

*a short introduction to*



Thanks to:

Vasia Kalavri

# This tutorial covers

- Scala basics
  - expressions, values, functions, control structures
- Basic functional programming in Scala
- Basic OOP in Scala
- Basic Types
- Pattern Matching

# **This tutorial does not cover**

- Polymorphism
- Concurrency
- SBT
- Unit Testing
- DSLs
- Java Interoperability

# A Scalable Language

a bit of history:

**2001:** Development starts at EPFL

**2004:** First release

**2006:** Second release

**2011:** Typesafe Inc. launch

large systems



small scripts

# **Why Scala?**

or a few motivating examples

# do more, type (a lot) less!

```
public class Person {  
    private final String firstName;  
    private final String lastName;  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    public String getFirstName() {  
        return firstName;  
    }  
    public String getLastName() {  
        return lastName;  
    }  
}
```

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

# it's a matter of taste

```
def printArgs(args: Array[String]): Unit = {  
  var i = 0  
  while (i < args.length) {  
    println(args(i))  
    i += 1  
  }  
}
```

**imperative**

```
def printArgs(args: Array[String]): Unit = {  
  args.foreach(println)  
}
```

**functional**

# sweet type inference

```
case class MyPair[A, B](x: A, y: B);

object InferenceTest extends App {
  def id[T](x: T) = x
  val p = MyPair(1, "scala") // type: MyPair[Int, String]
  val q = id(1)              // type: Int
}
```

```
val x: MyPair[Int, String] = MyPair[Int, String](1, "scala")
val y: Int = id[Int](1)
```



## and more...

- runs on the JVM
- can use Java libraries (and vice-versa)
- pattern matching
- concurrency made easy
- extensibility

# Scala Basics

# execution modes

```
object Hello {
  def main(args: Array[String]) = {
    val greeting = {
      if (args.length > 0)
        "Hello, " + args(0) + "!"
      else
        "Hello, there!"
    }
    println(greeting)
  }
}
```

*helloc.scala*

```
:~$ scalac helloc.scala
:~$ scala hello.scala
Hello, there!
```

```
scala> val name = "Vasia"
name: java.lang.String = Vasia

scala> println("Hello, " + name)
Hello, Vasia
```

```
val name = argv(0)
println("Hello, " + name + "!")
hello.scala
```

```
:~$ scala hello.scala Vasia
Hello, Vasia!
```

# expressions

- **identifiers:** `x`, `*`, `+`
- **literals:** `0`, `1.0`, `"abc"`
- **field or method selections:** `System.out.println`
- **function applications:** `sqrt(x)`
- **operator applications:** `y + x`
- **conditionals:** `if (x > 0) x else -x`
- **blocks:** `{ val x = abs(y) ; x * 2 }`
- **anonymous functions:** `x => x + 1`

# types

- **number** types Byte, Short, Char, Int, Long, Float and Double (as in Java)
- **Boolean** with values true and false
- **Unit** with the only value ()
- **String**
- **function types** such as (Int, Int) => Int or String => Int => String

# flow-control structures

```
> if (x > 0) println("positive") else println("negative")
```

```
val configFile = if (configFile.exists()) {  
    configFile.getAbsPath()  
} else {  
    configFile.createNewFile()  
}
```

```
> while (it.hasNext) foo(it.next)
```

```
> for (i <- 1 to 4) println(i)
```

```
> for (j <- List.range(1, 10)) println(j * 2)
```

```
> for (p <- persons if p.age > 20) yield p.name
```

- if-statements are expressions and we can assign the result of an if-expression to a variable

- <- iterates through the elements of a collection
- Use the yield keyword to generate a new collection

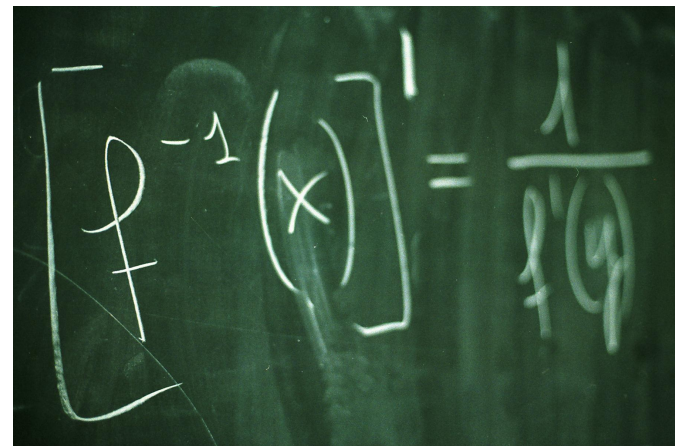
# functions

```
> def square(x: Double) = x * x
> square: (x: Double)Double

> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
> sumOfSquares: (x: Double, y: Double)Double

> def abs(x: Double) = if (x >= 0) x else -x
> abs: (x: Double)Double

> def timesTwo(i: Int): Int = {
  println("hello world")
  i * 2
}
> timesTwo: (i: Int)Int
```



A photograph of a chalkboard with a handwritten mathematical formula in white chalk. The formula is  $[f^{-1}(x)]' = \frac{1}{f'(y)}$ , representing the derivative of the inverse function. The chalkboard is dark green, and the handwriting is somewhat cursive and slightly blurred.

# val vs. var

```
val array: Array[String] = new Array(5)
array = new Array(3)      // error!
array(0) = 3              // ok :-)
```

- `val` declares an **immutable** (read-only) variable
- the *reference* cannot change, but the object state can still be modified
- both `val` and `var` must be initialized when declared



# **Basic OOP in Scala**

# classes

```
class Point (val x: Int, val y: Int) {  
  
    def vectorAdd (newpt: Point): Point = {  
        new Point (x + newpt.x, y + newpt.y)  
    }  
}
```

> val point = new Point(1, 5)

- the primary constructor is the *entire body* of the class
- the compiler generates a private field corresponding to each parameter and a public reader method with the same name
- single inheritance

# objects

*why define a class, if we only need a single object?*

- an object definition
  - has an optional extends clause and body
  - defines a single object
  - can appear anywhere in a Scala program; including at top-level

# object definition example

```
abstract class IntSet {  
    def contains(x: Int): Boolean  
    def incl(x: Int): IntSet  
}  
  
object EmptySet extends IntSet {  
    def contains(x: Int): Boolean = false  
    def incl(x: Int): IntSet = ...  
}
```

# traits

- similar to Java interfaces, allow some kind of multiple inheritance
- a class can extend a trait and a trait can extend a class
- when adding traits in a class, the order of declaration matters!

# traits example

```
trait Car {  
  val brand: String  
}
```

```
trait Shiny {  
  val shineRefraction: Int  
}
```

```
class BMW extends Car with Shiny {  
  val brand = "BMW"  
  val shineRefraction = 12  
}
```

# hands-on: classes, objects, traits

```
abstract class Shape {  
    def getArea(): Int  
}
```

```
class Rectangle(x: Int, y: Int) extends Shape {  
    def getArea(): Int = x*y  
}
```

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

```
class Circle(radius: Int, center: Point) extends Shape {  
    def getArea(): Int = radius*radius*Math.Pi  
    override def toString = "{(" + center.x + "," + center.y + ")," + radius +  
    "}"  
}
```

# Functional Programming



# the idea

- functions without *side-effects* that always behave the same way
- *immutable* data structures
- functions are first-class
  - you can assign them to variables
  - you can pass them to other functions
  - you can return them as values

# lists

```
scala> val lst = List(1, 2, 3, "boo", 3.14)
lst: List[Any] = List(1, 2, 3, boo, 3.14)
```

```
scala> val a_list = 'a'::lst
a_list: List[Any] = List(a, 1, 2, 3, boo, 3.14)
```

```
scala> lst(2)
res0: Any = 3
```

```
scala> lst.head
res1: Any = 1
```

```
scala> lst.tail
res2: List[Any] = List(2, 3, boo, 3.14)
```

# tuples

- a tuple groups items together

```
> val hostPort = ("localhost", 80)
hostPort: (String, Int) = (localhost, 80)
```

```
> val person = ("Tom", 42, "Bristol")
person: (java.lang.String, Int, java.lang.String) =
(Tom, 42, Bristol)
```

```
> val point = (1, 3)
point: (Int, Int) = (1, 3)
```

```
> point._1
res1: Int = 1
```

```
> point._2
res2: Int = 3
```

# anonymous functions

An expression that evaluates to a function:

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

```
> res1: (Int) => Int = <function>
```

You can pass them around or save them into vals:

```
> val addOne = (x: Int) => x + 1
```

```
addOne: (Int) => Int = <function1>
```

```
> addOne(1)
```

```
res4: Int = 2
```



# higher-order functions (1)

```
def lst = List(1, 2, 3, 4, 5)
```

```
> lst.map(x => x + 1)
```

```
res1: List[Int] = List(2, 3, 4, 5, 6)
```

```
> lst.filter(x => x > 3)
```

```
res2: List[Int] = List(4, 5)
```

```
> lst.foreach(i => println("element " + i) )
```

```
element 1
```

```
element 2
```

```
element 3
```

```
...
```

# higher-order functions (2)

```
val nestedNumbers = List(List(1, 2), List(3, 4))
nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

> nestedNumbers.flatMap(x => x.map(_ * 2))
res0: List[Int] = List(2, 4, 6, 8)

> nestedNumbers.map((x: List[Int]) => x.map(_ * 2)).flatten
res1: List[Int] = List(2, 4, 6, 8)

> def sum(xs: List[Int]) = (0 :: xs) reduceLeft {(x, y) => x + y}
sum: (List[Int])Int

> sum(List(1, 2, 3, 4))
> res0: Int = 10
```

# **\_ rules**

- $(x,y) \Rightarrow x + y$  can be replaced by `_+_`
- $v \Rightarrow v.Method$  can be replaced by `_.Method`

```
> val lst = List(11, 32, 3, 64, 9, 76)
```

```
> lst.filter(_ > 42) vs. lst.filter(x => x > 42)
```

```
> val lstw = List("Simplify", "your", "code", "with", "underscore")
```

```
> lstw.flatMap(_.toList).map(_.toUpperCase).removeDuplicates.sort(_ < _)
```

# Pattern Matching



# the pattern matching expression

`e match { case p1 => e1 ... case pn => en }`

- matches the patterns  $p_1, \dots, p_n$  in the order they are written against the selector value  $e$
- a variable pattern  $x$  matches any value and binds the variable name to that value
- the wildcard pattern `'_'` matches any value but does not bind a name to that value

# pattern matching example

```
val list_a = List(1, 2, 3, 4, 5)
val list_b = List("foo", "bar")
val list_c = List()
val lsts = List(list_a, list_b, list_c)

> lsts.foreach(l => l match {
  case List(_, 2, _, _, _) => println("list_a!")
  case List(x, _*) => println(x)
  case _ => "other"
}))
```

# case classes

```
abstract class TreeN

  case class InterN(key: String, left: TreeN, right: TreeN): Boolean
  case class LeafN(key:String, value: Int) extends TreeN

def find(t: TreeN, key: String): Int = {
  t match {
    case InterN(k, l, r) => find((if (k >= key) l else r), key)
    case LeafN(k, v) => if (k == key) v else 0
  }
}
```

# references and resources

- Scala Cheat Sheet  
<http://github.com/lrytz/progfun-wiki/blob/gh-pages/CheatSheet.md>
- Scala By Example  
[www.scala-lang.org/docu/files/ScalaByExample.pdf](http://www.scala-lang.org/docu/files/ScalaByExample.pdf)
- Scala School  
[http://twitter.github.io/scala\\_school/](http://twitter.github.io/scala_school/)
- Simply Scala  
<http://www.simplyscala.com/>
- A Tour of Scala  
<http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html>
- Effective Scala  
<http://twitter.github.io/effectivescala/>