



First steps in Scala

Andreas Neumann
a.neumann@smarchive.de

www.an-it.com



Überblick

- Scala - Kurzer Geschichte
- Teil 1:
 - Code ausführen / kompilieren
 - Syntax
 - Variablen und Values
 - Funktionen / Methoden
 - Kontrollstrukturen
 - Collections (kurzer Auszug)
 - Klassen, Traits und Objekte
- Teil 2
 - Pattern Matching
 - Option[Type]
 - XML erzeugen und bearbeiten
- Reguläre Ausdrücke
- Parallele Collections
- Implicits
- Funktional vs. imperative Konstrukte

Was zeichnet Scala aus ?

- JVM (.Net)
- Kombination objektorientierter und funktionaler Programmierparadigmen
- Interoperabel zu Java

Wer steht hinter Scala - Martin Odersky

- Professor an der EPFL
- Mitarchitekt am Java-Compiler
- Generics für Java 1.4
- PIZZA
- Scala
- Gründer Typesafe



Wer steht hinter Scala - Typesafe

- Training
- Consulting
- Scala
- Actor
- Play



Stärken

- Skalierbarkeit
- Parallelisierbarkeit
- Integrierbarkeit
- Erweiterbarkeit
- XML-Verarbeitung?
- DSL

Teil I

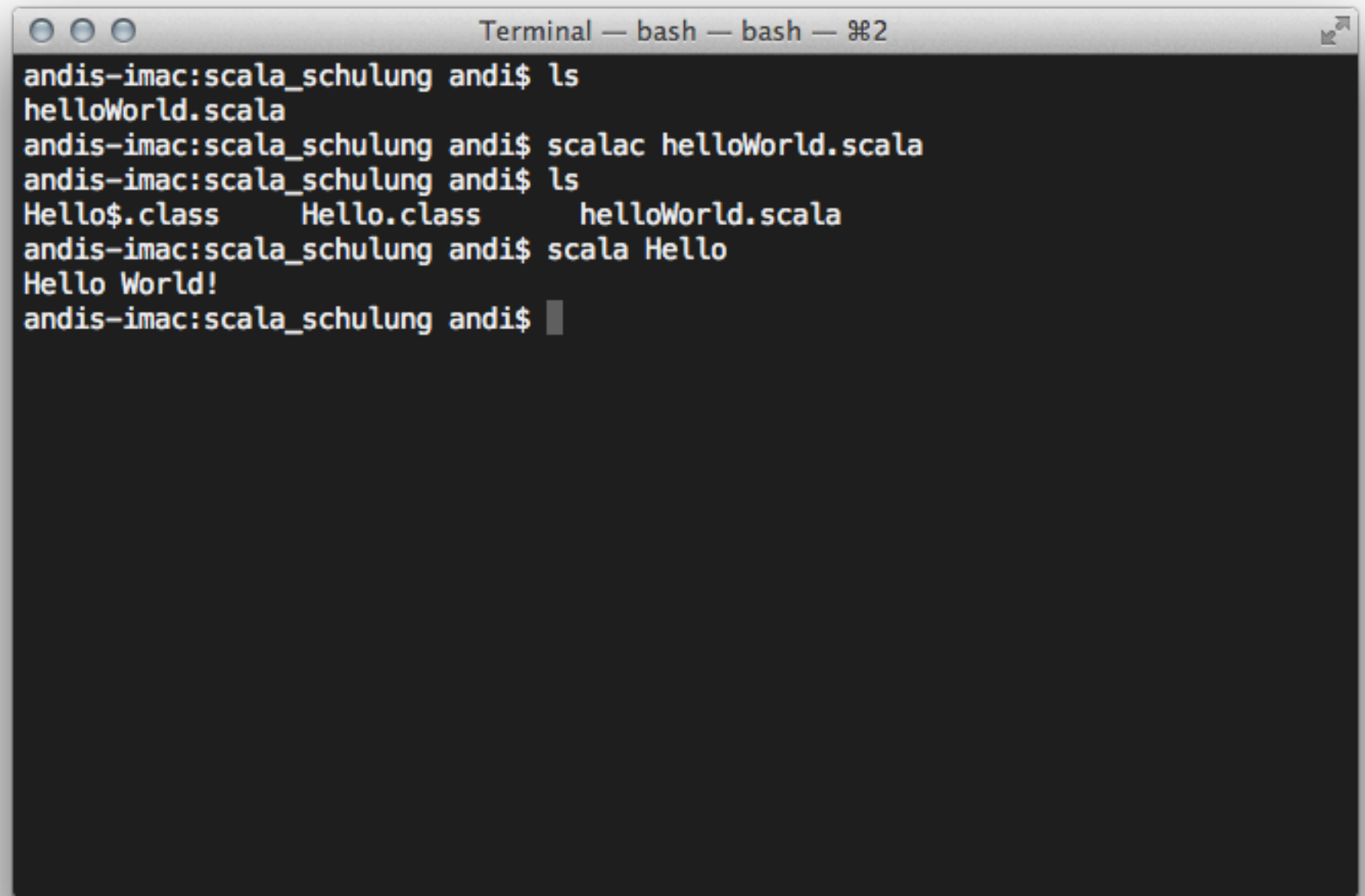
- Code ausführen / kompilieren
- Anmerkungen zur Syntax
- Variablen und Values
- Funktionen / Methoden
- Kontrollstrukturen
- Collections (kurzer Auszug)
- Klassen, Traits und Objekte

Code ausführen, compilieren

- Scala Code kann unter unterschiedlichsten Umgebungen ausgeführt und benutzt werden:
 - *Compilieren und auf einer JVM ausführen*
 - *Als Script direkt ausführen*
 - *In REPL, zum testen*

Scala - Compiler / scalac

- Compiler wird mit **scalac** aufgerufen
- Klassen werden in JavaBytecode kompiliert
- Aufruf mit **scala** **<Klassenname>**

A terminal window titled "Terminal — bash — bash — 2" showing the process of compiling and running a Scala program. The user is in the directory "andis-imac:scala_schulung". They list the files, showing "helloWorld.scala". Then they compile it with "scalac helloWorld.scala". They list the files again, showing "Hello\$.class", "Hello.class", and "helloWorld.scala". Finally, they run the program with "scala Hello", which outputs "Hello World!".

```
andis-imac:scala_schulung andi$ ls
helloWorld.scala
andis-imac:scala_schulung andi$ scalac helloWorld.scala
andis-imac:scala_schulung andi$ ls
Hello$.class      Hello.class      helloWorld.scala
andis-imac:scala_schulung andi$ scala Hello
Hello World!
andis-imac:scala_schulung andi$
```

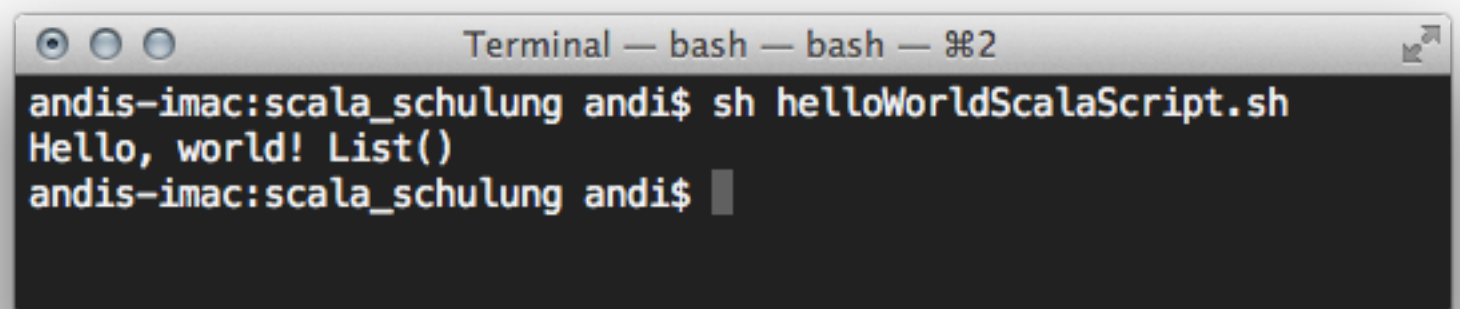
Scala als Scriptsprache - Unix

- Scala kann auch als Skriptsprache eingesetzt werden
- Der Aufruf erfolgt mit sh oder man kann die Datei ausführbar machen

```
#!/bin/sh
exec scala "$0" "$@"
!#

object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world! " +
            args.toList)
  }
}

HelloWorld.main(args)
```



A terminal window titled "Terminal — bash — bash — 2" showing the execution of the script. The prompt is "andis-imac:scala_schulung andi\$". The command "sh helloWorldScalaScript.sh" is entered, followed by the output "Hello, world! List()". The prompt returns to "andis-imac:scala_schulung andi\$".

```
Terminal — bash — bash — 2
andis-imac:scala_schulung andi$ sh helloWorldScalaScript.sh
Hello, world! List()
andis-imac:scala_schulung andi$
```

Scala als Scriptsprache - Windows

- Scala als Scriptsprache unter Windows
- In eine batch-Datei, Endung .bat speichern
- Aufrufen

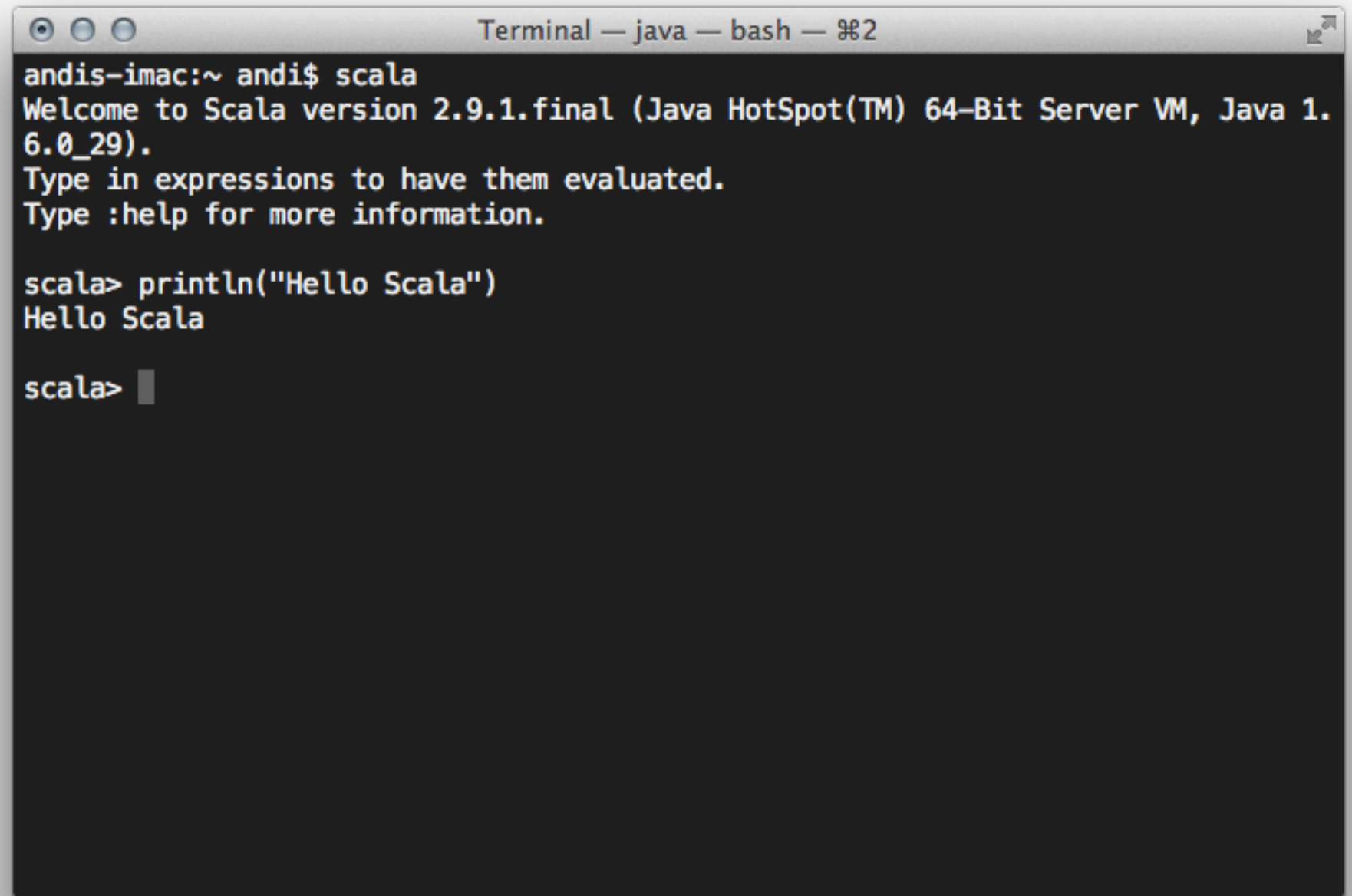
```
::#!  
@echo off  
call scala %0 %*  
goto :eof  
::!#
```

```
object HelloWorld {  
  def main(args: Array[String]) {  
    println("Hello, world! " +  
            args.toList)  
  }  
}
```

```
HelloWorld.main(args)
```

Scala - Interpreter / REPL

- Bei der Installation von Scala wird neben dem compiler auch ein Interpreter installiert
- Starten des Interpreters: `$ scala`



```
Terminal — java — bash — %2
andis-imac:~ andi$ scala
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_29).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello Scala")
Hello Scala

scala> █
```

Online REPL



Simply Scala

Created by Anthony Bagwell

Home	Comments	About	Learn More Scala
------	----------	-------	------------------

Welcome to Simply Scala 2.8

Scala is a modern computer programming language.
Here you can discover more about its features in
a simple interactive way.

Now available in [French](#) as well.

Creating user space...
Ready for code.

```
for(i<-List("olleH", " ", "!dlrow")){  
  print(i.reverse)  
}
```

Reset

Evaluate

- <http://www.simplyscala.com/>

Prinzipielles zur Syntax

- Keine abschließende Semikolons notwendig, eine Anweisung pro Zeile
- Mehrere Anweisungen pro Zeile können getrennt mit einem Semikolon angegeben werden
- Bei eindeutiger Syntax kann der Punkt bei Methodenaufrufen weggelassen werden

```
1.+ (5)
```

```
1 + 5
```

```
List(1,2.3).map(_ * 3).head  
List(1,2.3) map(_ * 3) head
```

val, vars

- Das Keyword **val** erzeugt einen Value, einen *unveränderbaren* Wert, vergleichbar mit **final** in Java
- Das Keyword **var** erzeugt eine Variable im klassischen Sinn

Beispiel: val und var

```
val x = 42
x: Int = 42
var y = 99
y: Int = 99
y = 1
y: Int = 1
x = 1
error: reassignment to val
      x = 1
      ^
```


Typen und Typinferenz

- Typangaben werden mit `:` eingeleitet und stehen hinter dem Value/der Variablen

```
s: String = "ein String"
```

- In vielen Fällen sind sie optional, da die Typinferenz den Typ selbst ermitteln kann

```
val a = "Hallo"  
a: java.lang.String = Hallo  
val b = 1  
b: Int = 1  
val c = 3.5  
c: Double = 3.5  
val d = List(1, 2.0)  
d: List[Double] = List(1.0, 2.0)
```

Methoden definieren

- Methoden werden mit dem Keyword **def** eingeleitet
- Darauf folgt eine optionale Parameterliste in runden Klammern
- Am Schluß folgt der Funktionsrumpf
- Hat die Methode eine Rückgabewert wird der Funktionsrumpf mit einem **=** angeschlossen
- Der letzte Wert in einer Methode ist der Rückgabewert

```
def write(aString: String) {  
    println(aString)  
}  
  
write("Hallo ihr alle da draußen!")  
Hallo ihr alle da draußen!
```

```
def add(x: Int, y: Int) : Int = {  
    x + y  
}  
  
add(40, 2)  
res0: Int = 42
```

Kontrollstrukturen

- if, else
- while
- foreach, map
- For-Comprehensions

Klassische Kontrollstrukturen

- Kontrollstrukturen wie **while**, **if** und **else** funktionieren wie gewohnt
- Im Gegensatz zu Java sind es aber echte Funktionen, d.h. Sie haben einen Rückgabewert
- if gibt den Wert des Blocks zurück wenn wahr
- while hat den Rückgabewert Unit

```
val x = if ( 1 < 2 ) true  
x: AnyVal = true
```

„Funktionale Kontrollstrukturen“

- Alles Funktionen
- Können gleiches Ergebnis erreichen wie klassische Kontrollstrukturen
- **Map** : Eine Funktion auf jedes Element anwenden, Ergebnis zurückliefern

```
List(1,2,3,4) map (x => x + 1)  
res1: List[Int] = List(2, 3, 4, 5)
```

- **Foreach**: Eine Funktion auf jedes Element anwenden, Ergebnis verwerfen

```
List(1,2,3) foreach( x => println("And a " + x))  
And a 1  
And a 2  
And a 3
```

For-Comprehensions I

- Eleganter Weg um lesbar mehrere Operationen zu verbinden
- Wird intern in eine folge von map, filter, flatMap und reduce Operationen umgewandelt

```
def isEven(x: Int) = x % 2 == 0
val integers = for {
  x <- 1 to 99
  if isEven(x)
  if x % 5 == 0
} yield x
integers: scala.collection.immutable.IndexedSeq[Int] =
Vector(10, 20, 30, 40, 50, 60, 70, 80, 90)
```

```
~ Umgewandelt ~
(1 to 99).filter(isEven(_)).filter( x % 5 == 0)
```

Klassen

- Klassen werden mit dem Keyword **class** eingeleitet
- Gefolgt von optionalen Konstruktorelementen in Klammern
- Daraufhin folgt optional der Körper der Klasse in geschweiften Klammern
- Besonderheiten
 - Für Konstruktorelemente die mit dem Keyword **val** eingeleitet werden wird automatisch ein **getter** unter dem gleichen Namen erzeugt (uniform access principle)
 - Für Konstruktorelemente die mit dem Keyword **var** eingeleitet werden wird automatisch ein **getter** und ein **setter** unter dem gleichen Namen erzeugt (uniform access principle)
 - Alle aufgerufenen Operationen im Body der Klasse werden bei deren Konstruktion aufgerufen

Beispiel: Klasse in Scala

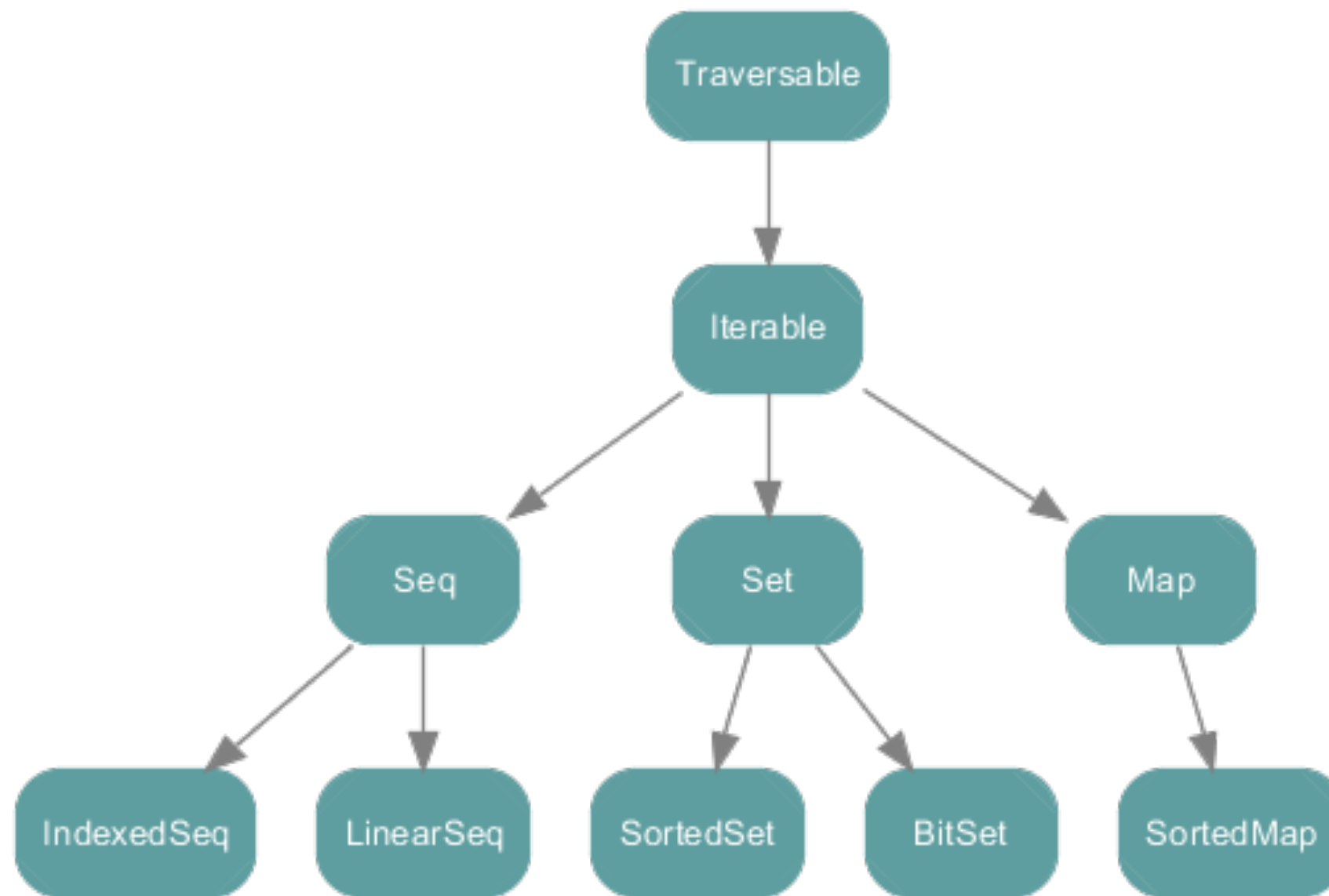
```
class Document(val title: String, val author: String, yearInt: Int) {  
    val year = yearInt.toString  
    def shortCitation: String = author + " : " + title + ". " + year  
}  
  
val scalaBook = new Document("Programming In Scala", "Martin Odersky", 2011)  
  
println(scalaBook.title)  
println(scalaBook.year)
```

- Instanzen werden mit **new** <Klassenname> erzeugt

Collections

- Scala wartet mit einer großen Collections-Bibliothek auf
- Collections bieten ein nach Möglichkeit einheitliches Interface
- Es gibt Collections in bis zu vier Ausführungen:
 - Basis
 - immutable : nicht Veränderbar
 - mutable : Veränderbar
 - (parallel: Parallele Version)

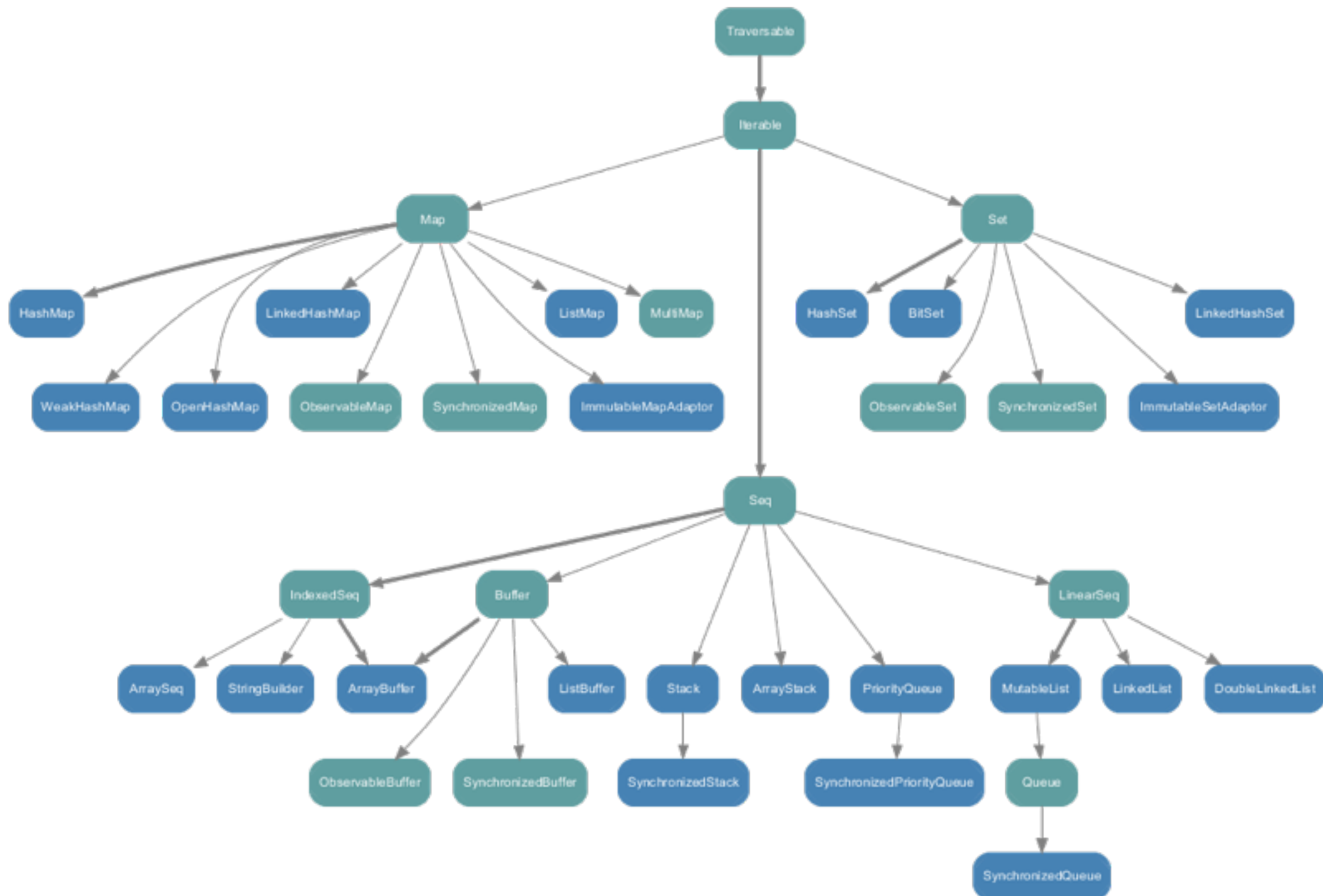
Scala Collection



Scala Collection - immutable



Scala Collection - mutable



Scala Collections : List

```
val a = List("a","b","c")
a: List[java.lang.String] = List(a, b, c)

a head
res5: java.lang.String = a

a tail
res6: List[java.lang.String] = List(b, c)

val b = List(1,2,3)
b: List[Int] = List(1, 2, 3)

0 :: b
res10: List[Int] = List(0, 1, 2, 3)

a ++ b
res11: List[Any] = List(a, b, c, 1, 2, 3)

a zip b
res12: List[(java.lang.String, Int)] = List((a,1), (b,2), (c,3))
```

Scala Collections : map

```
val counting = Map(1 -> "eins", 2 -> "zwei", 3 -> "drei")
counting:
scala.collection.immutable.Map[Int,java.lang.String] =
Map((1,eins), (2,zwei), (3,drei))

counting(2)
res2: java.lang.String = zwei

counting.get(2)
res3: Option[java.lang.String] = Some(zwei)

counting.get 99
res4: Option[java.lang.String] = None
```

Scala Object - Objekte

- Objekte werden mit dem keyword **object** eingeleitet
- Sie haben keine Konstruktorelemente
- Funktionsweise entspricht grob einer Javaklasse mit static modifizieren
- Aufrufe von Members und Methoden erfolgt über **Objektname.member** bzw. **Objektname.methode**
- Idiomatischer Weg um in Scala eine **Singleton-Object** zu erzeugen

Scala Object - Beispiel

```
object DeepThought {  
  val theMeaningOfLife =  
    "The meaning of life: 42"  
  
  def speak {  
    println(theMeaningOfLife)  
  }  
}
```

```
DeepThought.speak  
The meaning of life: 42
```


Companion Object

- Scala typisch
- Ein **object**, welches den gleichen Namen wie eine Klasse trägt
- Natives **Constructor Pattern**

Case Class

- Wird mit dem Keywords **case class** eingeleitet
- Ansonsten wie bei der „normalen Klasse“
- Definiert eine **companion object** mit **apply**, **unapply** und einigen anderen Methoden
- Für alle Konstruktorargumente werden **getter** erzeugt
- Sollte nicht vererben!

Case Class - Beispiel

```
case class Book(title: String, pages :Int)
defined class Book

val book = Book("Necronomicon",1000)
book: Book = Book(Necronomicon,1000)

println( book.title )
Necronomicon
```

Trait

- Wird mit dem keyword **trait** eingeleitet
- Ähnlich einem Java-Interface
- Erlaubt es einem der Mehrfachvererbung ähnlichen Effekt zu erzeugen ohne die Probleme, die dieses Verfahren üblicherweise mit sich bringt
- Bausteine
- Wie Ruby mixins

Trait - Beispiel Definition

- Es werden zwei Eigenschaftern als **Traits** definiert: *Edible* und *ExoticTaste*
- Zwei Klassen werden definiert: *Cake*, welcher vom Trait *Edible* „erbt“
- Klasse: *ChiliChocolate* welche sowohl von *Edible* als auch *ExoticTaste* erbt

```
trait Edible {  
    def taste: String  
    def eat = println(taste)  
}  
  
trait ExoticTaste {  
    def eat: Unit  
    def describeTaste = {  
        eat  
        println("It tastes exotic")  
    }  
}  
  
case class Cake extends Edible {  
    def taste = "sweet"  
}  
  
case class ChiliChocolate extends Edible with  
    ExoticTaste {  
    val taste = "sweet and hot"  
}
```

Trait - Beispiel : Anwendung

```
val cake = new Cake  
cake.eat
```

```
val chiliChoc = new ChiliChocolate  
chiliChoc.eat  
chiliChoc.describeTaste
```

```
val cake = new Cake  
cake: Cake = Cake()  
  
cake.eat  
sweet  
  
val chiliChoc = new ChiliChocolate  
chiliChoc: ChiliChocolate = ChiliChocolate()  
  
chiliChoc.eat  
sweet and hot  
  
chiliChoc.describeTaste  
sweet and hot  
It tastes exotic
```

Teil 2

- Pattern Matching
- Option[Type]
- XML erzeugen und bearbeiten
- Reguläre Ausdrücke
- Parallele Collections
- Implicits

Pattern-Matching

- Pattern matching
- Wird durch keyword **match** eingeleitet
- Mehrere cases die mit keyword **case** eingeleitet werden
- Ähnlich Java switch , aber mit weitaus mehr möglichkeiten
- **Pattern Guards** erweitern Möglichkeiten
- **Case Classes** und Extractor Patterns erlauben einfaches zerlegen

Matching Example

```
case class Book( title: String, pages: Int, year: Int)

val books = List(
  Book("Programming Scala", 883, 2012),
  Book("Programming Perl", 1104, 2000),
  Book("Necronomicon", 666, 666),
  "Ein String", 5, 42
)

val bookComments = books.map( book => book match {
  case Book("Programming Scala", pages, year) => "New Scala Book by Martin Odersky from " + year
  case Book(title, pages, year) => title + " " + pages + " " + year
  case x: Int if x > 10 => "an integer bigger than 10"
  case _ => "Something else"
})
```

```
// Ausgabe
bookComments: List[java.lang.String] = List(
New Scala Book by Martin Odersky from 2012,
Programming Perl 1104 2000,
Necronomicon 666 666,
Something else,
Something else,
an integer bigger than 10)
```

Der Option-Type

- Rückgabewert für Funktionen die ein wohldefiniertes Ergebnis liefern **können**, aber zu erwarten ist, dass dies nicht immer der Fall ist
- Scala-Implementierung von Haskells Maybe
- Ein Konzept um null-Checks bzw. null-Pointer-Exceptions zu vermeiden

Option-Type Beispiel: Option vs. null

```
val bigBangPHD = Map(  
  "Leonard" -> "Ph.D.",  
  "Sheldon" -> "Ph.D., Sc.D",  
  "Rajesh" -> "Ph.D"  
)  
  
val friends = List("Leonard", "Sheldon", "Rajesh", "Howard")
```

```
bigBangPHD("Leonard")  
res0: java.lang.String = Ph.D.  
  
bigBangPHD("Howard")  
java.util.NoSuchElementException:  
key not found: Howard  
    at scala.collection.MapLike  
$class.default(MapLike.scala:223)  
    at scala.collection.immutable.Map  
$Map3.default(Map.scala:132)
```

```
bigBangPHD.get("Leonard")  
res1: Option[java.lang.String] =  
Some(Ph.D.)  
  
bigBangPHD.get("Sheldon")  
res2: Option[java.lang.String] =  
Some(Ph.D., Sc.D)  
  
bigBangPHD.get("Howard")  
res3: Option[java.lang.String] =  
None
```

Option -Type : Beispiele für Integration in Scala

- Option ist tief in Scala integriert
- Es existieren viele Methoden die mit Option umgehen können

```
// Liste mit Options erzeugen
friends map(bigBangPHD.get(_))
res4: List[Option[java.lang.String]] = List(Some(Ph.D.), Some(Ph.D.,Sc.D),
Some(Ph.D), None)

// flatten entfernt None und „entpackt“ Some(thing)
friends map(bigBangPHD.get(_)) flatten
res5: List[java.lang.String] = List(Ph.D., Ph.D.,Sc.D, Ph.D)

// for comprehensions wenden Operationen nur auf Some() an und verwerfen
None
for {
  person <- friends
  phd <- bigBangPHD.get(person)
} yield person + " has a " + phd

res6: List[java.lang.String] = List(Leonard has a Ph.D., Sheldon has a
Ph.D.,Sc.D, Rajesh has a Ph.D)
```

Option -Type : Beispiele für Integration in Scala 2,

```
// getOrElse erlaubt es einen Standardrückgabewert für None anzugeben,  
ansonsten wird Some(thing) „ausgepackt“  
friends.map(p => (p, bigBangPHD.get(p))).  
map( m => m._1 + " " + m._2.getOrElse("Sheldon tells me you only have a  
master's degree."))
```

```
res7: List[java.lang.String] =  
List(Leonard Ph.D.,  
      Sheldon Ph.D.,Sc.D.,  
      Rajesh Ph.D.,  
      Howard Sheldon tells me you only have a master's degree.)
```

```
// Option Types besitzen Extraktoren für Pattern Matching  
friends.map(bigBangPHD.get(_)) zip friends map {  
  case (Some(phd), name ) => name + " : " + phd  
  case (None, name) => name + " is just an engineer"  
}
```

```
res10: List[java.lang.String] = List(Leonard : Ph.D.,  
      Sheldon : Ph.D.,Sc.D.,  
      Rajesh : Ph.D.,  
      Howard is just an engineer)
```

XML in Scala

- XML ist in Scala eine Sprachelement wie z.B. String in Java oder Ruby
- Es kann also als **Literal** in den Quellcode eingebunden werden

Scala - XML erzeugen

- XML darf direkt im Scala Code stehen
- Moderne Entwicklungsumgebungen (Eclipse, Netbeans, IntelliJ) bieten Syntaxhighlighting
- Variablen und Code können mit `{}` in XML Literale eingebettet werden

XML erzeugen - Beispielcode

```
case class Book( title: String, pages: Int, year: Int) {  
  def toXML =  
<book>  
  <title>{title}</title>  
  <pages>{pages toString}</pages>  
  <year>{year toString}</year>  
</book>  
  
}  
  
val books = List( Book("Programming Scala", 883, 2012), Book("Programming  
Perl", 1104, 2000), Book("Necronomicon", 666, 666) )  
  
for ( book <- books ) {  
  println(book.toXML)  
}
```


XML erzeugen - Ausgabe

```
<book>
  <title>Programming Scala</title>
  <pages>883</pages>
  <year>2012</year>
</book>
<book>
  <title>Programming Perl</title>
  <pages>1104</pages>
  <year>2000</year>
</book>
<book>
  <title>Necronomicon</title>
  <pages>666</pages>
  <year>666</year>
</book>
```

Scala XML verarbeiten

- XML kann mit XPath - ähnlicher Syntax verarbeitet werden (\\ anstatt // und \ anstatt /)
- <xml></xml> \ **"tag"** : für einen Shallow -Match
- <xml></xml> \\ **"tag"** : für einen Deep -Match
- <xml></xml> \\ **"@attribut"** : für einen Deep -Match auf ein Attribut
- (<xml></xml> \ "tag").**text** : Methode text gibt den Textwert des Knotens zurück

XML verarbeiten - Beispielcode

```
case class Book( title: String, pages: Int, year: Int) {
  def toXML =
    <book>
      <title>{title}</title>
      <pages>{pages}</pages>
      <year>{year}</year>
    </book>

  implicit def intToString(in : Int) : String = in.toString
}

object Book {
  def fromXML(bookXML: scala.xml.NodeSeq) : Book= {
    val title = (bookXML \\ "title").text
    val pages = (bookXML \\ "pages").text.toInt
    val year = (bookXML \\ "year").text.toInt
    new Book(title, pages, year)
  }
}
```

XML verarbeiten - Aufruf und Ausgabe

```
val books =
<books>
  <book>
    <title>Programming Scala</title>
    <pages>883</pages>
    <year>2012</year>
  </book>
  <book>
    <title>Programming Perl</title>
    <pages>1104</pages>
    <year>2000</year>
  </book>
  <book>
    <title>Necronomicon</title>
    <pages>666</pages>
    <year>666</year>
  </book>
</books>

val booksInstances = (books \\ "book").map(Book.fromXML(_))
val booksPages = (books \\ "pages").map(_.text.toInt)
```

```
booksInstances: scala.collection.immutable.Seq[Book] = List(Book(Programming
Scala,883,2012), Book(Programming Perl,1104,2000), Book(Necronomicon,666,666))
booksPages: scala.collection.immutable.Seq[Int] = List(883, 1104, 666)
```

Reguläre Ausdrücke

- .r aus einem String erzeugen: `""""href\s?=\s?" ([^"]+) """".r`
- Nutzt Java-Regex-Engine
- Automatische Erzeugung von Extraktoren für Pattern-Matching

Regex - Beispiel

```
import scala.io.Source

val html = Source.fromURL("http://www.an-it.com").getLines.mkString("")

val urlExtractor = """"href\s?=\s?"([^\"]+)"""".r

for {
  urlExtractor(url) <- (urlExtractor findAllIn html).matchData
  } {
    println("Url ->" + url)
  }
}
```

```
Url ->/stylesheets/an-it.css?1323020119
Url ->mobile_stylesheets/mobile.css
Url ->/
Url ->/
Url ->/vortraege
Url ->/websites
Url ->/projekte
Url ->/kontakt
Url ->/impressum
Url ->http://www.neumann.biz/cv
```

first-order-functions / Anonyme Funktionen

- Funktionen sind ein Sprachelemente wie Integer oder String
- Sie lassen sich als Argumente an Funktionen übergeben

```
val y = (x: Int) => x * x  
y: (Int) => Int =
```

```
y.apply(5)  
res10: Int = 25
```

```
y(5)  
res11: Int = 25
```

```
val add = (x: Int, y: Int) => x + y  
add: (Int, Int) => Int =
```

```
add(1, 2)  
res12: Int = 3
```

Implicits

- Werden mit dem keyword **implicit** eingeleitet
- Automatisch Umwandlung
- Nicht stapelbar
- Kann Code kürzen und duplikationen vermeiden
- **Pimp my library Pattern:** Ähnlich monkey-patching , kann aber lokal begrenzt werden

Implicit: Beispiel

```
case class Book( title: String, pages: Int, year: Int) {  
  def toXML =  
    <book>  
      <title>{title}</title>  
      <pages>{pages}</pages>  
      <year>{year}</year>  
    </book>  
  
  implicit def intToString(in : Int) : String = in.toString  
}
```

Parallele Collections

- Asynchrone, parallele Verarbeitung nutzt moderne Mehrkernprozessorarchitekturen aus
- Methode **.par** verwandelt normale Collection in parallele Version
- Methode **.seq** verwandelt parallele Collection in lineare Version
- Parallelität ist als Trait implementiert => eigene parallele Collections möglich
- Auch für Maps

Parallele Collections - Beispielcode

```
// Sequentiell  
(1 to 5).foreach(println)
```

```
scala> (1 to 10).foreach(println)  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

```
// Parallel  
(1 to 5).par foreach(println)
```

```
scala> (1 to 10).par.foreach(println)  
6  
7  
8  
9  
10  
3  
4  
5  
2  
1
```

Parallele Collections - Beispielcode II

```
// Ungeordnet
```

```
val tenTimes = (1 to 10).map(_ * 10)
```

```
//Geordnet
```

```
//.seq verwandelt parallele Collection //wieder in eine sequentielle
```

```
val tenTimes = (1 to 10).map(_ * 10).seq
```

Eigene Kontrollstrukturen definieren

- Curryfizierte Funktionen erlauben Funktionen zu definieren, die sich wie Kontrollstrukturen anfühlen

```
object ControlStructures {  
  def unless( test: => Boolean)(action: => Any) =  
    if (! test) {action}  
  
  def times( n: Int )(action: => Unit) {  
    (1 to n).foreach { i => action}  
  }  
}
```

```
times(2) { println("Hoorray :)")}  
Hoorray :)  
Hoorray :)
```

```
unless (5 < 10) { println("Math stopped working.") }  
  
val ifNot = unless (2 + 2 != 4) { "Math still works." }  
ifNot: Any = Math still works.
```

Scala - imperativ, objektorientiert funktional

Scala - imperativ, objektorientiert funktional - Faustregeln

- Funktional wenn möglich
- Aber: Manches lässt sich imperativ intuitiver und performanter lösen
- Mit vals und immutable Collections beginnen, auf vars und mutable collections ausweichen
- Objektorientierung nutzen um Seiteneffekte oder imperative Programmteile zu kapseln

Funktional - Vorteile

- Kurz
- keine Seiteneffekt -> leichter parallelisierbar

Imperativ - Vorteile

- Intuitiv
- Endgültige Abbildung ist (bis jetzt) stets imperativ

Imperativ vs. Funktional, ein paar Beispiele

Imperativ

```
var x = 1
var sum = 0
while (x <= 9999) {
    sum += x
    x += 1
}
```

```
var i = 0
while (i < args.length) {
    if ( i != 0 )
        print(" ")
    print( args(i) )
    i += 1
}
println()
```

Funktional

```
(1 to 9999) foldLeft(0)(_ + _)
```

```
(1 to 9999) sum
```

```
println(
    args reduce ( (acc,arg) =>
        acc + " " + arg
    )
)
```

```
println( args mkString " " )
```

Literatur:

- Wampler, D., & Payne, A. (2009). Programming Scala. Sebastopol, CA: O'Reilly.
- Odersky, M., Spoon, L., & Venners, B. (2008). Programming in Scala. Mountain View, Calif: Artima.
- Odersky M., Zenger M. Scalable Component Abstractions
- Malayeri, M. “Pimp My Library” Is An Affront To Pimpers Of The World, Everywhere
- <http://www.scala-lang.org>

Vielen Dank für die Aufmerksamkeit :)