keine Zugriffssynchronisation nötig!
  - Request/Response asynchron mit Message Passing
  - Nur unveränderliche Werte übergeben
  - Benötigt auch bei 1000en Actors nur 1 Thread je CPU
- Syntax:
  - Senden: actor ! message
  - Empfangen: **`def`** `receive =` { **`case`** *pattern* => *action*; ... }

Actors: Beispiel

- Benchmark-Beispiel:
  - Producer-Consumer-Kette aus n Threads | n Actors
  - Nachricht n-mal versenden ⇒ $O(n^2)$
  - In Java als Kette Thread-Buffer-Thread-Buffer ...
  - In Scala als Kette Actor-Message-Actor-Message ...
- Benchmark-Ergebnisse:

number	Threads [ms]	Actors [ms]
10	9	34
100	85	220
1,000	5,390	2,470
10,000	529,856	37,988

  - Threads für geringe Parallelität, Actors für **hohe**!
  - Siehe https://github.com/ChristophKnabe/actor-thread-benchmark

Skalierbarkeit: Hierarchische Pakete

- Hierachische Pakete:
  - Generischer Sichtbarkeitsmodifier **`private`**`[`*Component*`]`
  - *Component* kann Paket oder Klasse sein.
  - **`private`**`[a.b]` erlaubt Zugriff aus `a.b`, `a.b.c` usw., aber nicht aus `a.`*x*.
  - Bsp.: Pakete `probe.math`, `probe.math.impl`, Programm `probe.MathTest`
- Skalierbarkeit:
  - Von Skripten zu echt hierarchischen Paketen
  - Von sequentiell bis massiv parallel
  - Um eigene Sprachkonstrukte erweiterbar
  - Erweiterbarkeit zu DSLs

Das Web-Framework Play

- Skalierbar
  - Durchgängig asynchron (nichtblockierend)
  - Basiert auf Aktoren
- Modern Web & Mobile
  - REST-Stil standardmäßig
  - Websockets, Comet, EventSource
  - NoSQL & Big Data Support
- Entwicklerfreundlich
  - Ändern, Speichern, Laufen
  - Typsichere HTML-Templatemaschine

https://www.playframework.com/

Einsatz von Scala

- Einsatzbereiche:
  - Skalierbare Web-Backends
  - Big Data Analysis, Fast Data Serving
  - Komplexe Modellierung
- Job-Angebote in Berlin auf https://stackoverflow.com/jobs
  - Scala: 62 results
  - JavaScript: 144 results
  - Java: 169 results
- Firmen in Berlin:
  - Zalando
  - Lieferheld
  - SoundCloud
  - eBay Kleinanzeigen
  - ...

Fazit Scala

- Mächtige, orthogonale Konzepte
- Java-integriert
- Stark wachsende Community
- Leichter Übergang von Java aus
- IDE-Unterstützung: IDEA, Eclipse, Netbeans
- Für umfangreiche Geschäftslogik, Parallelität, Leistung
  besser als Java, Ruby, Python, JavaScript