Scala 101

Scala 101


```
repeat {
  I.explain(Some("Scala features"))
  you.do(exercises)
  we.interact()
}
```

Hi there!

Let me tell you a little bit about myself

What will you learn?

- Syntax of Scala and differences from Java
- Object-Oriented concepts in Scala
- Basics of functional programming
- Effective use of Scala's collections
- Basic working with SBT and ScalaTest

What is Scala?

- Functional *and* Object-Oriented
- Fully compatible with Java
  - But more concise
- *Very* strongly typed
  - But *inferred*
- Proven in production

What's it good at?

- Concurrency
- Programs that are correct from the beginning
- Internal DSLs
- All the things Java is good at

Challenges with Scala

- Complexity
  - Syntax
  - Type system
  - Operator overloading
- Paradigm shift
  - OO vs FP
- Concurrency
  - Mutability is still possible

Scala 101: block 1

"A cleaner Java"

```
System.out.println("Hello, world!")
```

Type inference


```
val s: String = "Hello, world!"
```

is equivalent to

```
val s = "Hello, world!"
```

String interpolation


```
val hello = "Hello"
val world = "world"
```

```
val greeting = hello + ", " + world + "!"
```

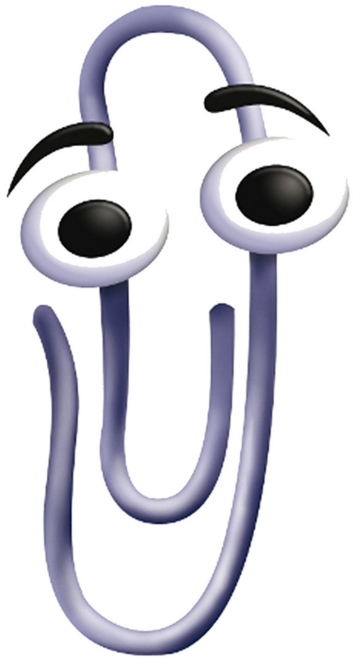
is equivalent to

```
val greeting = s"$hello, $world!"
```

Java interop


```
import java.util.ArrayList

val dates = new ArrayList[LocalDate]

dates.add(new LocalDate(2015, 4, 12))
dates.add(new LocalDate(1999, 12, 31))

System.out.println(dates.size)
```

A note about style



- Never omit () to indicate a side-effecting call:

```
dates.clear()

control.fireTheMissiles()
```

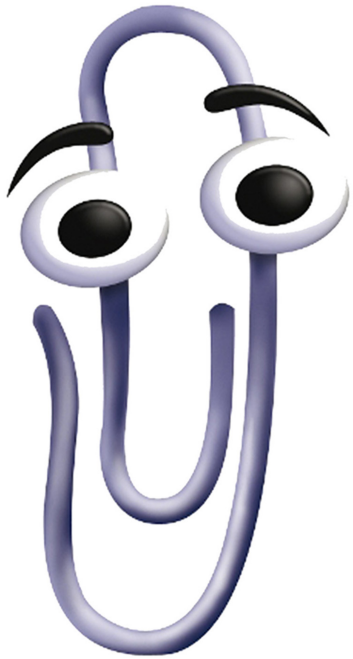
- This improves readability a *lot*

Infix notation


```
val s = "hello"
```

```
s.contains("e")
s contains "e"
(s contains "e")
```

A note about style



- Never use infix notation to indicate a side-effecting call:

```
dates add new LocalDate(2015, 4, 12)  // no

dates.add(new LocalDate(2015, 4, 12)) // yes
```

- This improves readability a *lot*

Infix notation


```
1.to(3)
1 to 3
(1 to 3)
```

Symbolic methods

1.+(3)
1 + 3
(1 + 3)

Symbolic methods

- Other examples

```
bigDecimal1 + bigDecimal2
```

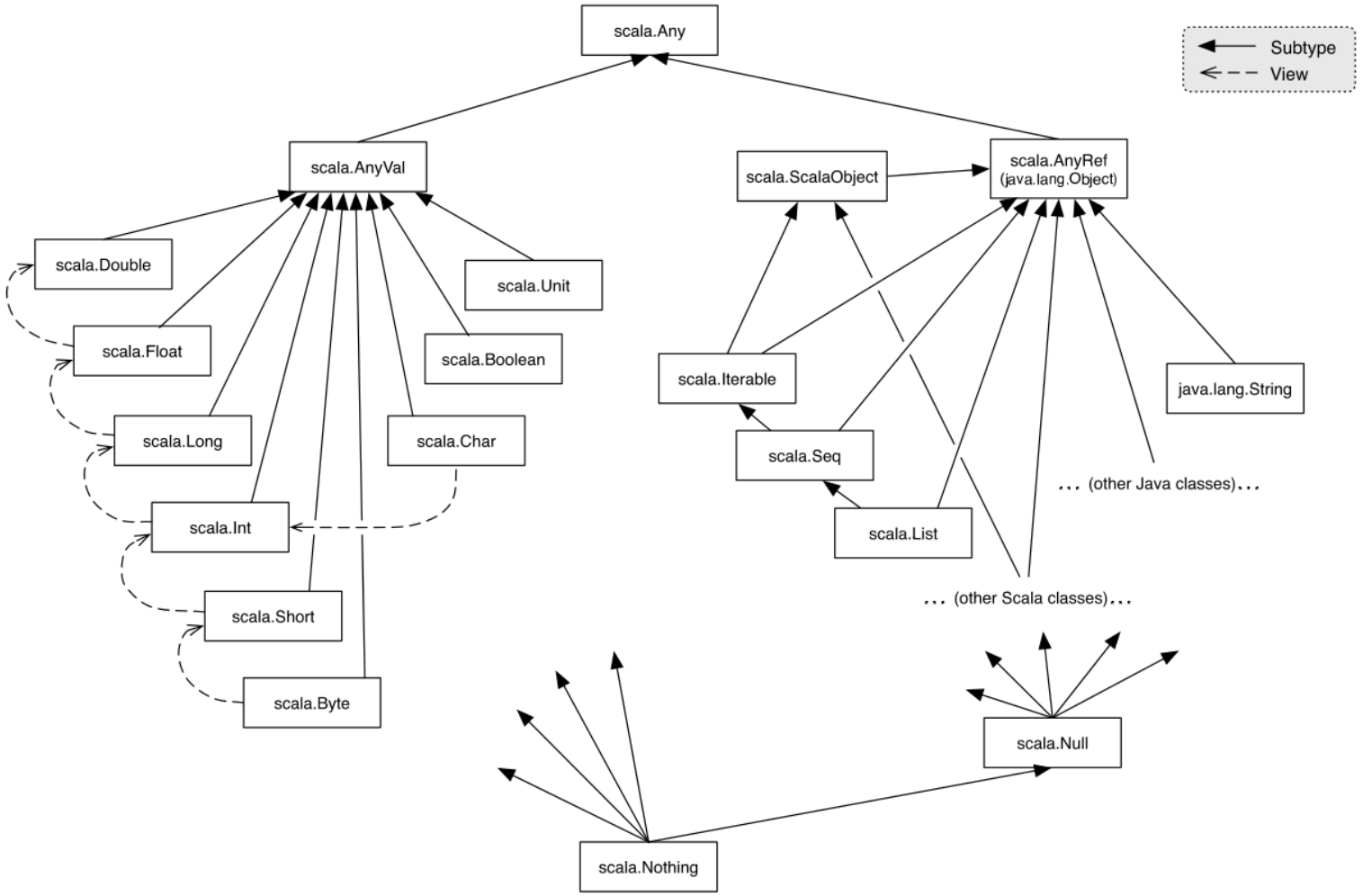
```
actor ! "message"
```

```
val todo = ???
```

???


```
def ??? : Nothing = throw new NotImplementedError
```

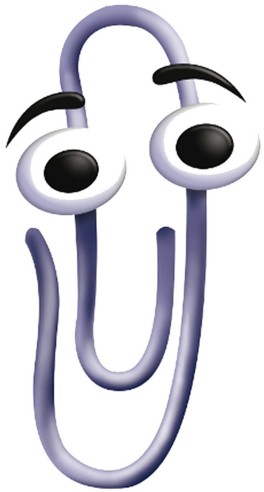
???


scala.Any
Subtype
View
scala.AnyVal
scala.ScalaObject
scala.AnyRef
(java.lang.Object)
scala.Double
scala.Unit
scala.Float
scala.Boolean
scala.Long
scala.Char
scala.Int
scala.Short
scala.Byte
scala.Iterable
java.lang.String
scala.Seq
... (other Java classes)...
scala.List
... (other Scala classes)...
scala.Null
scala.Nothing

A note about style

- Don't go crazy!

x :-| *y*



Periodic Table of Dispatch Operators

All operators of Scala's marvelous Dispatch library on a single page

Start building a Request from scratch

/	<<? (values)	POST	>> ((in, charset) => result)	as_source
:/ (host, port)	/ (path)	PUT	>> ((in) => result)	as_str
:/ (host)	<<< (text)	DELETE	>~ ((source) => result)	>>> (out)
/ (path)	<<< (file, content_type)	HEAD	>- ((text) => result)	>:> ((map) => result)
url (url)	<<< (values)	secure	>>~ ((reader) => result)	>+ (block)
	<< (text)	<& (request)	<> ((elem) => result)	~> ((conversion) => result)
	<< (values)	>\ (charset)	</> ((nodeseq) => result)	>+> (block)
	<< (text, content_type)	to_uri	># ((json) => result)	>! (listener)
	<< (bytes)		>|	
	<:< (map)			
	as (user, passwd)			

ORDINA

Equals and hashCode

- Scala does things a little differently:

```
val s = "hello"
val t = "world"

// Don't do this
val same = s.equals(t)
val hash = s.hashCode

// Do this instead
val same = s == t
val hash = s.##
```

Scala 101: block 2

Object-oriented programming

ORDINA

Classes


```
class Shape(area: Double) {
  // area is private
}

val shape = new Shape(1.0)
```

Classes


```
class Shape(a: Double) {
  // a is private, area is public
  val area = a
}

val shape = new Shape(1.0)
println(shape.area)
```

Classes


```
class Shape(val area: Double) {
  // area is public
}


val shape = new Shape(1.0)
println(shape.area)
```

Subclasses


```
class Circle(val r: Double)
                    extends Shape(Math.PI * r * r) {

  // ...

}

val circle = new Circle(1.0)
println(circle.area)
```

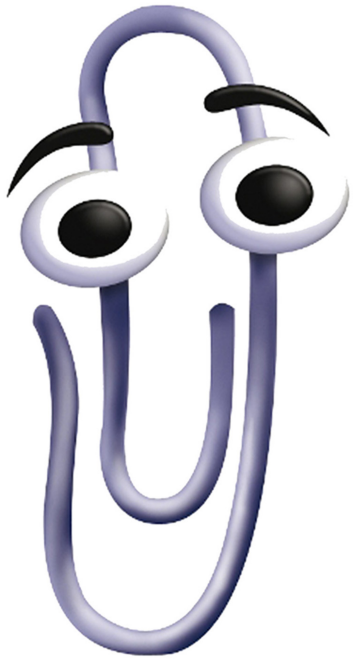
Methods


```
class Circle(val r: Double) {

  def circumference: Double = {
    return 2 * Math.PI * r
  }

  // is equivalent to

  def circumference: Double = 2 * Math.PI * r

}
```

Methods


```
class Circle(val r: Double) {

  def draw(c: Canvas): Unit = {
    val box = determineBoundingBox
    // draw using box
  }

  private def determineBoundingBox =
    new Rectangle(2 * r, 2 * r)

}
```

A note about style



- Never use explicit **return** statements

- Always make a *public* **def**'s return type explicit
  - Even though it's optional in Scala

Case class

- Java

```
public class Point {
   private final int x;
   private final int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public int getX() {
      return x;
   }

   public int getY() {
      return y;
   }

   @Override
   public String toString() {
      return "Point [x=" + x + ", y=" + y + "]";
   }

   @Override
   public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + x;
      result = prime * result + y;
      return result;
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      Point other = (Point) obj;
      if (x != other.x)
         return false;
      if (y != other.y)
         return false;
      return true;
   }
}
```

Case class

- Scala

```
case class Point(x: Int, y: Int)
```

Note:

- Parameters are **val**s automatically

Case class

- Advantages of case classes
  - Constructor, equals, hashCode
  - Immutable
  - copy method
  - Easy serialization with the apply and unapply methods
  - Put your whole domain in a single, short file

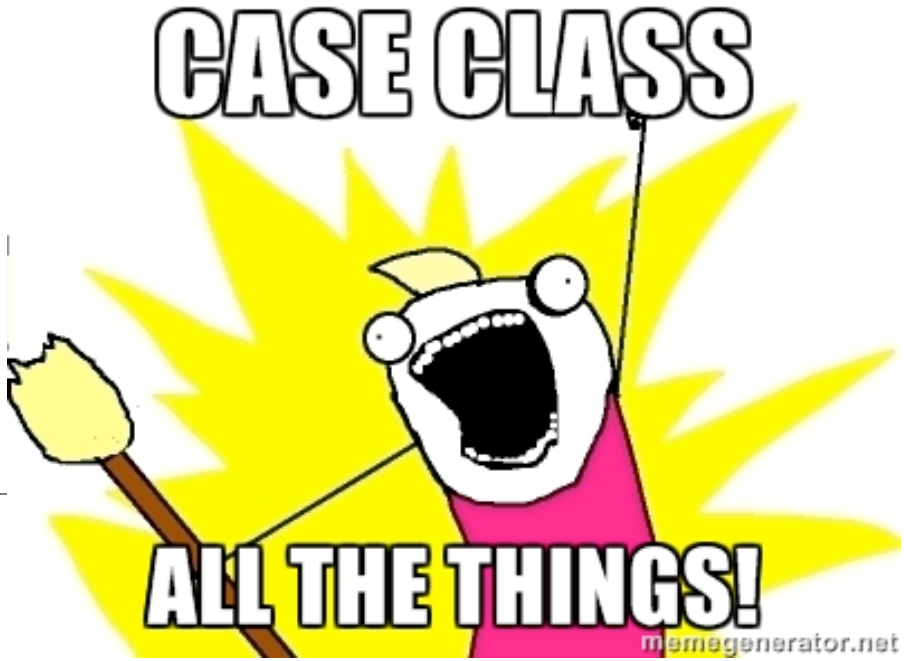
Case class

- Stringly typed:

```
def execute(sql: String): Result
```

- Strongly typed:

```
case class Sql(s: String)
def execute(sql: Sql): Result
```



Singleton object

- Similar to Java's static methods

```
object Main {

  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }

}
```

Singleton object

- You could write Java-like code in Scala:

```
class Circle(val r: Double)

object CircleFactory {
  def create(diam: Double): Circle =
                          new Circle(diam / 2)
}

val c = CircleFactory.create(1.0)
```

Companion object

- In Scala, we don't like factory classes

```
class Circle(val r: Double)

object Circle {
  def create(diam: Double): Circle =
                          new Circle(diam / 2)
}

val c = Circle.create(1.0)
```

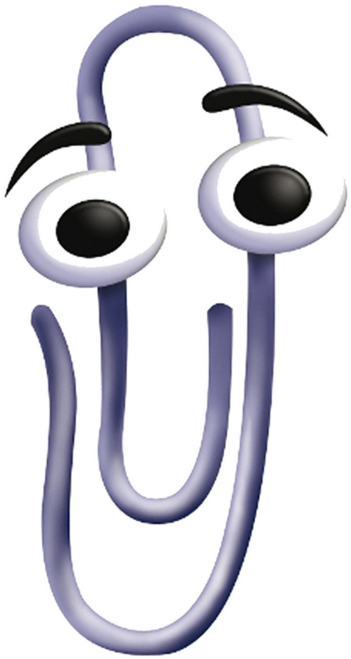
- A companion object must be in the same file as its class

Apply method

- Even more Scala-esque:

```
class Circle(val r: Double)

object Circle {
  def apply(diam: Double): Circle =
                          new Circle(diam / 2)
}

val c = Circle(1.0)
```

A note about style



- Don't go crazy!
  - Use apply for factory methods in companion objects
  - Use apply in DSLs

Trait


```
trait Ordered[T] {
  def compareTo(other: T): Int
}
```

```
case class Player(skill: Int) extends Ordered[Player] {
  override def compareTo(other: Player): Int =
    skill.compareTo(other.skill)
}
```

```
player1.compareTo(player2) < 0
```

Trait


```
trait Ordered[T] {
  def compareTo(other: T): Int

  def < (other: T): Boolean = this.compareTo(other) < 0
  def > (other: T): Boolean = this.compareTo(other) > 0
}

case class Player(skill: Int) extends Ordered[Player] {
  override def compareTo(other: Player): Int =
    skill.compareTo(other.skill)
}
```

player1 < *player2*

Trait mixins

- It's similar to multiple inheritance

```
trait A {
  def doA(): Unit = println("A")
}
trait B {
  def doB(): Unit = println("B")
}

class X extends A with B

val x = new X
x.doA()
x.doB()
```

Import

- Importing classes and traits is like Java:

```
import nl.codestar.scala101.MyTrait
```

- Or

```
import nl.codestar.scala101._
```

Import

- But you can be more specific

```
import nl.codestar.scala101.Circle


val c = Circle.create(1.0)
```

Import

- But you can be more specific

```
import nl.codestar.scala101.Circle.create
```

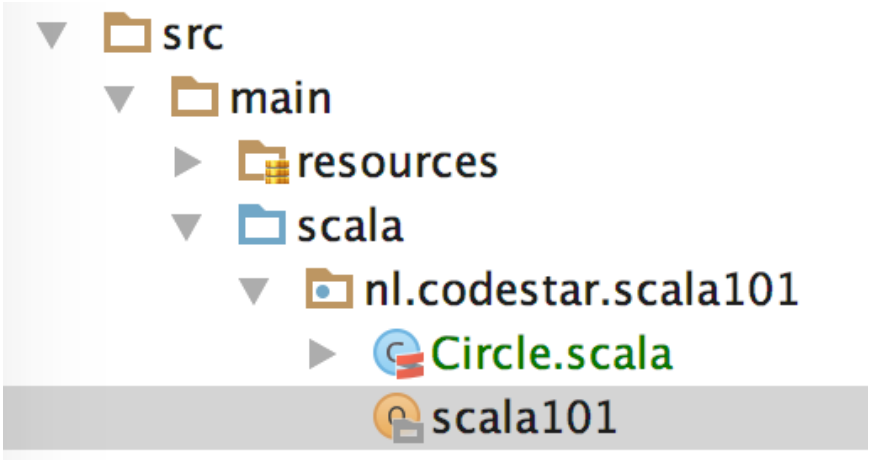
```
val c = create(1.0)
```

Import

- But you can be more specific

```
import nl.codestar.scala101.Circle
import Circle.create

val c = create(1.0)
```

Package object




```
package nl.codestar

package object scala101 {
  val circle1 = Circle(1.0)
}
```

Package object


```
import nl.codestar.scala101._
```

- Gives you everything in the package object as well:

```
println(circle1)
```

Implicit classes


```
implicit class RichInt(i: Int) extends AnyVal {
  def twice = i * 2
}

println(2.twice)
```

- Standard Java types enhanced by Scala's standard library:

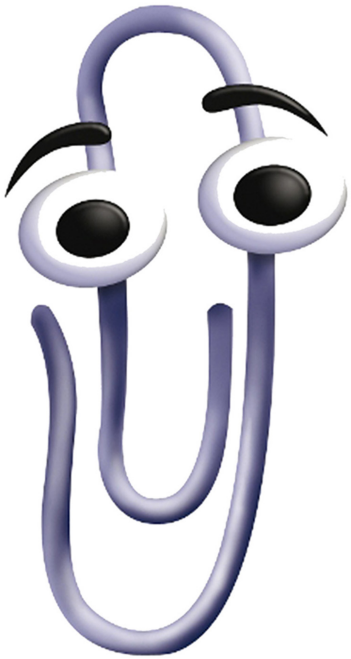
```
1.to(3)              //> Range(1, 2, 3)
"42".toInt           //> 42
```

- Notice the underline in Eclipse and IntelliJ

Implicit parameters


```
def tellMeASecret(implicit secret: String): Boolean = ???

implicit val password = "admin"

if (tellMeASecret) // look mommy, no parameter!
```

A note about style



- Implicits are a controversial Scala feature
- Implicits are an advanced Scala feature
  - Only for API designers
  - But you need to know they exist
- Often it only makes things hard to understand

Exercise: classes and objects

- Given this code:

```
trait PrettyPrintable {
  def prettyPrint: String
}

val shakespeare = Author("William", "Shakespeare")
val hamlet = Book("Hamlet", shakespeare, 1603)

PrettyPrintable.toScreen(hamlet)
// prints "Hamlet, by William Shakespeare, as published in 1603"
```

- How would you implement the classes Author and Book?
- Download the code at: https://github.com/code-star/scala101-exercises

Exercise: one possible solution

```
case class Author(firstName: String, lastName: String)
        extends PrettyPrintable {

  override def prettyPrint: String = s"$firstName $lastName"
}


case class Book(title: String, author: Author, year: Int)
        extends PrettyPrintable {

  override def prettyPrint: String =
    s"$title, by ${author.prettyPrint}, as published in $year"
}


object PrettyPrintable {
  def toScreen(p: PrettyPrintable): Unit = println(p.prettyPrint)
}
```

Scala 101: block 3

Tools

SBT

- Scala Build Tool
- Simple Build Tool
- Interactive Build Tool

SBT

- You can use Ant, Maven, Gradle, Make, …
- But most Scala projects use SBT
  - It's faster for Scala
  - Interactive mode

build.sbt


```
name := "My Cool Project"
version := "1.0"
scalaVersion := "2.11.7"


libraryDependencies ++= Seq(
    "com.typesafe.akka" %% "akka-actor" % "2.4.0",
    "org.scalatest" %% "scalatest" % "2.2.5" % Test
)
```

Running SBT

- sbt clean
- sbt compile
- sbt test

Running SBT interactively


```
$ sbt
[info] Loading project definition from …
[info] Set current project to My Cool Project
> _
```

Running SBT interactively


```
$ sbt
[info] Loading project definition from …
[info] Set current project to My Cool Project
> compile
[success] Total time: 0 s, completed Sep 18, 2015 1:11:49 PM
> _
```

Running SBT interactively


```
$ sbt
[info] Loading project definition from …
[info] Set current project to My Cool Project
> compile
[success] Total time: 0 s, completed Sep 18, 2015 1:11:49 PM
> ~compile
[success] Total time: 0 s, completed Sep 18, 2015 1:12:44 PM
1. Waiting for source changes... (press enter to interrupt)
```

SBT with IntelliJ

- Using IntelliJ?
- You're lucky! IntelliJ's Scala plugin supports SBT out-of-the-box
- Import a build.sbt and follow the wizard

SBT with Eclipse

- Using Eclipse?
- You have to do a little work!
  - Create a file project/plugins.sbt in the root of your project
  - Add this line to it:
    addSbtPlugin(**"com.typesafe.sbteclipse"** % **"sbteclipse-plugin"** % **"4.0.0"**)
  - Run sbt eclipse to generate an Eclipse project
  - Import the project in Eclipse
  - Run sbt eclipse again when your dependencies change

ScalaTest

- Scala's most used testing framework
- Very beautifully designed DSL

ScalaTest

- The problem with JUnit:

```
@Test public void
    twoIntsAddedWithPlusShouldBeTheSumOfTheTwoInts() {}

@Test public void plus() {}
```

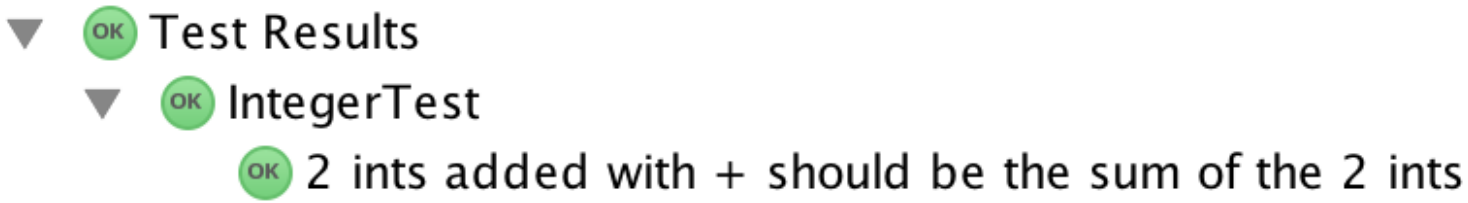
- Test names are usually either long and unreadable, or short and unclear.

ScalaTest

- The solution in ScalaTest:

```
test("2 ints added with + should be the sum of the 2 ints") {
  // ...
}
```

- It also works well with tooling:



ScalaTest

- Assertions:

```
test("2 ints added with + should be the sum of the 2 ints") {

  1 + 1 should be (2)
}


test("my HashMap should contain 1 as a key") {

  map should contain key 1
}
```

ScalaTest

- Testing exceptions:

```
test("division by 0 should throw an exception") {
  intercept[ArithmeticException] {
    1 / 0
  }
}
```

ScalaTest

- There are many styles in which you can write tests

- ScalaTest lets you choose your favourite style
  - By mixing in traits

Configure ScalaTest with traits

- org.scalatest.FunSuite
  gives you the test method

- org.scalatest.Matchers
  gives you the should method

```
class IntegerTest extends FunSuite with Matchers {

  test("2 ints added with + should be the sum of the 2 ints") {

    1 + 1 should be (2)
  }
}
```

Configure ScalaTest with traits

- org.scalatest.FlatSpec
  gives you behavior of and it should

- org.scalatest.Matchers
  gives you the should method

```
class IntegerTest extends FlatSpec with Matchers {

  behavior of "Integer"

  it should "add 2 ints with +" in {
    1 + 1 should be (2)
  }
}
```

Configure ScalaTest with traits

- org.scalatest.MustMatchers
  gives you the must method instead of should

```
class IntegerTest extends FunSuite with MustMatchers {

  test("2 ints added with + should be the sum of the 2 ints") {

    1 + 1 must be (2)
  }
}
```

Scala 101: block 4

Collections

ORDINA

Collections

- Immutable by default
  - But still very performant
- Mutable if you need it

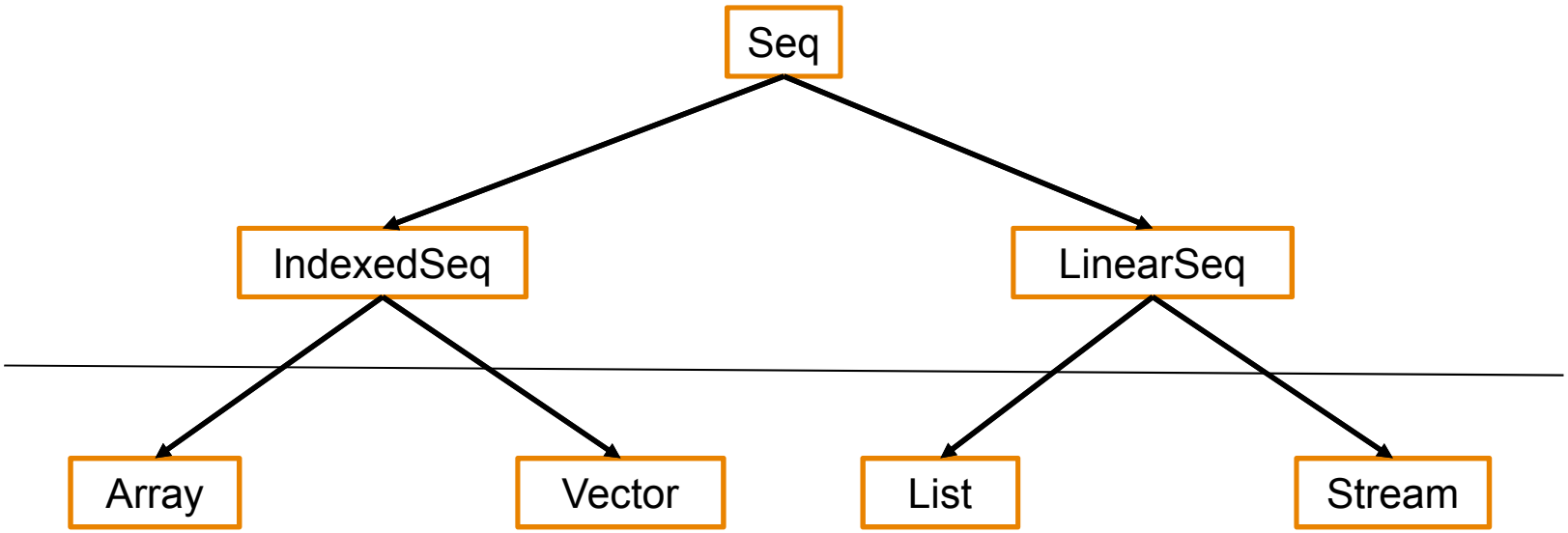
Lists


```
val xs = List(1, 2, 3)       // [1, 2, 3]
val ys = 4 :: 5 :: 6 :: Nil  // [4, 5, 6]
val zs = xs ++ ys            // [1, 2, 3, 4, 5, 6]

zs.head                      // 1
zs.tail                      // [2, 3, 4, 5, 6]

zs(2)                        // 3
zs take 3                    // [1, 2, 3]
zs drop 2                    // [3, 4, 5, 6]

zs.size                      // 6
zs.isEmpty                   // false
```

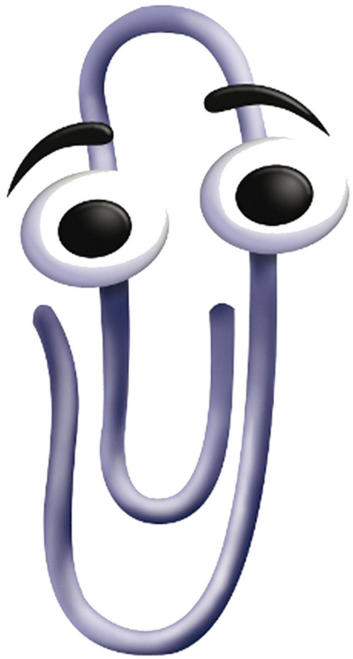
Lists


```
zs mkString ","          // "1,2,3,4,5,6"
zs.reverse               // [6, 5, 4, 3, 2, 1]

zs.min                   // 1
zs.max                   // 6
zs.sum                   // 21

xs zip ys                // [(1,4), (2,5), (3,6)]


val rs = List(2, 4, 1, 3)
rs.zipWithIndex          // [(2,0), (4,1), (1,2), (3,3)
rs.sorted                // [1, 2, 3, 4)
```

Lists



A note about style



- Consider List
  - For pattern matching
  - For recursion
- Consider Vector
  - If you want to index elements
  - It's like Java's ArrayList, ***not*** like Java's Vector

Sets


```
val s = Set(3, 1, 2, 1)     // HashSet(3, 1, 2)
val t = SortedSet(3, 5, 4)  // TreeSet(3, 4, 5)

t + 42                      // Set(3, 4, 5, 42)
t - 3                       // Set(4, 5)

t.size                      // 3
t.isEmpty                   // false
t contains 5                // true

t intersect s               // Set(3)
t union s                   // Set(1, 2, 3, 4, 5)
t diff s                    // Set(4, 5)
```

Maps


```
val m = Map(1 -> "a", 2 -> "b", 3 -> "c") // HashMap(…)

m + (4 -> "d")       // Map(1 -> a, 2 -> b, 3 -> c, 4 -> d)
m - 1                // Map(2 -> b, 3 -> c)
m.updated(3, "???")  // Map(1 -> a, 2 -> b, 3 -> ???)

m(2)                 // "b"
m(5)                 // NoSuchElementException
m.get(2)             // Some("b")
m.get(5)             // None
m.getOrElse(5, "???") // "???"

m.isEmpty            // false
m contains 5         // false

m.keys               // Set(1, 2, 3)
m.unzip              // (List(1, 2, 3), List("a", "b", "c"))
```

Option


```
def findUser(id: Int): Option[Person] = {
  val result = doQuery(id)

  if (result == null)
    None
  else
    Some(result)
}
```

- Scala avoids **null**

Option


```
val user: Option[Person] = findUser(42)


if (user.isDefined) {
  val u = user.get
  // do something
}
else {
  // do something else
}


val u = user.getOrElse(defaultUser)


val a: Option[Address]  = user.map(_.address)
```

Exercise: Shakespeare

- Input:

```
val phrase = "brevity is the soul of wit"
```

- Output:

```
(brevity,0)
(is,1)
(of,4)
(soul,3)
(the,2)
(wit,5)
```

Exercise: one possible solution

```
phrase.split(" ").zipWithIndex.sorted.mkString("\n")
```

Scala 101: block 5

Functional programming

Imperative vs Functional

- Imperative programming is about *executing instructions* and *updating state*
  - First do X, then do Y
  - Z is updated

- Functional programming is about *transforming data*
  - Call X with the result of Y
  - X and Y have no side-effects
  - Z is immutable

Referential transparency

- An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program.

```
  1 + 1
=
  2
```

val vs var

- Functional programming is about transforming *immutable* data

```
val i = 0
i = i + 1  // does not compile
```

```
var j = 0
j = j + 1  // ok
```

lazy val

Executed *when needed* and only *once*:

```
lazy val pi = very_expensive_computation
```

Everything is an expression


```
val discount = if (totalPrice > 100)
                 totalPrice * 0.05
               else
                 0.0
```

```
val x: Unit = println("Hello, world!")
println(x)
```

```
Hello, world!
()
```

Pattern matching


```
def intToString(x: Int): String =
  x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "I don't know"
  }
```

Pattern matching


```
def maybeIntToString(x: Option[Int]): String =
  x match {
    case Some(i) => i.toString
    case None    => "I don't know"
  }
```

Pattern matching


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

```
def isAwesome(person: Person): Boolean =
  person match {
    case Person("Martin", "Odersky")      => true
    case Person(n, _) if n startsWith "J" => true
    case _                                => false
  }
```

Partial functions


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


val isAwesome: PartialFunction[Person, Boolean] = {
  case Person("Martin", "Odersky")      => true
  case Person(n, _) if n startsWith "J" => true
  case _                                => false
}
```

Partial functions


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

```
val isAwesome: PartialFunction[Person, Boolean] = {
  case Person("Martin", "Odersky")      => true
  case Person(n, _) if n startsWith "J" => true
}

isAwesome(Person("Martijn", "Blankestijn"))
```

Partial functions


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


val isAwesome: PartialFunction[Person, Boolean] = {
  case Person("Martin", "Odersky")      => true
  case Person(n, _) if n startsWith "J" => true
}

isAwesome.isDefinedAt(Person("Martijn", "Blankestijn"))
```

Lambdas


```
def twice(n: Int): Int = n * 2
```

Lambdas


```
def twice(n: Int): Int = n * 2

val twice: Int => Int = (n: Int) => n * 2

val twice: Int => Int = _ * 2
```

Lambdas


```
def twice(n: Int): Int = n * 2

val twice: Int => Int = (n: Int) => n * 2

val twice: Int => Int = _ * 2
```

Lambdas


```
def twice(n: Int): Int = n * 2

val twice: Int => Int = (n: Int) => n * 2

val twice: Int => Int = _ * 2
```

Scala 101: block 6

Higher-order functions

ORDINA

Higher order functions


```
// higher order function:
def applyFunction(n: Int, f: Int => Int): Int = f(n)


applyFunction(2, n => n + 1)  // returns 3
applyFunction(2, _ + 1)       // also returns 3

applyFunction(3, twice)       // returns 6
```

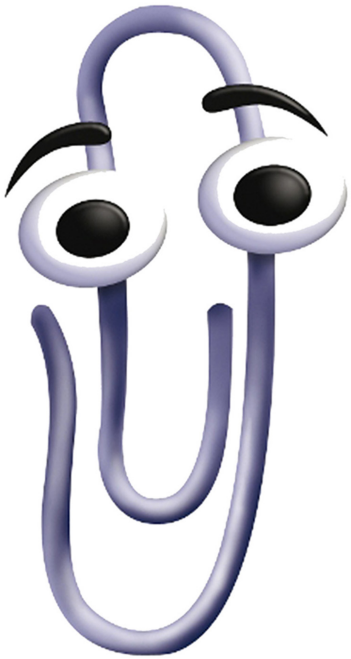
Higher order functions: collections


```
val langs = List("Scala", "Java", "C#", "Python")


langs.filter(s => s.contains("a"))
                // List(Scala, Java)

langs.map(_.toUpperCase)
                // List(SCALA, JAVA, C#, PYTHON)

langs.sortBy(_.length)
                // List(C#, Java, Scala, Python)
```

Higher order functions: alternative notations


```
val langs = List("Scala", "Java", "C#", "Python")


langs filter { s => s.contains("a") }
                // List(Scala, Java)

langs map { _.toUpperCase }
                // List(SCALA, JAVA, C#, PYTHON)

langs sortBy { _.length }
                // List(C#, Java, Scala, Python)
```

A note about style



- Method-call notation

```
langs.filter(_.contains("a")).map(_.toUpperCase)
```

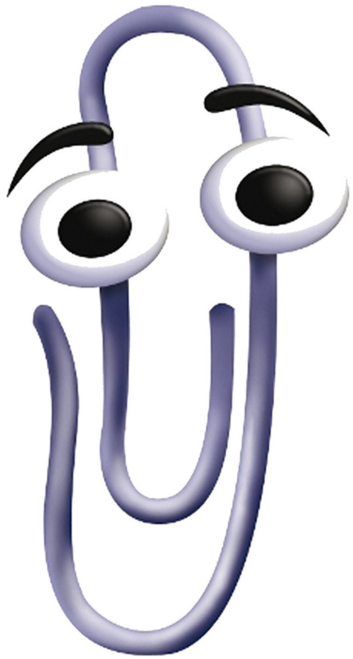
- Infix notation

```
langs filter { _.contains("a") } map { _.toUpperCase }
```

- **Do not mix the notations!**
  - It doesn't do what you think it does

```
langs.filter { _.contains("a") }.map { _.toUpperCase }
```

A note about style



- Method-call notation

```
langs.filter(_.contains("a")).map(_.toUpperCase)
```

- Infix notation

```
langs filter { _.contains("a") } map { _.toUpperCase }
```

- **Do not mix the notations!**
  - It doesn't do what you think it does

```
langs.filter { _.contains("a") }.map { _.toUpperCase }
```

Higher order collections functions


```
val xs = List(1, 2, 3, 4, 5)


xs takeWhile {n => n < 3}        // List(1, 2)
xs dropWhile {n => n < 3}        // List(3, 4, 5)
xs find {n => n > 3}             // Some(4)

xs exists {n => n > 10}          // false
xs forall {n => n < 10}          // true

xs maxBy {n => -n}               // -1
xs minBy {n => -n}               // -5

xs sortWith {(n, m) => n > m}    // List(5, 4, 3, 2, 1)
xs groupBy {n => n % 2}          // Map(0 -> List(2,4),
                                 //     1 -> List(1,3,5))
```

Higher order collections functions: flatMap


```
val xs = List(1, 2, 3, 4, 5)


val ys = xs map (List(_))  // List(List(1), List(2), List(3),
                           //      List(4), List(5))
ys.flatten                 // List(1, 2, 3, 4, 5)


xs flatMap (List(_))       // List(1, 2, 3, 4, 5)
```

Higher order collections functions: flatMap


```
val xs = List(1, 2, 3, 4, 5)


def even(n: Int): Option[Int] =
            if (n % 2 == 0) Some(n) else None


xs map even     // List(None, Some(2), None, Some(4), None)

xs flatMap even // List(2, 4)
```

Exercise: sets of integers

- How would you represent the set of **all** negative integers?
  - You cannot list them all…

Exercise: sets of integers

- How would you represent the set of **all** negative integers?
  - You cannot list them all…
- Let's do it as a function!
  - You give it a number
  - It says 'yes' or 'no'

Exercise: sets of integers

- We can define the set of negative integers by the "characteristic function":

```
val negative: Int => Boolean =
  (x: Int) => x < 0
```

- Similarly, this is the set of all even integers:

```
val even: Int => Boolean =
  (x: Int) => x % 2 == 0
```

Exercise: sets of integers

```
type Set = Int => Boolean
```

- We can define the set of negative integers by the "characteristic function":

```
val negative: Set =
  (x: Int) => x < 0
```

- Similarly, this is the set of all even integers:

```
val even: Set =
  (x: Int) => x % 2 == 0
```

Exercise: sets of integers

- Using this representation, we define a function that tests for the presence of a value in a set:

```
def contains(s: Set, elem: Int): Boolean = s(elem)
```

Exercise: sets of integers

- We need some more functions to work effectively with Sets.
- Let's define them!
- Hint: return lambdas…

```
def singletonSet(elem: Int): Set

def union(s: Set, t: Set): Set

def intersect(s: Set, t: Set): Set

def diff(s: Set, t: Set): Set
```

Note: diff(s, t) returns a set which contains all the elements of the set s that are not in the set t.

Exercise: a possible solution

```
def singletonSet(elem: Int): Set =
  (x: Int) => x == elem
def union(s: Set, t: Set): Set =
  (x: Int) => contains(s, x) || contains(t, x)
def intersect(s: Set, t: Set): Set =
  (x: Int) => contains(s, x) && contains(t, x)
def diff(s: Set, t: Set): Set =
  (x: Int) => contains(s, x) && !contains(t, x)
```

Exercise: another possible solution

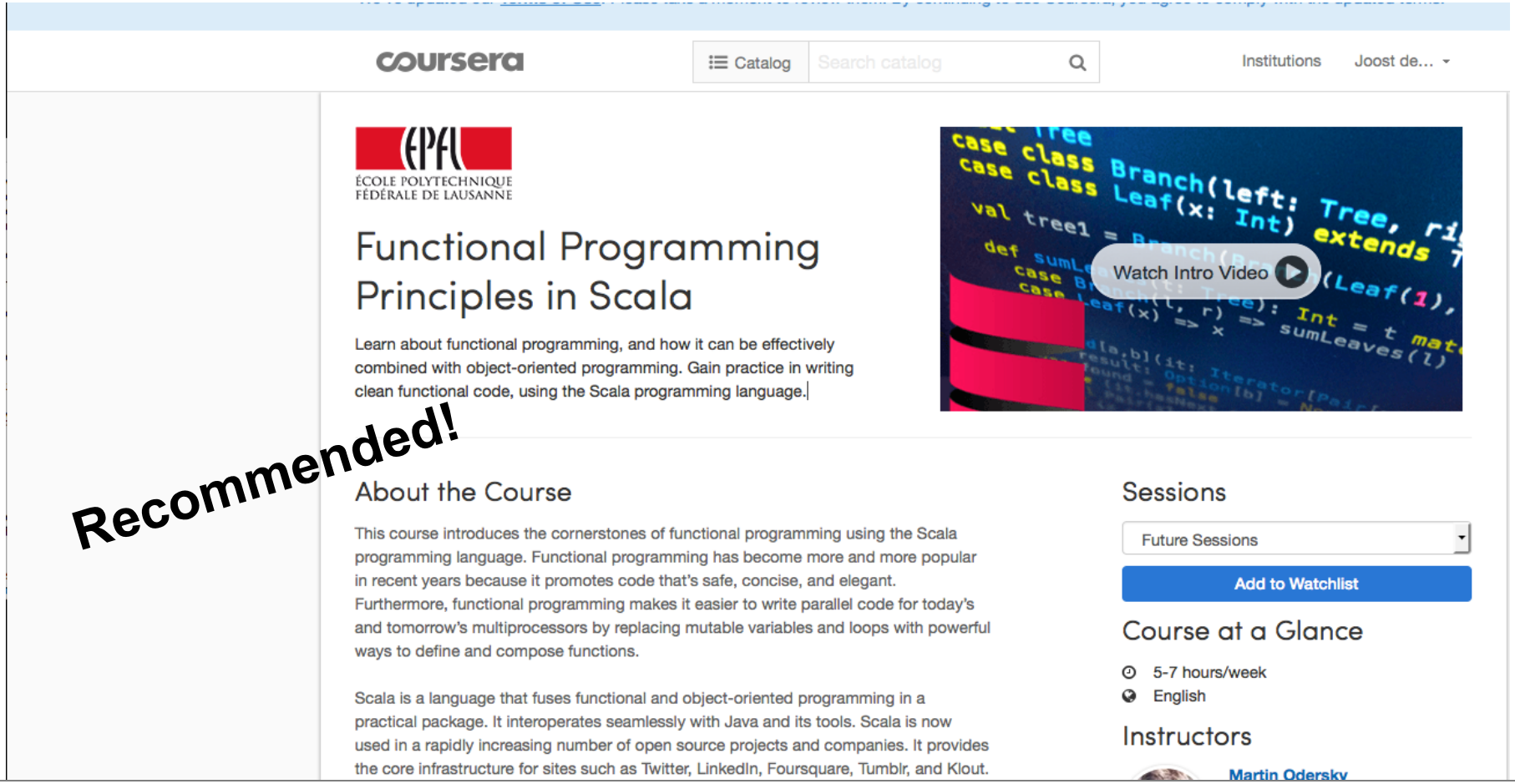
```
def singletonSet(elem: Int): Set =
  _ == elem
def union(s: Set, t: Set): Set =
  (x: Int) => contains(s, x) || contains(t, x)
def intersect(s: Set, t: Set): Set =
  (x: Int) => contains(s, x) && contains(t, x)
def diff(s: Set, t: Set): Set =
  (x: Int) => contains(s, x) && !contains(t, x)
```

Resources to go further with Scala

- Twitter's Scala School
  http://twitter.github.io/scala_school/
- Free e-book: Scala for the Impatient
  https://www.typesafe.com/resources/e-book/scala-for-the-impatient
- Book: Functional Programming in Scala
  https://www.manning.com/books/functional-programming-in-scala

Coursera – Functional Programming in Scala

https://www.coursera.org/course/progfun



Recommended!

ICT. MAAR DAN VOOR MENSEN

www.ordina.nl