A Gentle Introduction to

Scala

OBJECT - ORIENTED

FUNCTIONAL

Apostolos N. Papadopoulos

Assistant Professor
Data Engineering Lab,
Department of Informatics
Aristotle University of Thessaloniki
GREECE

The Very Basics

"*If I were to pick a language to use today other than Java, it would be Scala.*"

—James Gosling

(father of Java)

The Very Basics

Scala = **SCA**lable **LA**nguage

i.e., designed to grow with the demands of its users

Development started in 2001 by **Martin Odersky** and his team at EPFL

First release in **January 2004**

Current version **2.11.5** (release date 16/12/2014)

The Very Basics

Scala is a general-purpose programming language that runs on Java Virtual Machine (JVM) and .NET

Expresses common programming patterns in a **concise**, **elegant**, and **type-safe** way.

Scala supports both the **object-oriented** and the **functional** programming model.

The Very Basics

Scala is object-oriented:

- Encapsulation
- Inheritance
- Polymorphism
- All predefined types are objects
- All user-defined types are objects
- Objects communicate by message exchange

The Very Basics

Scala is functional:

- Functions are first-class values
- Can define a function in another function
- Can map input values to output values
- Can do lazy evaluation
- Supports pattern matching
- Higher-order functions

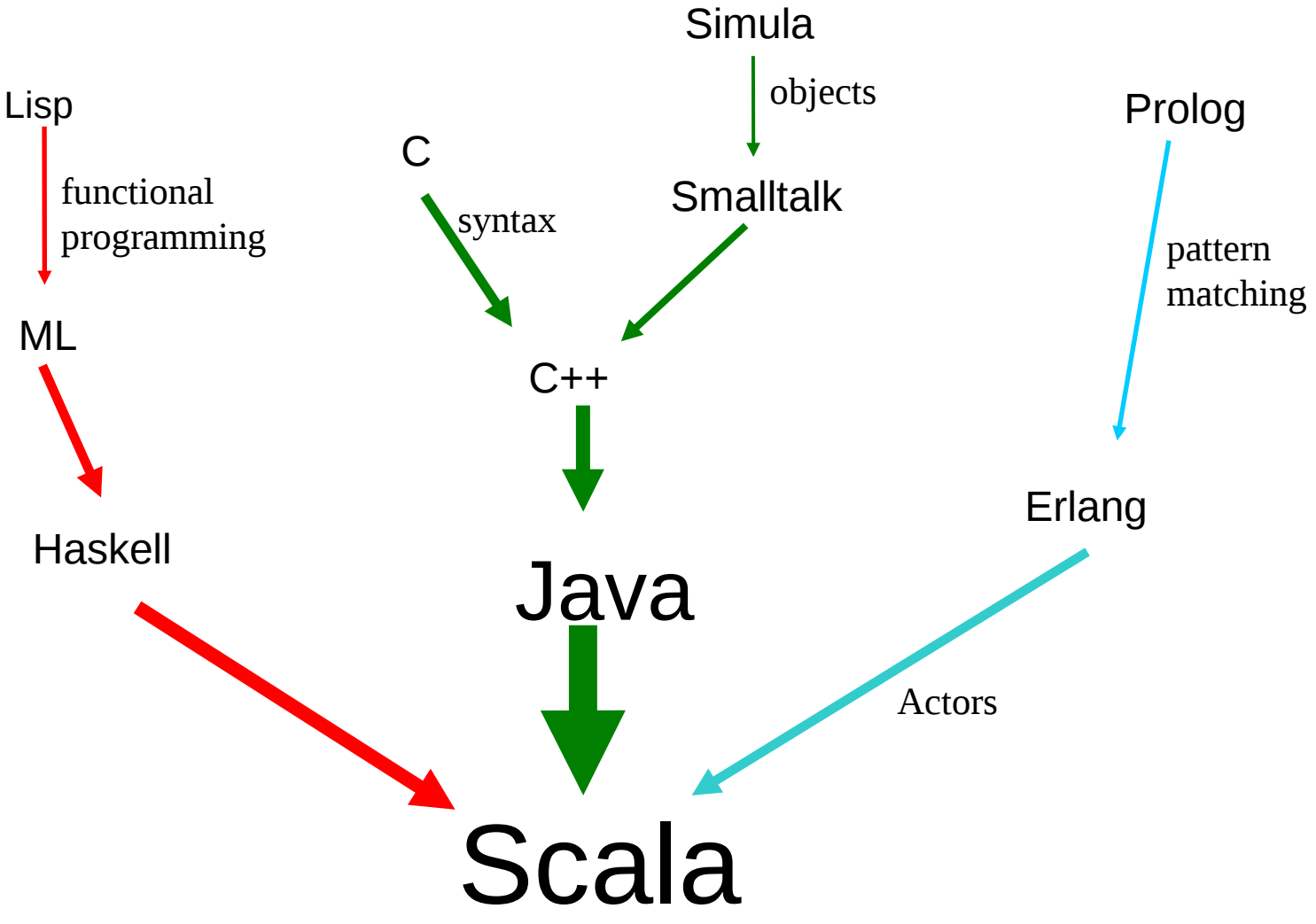
Scala is **not a PURE** functional language however.

The Very Basics

Scala has been inspired by other programming languages:

- Scala's object model was pioneered by **Smalltalk** and taken up subsequently by **Ruby**
- Scala adopts a large part of the syntax of **Java** and **C#**
- Its idea of universal nesting (almost every construct in Scala can be nested inside any other construct) is also present in **Algol**, **Simula**, and, more recently in **Beta**
- Its uniform access principle for method invocation and field selection comes from **Eiffel**
- Its approach to functional programming is quite similar in spirit to the **ML family** of languages, which has **SML**, **OCaml**, and **F#** as prominent members
- It adopts the Actor model for concurrent computation from **Erlang**

The Very Basics



The Very Basics

Scala is a **statically typed language** like Java.

In static typing, a variable is bound to a particular type for its lifetime. Its type can’t be changed and it can only reference type-compatible instances.

That is, if a variable refers to a value of type A , you can’t assign a value of a different type B to it, unless B is a subtype of A , for some reasonable definition of “subtype.”

This is different than in dynamically typed languages such as Ruby, Python, Groovy, JavaScript, Smalltalk and others.

Scala vs Java

Java code:

```
class Book {
      private String authorName;
      private String bookTitle;
      private int    year;

      public Person(String authorName, String bookTitle, int year) {
          this.authorName = authorName;
          this.bookTitle  = bookTitle;
          this.year  = year;
      }

      public void setAuthorName(String authorName) { this.authorName = authorName; }
      public void String getAuthorName() { return this.authorName; }
      public void setBookTitle(String bookTitle) { this.bookTitle = bookTitle; }
      public void String getBookTitle() { return this.bookTitle; }
      public void setYear(int year) { this.age = year; }
      public void int getYear() { return this.year; }
   }
```

Scala code:

```
class Book (var authorName: String, var bookTitle: String, var year: Int)
```

The First Scala Program

A very simple program in Scala:

```
/* Our first Scala program */
object HelloWorld {
  /* The main function */
  def main(args: Array[String]) {
    println("Hello, world!")
    }
}
```

To define a singleton we use the keyword **object**

Scala Data Types

`Byte`	8 bit signed value, Range from -128 to 127
`Short`	16 bit signed value. Range -32768 to 32767
`Int`	32 bit signed value. Range -2147483648 to 2147483647
`Long`	64 bit signed value. -9223372036854775808 to 9223372036854775807
`Float`	32 bit IEEE 754 single-precision float
`Double`	64 bit IEEE 754 double-precision float
`Char`	16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
`String`	a sequence of Chars

Simple Functions

```
// count from 1 to 10
def countTo(n: Int) {
    for (i <- 1 to 10) {
      println(i)
    }
}
```

```
// return true if a number is even, false otherwise
def isEven(n: Int) = {
    val m = n % 2
    m == 0
}
```

Immutable vs Mutable

An **immutable** element cannot change its value. Immutable elements help in creating more robust parallel programs because concurrency is much more easier for immutable values.

A **mutable** element can change its value. Scala supports both immutable and mutable elements and it is up to the program to use these features.

e.g.,

```
val pi = 3.14 // pi is immutable, is defined as a val (value)
var sal = 10,000 // salary is mutable, it is defined as a var (variable)
```

Imperative vs Functional

Imperative programming is the way we program in C, C++, Java and similar languages. It is heavily based on **mutable elements** (i.e., variables) and we need to specify every single step of an algorithm.

Scala supports imperative programming, but the real power of the language is the **functional** perspective which is based on **immutable elements**.

Counting Lines of a File

```
object LineCount {
    def main(args: Array[String]) {
      val inputFile = "leonardo.txt"
      val src = scala.io.Source.fromFile(inputFile)
      val counter = src.getLines().map(line => 1).sum
      println("Number of lines in file: "+counter)
  }
}
```

WordCount v1

```
import scala.io.Source
object WordCount {
  def main(args: Array[String]) {

    val lines = Source.fromFile("leonardo.txt").getLines.toArray
    val counts = new collection.mutable.HashMap[String, Int].withDefaultValue(0)
    lines.flatMap(line => line.split(" ")).foreach(word => counts(word) += 1)

    println(counts)

  }
}
```

WordCount v2

```
import scala.io.Source
object WordCount {
  def main(args: Array[String]) {

    val counts = Source.fromFile("leonardo.txt").
      getLines().
      flatMap(_.split("\\W+")).
      toList.
      groupBy((word: String) => word).
      mapValues(_.length)

    println(counts)
  }
}
```

Quicksort v1

```
var xs: Array[Double]
def swap(i: Int, j: Int) {
      val t = xs(i); xs(i) = xs(j); xs(j) = t
    }

def sort1(l: Int, r: Int) {
      val pivot = xs((l + r) / 2)
      var i = l
      var j = r
      while (i <= j) {
          while (xs(i) < pivot) i += 1
          while (xs(j) > pivot) j -= 1
          if (i <= j) {
            swap(i, j); i += 1; j -= 1
          }
      }
      if (l < j) sort1(l, j)
      if (j < r) sort1(i, r)
}
sort1(0, xs.length - 1)
```

Quicksort v2

```
object QuickSort {

  def quick(xs: Array[Int]): Array[Int] = {

    if (xs.length <= 1) xs
    else {
      val pivot = xs(xs.length / 2)
      Array.concat(quick(xs filter (_ < pivot)),
         xs filter (_ == pivot), quick(xs filter (_ > pivot)))
    }
  }
```

Array Example

```
object ArrayDemo {
  def mySquare(arr: Array[Int]): Array[Int] = {
    arr.map(elem => elem * elem)
  }
  def myCube(arr: Array[Int]): Array[Int] = {
    arr.map(elem => elem*elem*elem)
  }

  def main(args: Array[String]) {
    // fill the array with random numbers
    val vector = Array.fill(10){scala.util.Random.nextInt(9)}

    println(vector.mkString(","))
    println(mySquare(vector).mkString(","))
    println(myCube(vector).mkString(","))
  }
}
```

Traits

- Similar to **interfaces** in Java
- They may have implementations of methods
- But cannot contain state
- Can be multiply inherited from

Trait Example

```
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}

class Point(xc: Int, yc: Int) extends Similarity {
  var x: Int = xc
  var y: Int = yc
  def isSimilar(obj: Any) =
    obj.isInstanceOf[Point] &&
    obj.asInstanceOf[Point].x == x
}

object TraitsTest extends Application {
  val p1 = new Point(2, 3)
  val p2 = new Point(2, 4)
  val p3 = new Point(3, 3)
  println(p1.isNotSimilar(p2))
  println(p1.isNotSimilar(p3))
  println(p1.isNotSimilar(2))
}
```

Actors

A strong aspect of Scala is its ability to develop concurrent programs.

The language supports the Actor model (adopted from Erlang)

What is an actor?

Actors are normal objects that are created by instantiating subclasses of the **Actor class**.

Actor Example

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")
  helloActor ! "hello"
  helloActor ! "buenos dias"
}
```

Reference: Scala Cookbook

Resources

Recommended links for Scala programming

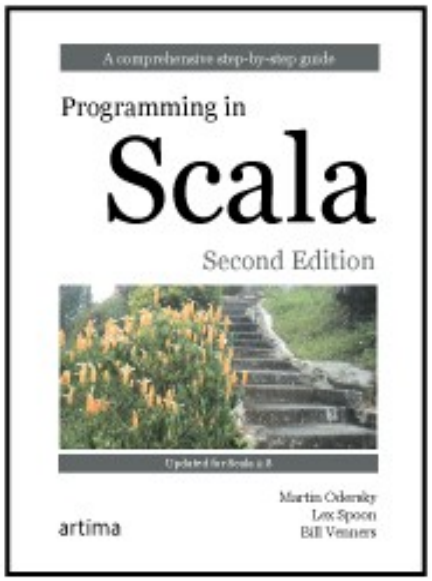
Official Scala Website

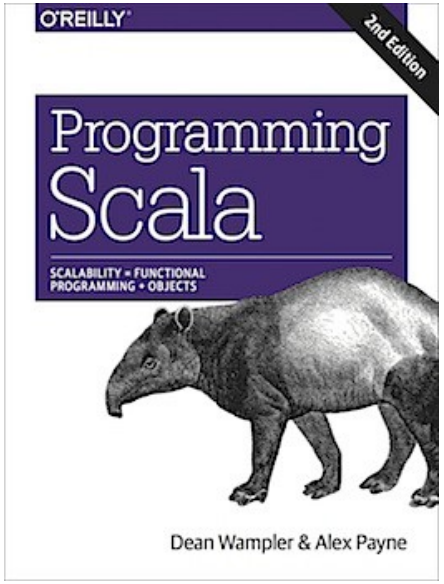
http://www.scala-lang.org

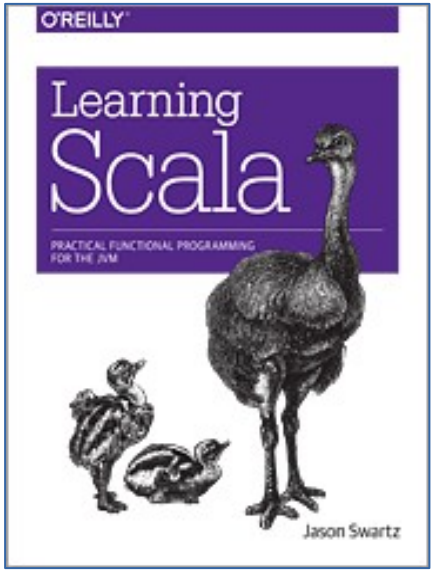
Scala School

https://twitter.github.io/scala_school

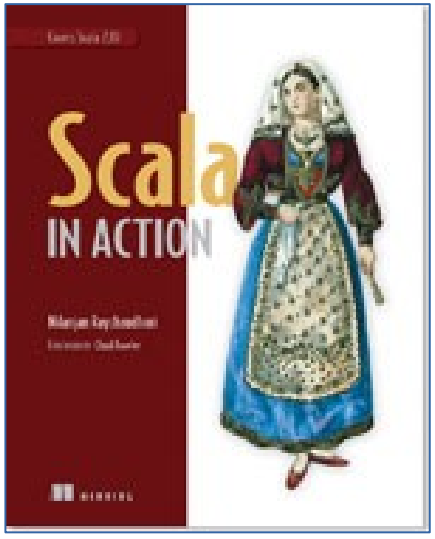
Resources

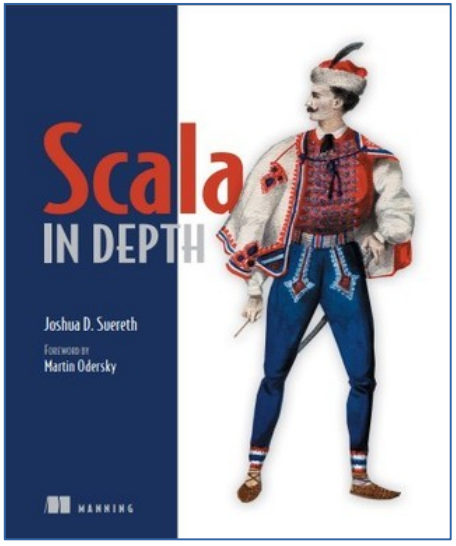

A comprehensive step-by-step guide
Programming in
Scala
Second Edition
artima
Martin Odersky
Lex Spoon
Bill Venners

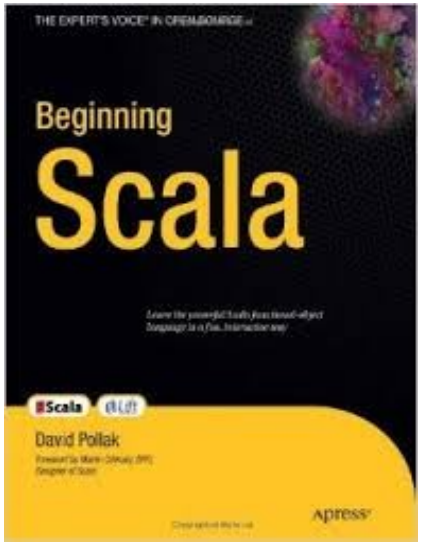

O'REILLY
2nd Edition
Programming
Scala
SCALABILITY = FUNCTIONAL
PROGRAMMING + OBJECTS
Dean Wampler & Alex Payne

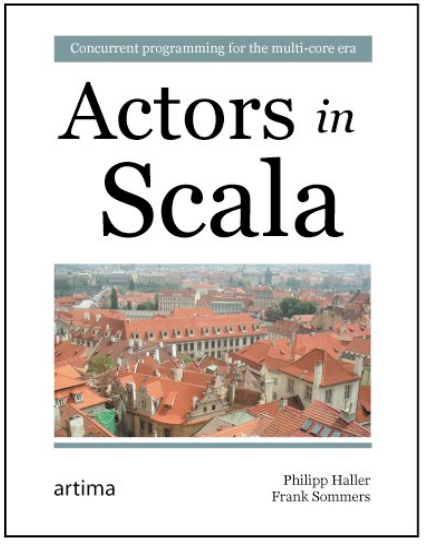

O'REILLY
Learning
Scala
PRACTICAL FUNCTIONAL PROGRAMMING
FOR THE JVM
Jason Swartz


SCALA
for the
Impatient
Cay Horstmann


Scala
IN ACTION


Scala
IN DEPTH
Joshua D. Suereth
Martin Odersky
MANNING


Beginning
Scala
David Pollak
Apress


Concurrent programming for the multi-core era
Actors in
Scala
artima
Philipp Haller
Frank Sommers

Thank You

Questions ?