A Crash Course in Scala

Amir H. Payberah
Swedish Institute of Computer Science

amir@sics.se
April 15, 2014
Scala: scalable language

A blend of object-oriented and functional programming.

Runs on the Java Virtual Machine.

Designed by Martin Odersky at EPFL.
Two metaphors for software development (Eric S. Raymond)
Cathedral vs. Bazaar

- **The cathedral**
  - A near-perfect building that takes a long time to build.
  - Once built, it stays unchanged for a long time.

- **The bazaar**
  - Adapted and extended each day by the people working in it.
  - Open-source software development.
Cathedral vs. Bazaar

- The cathedral
  - A near-perfect building that takes a long time to build.
  - Once built, it stays unchanged for a long time.

- The bazaar
  - Adapted and extended each day by the people working in it.
  - Open-source software development.

Scala is much more like a bazaar than a cathedral!
Functional Programming (FP)

- In a **restricted** sense: programming **without** mutable variables, assignments, loops, and other imperative control structures.

- In a **wider** sense: focusing on the functions.
Functional Programming (FP)

- In a restricted sense: programming without mutable variables, assignments, loops, and other imperative control structures.

- In a wider sense: focusing on the functions.

- Functions can be values that are produced, consumed, and composed.
In a **restricted** sense: a language that does **not** have **mutable** variables, assignments, or **imperative control** structures.

In a **wider** sense: it enables the construction of programs that **focus on functions**.
In a **restricted** sense: a language that does **not** have **mutable variables**, assignments, or **imperative control** structures.

In a **wider** sense: it enables the construction of programs that focus on **functions**.

**Functions** are **first-class** citizens:
- **Defined anywhere** (including inside other functions).
- **Passed as parameters** to functions and **returned as results**.
- **Operators** to compose functions.
In the **restricted** sense:

- Pure Lisp, XSLT, XPath, XQuery, Erlang

In the **wider** sense:

- Lisp, Scheme, Racket, Clojure, SML, Ocaml, Haskell (full language), Scala, Smalltalk, Ruby
Hello World!
object HelloWorld {
    def main(args: Array[String]) {
        println("Hello, world!")
    }
}
> scala
This is a Scala shell.
Type in expressions to have them evaluated.
Type :help for more information.

scala> object HelloWorld {
    | def main(args: Array[String]) {
    |     println("Hello, world!"")
    | }
    | }
defined module HelloWorld

scala> HelloWorld.main(null)
Hello, world!

scala> :q
>

Amir H. Payberah (SICS)
// Compile it!
> scalac HelloWorld.scala
> scalac -d classes HelloWorld.scala

// Execute it!
> scala HelloWorld
> scala -cp classes HelloWorld
```bash
#!/bin/bash
exec scala $0 $@

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

HelloWorld.main(null)

# Execute it!
> ./script.sh
```
- Scala basics
- Functions
- Collections
- Classes and objects
- SBT
Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT
Scala Variables

- **Values**: immutable
- **Variables**: mutable

```scala
var myVar: Int = 0
val myVal: Int = 1

// Scala figures out the type of variables based on the assigned values
var myVar = 0
val myVal = 1

// If the initial values are not assigned, it cannot figure out the type
var myVar: Int
val myVal: Int
```
Scala Data Types

- **Boolean**: true or false
- **Byte**: 8 bit signed value
- **Short**: 16 bit signed value
- **Char**: 16 bit unsigned Unicode character
- **Int**: 32 bit signed value
- **Long**: 64 bit signed value
- **Float**: 32 bit IEEE 754 single-precision float
- **Double**: 64 bit IEEE 754 double-precision float
- **String**: A sequence of characters

```scala
var myInt: Int
var myString: String
```
```scala
var x = 30;

if (x == 10) {
    println("Value of X is 10");
} else if (x == 20) {
    println("Value of X is 20");
} else {
    println("This is else statement");
}
```
var a = 10

// do-while
do {
    println("Value of a: " + a)
    a = a + 1
} while(a < 20)

// while loop execution
while(a < 20) {
    println("Value of a: " + a)
    a = a + 1
}
Loops (2/3)

```scala
var a = 0
var b = 0

for (a <- 1 to 3; b <- 1 until 3) {
    println("Value of a: ", a + ", b: " + b)
}

Value of a: 1, b: 1
Value of a: 1, b: 2
Value of a: 2, b: 1
Value of a: 2, b: 2
Value of a: 3, b: 1
Value of a: 3, b: 2
```
// loop with collections
val numList = List(1, 2, 3, 4, 5, 6)
for (a <- numList) {
  println("Value of a: " + a)
}

// for loop with multiple filters
for (a <- numList if a != 3; if a < 5) {
  println("Value of a: " + a)
}

// for loop with a yield
// store return values from a for loop in a variable
var retVal = for(a <- numList if a != 3; if a < 6) yield a
println(retVal)
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
  def main(args: Array[String]) {
    try {
      val f = new FileReader("input.txt")
    } catch {
      case ex: FileNotFoundException => { println("Missing file exception") }
      case ex: IOException => { println("IO Exception") }
    } finally {
      println("Exiting finally...")
    }
  }
}
def functionName([list of parameters]): [return type] = {
    function body
    return [expr]
}

def addInt(a: Int, b: Int): Int = {
    var sum: Int = 0
    sum = a + b
    sum
}

println("Returned Value: " + addInt(5, 7))
Functions - Default Parameter Values

```scala
def addInt(a: Int = 5, b: Int = 7): Int = {
  var sum: Int = 0
  sum = a + b
  return sum
}
println("Returned Value :" + addInt())
```
def printStrings(args: String*) = {
  var i : Int = 0;
  for (arg <- args) {
    println("Arg value[" + i + "] = " + arg )
    i = i + 1;
  }
}

printStrings("SICS", "Scala", "BigData")
def factorial(i: Int): Int = {
    def fact(i: Int, accumulator: Int): Int = {
        if (i <= 1)
            accumulator
        else
            fact(i - 1, i * accumulator)
    }
    fact(i, 1)
}

println(factorial(5))
Lightweight syntax for defining anonymous functions.

```scala
class Functions {
  val inc = (x: Int) => x + 1
  val x = inc(7) - 1

  val mul = (x: Int, y: Int) => x * y
  println(mul(3, 4))

  val userDir = () => { System.getProperty("user.dir") }
  println(userDir())
}
```
def **apply**(*f*: Int => String, *v*: Int) = *f*(*v*)

def **layout**(*A*(*x*: *A*)) = "[" + *x*.toString() + "]"

println(apply(layout, 10))
Call-by-Value: the value of the parameter is determined before it is passed to the function.

def time() = {
    println("Getting time in nano seconds")
    System.nanoTime
}

def delayed(t: Long) {
    println("In delayed method")
    println("Param: " + t)
}

delayed(time())

Getting time in nano seconds
In delayed method
Param: 2532847321861830
Call-by-Name: the value of the parameter is not determined until it is called within the function.

def time() = {
    println("Getting time in nano seconds")
    System.nanoTime
}

def delayed2(t: => Long) {
    println("In delayed method")
    println("Param: " + t)
}

delayed2(time())

In delayed method
Getting time in nano seconds
Param: 2532875587194574
If you do not pass in arguments for all of the parameters.

```scala
def adder(m: Int, n: Int, p: Int) = m + n + p

val add2 = adder(2, _: Int, _: Int)

add2(3, 5)
```
Transforms a function with **multiple arguments** into a **chain of functions**, each accepting a **single argument** and returning another function.

For example transforms \( f(x, y, z) // (\text{int, int, int}) \rightarrow \text{int} \) to \( g(x)(y)(z) // \text{int} \rightarrow (\text{int} \rightarrow (\text{int} \rightarrow \text{int})) \), in which \( g(x) \) returns another function, \( h(y) \) that takes an argument and returns \( k(z) \).

Used to **partially apply** a function to some value while leaving other values undecided,
def adder(m: Int)(n: Int)(p: Int) = m + n + p
adder: (m: Int)(n: Int)(p: Int)Int

// The above definition does not return a curried function yet
// (adder: (m: Int)(n: Int)(p: Int)Int)
// To obtain a curried version we still need to transform the method.
// into a function value.

val currAdder = adder _
currAdder: Int => Int => Int => Int = <function1>

val add2 = currAdder(2)

val add5 = add2(3)

add5(5)
Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT
Scala collections can be **mutable** and **immutable** collections.

- **Mutable** collections can be updated or extended in place.

- **Immutable** collections never change: additions, removals, or updates operators return a **new collection** and leave the old collection unchanged.
Collections

- Arrays
- Lists
- Sets
- Maps
- Tuples
- Option
Collections - Arrays

- A fixed-size sequential collection of elements of the same type
- Mutable

```scala
// Array definition
val t: Array[String] = new Array[String](3)
val t = new Array[String](3)

// Assign values or get access to individual elements
\t0 = "zero"; t(1) = "one"; t(2) = "two"

// There is one more way of defining an array
val t = Array("zero", "one", "two")
```
Collections - Lists

- A sequential collection of elements of the same type
- Immutable
- Lists represent a linked list

```scala
// List definition
val l1 = List(1, 2, 3)
val l1 = 1 :: 2 :: 3 :: Nil

// Adding an element to the head of a list
val l2 = 0 :: l1

// Adding an element to the tail of a list
val l3 = l1 :+: 4

// Concatenating lists
val t3 = List(4, 5)
val t4 = l1 ::: t3
```
Collections - Sets

- A sequential collection of elements of the same type
- Immutable and mutable
- No duplicates.

```scala
// Set definition
val s = Set(1, 2, 3)

// Add a new element to the set
val s2 = s + 0

// Remove an element from the set
val s3 = s2 - 2

// Test the membership
s.contains(2)
```
A collection of **key/value pairs**

**Immutable and mutable**

```
// Map definition
var m1: Map[Char, Int] = Map()
val m2 = Map(1 -> "Carbon", 2 -> "Hydrogen")

// Finding the element associated to a key in a map
m2(1)

// Adding an association in a map
val m3 = m2 + (3 -> "Oxygen")

// Returns an iterable containing each key (or values) in the map
m2.keys
m2.values
```
Collections - Tuples

- A **fixed** number of items of **different types** together

- **Immutable**

```scala
// Tuple definition
val t = (1, "hello", Console)
val t = new Tuple3(1, "hello", 20)

// Tuple getters
t._1
t._2
t._3
```
Sometimes you might or might not have a value.

Java typically returns the value null to indicate nothing found.
  • You may get a NullPointerException, if you don’t check it.

Scala has a null value in order to communicate with Java.
  • You should use it only for this purpose.

Everyplace else, you should use Option.
// the value of an Option[type] variable is either Some or None.
scala> var s = Some("abc")
scala> var t: Option[String] = None
scala> val numbers = Map(1 -> "one", 2 -> "two")
numbers: scala.collection.immutable.Map[Int, String] = Map((1, one), (2, two))

scala> numbers.get(2)
res0: Option[String] = Some(two)

scala> numbers.get(3)
res1: Option[String] = None

// Check if an Option value is defined (isDefined and isEmpty).
scala> val result = numbers.get(3).isDefined
result: Boolean = false

// Extract the value of an Option.
scala> val result = numbers.get(3).getOrElse("zero")
result: String = zero
Functional Combinators

- map
- foreach
- filter
- zip
- partition
- find
- drop and dropWhile
- foldRight and foldLeft
- flatten
- flatMap
Evaluates a function over each element in the list, returning a list with the same number of elements.

```scala
scala> val numbers = List(1, 2, 3, 4)
nnumbers: List[Int] = List(1, 2, 3, 4)

scala> numbers.map((i: Int) => i * 2)
res0: List[Int] = List(2, 4, 6, 8)

scala> def timesTwo(i: Int): Int = i * 2
timesTwo: (i: Int)Int

scala> numbers.map(timesTwo _)
or
scala> numbers.map(timesTwo)
res1: List[Int] = List(2, 4, 6, 8)
```
It is like map but returns nothing

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> val doubled = numbers.foreach((i: Int) => i * 2)
doubled: Unit = ()

scala> numbers.foreach(print)
1234
```
Removes any elements where the function you pass in evaluates to false

```scala
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> numbers.filter((i: Int) => i % 2 == 0)
res0: List[Int] = List(2, 4)

scala> def isEven(i: Int): Boolean = i % 2 == 0
isEven: (i: Int)Boolean

scala> numbers.filter(isEven)
res2: List[Int] = List(2, 4)
```
Aggregates the contents of two lists into a single list of pairs

```scala
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> val chars = List("a", "b", "c")
chars: List[String] = List(a, b, c)

scala> numbers.zip(chars)
res0: List[(Int, String)] = List((1, a), (2, b), (3, c))
```
Functional Combinators - partition

- Splits a list based on where it falls with respect to a predicate function

```scala
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
nnumbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> numbers.partition(_ % 2 == 0)
res0: (List[Int], List[Int]) = (List(2, 4, 6, 8, 10), List(1, 3, 5, 7, 9))
```
Functional Combinators - find

- Returns the first element of a collection that matches a predicate function

```scala
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
native::numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> numbers.find(i => i > 5)
res0: Option[Int] = Some(6)
```
Functional Combinators - drop and dropWhile

- **drop** drops the first i elements
- **dropWhile** removes the first elements that match a predicate function

```scala
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> numbers.drop(5)
res0: List[Int] = List(6, 7, 8, 9, 10)

scala> numbers.dropWhile(_ % 3 != 0)
res1: List[Int] = List(3, 4, 5, 6, 7, 8, 9, 10)
```
Functional Combinators - foldLeft

- It goes through the whole List, from head to tail, and passes each value to \( f \).

- For the first list item, that first parameter, \( z \), is used as the first parameter to \( f \).

- For the second list item, the result of the first call to \( f \) is used as the \( B \) type parameter.

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B

scala> val numbers = List(1, 2, 3, 4, 5)
scala> numbers.foldLeft(0) { (m, n) => println("m: "+m+" n: "+n); m+n }
m: 0 n: 1
m: 1 n: 2
m: 3 n: 3
m: 6 n: 4
m: 10 n: 5
res0: Int = 15
```
It is the same as foldLeft except it runs in the opposite direction

```
def foldRight[B](z: B)(f: (A, B) => B): B

scala> val numbers = List(1, 2, 3, 4, 5)
scala> numbers.foldRight(0) { (m, n) => println("m: " + m + " n: " + n);
  m + n }
  m: 5 n: 0
  m: 4 n: 5
  m: 3 n: 9
  m: 2 n: 12
  m: 1 n: 14
res52: Int = 15
```
It collapses one level of nested structure

scala> List(List(1, 2), List(3, 4)).flatten
res0: List[Int] = List(1, 2, 3, 4)
It takes a function that works on the nested lists and then concatenates the results back together.

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

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

// Think of it as short-hand for mapping and then flattening:
scala> nestedNumbers.map((x: List[Int]) => x.map(_ * 2)).flatten
res1: List[Int] = List(2, 4, 6, 8)
```
Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT
Scala is a pure object-oriented language.

Everything is an object, including numbers.

Functions are also objects, so it is possible to pass functions as arguments, to store them in variables, and to return them from other functions.
class Calculator {
  val brand: String = "HP"
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
class Calculator(brand: String) {
  // A constructor.
  val color: String = if (brand == "TI") {
    "blue"
  } else if (brand == "HP") {
    "black"
  } else {
    "white"
  }

  // An instance method.
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator("HP")
println(calc.color)
Scala allows the inheritance from just one class only.

class SciCalculator(brand: String) extends Calculator(brand) {
    def log(m: Double, base: Double) = math.log(m) / math.log(base)
}

class MoreSciCalculator(brand: String) extends SciCalculator(brand) {
    def log(m: Int): Double = log(m, math.exp(1))
}
A singleton is a class that can have only one instance.

class Point(val xc: Int, val yc: Int) {
  var x: Int = xc
  var y: Int = yc
}

object Test {
  def main(args: Array[String]) {
    val point = new Point(10, 20)
    printPoint

    def printPoint {
      println("Point x location : " + point.x);
      println("Point y location : " + point.y);
    }
  }
}

Test.main(null)
abstract class Shape {
   // subclass should define this
   def getArea(): Int
}

class Circle(r: Int) extends Shape {
   def getArea(): Int = { r * r * 3 }
}

val s = new Shape // error: class Shape is abstract
val c = new Circle(2)
c.getArea
A class can mix in any number of traits.

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

trait Shiny {
  val shineRefraction: Int
}

class BMW extends Car with Shiny {
  val brand = "BMW"
  val shineRefraction = 12
}
```
trait Cache[K, V] {
  def get(key: K): V
  def put(key: K, value: V)
  def delete(key: K)
}

def remove[K](key: K)
Case Classes and Pattern Matching

- **Case classes** are used to store and match on the contents of a class.
- They are designed to be used with **pattern matching**.
- You can construct them **without using new**.

```scala
scala> case class Calculator(brand: String, model: String)
scala> val hp20b = Calculator("hp", "20B")

def calcType(calc: Calculator) = calc match {
  case Calculator("hp", "20B") => "financial"
  case Calculator("hp", "48G") => "scientific"
  case Calculator("hp", "30B") => "business"
  case _ => "Calculator of unknown type"
}

scala> calcType(hp20b)
```
Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT
Simple Build Tool (SBT)

- An open source build tool for Scala and Java projects.
- Similar to Java’s Maven or Ant.
- It is written in Scala.
$ mkdir hello
$ cd hello
$ cp <path>/HelloWorld.scala .
$ sbt
...
> run
Running SBT

- **Interactive** mode

  ```
  $ sbt
  > compile
  > run
  ```

- **Batch** mode

  ```
  $ sbt clean run
  ```

- **Continuous build** and **test**: automatically recompile or run tests whenever you save a source file.

  ```
  $ sbt
  > ~ compile
  ```
Common Commands

- **clean**: deletes all generated files (in target).
- **compile**: compiles the main sources (in src/main/scala).
- **test**: compiles and runs all tests.
- **console**: starts the Scala interpreter.
- **run <argument>***: run the main class.
- **package**: creates a jar file containing the files in src/main/resources and the classes compiled from src/main/scala.
- **help <command>**: displays detailed help for the specified command.
- **reload**: reloads the build definition (build.sbt, project/*.scala, project/*.sbt files).
Create a Simple Project

- Create `project` directory.
- Create `src/main/scala` directory.
- Create `build.sbt` in the project root.
A list of Scala expressions, separated by blank lines.

Located in the project’s base directory.

$ cat build.sbt
name := "hello"
version := "1.0"
scalaVersion := "2.10.3"
Add Dependencies

- Add in `build.sbt`.

- Module ID format:
  
  "groupID" %% "artifact" % "version" % "configuration"

```scala
libraryDependencies += "org.apache.spark" %% "spark-core" % "0.9.0-incubating"

// multiple dependencies
libraryDependencies ++= Seq(  
  "org.apache.spark" %% "spark-core" % "0.9.0-incubating",  
  "org.apache.spark" %% "spark-streaming" % "0.9.0-incubating"
)
```

- sbt uses the standard Maven2 repository by default, but you can add more `resolvers`.

```scala
resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```
Questions?