Scala (Programmiersprache)

Ein Teaser und allgemeinere Gedanken

Sebastian Eidecker

16. März 2016

Wer als Werkzeug nur einen Hammer hat, sieht in jedem Problem einen Nagel.

Paul Watzlawick

Worüber reden wir? IT im Wandel

IT im Wandel

Herausforderungen

Manifeste

IT im Wandel

Herausforderungen



IT im Wandel Herausforderungen

Manifeste Scala

IT im Wandel

Herausforderungen

Manifeste

Scala

Management Summary

IT im Wandel

Herausforderungen

Manifeste

Scala

Management Summary

Ein wenig Code

IT im Wandel

Herausforderungen

Manifeste

Scala

Management Summary

Ein wenig Code

Spannendes

IT im Wandel

Herausforderungen

Software Engineering

Software Engineering

· Stabilität und Resilienz

- · Stabilität und Resilienz
- Wertbeitrag

- · Stabilität und Resilienz
- Wertbeitrag
- Businesstreiber

- · Stabilität und Resilienz
- Wertbeitrag
- Businesstreiber

Matthias Magnor – CEO Surface und Contract Logistics

IT im Wandel

Manifeste



 Antwortbereit, Widerstandsfähig, Elastisch, Nachrichtenorientiert (2013)

• Gut gefertigt, Stets Mehrwert, Gemeinschaft aus Experten, Produktive Partnerschaften (2009)

 Individuen und Interaktionen, Funktionierende Software, Zusammenarbeit mit dem Kunden, Reagieren auf Veränderung (2001)

- Antwortbereit, Widerstandsfähig, Elastisch, Nachrichtenorientiert (2013)
- Gut gefertigt, Stets Mehrwert, Gemeinschaft aus Experten, Produktive Partnerschaften (2009)
- Individuen und Interaktionen, Funktionierende Software, Zusammenarbeit mit dem Kunden, Reagieren auf Veränderung (2001)

Wo stehen wir?

Scala

Management Summary

Scalable Language

Scalable Language

This means that Scala grows with you. You can play with it by typing one-line expressions and observing the results. But you can also rely on it for large mission critical systems [...]

— www.scala-lang.org

Objektorientiert

- Objektorientiert
- Funktional

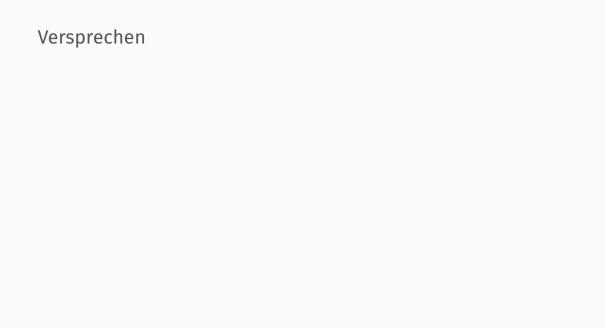
- Objektorientiert
 - Funktional
 - Statisch typisiert mit Type Inference

- Objektorientiert
- Funktional
- Statisch typisiert mit Type Inference
- Immutable by default

- Objektorientiert
- Funktional
- Statisch typisiert mit Type Inference
- · Immutable by default
- Gewohnte Syntax ("Java ohne Semikolon")

- Objektorientiert
- Funktional
- Statisch typisiert mit Type Inference
- Immutable by default
- Gewohnte Syntax ("Java ohne Semikolon")
- Ausdrucksstark (APIs/DSLs)

- Objektorientiert
- Funktional
- Statisch typisiert mit Type Inference
- · Immutable by default
- · Gewohnte Syntax ("Java ohne Semikolon")
- Ausdrucksstark (APIs/DSLs)
- Jung (2004, Hype 2011)



Produktivitätssteierung

- Produktivitätssteierung
- · Höhere Codequalität

- Produktivitätssteierung
- · Höhere Codequalität
- Mehr Spaß

- Produktivitätssteierung
- Höhere Codequalität
- Mehr Spaß durch
- Weniger Code

- Produktivitätssteierung
- Höhere Codequalität
- Mehr Spaß durch
- Weniger Code
- · Höheres Abstraktionsniveau

- Produktivitätssteierung
- · Höhere Codequalität
- Mehr Spaß
 durch
- Weniger Code
- · Höheres Abstraktionsniveau
- Skalierbarkeit

· Java-Bytecode, läuft auf JVM

Scala und die Java-Plattform

· Java-Bytecode, läuft auf JVM

· Java-Bibliotheken nutzbar

Bekannte IDFs

- · Java-Bytecode, läuft auf JVM
- · Java-Bibliotheken nutzbar

- · Java-Bytecode, läuft auf JVM
- · Java-Bibliotheken nutzbar
- Bekannte IDEs
- · Ähnliches Toolset, oft wiederverwendbar

Scala

Ein wenig Code



Eine Java-Klasse

13

```
public class Person {
    private final String firstName;
    private final String lastName:
    public Person(String firstName, String lastName) {
        this.firstName = firstName:
5
         this.lastName = lastName:
7
    public String getFirstName() {
8
        return firstName:
10
    public String getLastName() {
11
         return lastName:
12
```

Eine Java-Klasse

14

15

16

17

18

19

20

21

22

23

24

25 26

```
aOverride
public boolean equals(Object o) {
    if (this == 0) return true:
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o:
    if (firstName != null ?
        !firstName.equals(person.firstName) :
        person.firstName != null) return false:
    if (lastName != null ?
        !lastName.equals(person.lastName) :
        person.lastName != null) return false:
    return true:
```

Eine Java-Klasse

Businesslogik?



Dasselbe in Scala

Dasselbe in Scala

case class Person(firstName:String, lastName:String)

Es wird funktional – Quicksort

Es wird funktional – Quicksort

Schnelldurchlauf Scala – Endlich!

Variablen – val und var

```
Variablen – val und var
```

```
var j = 3
j = 4
```

```
Variablen – val und var
```

```
_{1} var j = 3
```

2 **j = 4**

3 **val** k = 3

 $_{4}$ k = 4

```
Variablen – val und var
```

```
2  j = 4
3  val k = 3
```

 $_{1}$ **var** j = 3

3 **val** k = 3

Compile-Fehler für k, da immutable

```
Variablen – val und var
```

```
_1 var j = 3
2 j = 4
```

 $_{4}$ k = 4

```
3 val k = 3
```

5 **val** k: Int = 3

```
Compile-Fehler für k, da immutable
```

Methoden

Methoden

1 **def** f = 3 * 2

```
Methoden
```

1 **def** f = 3 * 2

```
def f(): Int = 3 * 2
```

```
Methoden
```

```
_{1} def f = 3 * 2
```

2 def f(): Int = 3 * 2

3 def f(i: Int) = 3 * i

```
Methoden

def f = 3 * 2
```

```
def f(): Int = 3 * 2
```

```
3 def f(i: Int) = 3 * i
```

4 def f(i: Int) = {

5 3 * i



Implizites return

Implizites return

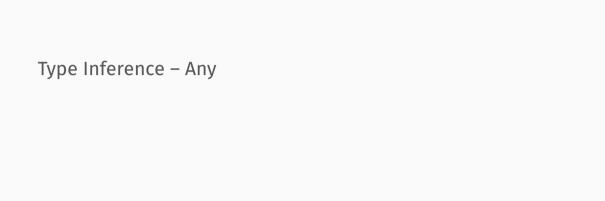
```
1 def f() = {
2    if (something)
3    "A"
4    else
```

"B"

Implizites return

```
1 def f() = {
2    if (something)
3    "A"
4    else
5    "B"
6  }
```

Letzte Anweisung wird zurückgegeben, impliziter Typ String.



```
Type Inference – Any
```

```
1 def f() = {
```

```
if (something)
```

else

```
Type Inference – Any def f() = {
```

```
1 def f() = {
2    if (something)
3    "A"
4    else
5    1
```

Erste gemeinsame Oberklasse, zur Not Any



Unit

```
def print(s: String): Unit = println(s)
```

Parameter sind vals

```
Parameter sind vals
```

def addOne(i: Int): Int = { i += 1; i }

Parameter sind vals

def addOne(i: Int): Int = { i += 1; i }

Compile-Fehler, da i immutable



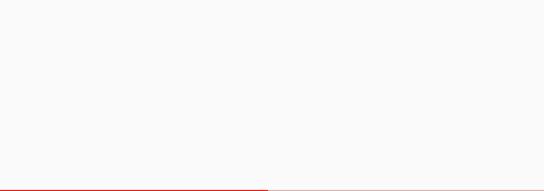
```
val numbers = List(1, 2, 3, 4)
```

```
val numbers = List(1, 2, 3, 4)
```

val bagContent = List(1, "a", Time(12,30), 4)

```
val numbers = List(1, 2, 3, 4)
val bagContent = List(1, "a", Time(12,30), 4)
```

Immutable-Listen bevorzugen



Funktionen höherer Ordnung

val numbers = List(1, 2, 3, 4)

Funktionen höherer Ordnung

Funktionen höherer Ordnung

```
val numbers = List(1, 2, 3, 4)
```

```
val numbersDecreasing = numbers.sortWith((x, y) => x > y)
```

Funktionen höherer Ordnung

val numbersPlusOne = numbers.map(x => x + 1)

```
val numbers = List(1, 2, 3, 4)
```

```
val numbersDecreasing = numbers.sortWith((x, y) => x > y)
```

val numbersPlusOne = numbers.map(x => x + 1)

```
val numbersPlusOne = numbers.map(x => x + 1)
```

```
val addTwo = (n: Int) => n + 2
```

```
val numbersPlusOne = numbers.map(x => x + 1)
```

```
_2 val addTwo = (n: Int) => n + 2
```

val numbersPlusTwo = numbers.map(addTwo)

```
val numbersPlusOne = numbers.map(x => x + 1)
```

```
val addTwo = (n: Int) => n + 2
```

```
val numbersPlusTwo = numbers.map(addTwo)
```

```
4 val descending = (n: Int, m: Int) => n > m
```

```
val numbersPlusOne = numbers.map(x => x + 1)
```

```
val addTwo = (n: Int) => n + 2
```

```
val numbersPlusTwo = numbers.map(addTwo)
```

```
4 val descending = (n: Int, m: Int) => n > m
```

5 val sortedNumbers = numbers.sortWith(descending)



Klassen

def this() = this(0)

5

```
class Time(val hours: Int = 0, val minutes: Int = 0) {
  // Primärer Konstruktor
  require(hours < 24 && hours >= 0)
  require(minutes < 60 && minutes >= 0)
```

Klassen

5

```
class Time(val hours: Int = 0, val minutes: Int = 0) {
  // Primärer Konstruktor
  require(hours < 24 && hours >= 0)
  require(minutes < 60 && minutes >= 0)
```

```
def this() = this(0)
```

```
8 val t = new Time(1, 14)
9 val t2 = new Time
```

Benannte Parameter und Standardwerte

Benannte Parameter und Standardwerte

class Time(val hours: Int = 0, val minutes: Int = 0)

Benannte Parameter und Standardwerte

```
class Time(val hours: Int = 0, val minutes: Int = 0)
```

```
val t = new Time(1)
```

```
3 val t2 = new Time(minutes = 13)
```

Singleton/Companion Objects

Singleton/Companion Objects

```
object Time {
  def fromMinutes(minutes: Int): Time =
     new Time(minutes / 60, minutes % 60)
```

Singleton/Companion Objects

5 val t = Time.fromMimutes(100)

```
object Time {
  def fromMinutes(minutes: Int): Time =
       new Time(minutes / 60, minutes % 60)
}
```



Case classes

case class Person(nachname: String, vorname: String)

case classe

Case classes

```
case class Person(nachname: String, vorname: String)
```

- 5 val ich = Person("Eidecker", "Sebastian")
- 6 val sohn = ich.copy(vorname = "Nils")

Vererbung und Traits

Vererbung und Traits

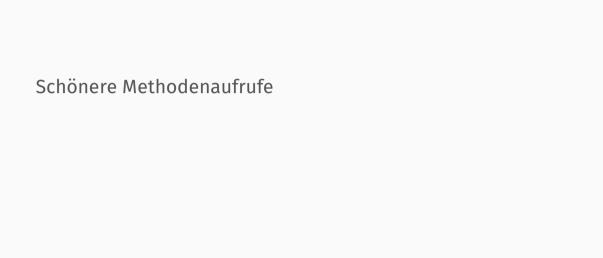
• Geht

Vererbung und Traits

- Geht
- Mehrfachvererbung durch Traits

Vererbung und Traits

- Geht
- Mehrfachvererbung durch Traits
- · Zu wenig Zeit heute



Schönere Methodenaufrufe

```
"Test".startsWith("T")
List(1,2,3).isEmpty
```

Schönere Methodenaufrufe

```
1 "Test".startsWith("T")
2 List(1,2,3).isEmpty
1 "Test" startsWith "T"
```

2 List(1,2,3) isEmpty

```
case class Time(hour:Int, minute:Int) {
   def minus(time: Time) = {new Time(this.hour - time.hour, this.
        minute - time.minute)}
```

Time(10.20).minus(Time(1.10))

```
case class Time(hour:Int, minute:Int) {
  def minus(time: Time) = {new Time(this.hour - time.hour, this.
      minute - time.minute)}
}
```

Time(10,20).minus(Time(1,10))
Time(10.20) minus Time(1.10)

Time(10,20).minus(Time(1,10))
Time(10.20) minus Time(1.10)

```
case class Time(hour:Int, minute:Int) {
  def minus(time: Time) = {new Time(this.hour - time.hour, this.
      minute - time.minute)}
  def -(time: Time) = minus(time)
}
```

Time(10,20).minus(Time(1,10))
Time(10,20) minus Time(1,10)
Time(10,20) - Time(1,10)

```
case class Time(hour:Int, minute:Int) {
  def minus(time: Time) = {new Time(this.hour - time.hour, this.
      minute - time.minute)}
  def -(time: Time) = minus(time)
}
```

Pattern Matching

Pattern Matching

4

8

11

```
val alice = new Person("Alice", 25)
 val bob = new Person("Bob", 32)
 val charlie = new Person("Charlie", 32)
  for (person <- List(alice, bob, charlie)) {</pre>
    person match {
      case Person("Alice", 25) => println("Hallo Alice!")
      case Person("Bob", 32) => println("Hallo Bob!")
      case Person(name, age) => println("Alter: " + alter + ", Name:
            + name)
```

Pattern Matching – Funktional Länge einer Liste

```
Pattern Matching – Funktional Länge einer Liste
```

```
def length[A](list : List[A]) : Int = {
 list match {
```

case :: tail => 1 + length(tail)

case Nil => 0



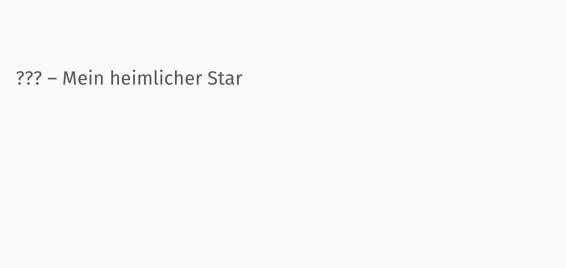
Tupel

```
val ichMitAlter = (Person("Eidecker", "Sebastian"), 37)
```

Tupel

```
val ichMitAlter = (Person("Eidecker", "Sebastian"), 37)
val ich = ichMitAlter._1
val alter = ichMitAlter. 2
```

Tupel



```
??? - Mein heimlicher Star
case class Time(hour:Int, minute:Int) {
```

```
def minus(time: Time) = ???
```

```
??? - Mein heimlicher Star

case class Time(hour:Int, minute:Int) {
  def minus(time: Time) = ???
}
```

Kompilierbar, aber nicht gefährlich.

Und noch viel, viel mehr ...

Scala

Spannendes

Fachliche Abstraktion

- Fachliche Abstraktion
- Verständlicher

- Fachliche Abstraktion
- Verständlicher
- Entwicklung schwierig

- Fachliche Abstraktion
- Verständlicher
- Entwicklung schwierig
- · Einschränkungen vorhanden

DSL - Beispiel

- 2 110 INPUT 'burn
- 3 120 IF ABS('burn) <= 'fuel THEN 150
- 4 130 PRINT "You don't have that much fuel"
- 140 GOTO 100
- 5 150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))

Scalatest

```
"Creating a Time" should {
    "throw an IllegalArgumentException for hours less than 0 or
       greater equal 24" in {
      forAll("hours") { (hours: Int) =>
        whenever(hours < 0 || hours >= 24) {
          an[IllegalArgumentException] should be thrownBy Time(
             hours)
```

· Nebenläufige Einheiten

- Nebenläufige Einheiten
 - Empfangen Nachrichten (Ereignisse)

- Nebenläufige Einheiten
- Empfangen Nachrichten (Ereignisse)
- Abarbeitung FIFO

- Nebenläufige Einheiten
- Empfangen Nachrichten (Ereignisse)
- Abarbeitung FIFO
- Verhaltensänderung

- Nebenläufige Einheiten
- Empfangen Nachrichten (Ereignisse)
- Abarbeitung FIFO
- Verhaltensänderung
- Asynchrone Kommunikation mit Aktoren



Aktoren

- Aktoren
- Fehlertoleranz

- Aktoren
- Fehlertoleranz
- Standort-Transparenz

- Aktoren
- Fehlertoleranz
- Standort-Transparenz
- Nachrichten-Persistenz

- Aktoren
- Fehlertoleranz
- Standort-Transparenz
- Nachrichten-Persistenz
- · Reaktiv laut Manifest

Akka – Ping-Aktor

Akka – Ping-Aktor

```
case object PingMessage
case object PongMessage
```

case object StartMessage case object StopMessage

```
5
```

class Ping(pong: ActorRef) extends Actor {

```
var count = 0
```

def incrementAndPrint { count += 1; println("ping") }

```
Akka - Ping-Aktor

def receive = {
    case StartMessage =>
        incrementAndPrint
        pong ! PingMessage
    case PongMessage =>
```

} else {

incrementAndPrint

if (count > 99) {

sender! StopMessage

sender! PingMessage

context.stop(self)

println("ping stopped")

Akka - Pong-Aktor

```
class Pong extends Actor {
  def receive = {
    case PingMessage =>
        println(" pong")
        sender ! PongMessage
    case StopMessage =>
        println("pong stopped")
        context.stop(self)
```

Mögliche Bedeutung

· Start, Stop, Nachrichten, Zustand

- · Start, Stop, Nachrichten, Zustand
- · Klingt verdächtig nach Prozessen

- · Start, Stop, Nachrichten, Zustand
- · Klingt verdächtig nach Prozessen
- · Nebenläufig, skalierbar, resilient
- · Mein Traum: Direkte Abbildung

- · Start, Stop, Nachrichten, Zustand
- · Klingt verdächtig nach Prozessen
- · Nebenläufig, skalierbar, resilient
- · Mein Traum: Direkte Abbildung
- Scheint machbar

Pattern Matching



- Pattern Matching
 - Funktional

- Pattern Matching
- Funktional
- Weniger Code

- Pattern Matching
- Funktional
- Weniger Code
- Besser verständlich

Vorteile

 Für moderne Architekturen

- Für moderne Architekturen
- · Verständlich funktional

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem

- Für moderne Architekturen
- · Verständlich funktional
- · Java-Ökosystem
- Macht Spaß

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem
- Macht Spaß
- Statisch typisiert

Vorteile

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem
- Macht Spaß
- Statisch typisiert

Vorteile

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem
- Macht Spaß
- Statisch typisiert

Nachteile

Komplex

Vorteile

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem
- Macht Spaß
- Statisch typisiert

- Komplex
- · Zukunftssicher?

Vorteile

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem
- Macht Spaß
- Statisch typisiert

- Komplex
- · Zukunftssicher?
- Anzahl
 Entwicklungssklaven

Vorteile

- Für moderne Architekturen
- · Verständlich funktional
- Java-Ökosystem
- Macht Spaß
- Statisch typisiert

- Komplex
- · Zukunftssicher?
- Anzahl
 Entwicklungssklaven
- Binärkompatibilität nicht in alle Ewigkeit

We've found that Scala has enabled us to deliver things faster with less code. It's reinvigorated the team.

— Graham Tackley, Guardian



Sprecht mich an

- Sprecht mich an
- · Hands on-Termin bei Interesse

- Sprecht mich an
- · Hands on-Termin bei Interesse
- Heiko Seeberger: "Durchstarten mit Scala. Tutorial für Einsteiger (2. Aufl.)"

- · Sprecht mich an
- · Hands on-Termin bei Interesse
- Heiko Seeberger: "Durchstarten mit Scala. Tutorial für Einsteiger (2. Aufl.)"
- Martin Odersky: ""

