Clueda

Beyond big data

An Introduction to Scala

Andreas Neumann, Clueda
a.neumann@clueda.com

Overview

- A short history of Scala
- Using Scala Basics: Compile and run
- Introduction to the core concepts : vals, classes, objects, traits and functions
- Using Scala: Option, XML, Futures
- Patterns

About Scala

- A JVM Languge
- mutltiparadigmatic
- functional object orientated

The father of Scala - Martin Odersky

- Professor EPFL
- Java-Compiler
- Java 1.4 Generics
- PIZZA
- Scala
- Co-Founder Typesafe

The company behind Scala - Typesafe

- Training
- Consulting
- Scala
- Actor
- Play

Who is using Scala

- Xerox
- Twitter
- Foursquare
- Sony
- Siemens
- clueda
- …

Nice features

- Can be integrated in any Java based architecture
- support concurrent and asynchronous operations out of the box in many flavors (Futures,Parallel Collections, Akka)
- great for embedded DSLs

Introduction

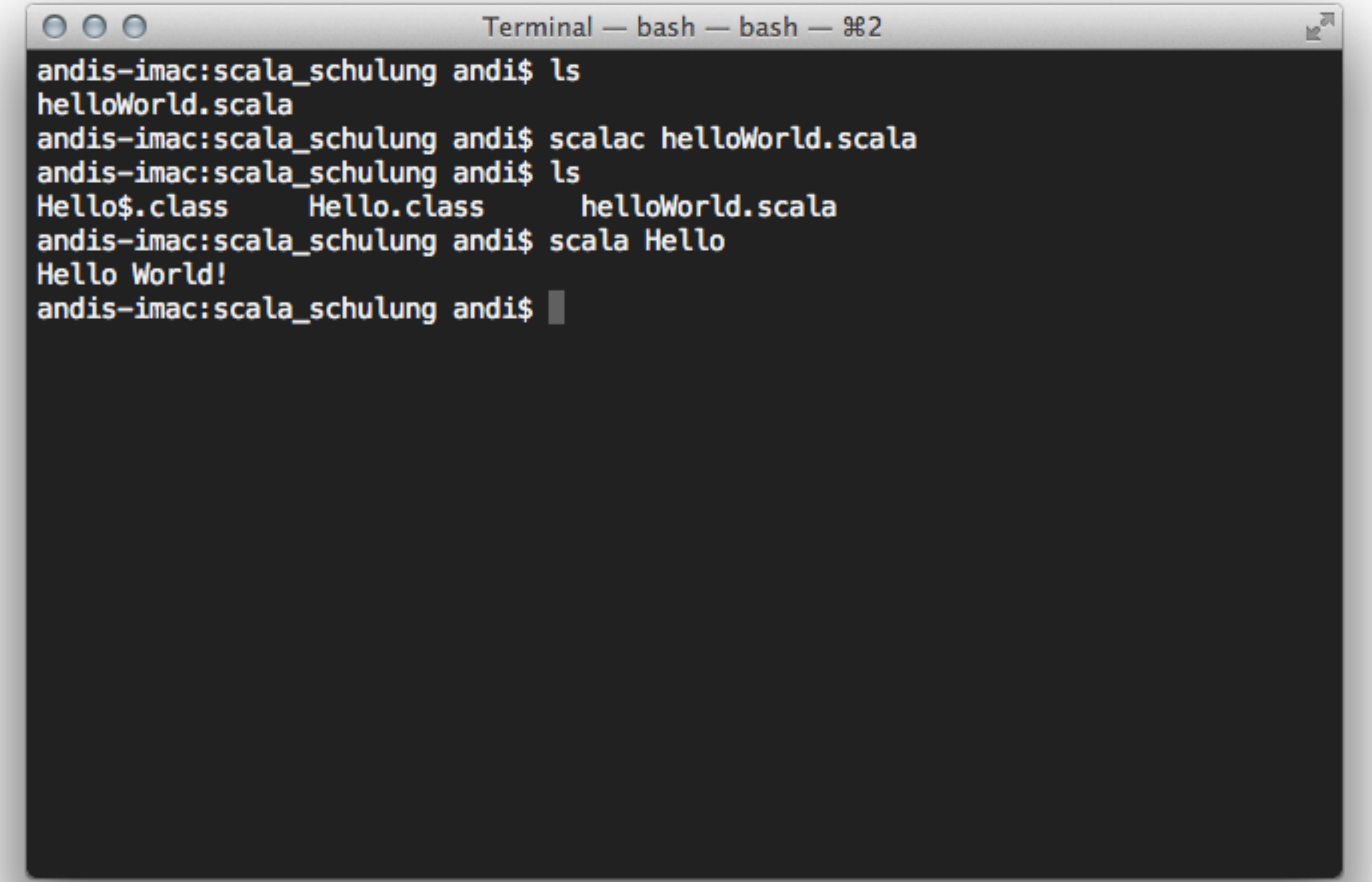
- Run code / compile Code
- Syntax (some tips)
- declare variables
- functions
- clases, traits and objects

Run code, compile sources

- Scala can be used in many different environments:
  - *compile and run on a **JVM***
  - *run as a **script***
  - *in an interactive console, called **REPL***

Scala - compiler / scalac

- the compiler is called **scalac**
- The sources are compiled to Java byte code
- *.class
- run class with **scala <name>**

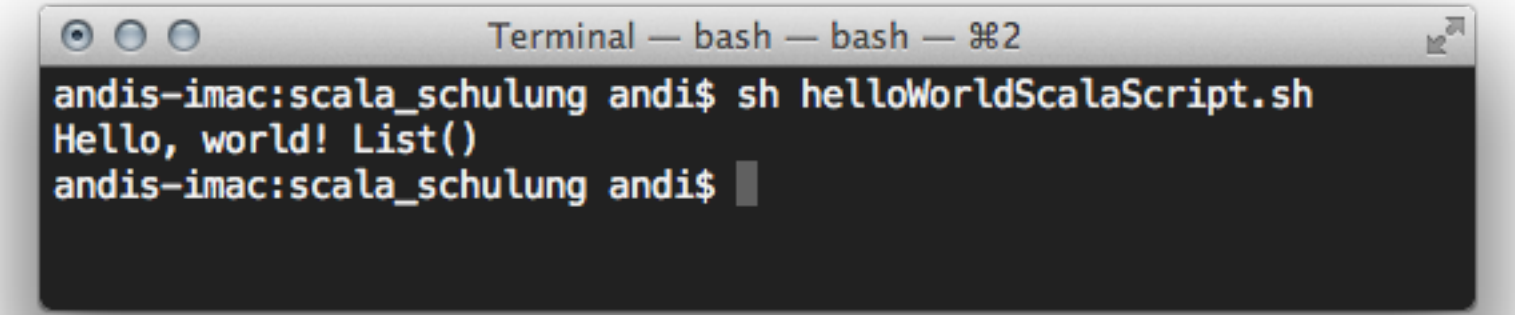


```
andis-imac:scala_schulung andi$ ls
helloWorld.scala
andis-imac:scala_schulung andi$ scalac helloWorld.scala
andis-imac:scala_schulung andi$ ls
Hello$.class     Hello.class      helloWorld.scala
andis-imac:scala_schulung andi$ scala Hello
Hello World!
andis-imac:scala_schulung andi$
```

Scripting with Scala - Unix

- Scala can be used as scripting language
- change mode to excuteable or run by sh

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

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

HelloWorld.main(args)
```



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

Scripting with Scala - Windows

- Resembles UNIX use a batch script instead
- *.bat
- run

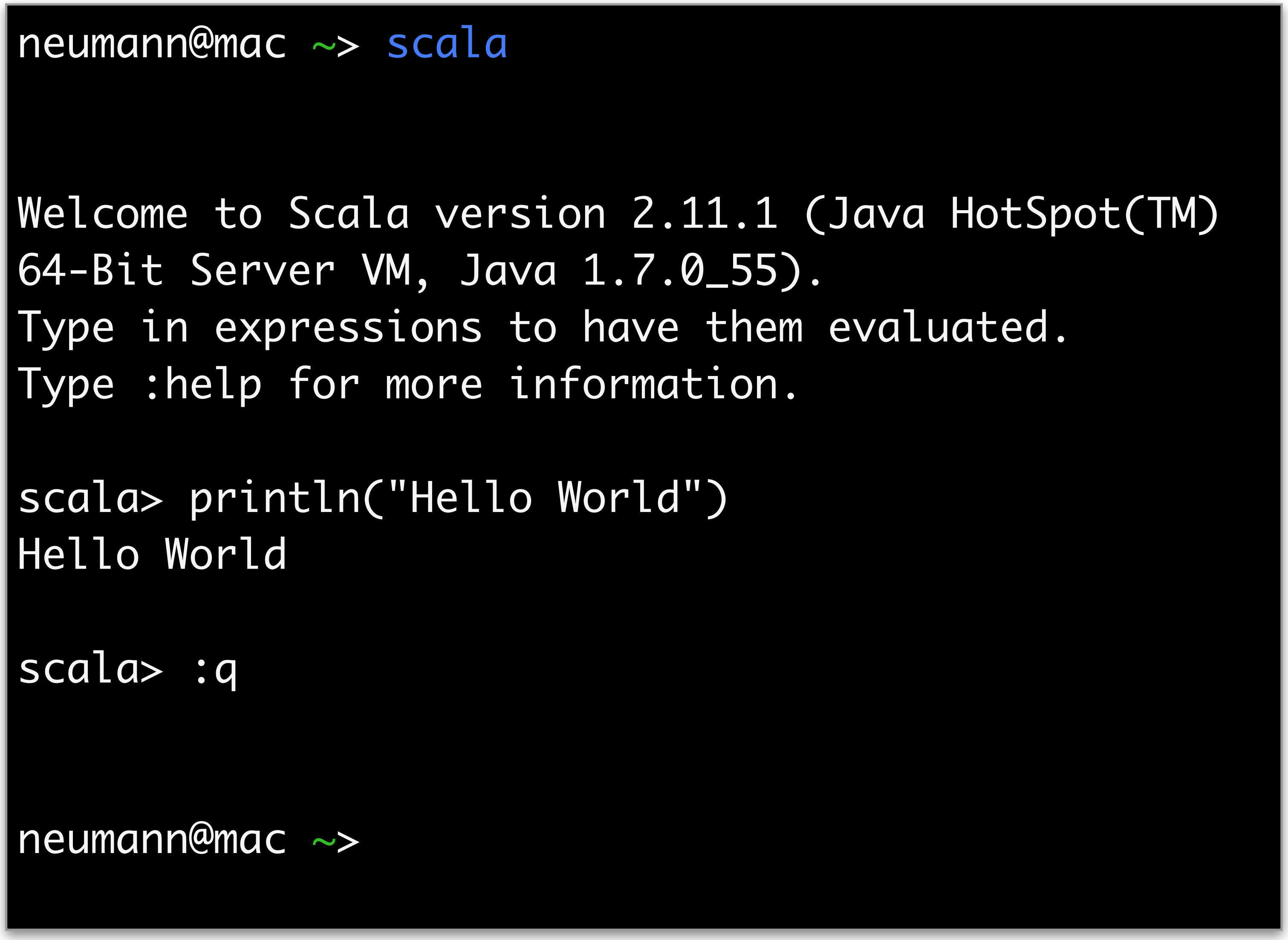
```
::#!
@echo off
call scala %0 %*
goto :eof
::!#

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

HelloWorld.main(args)
```

Scala - Interpreter / REPL

- part of the scala installation

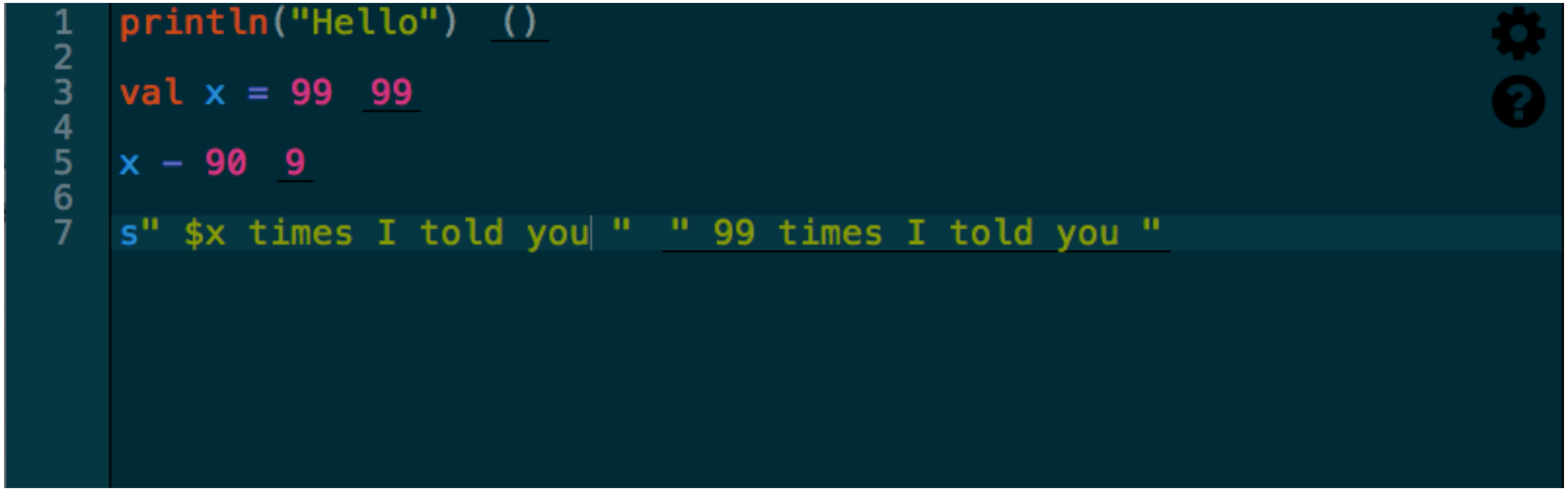
- Start by typing **scala** the shell

- **:q** or **CTRL + d** to quit



```
neumann@mac ~> scala


Welcome to Scala version 2.11.1 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_55).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello World")
Hello World

scala> :q


neumann@mac ~>
```

Online REPL



```
println("Hello")  ()

val x = 99  99

x - 90  9

s" $x times I told you "  " 99 times I told you "
```

- https://codebrew.io/

Syntax

- No need for „;“ , one expression per line
- Still possible to add ; if for example you want several expression within one line
- The dot on method invocations can be dropped. It is best practice to do so with infix invocation but not with postfix operations.

```
"Hallo Welt !".split(" ")
//res0: Array[String] = Array(Hallo, Welt, !)

scala> "Hallo Welt !" split " "
//res2: Array[String] = Array(Hallo, Welt, !)
```

```
scala> List(1,2.3).map(_ * 3).head
//res3: Double = 3.0

scala> ( List(1,2.3) map (_ * 3) ).head
//res4: Double = 3.0
```

val, vars

- **val** creates a Value which is not changeable (like final modifier in Java)
- **var** creates a Variable , which can be reassigned different values

Example: val and 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
         ^
```

Types and type inferrence

- Types are introduced after **:** which is written behind the var/val

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

- Giving the type explicitly is optional as the type inference can infer the type. It's considered good style to add the type information nonetheless.

```
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)
```

define methods

- methods are introduced with **def**
- optionally the can be a list of parameters enclosed by parentheses
- then the body of the function
- methods returning a value put a between name arguments and the body
- the result of the last expression evaluated within the body is the return value

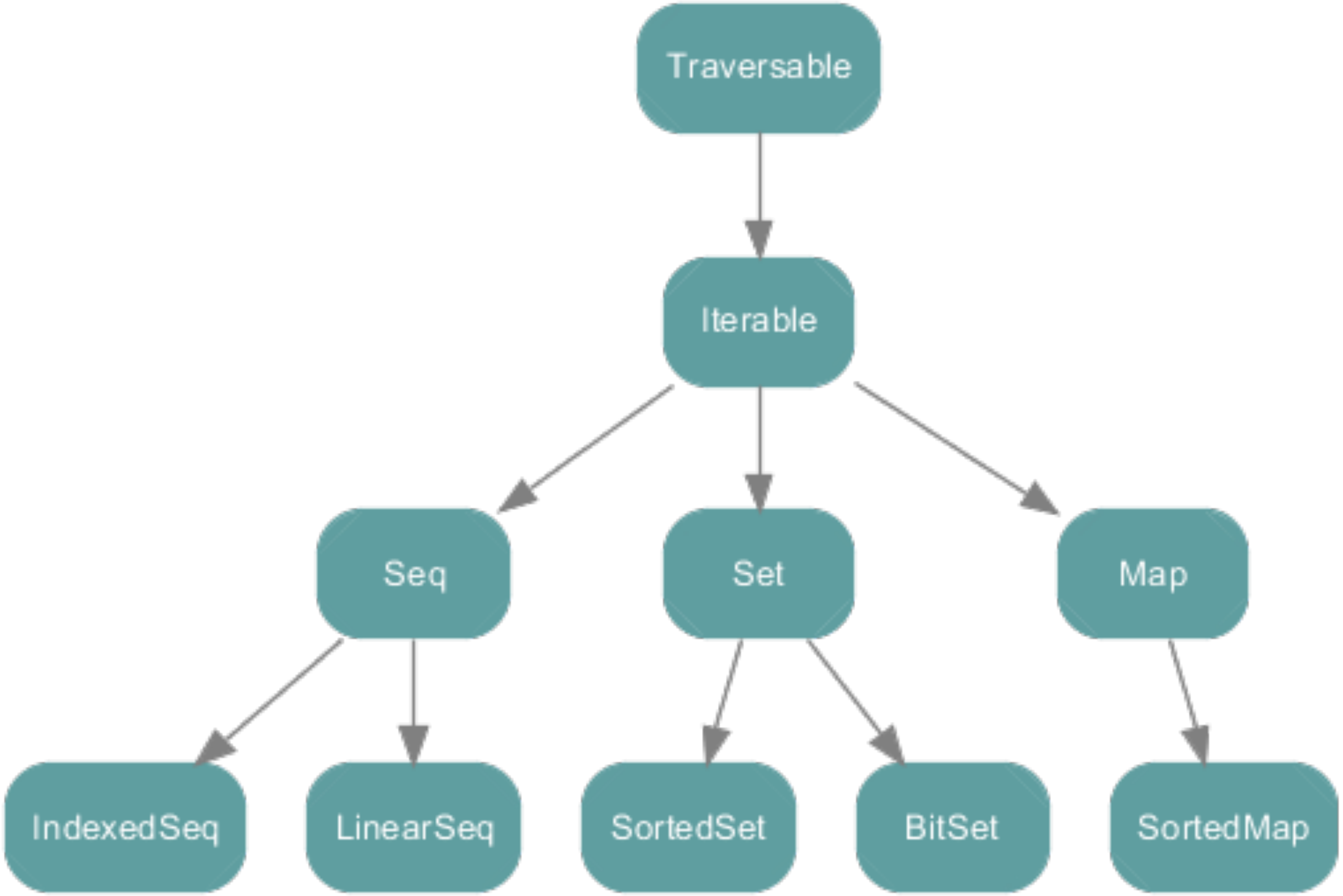
```
def write(aString: String) {
  println(aString)
}

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
```

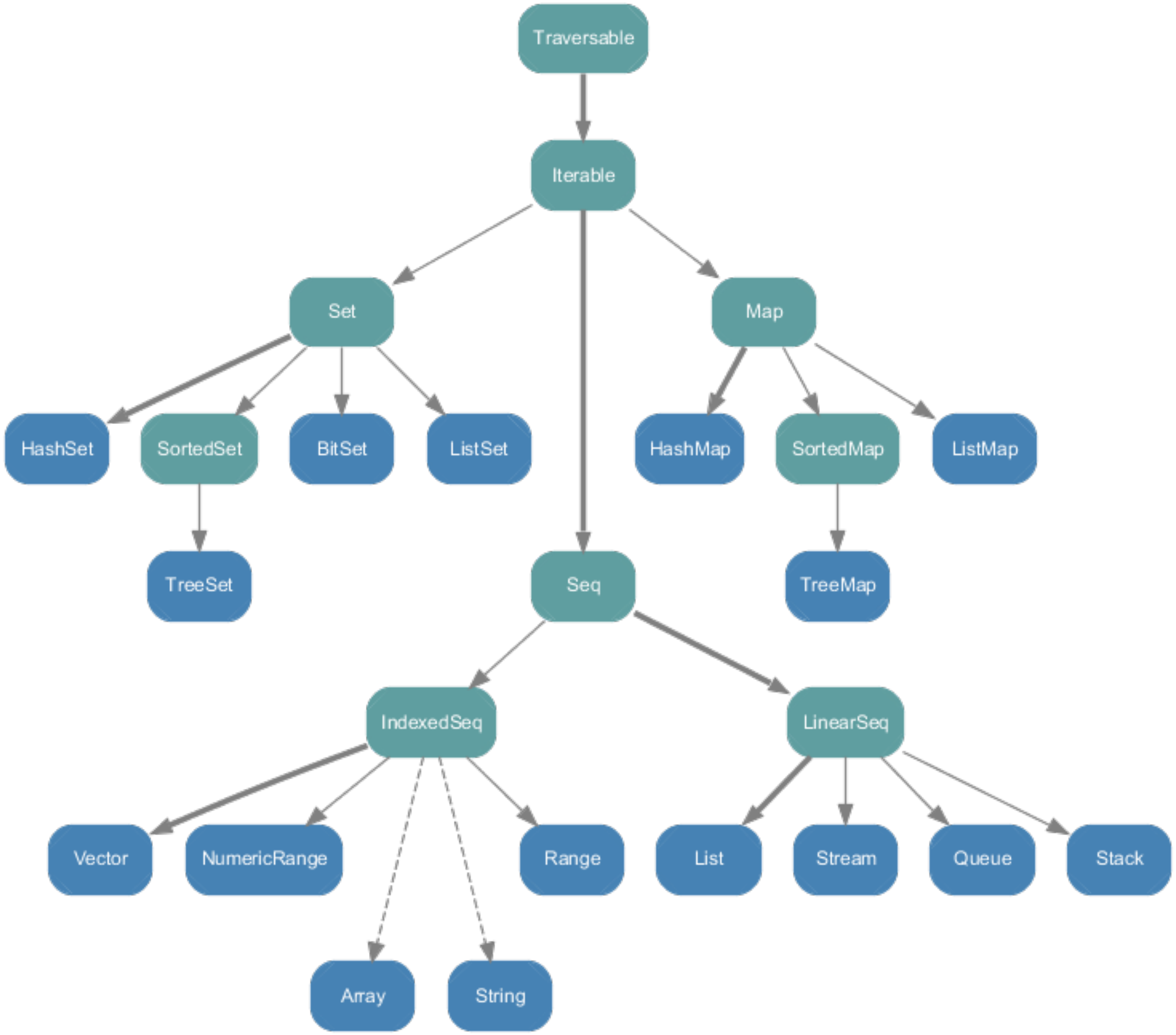
Collections

- Scala has a big Collections library
- Collections provide similar interfaces as far as possible
- Most collections come in up to four flavors:
  - basic (mostly = immutable)
  - immutable
  - mutable
  - parallel

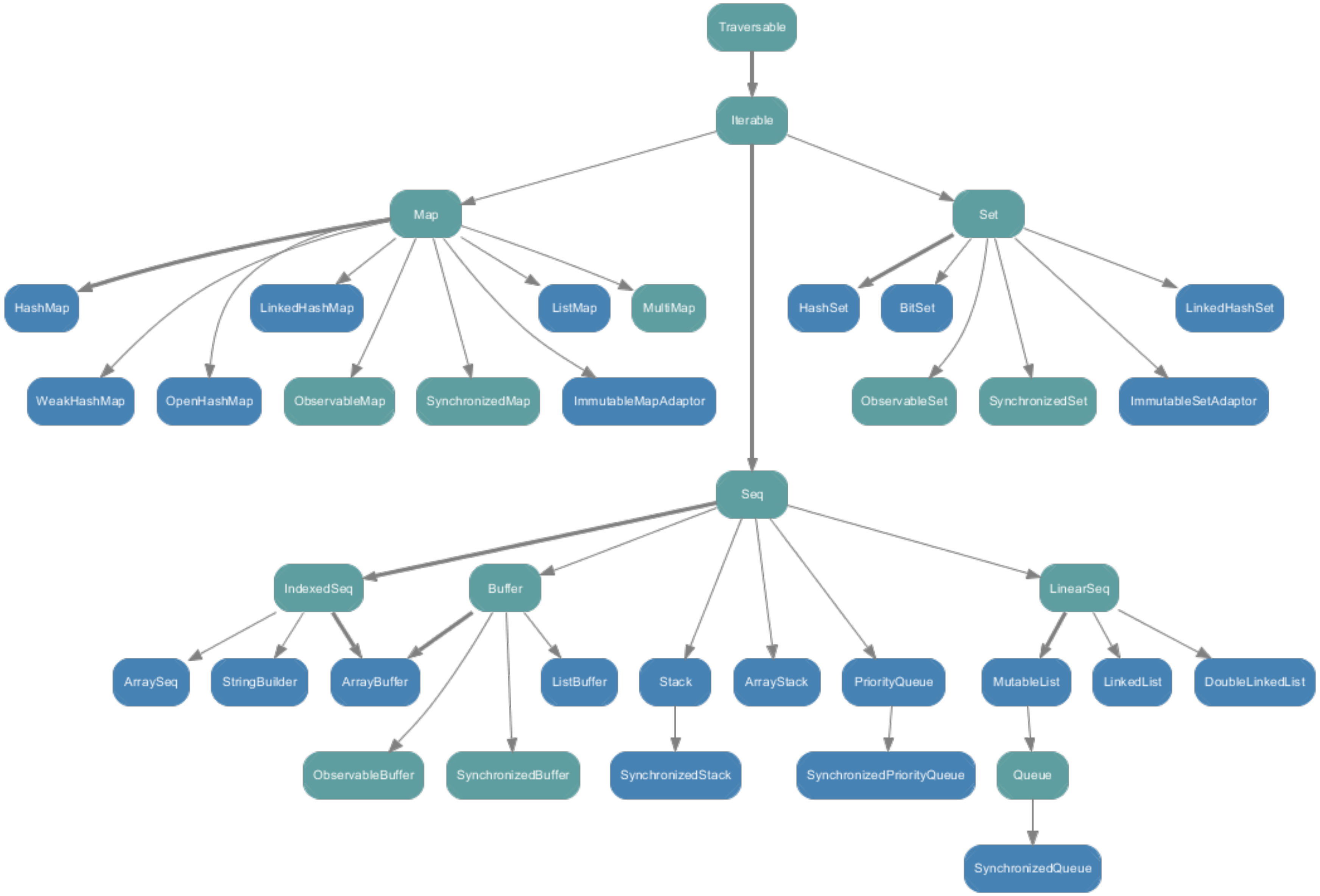
Scala basic collection tree



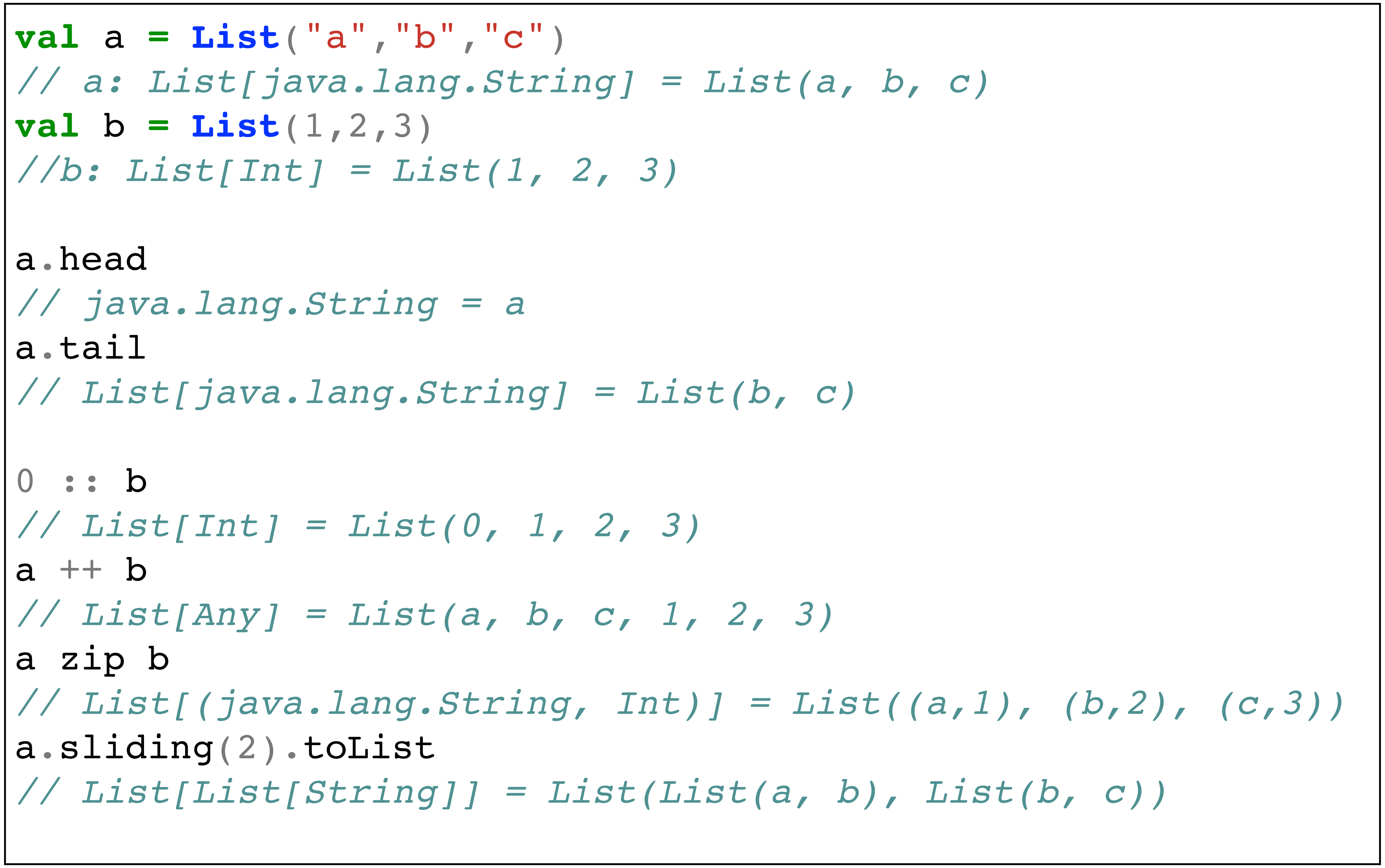
Scala collections - immutable



Scala collections - mutable



Scala Collections Example : List (1 / 2)



```scala
val a = List("a","b","c")
// a: List[java.lang.String] = List(a, b, c)
val b = List(1,2,3)
//b: List[Int] = List(1, 2, 3)

a.head
// java.lang.String = a
a.tail
// List[java.lang.String] = List(b, c)

0 :: b
// List[Int] = List(0, 1, 2, 3)
a ++ b
// List[Any] = List(a, b, c, 1, 2, 3)
a zip b
// List[(java.lang.String, Int)] = List((a,1), (b,2), (c,3))
a.sliding(2).toList
// List[List[String]] = List(List(a, b), List(b, c))
```

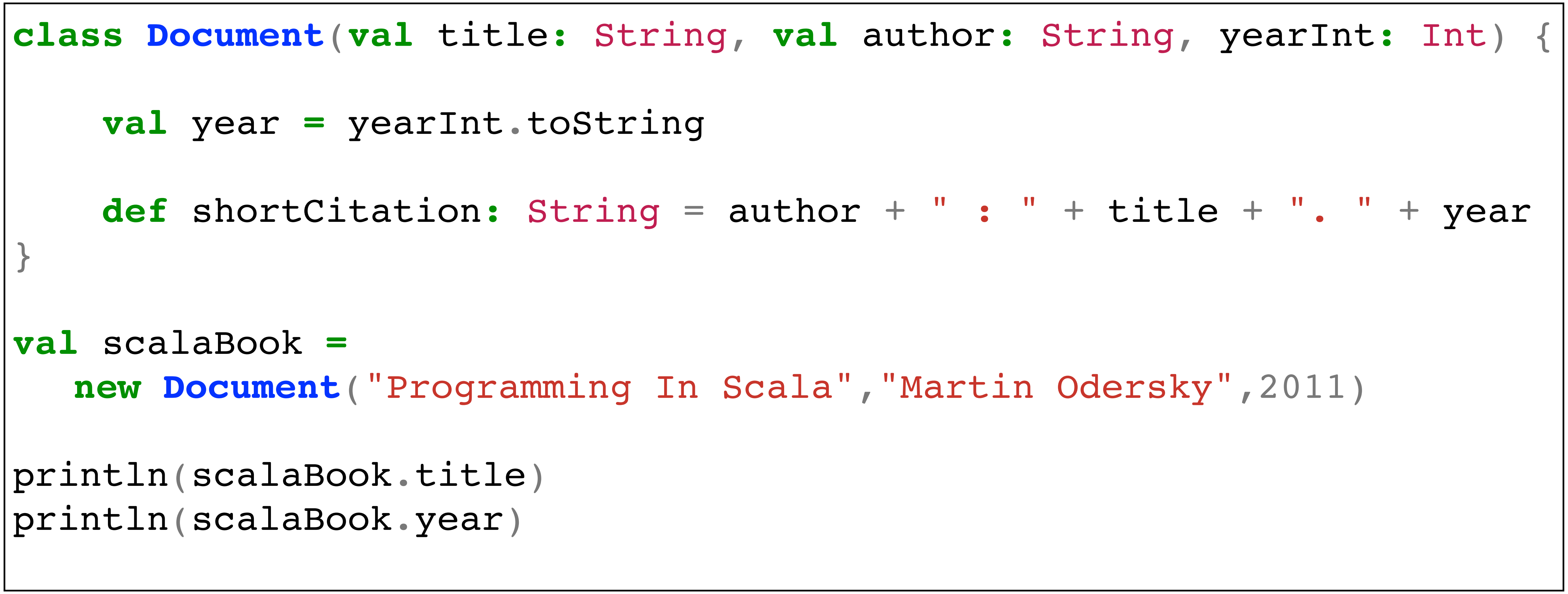
Scala Collections : Map

```
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)
//java.lang.String = zwei

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

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

Classes

- Classes are introduced by the keyword **class**
- Optionally each class has constructor elements in parentheses
- optionally there is a class body
- Things to look out for
  - Constructor elements prepended with the keyword **val** automatically get a **getter** method with the same name as the val (uniform access principle)
  - Constructor elements prepended with the keyword **var** get a **getter** method and a **setter** method with the same name as the var (uniform access principle)
  - Every expression within the body gets evaluated and called on object creation time

Example: A scala class



```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)
```

- Instances are created with **new <ClassName>**

Scala Objects

- Objects are created using the keyword **object**
- They have NO constructor
- Works roughly like a java class with static methods
- Calling members **ObjectName.member** or **ObjectName.method**
- **Singleton-Object**

Scala Object - Example

```scala
object DeepThought {

   val theMeaningOfLife =
    "The meaning of life: 42"

   def speak {
    println(theMeaningOfLife)
   }

}


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

Companion Object

- Widely used pattern in Scala
- A **object** with the same name as a **class**
- Native **Constructor Pattern**

Case Class

- Introduce using the keywords **case class**
- Like a „normal class“
- Defines a **companion object** automatically with **apply**, **unapply** and some other methods
- All constructor elements have **getter** methods as if prepended by val
- Adds **hashCode** and **equals** based on the constructor Elements
- Other classes must not inherit from case classes

Case Class - Example

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

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

println( book.title )
//Necronomicon

book == Book("Necronomicon",1000)
//Boolean = true

scala> book.hashCode
//-364191203

scala> Book("Necronomicon",1000).hashCode
//-364191203
```

Trait

- introduced with keyword **trait**
- roughly comparable to a java interface
- allows an effect resembling multiple inheritance without the dangers (i.e. diamond of death)
- small building blocks
- like ruby mixins

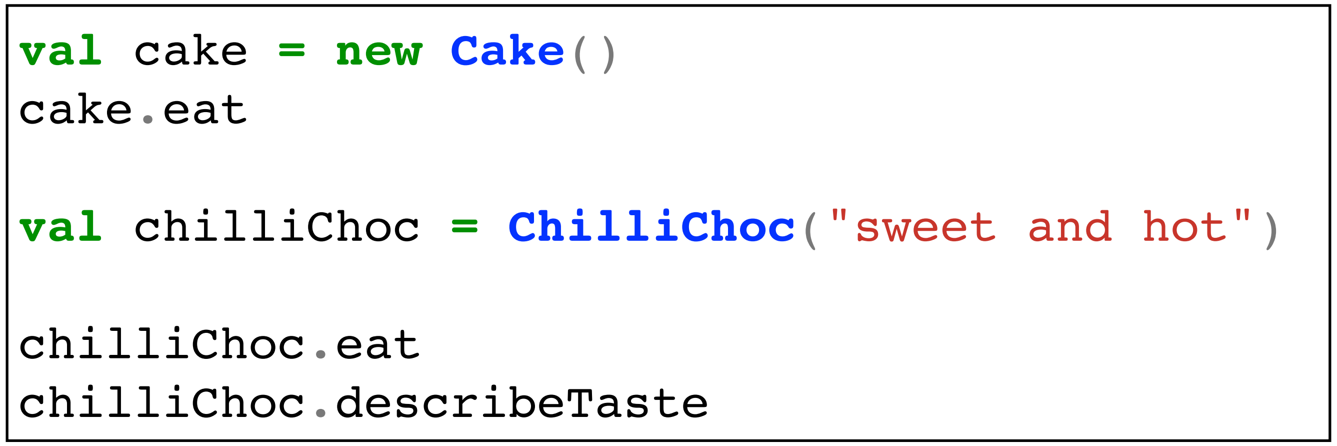
Trait - Example

- Two „traits“ are defined as **Traits**: *Edible* and *ExoticTaste*

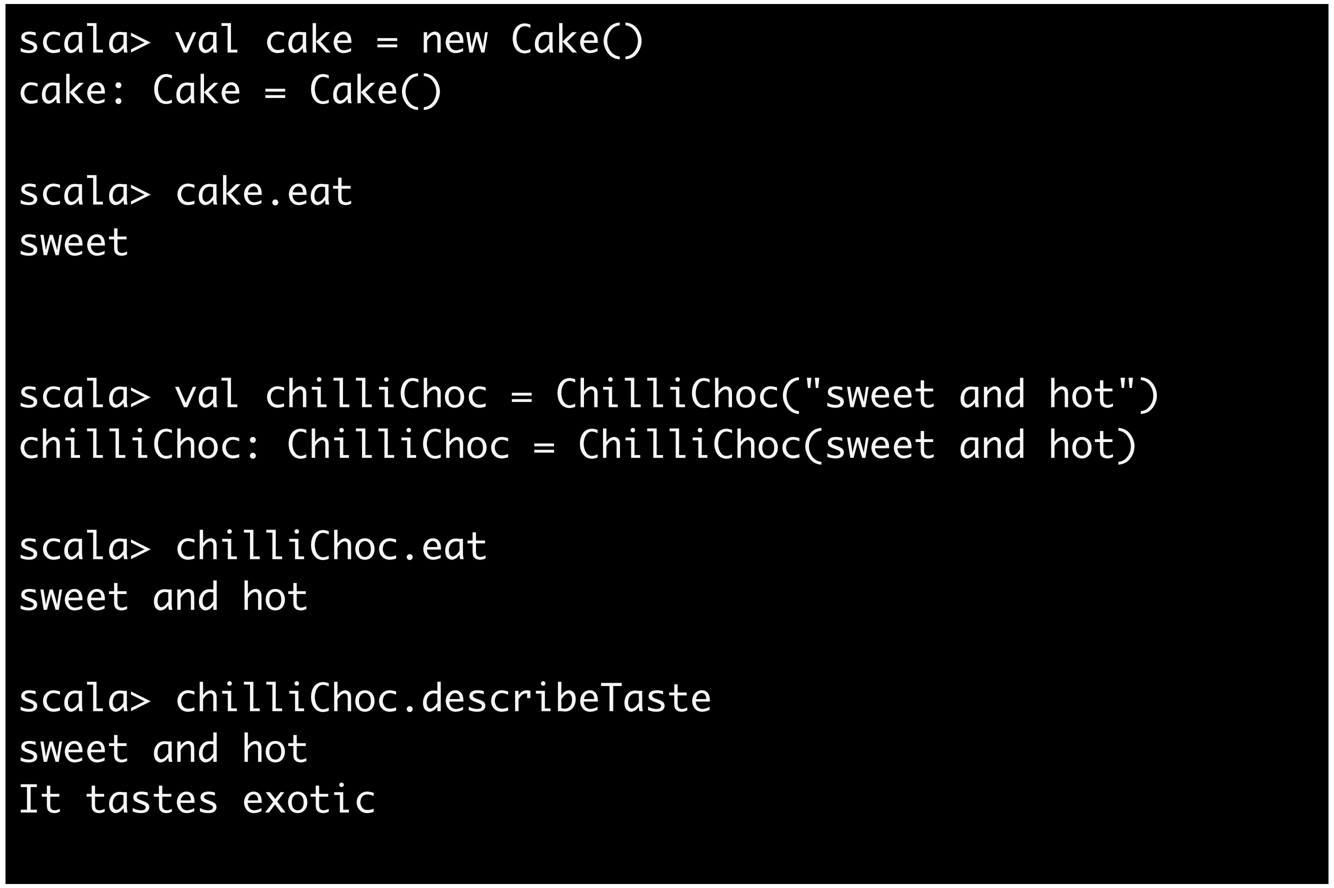
- Two classes are defined: **Cake**, which implements edible and **ChiliChoc** implements *Edible* and *ExoticTaste*

```
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 ChilliChoc(taste: String)
                        extends Edible with ExoticTaste
```

Trait - Example : Usage



```scala
val cake = new Cake()
cake.eat

val chilliChoc = ChilliChoc("sweet and hot")

chilliChoc.eat
chilliChoc.describeTaste
```



```
scala> val cake = new Cake()
cake: Cake = Cake()

scala> cake.eat
sweet


scala> val chilliChoc = ChilliChoc("sweet and hot")
chilliChoc: ChilliChoc = ChilliChoc(sweet and hot)

scala> chilliChoc.eat
sweet and hot

scala> chilliChoc.describeTaste
sweet and hot
It tastes exotic
```

control structures

- if, else
- while
- foreach, map
- for-comprehensions

Control structures

- Control structures like **while**, **if** and **else** work as in Java or C
- In contrast to Java, control structures are functions, i.e. they return a value
- while always returns returns Unit

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

functional control structures

- some Examples

- **map :** Apply function to each given element , keep result (like looping and collecting the results)

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

- **foreach:** Apply function to each given element , drop result

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

//And a 1
//And a 2
//And a 3
```

matching

- Pattern matching
- keyword **match**
- A group of *cases* introduced with the keyword **case**
- Switch on Steroids
- **Pattern Guards** allow better control flow
- **Case Classes** and **Extractor Patterns** for easy deconstruction and extraction of input

Matching Example

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

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

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

```
books: List[Any] = List(Book(Programming Scala,883,2012), Book(Programming Pearl,1104,2000), Book(Necronomicon,666,666), Ein String, 5,
42)
bookComments: List[String] = List(New Scala Book by Martin Odersky from 2012, Programming Pearl 1104 2000, Necronomicon 666 666, Something
else, Something else, an integer bigger than 10)
```

For-Comprehensions

- program with a DSL that looks a lot like pseudocode

- will be translated to map, filter, flatMap and reduce operations by the compiler

```
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)
```

```
                ~ Translated ~
(1 to 99) filter isEven filter ( _ % 5 == 0)
```

Some language features

- Strings: Interpolation and MultilineStrings
- Option[Type]
- Future[X]
- XML erzeugen und bearbeiten
- Reguläre Ausdrücke
- Parallele Collections
- Implicits

String Interpolation

- String concatenation works with „**+**“

```
"hello" + " " + "world"
//res2: String = hello world
```

- String interpolation: prepending String with „**s**“ marking Variable with **$**

```
val w = "world"
s"hello $w"
//res3: String = hello world
```

- Complex Expressions are enclosed within **${ }**

```
val w = "world"
s"$w has length:${w.length}"
//res4: String = world has length:5
```

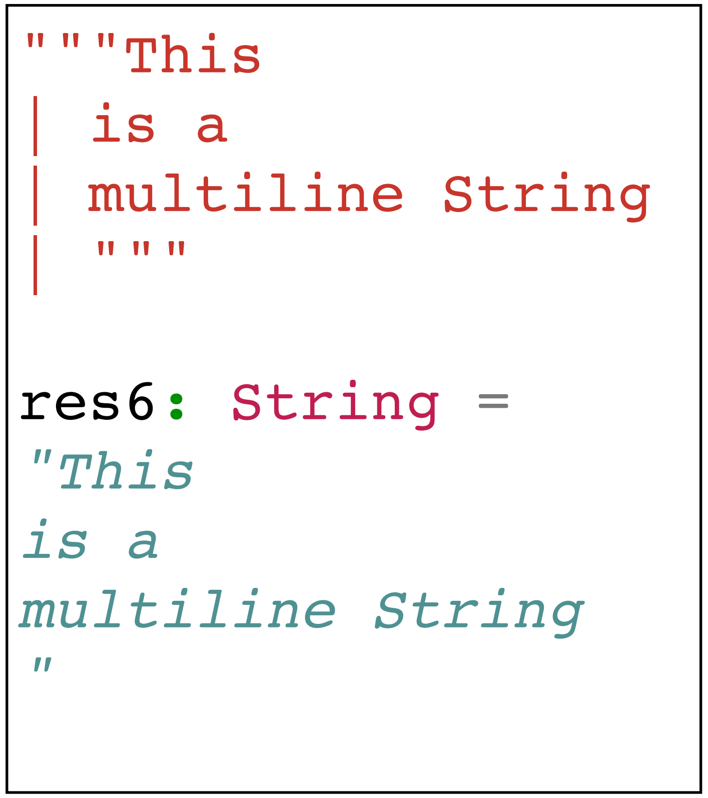
String Concatenation/Interpolation - Example

```
val names = List("Roger","Felix", "Bene")

for (name <- names) println("Hello" + name)

//HelloRoger
//HelloFelix
//HelloBene
```

```
val names = List("Roger","Felix", "Bene")

for (name <- names) println(s"Hello $name")


//Hello Roger
//Hello Felix
//Hello Bene
```

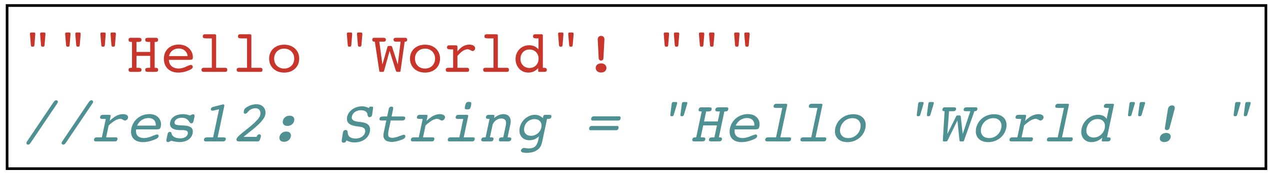
Multiline String

- Multiline String is created by using """""" instead of ""



```
"""This
| is a
| multiline String
| """

res6: String =
"This
is a
multiline String
"
```

- allows embedding " in Strings and gets rid of double Escapes



```
"""Hello "World"! """
//res12: String = "Hello "World"! "
```

Combined example Strings

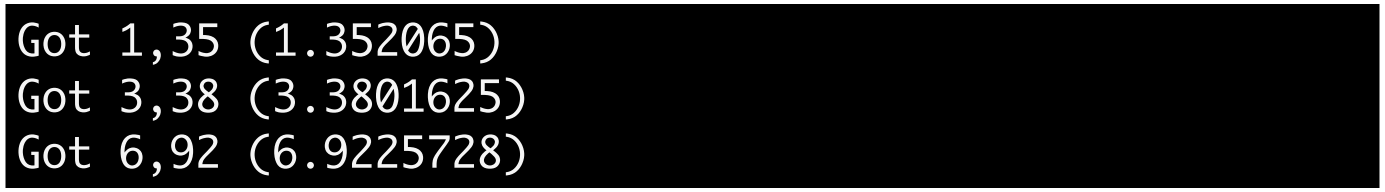
```
val names = List("Roger","Felix", "Bene")

for (name <- names) println(
s"""Hello $name your name
    has length:${ name.size }
    and reads backwards as:"${ name.reverse}"
"""
)
```

```
Hello Roger your name
has length:5
and reads backwards as:"regoR"

Hello Felix your name
has length:5
and reads backwards as:"xileF"

Hello Bene your name
has length:4
and reads backwards as:"eneB"
```

String format - The f interpolator

- prepend String with **f “““**
- Syntax like C printf

```
def euroToDollar(euro: Double): Double =
      euro * 1.352065

val euro = List(1,2.5,5.12)

euro map euroToDollar foreach { d =>
  println(f"Got $d%2.2f ($d)")
}
```

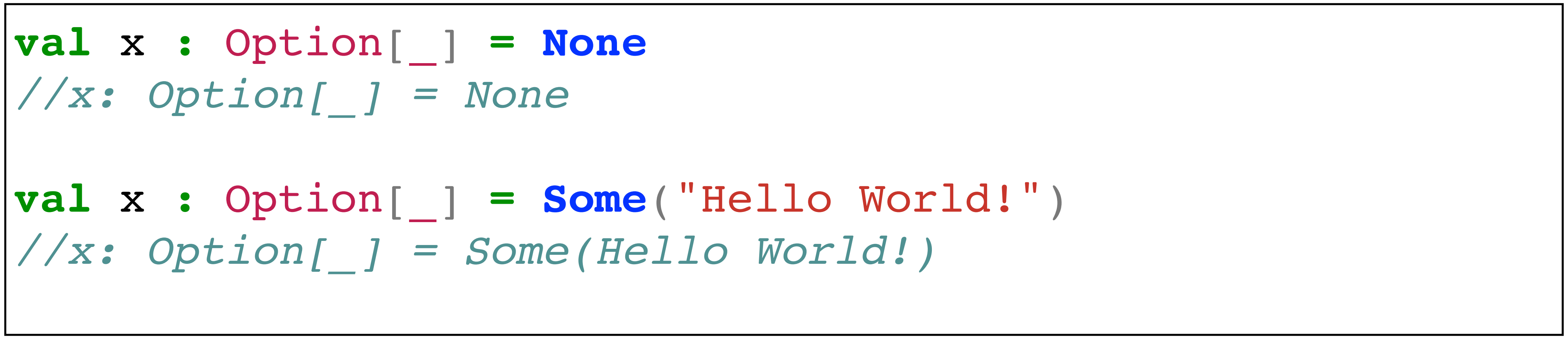


```
Got 1,35 (1.352065)
Got 3,38 (3.3801625)
Got 6,92 (6.9225728)
```

The Option-Type

- Marks functions that may return a result but also may not return a result

- Comes in two flavors: **Some** and **None**



```
val x : Option[_] = None
//x: Option[_] = None

val x : Option[_] = Some("Hello World!")
//x: Option[_] = Some(Hello World!)
```

- like maybe in Haskell

- A way to get around checking for null all the time

Option / List Comparison

- Option behaves like a list with one element

```
List( 1 ) map (i => i + 0.5 )
//List[Double] = List(1.5)
```

```
Some( 1 ) map (i => i + 0.5 )
//Option[Double] = Some(1.5)
```

- An empty Option is called None. None behaves like an empty list.

```
val y : List[Int] = List()
//y: List[Int] = List()

y map (i => i+ 0.5)
//List[Double] = List()
```

```
// Like an empty List
val x : Option[Int] = None
//x: Option[Int] = None
x map (i => i+ 0.5)
// Option[Double] = None
```

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 :Examples 1

- Used widely throughout Scala

- many builtin methods to handle Option

```
// Liste mit Options erzeugen
friends map (bigBangPHD.get(_))
friends map bigBangPHD.get
//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
friends flatMap (f => bigBangPHD.get(f))
//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 s"$person has a $phd"
//List[java.lang.String] =
//List(Leonard has a Ph.D., Sheldon has a Ph.D.,Sc.D, Rajesh has a Ph.D)
```

Option -Type : Examples 2,

```
// getOrElse erlaubt es einen Standardrückgabewert für None anzugeben, ansonsten wird
Some(thing) „ausgepackt“
friends
    .map( n =>(n,bigBangPHD.get(n)) ) // creates Tuple
    .map{ case (n,d) =>
              n + " " + d.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)
```

Futures

- Are a way to abstract over asynchronous computation
- non blocking
- can be used much like Option
- used in many popular Scala libraries

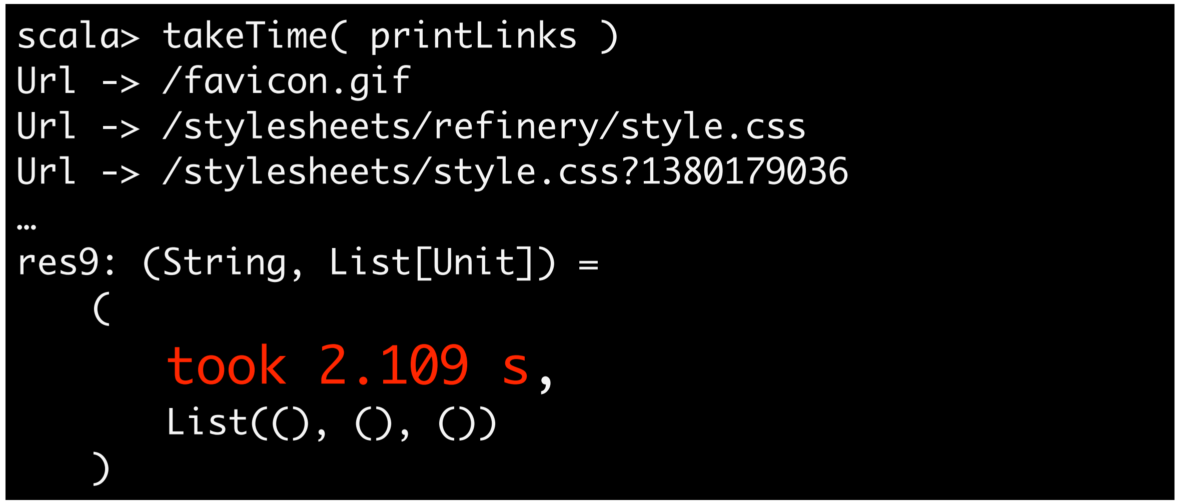
Futures - Plumbing

- Import com.scala.concurrent._ for future helpers
- Every future needs an **ExcecutionContext**

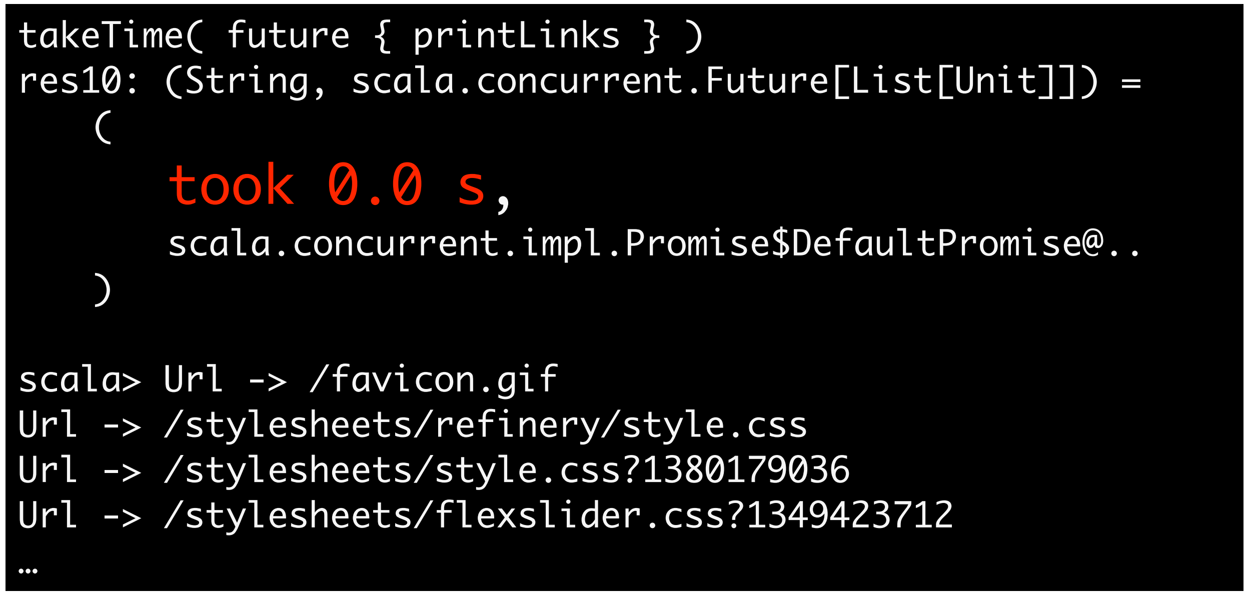
```
import scala.concurrent._
import ExecutionContext.Implicits.global
```

Using Futures - Example

```
val urls =
List("http://www.clueda.de","http://www.neumann.biz","http://www.an-it.com")

def takeTime = { code to measure time }
def extractURLs(data: String) : Iterator[String] = …
def printURLs(data: String) : Unit = extractURLs(data) foreach println
def printLinks = urls map ( url => printURLs( getData(url) ) )

takeTime( printLinks )
takeTime( future { printLinks } )
```



```
scala> takeTime( printLinks )
Url -> /favicon.gif
Url -> /stylesheets/refinery/style.css
Url -> /stylesheets/style.css?1380179036
…
res9: (String, List[Unit]) =
   (
      took 2.109 s,
      List((), (), ())
   )
```



```
takeTime( future { printLinks } )
res10: (String, scala.concurrent.Future[List[Unit]]) =
   (
      took 0.0 s,
      scala.concurrent.impl.Promise$DefaultPromise@..
   )

scala> Url -> /favicon.gif
Url -> /stylesheets/refinery/style.css
Url -> /stylesheets/style.css?1380179036
Url -> /stylesheets/flexslider.css?1349423712
…
```

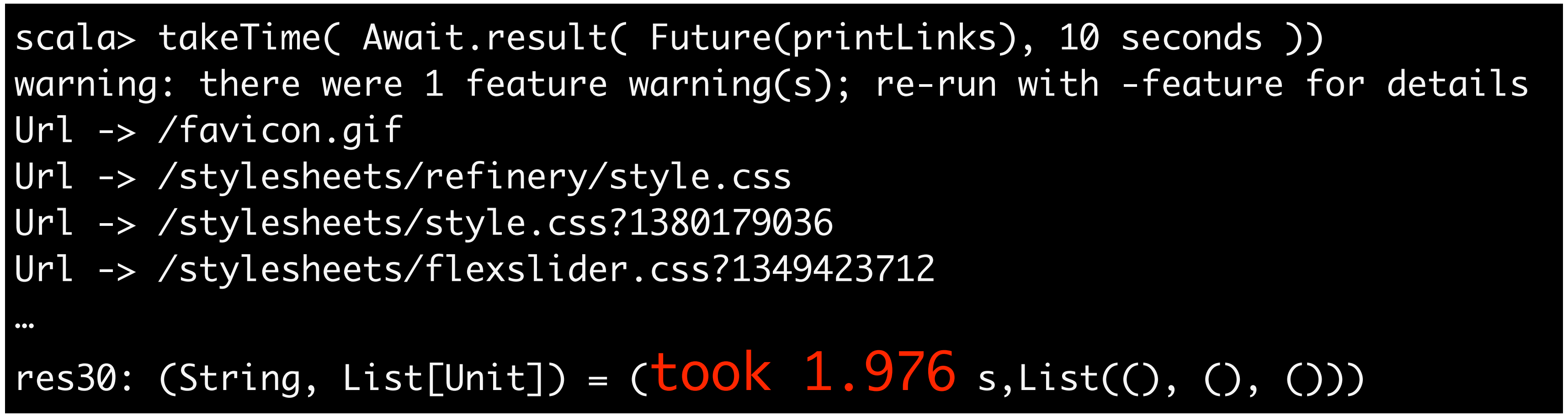
Futures - Getting the result

- To get the result of a Future you have to block and wait

```
import scala.concurrent.duration._

takeTime( Await.result( Future(printLinks), 10 seconds ))
```

- This is usually bad

- Awaiting the result should happen as late as possible as it negates the benefits one gets using futures

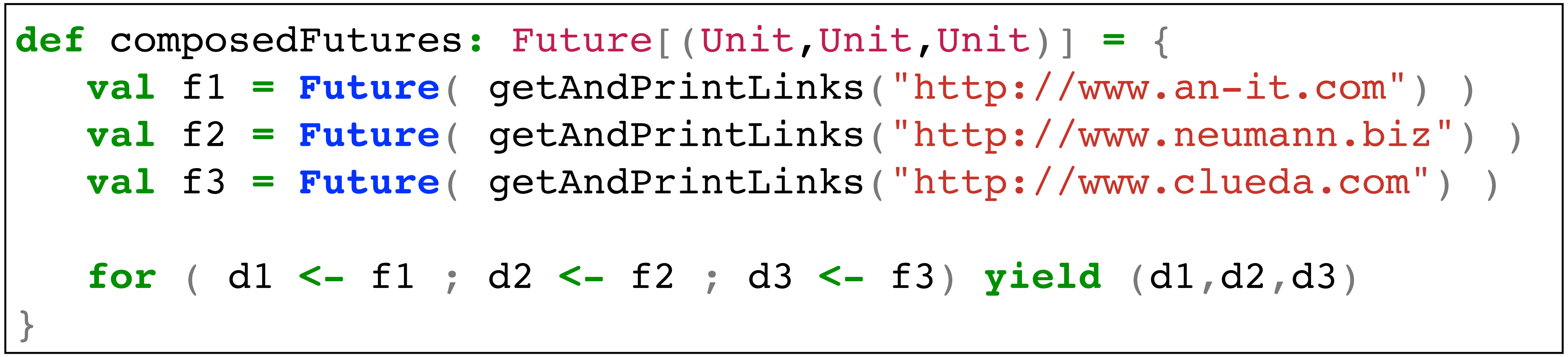


```
scala> takeTime( Await.result( Future(printLinks), 10 seconds ))
warning: there were 1 feature warning(s); re-run with -feature for details
Url -> /favicon.gif
Url -> /stylesheets/refinery/style.css
Url -> /stylesheets/style.css?1380179036
Url -> /stylesheets/flexslider.css?1349423712
…
res30: (String, List[Unit]) = (took 1.976 s,List((), (), ()))
```

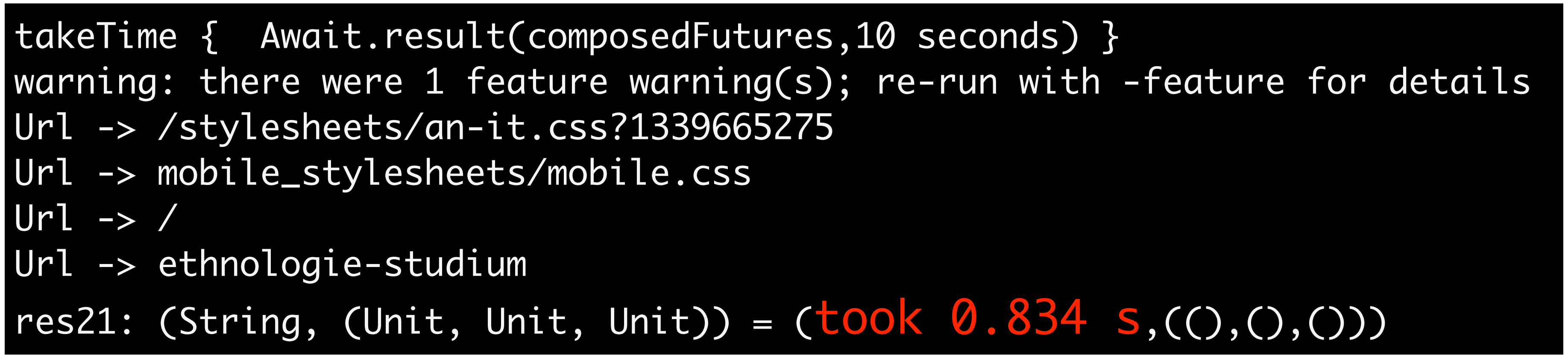
Futures - Composing

- As futures are Monads (said it, done!) they can be composed

- The futures run asynchronously and will not wait for each other but await will wait till the last of the futures has completed or the timeout is reached.



```
def composedFutures: Future[(Unit,Unit,Unit)] = {
   val f1 = Future( getAndPrintLinks("http://www.an-it.com") )
   val f2 = Future( getAndPrintLinks("http://www.neumann.biz") )
   val f3 = Future( getAndPrintLinks("http://www.clueda.com") )

   for ( d1 <- f1 ; d2 <- f2 ; d3 <- f3) yield (d1,d2,d3)
}
```



```
takeTime {  Await.result(composedFutures,10 seconds) }
warning: there were 1 feature warning(s); re-run with -feature for details
Url -> /stylesheets/an-it.css?1339665275
Url -> mobile_stylesheets/mobile.css
Url -> /
Url -> ethnologie-studium
res21: (String, (Unit, Unit, Unit)) = (took 0.834 s,((),(),()))
```

XML in Scala

- XML is a first class language citizen as string is in Java or Ruby

- It’s possible to embed XML in Scala source code and get syntax highlighting

Scala - XML

- Xml can be written within scala sources
- IDE s provide syntax-highlighting (Eclipse, Netbeans, IntelliJ)
- Code can be embedded using **{ }** within XML literals

Emit XML - Example

```scala
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)
}
```

Emitted XML

```
<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>
```

Processing XML

- Scala provides an internal DSL influence providing a XPath like syntax (\\ instead of // and \ instead of /)

- <xml></xml> **\ "tag"** : Shallow -Match

- <xml></xml> **\\ "tag**" : Deep -Match

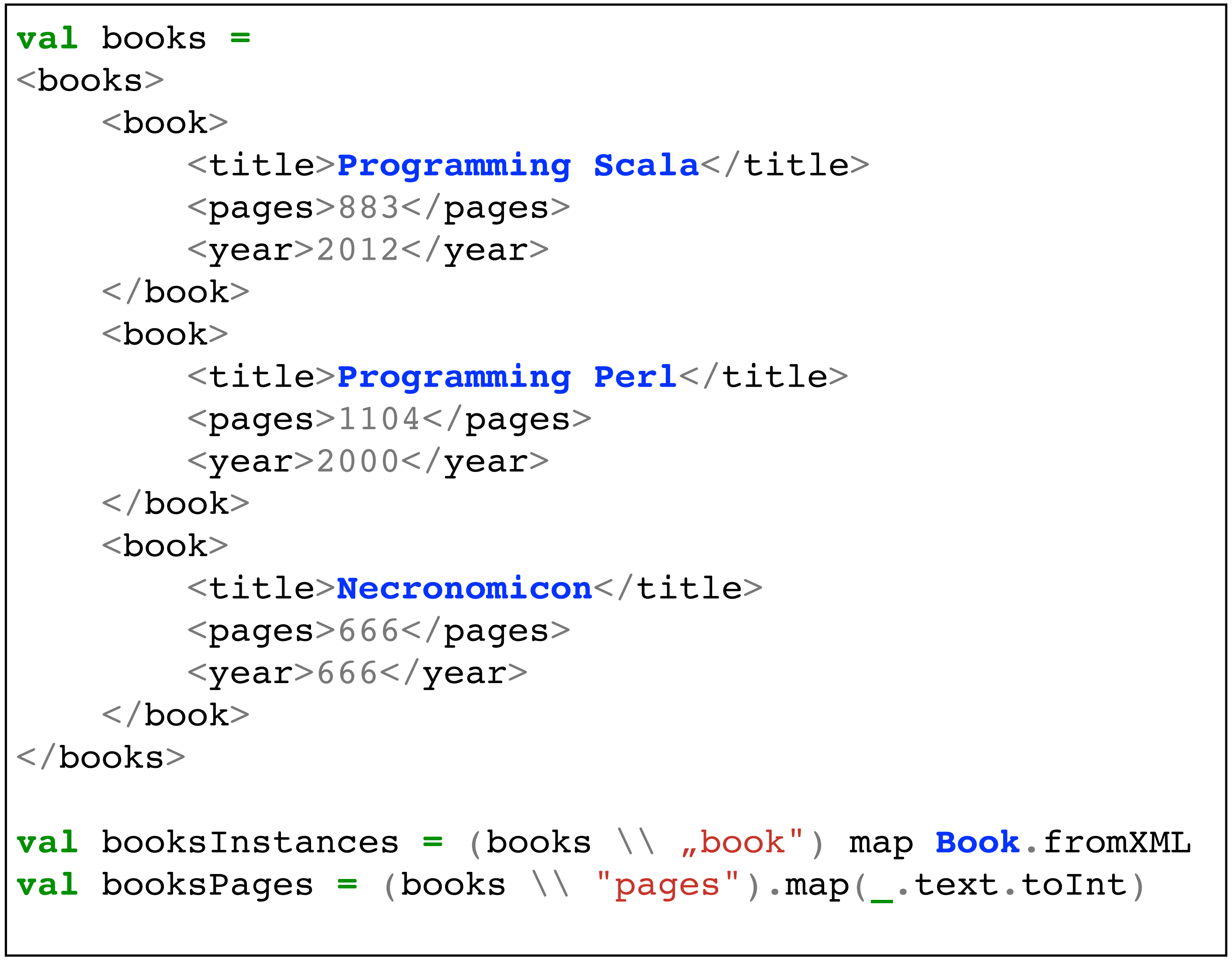
- <xml attribute=„wert“></xml> **\\ "@attribut"** : Deep -Match on a XML attribute

- (<xml></xml> \ "tag")**.text** : Extracts the text value of an xml node

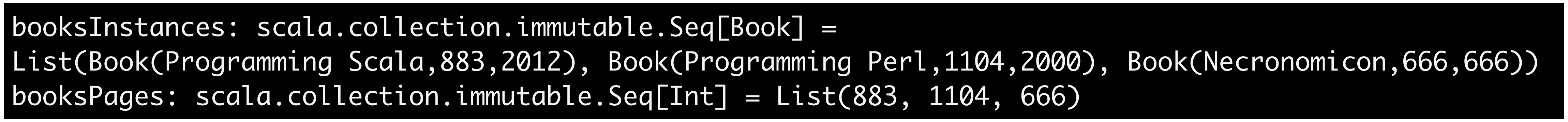
processing XML - Example

```
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)
    }
}
```

processing XML - Result



```
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)
```

Regular Expressions

- Creating a regular Expression:.r aus einem String erzeugen:

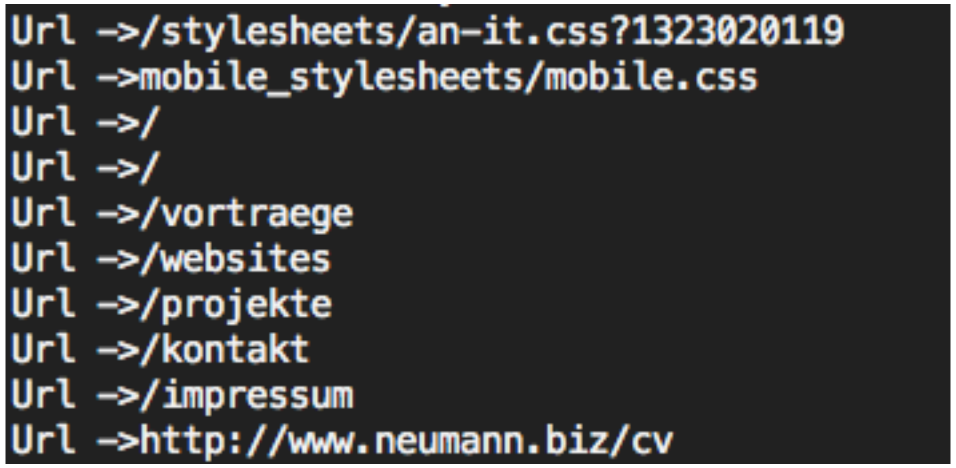
```
// Using the Constructor
 new scala.util.matching.Regex("href\\s?=\\s?\"([^\"]+)\"")
//Changing a string to a regex with the .r method
"href\\s?=\\s?\"([^\"]+)\"".r
// Using """ , no need to escape " and double escaping of \
"""href\s?=\s?"([^"]+)"""".r
```

- Uses Java-Regex-Engine to create a NFA

- Regex-Object also implement extractors for pattern matching

Regex - Usage

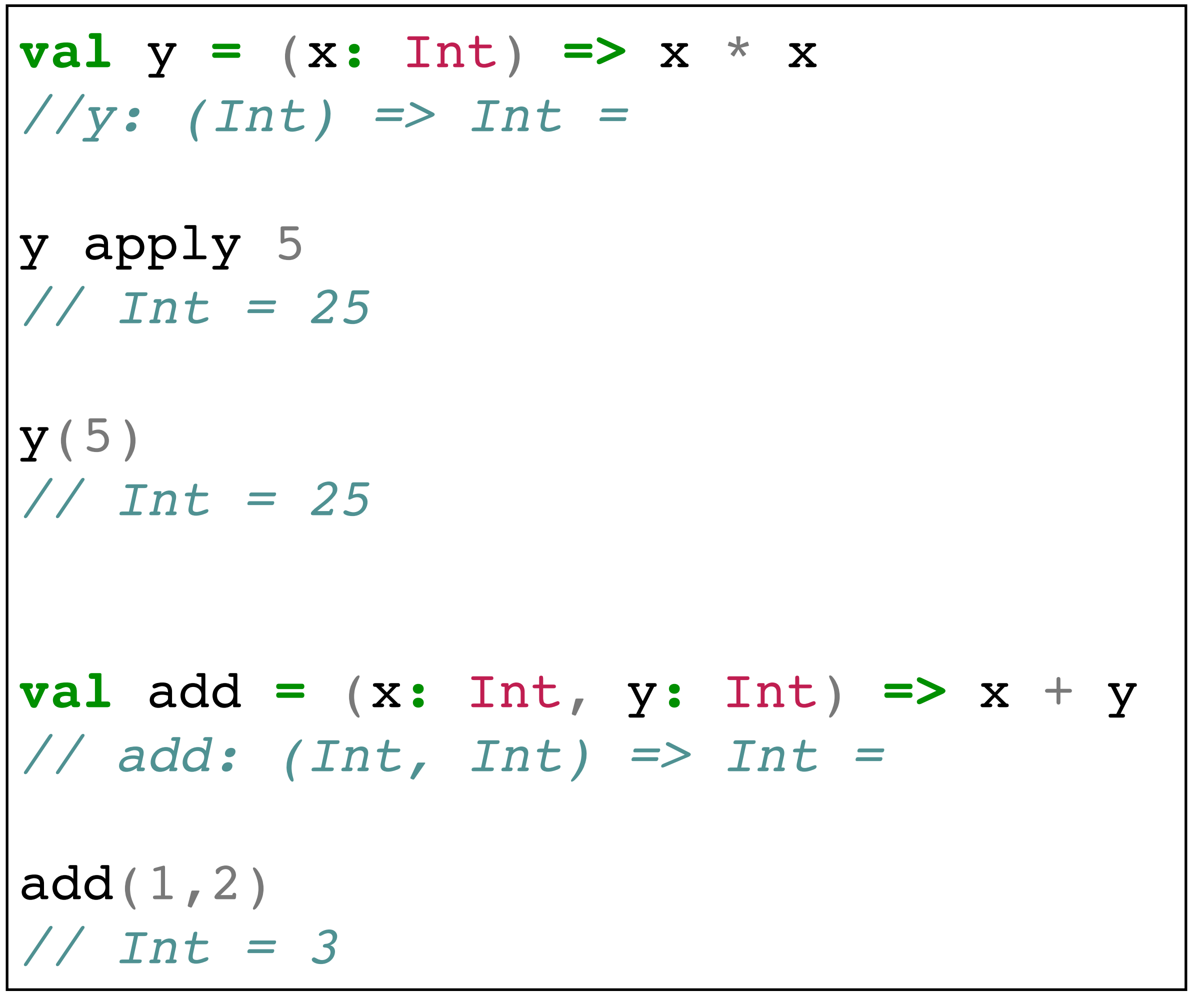
```
import scala.io.Source

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

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

for ( urlExtractor(url) <- urlExtractor findAllIn html ) {
   println(s"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 / anonymous functions

- functions have a type like Integer or String
- They can be arguments to function and passed around as values



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

y apply 5
// Int = 25

y(5)
// Int = 25


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

add(1,2)
// Int = 3
```

Implicits

- are introduced using the keyword **implicit**
- trigger an automatic transformation
- not stackable
- shorter, more readable
- may introduce „magic“
- **Pimp my library Pattern**: Locally scopefied monkey patching

Implicits: Example

- no more need to manually transform year to string when using xml
- will also work for all other integers in scope of Book

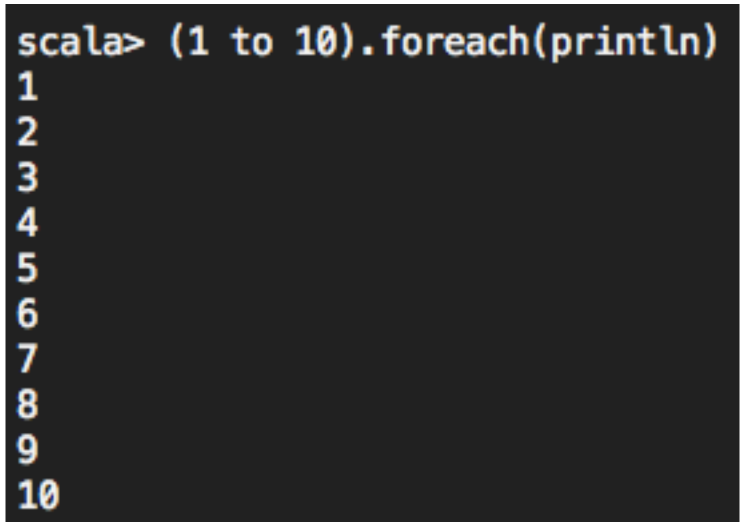
```
case class Book( title: String, pages: Int, year: Int) {
    def toXML =
      <book>
        <title>{title}</title>
        <pages>{pages}</pages>
        <year>{year}</year>
      </book>

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

Parallel Collections

- Asynchronous, parallel processing to take advantage of multicore processors
- **.par** transforms a Collection to it’s parallel counterpart
- **.seq** transforms a parallel Collection to a sequential one
- **Parallel** is implemented as a trait => can be used to create own par collections
- Also works for Map

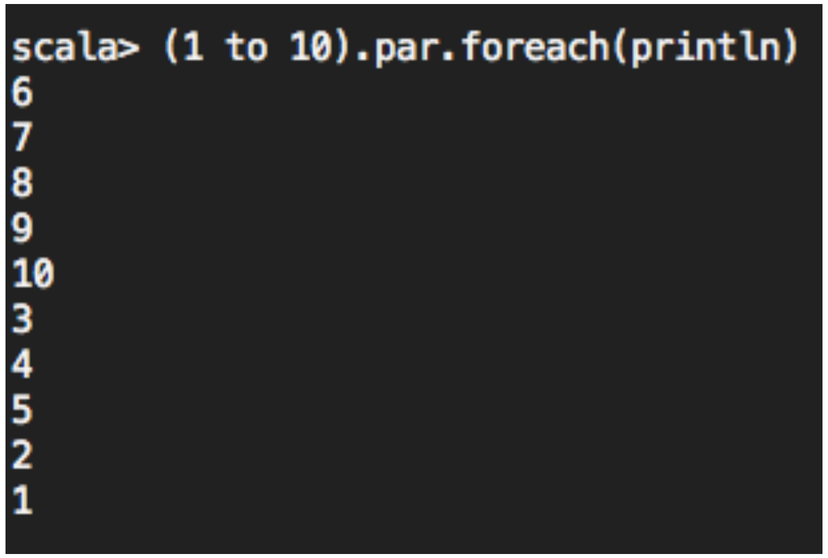
Parallel Collections - Example

```
// Sequential
(1 to 10) foreach println
```



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

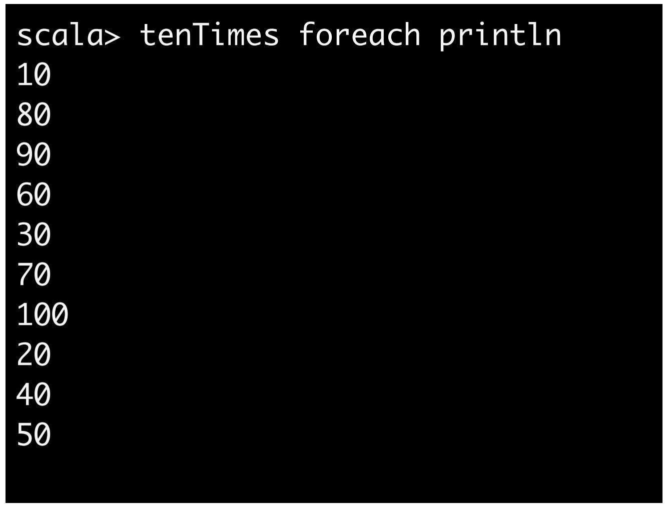
```
// Parallel
(1 to 10).par foreach println
```



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

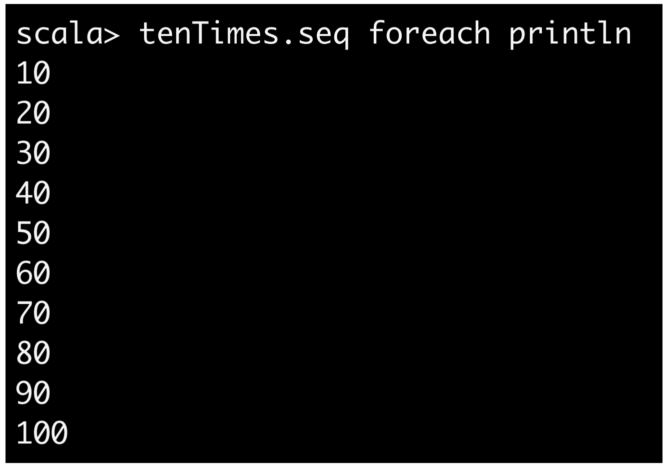
Parallele Collections - Examples II

```
// Unordered
val tenTimes = (1 to 10).par map (_ * 10)
tenTimes foreach println
```



```
scala> tenTimes foreach println
10
80
90
60
30
70
100
20
40
50
```

```
// Unordered
val tenTimes = (1 to 10).par map (_ * 10)
tenTimes foreach println

//Ordered
//.seq  transforms a parallel collection to a sequential one
tenTimes.seq foreach println
```



```
scala> tenTimes.seq foreach println
10
20
30
40
50
60
70
80
90
100
```

Build your own control structures

- Currified functions can be used to build control structures

```scala
object ControlStructures {
  def unless( test: => Boolean)(action: => Any) =
    if (! test) action

  def times( n: Int )(action: => Unit) {
    (1 to n) foreach { _ => action}
  }
}
```

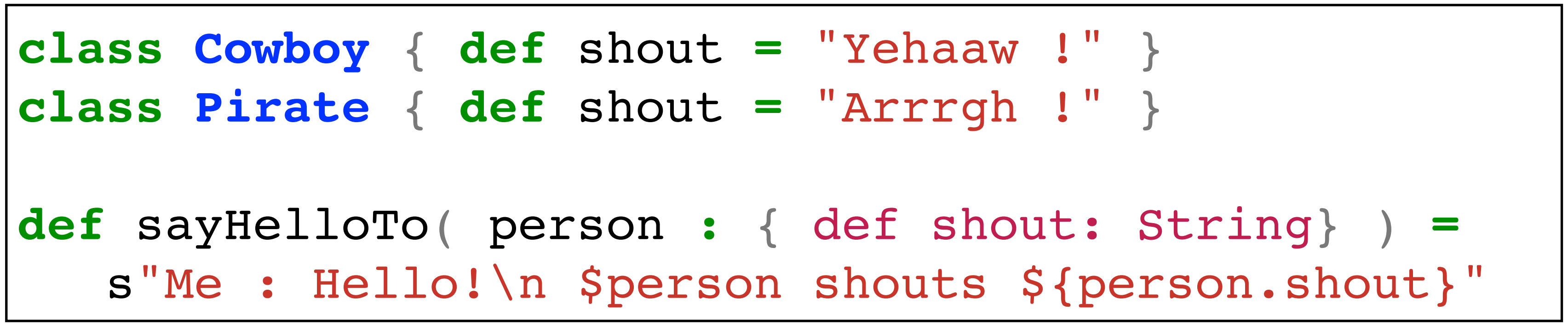
```
scala> import ControlStructures._
//import ControlStructures._

scala> times(2) { println("Hoorray :)")}
Hoorray :)
Hoorray :)

scala> unless (5 < 10) { println("Math stopped working.") }
// Any = ()

scala> val ifNot = unless (2 + 2 != 4) { "Math still works." }
// Any = Math still works.
```

Scala - Patterns

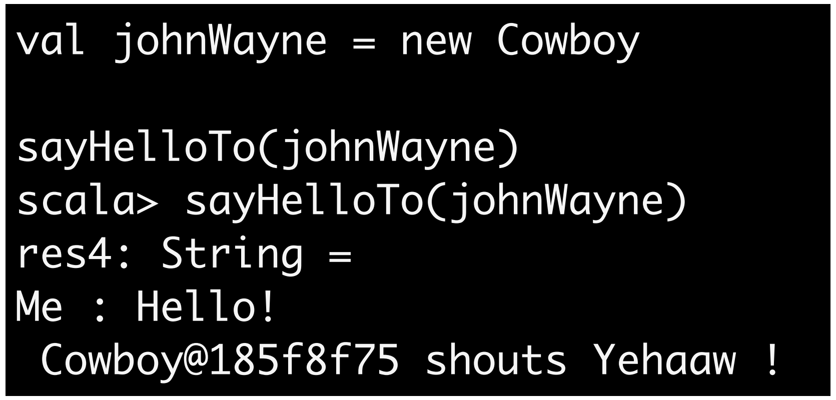
- Structural Typing
- Pimp-My-Library-Pattern

Structural Typing

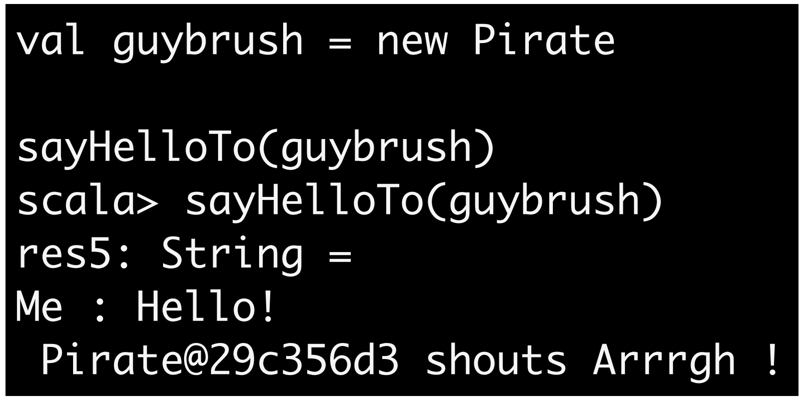
- Classed are described by methods and return types they provide
- Works like **duck typing** but the checking happens in compile time, not run time



```
class Cowboy { def shout = "Yehaaw !" }
class Pirate { def shout = "Arrrgh !" }

def sayHelloTo( person : { def shout: String} ) =
   s"Me : Hello!\n $person shouts ${person.shout}"
```



```
val johnWayne = new Cowboy

sayHelloTo(johnWayne)
scala> sayHelloTo(johnWayne)
res4: String =
Me : Hello!
 Cowboy@185f8f75 shouts Yehaaw !
```



```
val guybrush = new Pirate

sayHelloTo(guybrush)
scala> sayHelloTo(guybrush)
res5: String =
Me : Hello!
 Pirate@29c356d3 shouts Arrrgh !
```

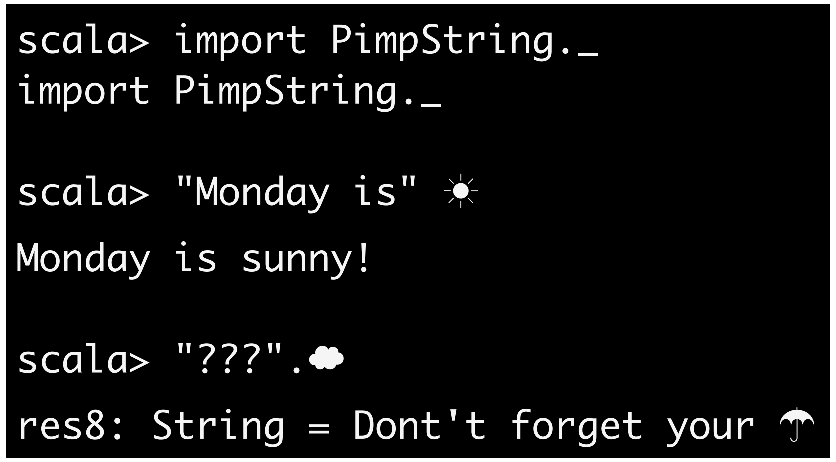
Pimp-My-Library-Pattern

- Add new functions to existing libraries without changing the code
- Like monkey patching
- type safe
- scoped

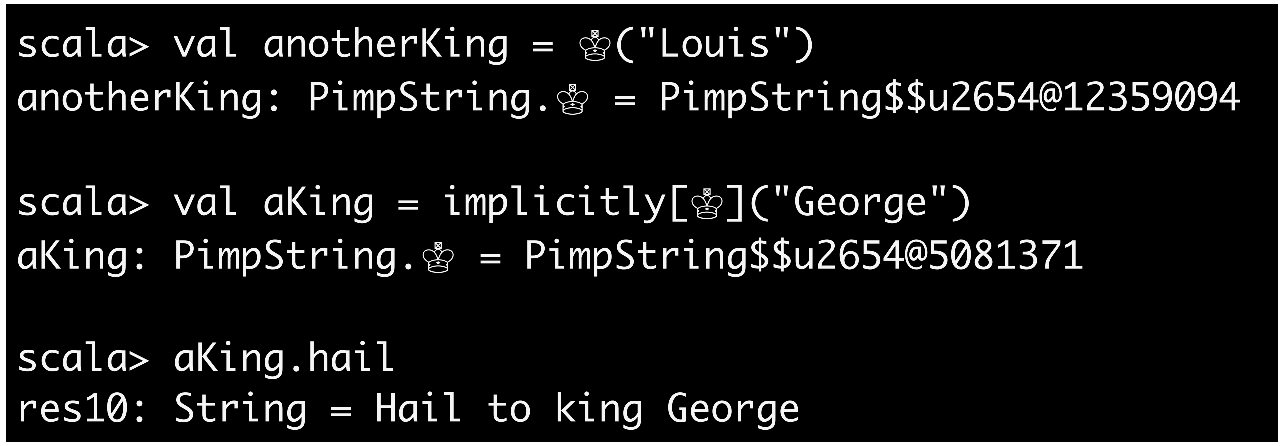
Pimp-My-Library-Pattern : Example Source

```
object PimpString {

    class WeatherString(s: String) {
        def ☀ = { println(s"$s sunny!") }
        def ☁ = "Dont't forget your ☂!“
    }

    implicit class ♔(name : String) {
        def hail = s"Hail to king $name"
    }

    implicit def pimpString(in: String) : WeatherString =
      new WeatherString(in)
}
```

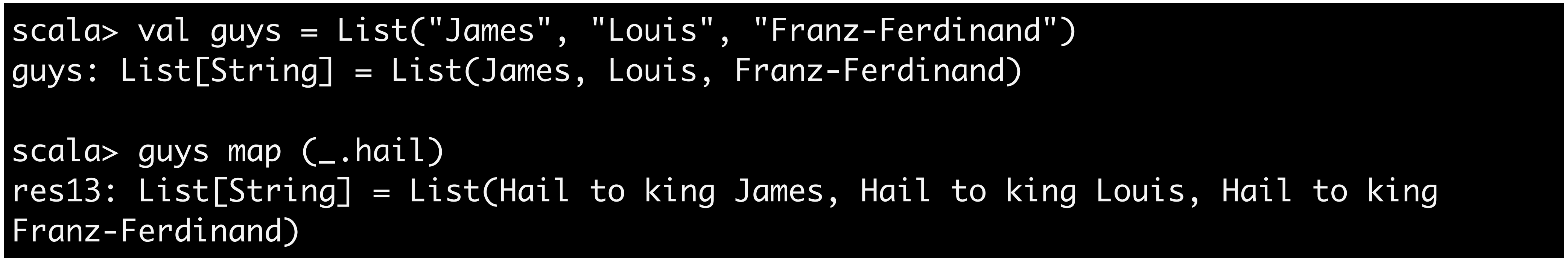
Pimp-My-Library-Pattern : Example Usage



```
scala> import PimpString._
import PimpString._

scala> "Monday is" ☀
Monday is sunny!

scala> "???".☁
res8: String = Dont't forget your ☂
```



```
scala> val anotherKing = ♔("Louis")
anotherKing: PimpString.♔ = PimpString$$u2654@12359094

scala> val aKing = implicitly[♔]("George")
aKing: PimpString.♔ = PimpString$$u2654@5081371

scala> aKing.hail
res10: String = Hail to king George
```



```
scala> val guys = List("James", "Louis", "Franz-Ferdinand")
guys: List[String] = List(James, Louis, Franz-Ferdinand)

scala> guys map (_.hail)
res13: List[String] = List(Hail to king James, Hail to king Louis, Hail to king
Franz-Ferdinand)
```

- Use with caution !

Scala - imperative, object oriented, functional

Scala -imperative, object oriented, functional - Rules of the thumb

- functional if possible
- Sometimes imperative is better and faster
- start out with **val** and **immutable collections**,switch to **var** or **mutable** collections if needed
- Use object orientation to encapsulate side effects and imperative code

Advantage of the functional approach

- short
- no side effect -> easier to reason about
- composable

Advantage of the imperative approach

- familar
- Eventually everything will be iterative after being translated to machine code

Imperative vs. functional, Examples

Imperative

```
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()
```

Functional

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

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

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

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

Imperative vs. functional, Examples 2

Imperative

```
var i = null
var data = gettingData()

if (data != null && data.size > 0)
    i = data(0)
else
    i = 42
```

Functional

```
val i =
    if (data != null && data.size > 0)
        data(0)
    else
        42
```

```
val i =
    gettingData().headOption getOrElse 42
```

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.
- Malayeri, M. “Pimp My Library” Is An Affront To Pimpers Of The World, Everywhere
- http://www.scala-lang.org
- http://www.an-it.com

Thanks for participating :)