

Scala

Lightning Talk
@JSUG

Future Languages for the Java VM

Michael Greifeneder

Overview

- Mix of functional and OO programming
- Static typing (performance)
- FP: Pattern matching, High order functions, Parametric Polymorphism, Operator overloading
- Closures
- Open Source
- Functions are objects
- Perfect for domain specific languages

Vorteile

- Scala ist kompatibel mit Java.
 - Alle Bibliotheken können weiterverwendet werden.
- Scala skaliert
 - Sprache kann fuer hoehere Abstraktionsebenen angepasst werden. zB Actors!
- Kann als Scripting Sprache verwendet werden
- Kurzer Code, aussagekraeftig

- `abstract class` `SomethingWithName` (`name:String`) {
 `override def toString():String = name`
 `def` `does` (`what:(SomethingWithName)=>unit`) {
 `what(this);}`}
- `case class` `Person` (`name:String`) `extends`
`SomethingWithName` (`name:String`) {
 `def` `says` (`msg: String`) = {
 `print(name + " says " + msg)`}
 `def` `apply` (`name:String`):`Person` = `new Person(name)`}
- `case class` `Stone` (`name:String`) `extends`
`SomethingWithName` (`name:String`) {
 `def` `apply` (`name:String`):`Bagger` = `new Bagger(name)`}
 `def` `swimming` (`who:SomethingWithName`) = `who match` {
 `case Person(name)=>print("Swims in pool: " + name)`
 `case Stone(name)=>print("On the ground: " + name)`
- `var` `Tom` = `Person("Tom")`
`var` `YellowStone` = `Stone("YellowStone")`
`Tom says "Hello"`
`Tom does swimming //Tom.does(swimming)`
`YellowStone does swimming;`

Code

Output

- `>scala dsl.scala`
- `Tom says Hello`
- `Swimming in pool: Tom`
- `On the ground: YellowBagger`

Skriptsprache !?

- Folge von Statements
- Zugriff auf Kommandozeilenargumente

```
println("Hello " + args(0) + "!")
```

- Ausführen

```
scala test.scala
```

- Shellscript

```
- #!/bin/sh
```

```
exec scala "$0" "$@"
```

```
!#
```

```
println("Hello, " + args(0) + "!")
```

Closures

- Tom does `((in:SomethingWithName) => { print("Who needs closures? Maybe " + in + "?") })`
- Output:
 - Who needs closures? Maybe Tom?
- Auch nett:
 - `args.foreach(arg => println(arg))`
- Noch kürzer
 - `args.forach(println)`

XML Processing

Syntax Sugar

Functional Style

Ease of Use

- Maps

- `val map = Map(
 "Austria" -> "Vienna",
 "Germany" -> "Berlin",
 "Switzerland" -> "Bern")
 map("Austria"`

- List

- `val list = List(1,2,3)`
`val list2 = 1 ::: 2 ::: 3 ::: Nil`

Short Code

- Keine Strichpunkt notwendig.
- Keine geschwungenen Klammern notwendig bei einem Ausdruck
 - `def method = x`
- First class functions
 - `args.foreach(println)`

-

Matching Expressions

- ```
val friend = args(0) match {
 case "salt" => "pepper"
 case "chips" => "salsa"
 case "eggs" => "bacon"
 case _ => "huh?"
}
```
- Keine Breaks notwendig
- Match kann auch Ergebnis liefern

# Catching Exceptions

- ```
try {
    doSomething()
} catch {
    case ex: IOException =>
        println("Oops!")
    case ex: NullPointerException =>
        println("Oops!!")
}
```

Actors! (And Action!)

- ```
object PrintActor extends Actor{
 def act() = {
 var num = 0;
 react{
 case msg:String =>
 println(num + msg)
 case c:Int => num+=c
 }
 }
}
```

```
Tester.start(); Tester ! 2
Tester ! "First Message";
```