A Crash Course on Scala

Amir H. Payberah
payberah@kth.se

2022-09-08

The Course Web Page

https://id2221kth.github.io

The Questions-Answers Page

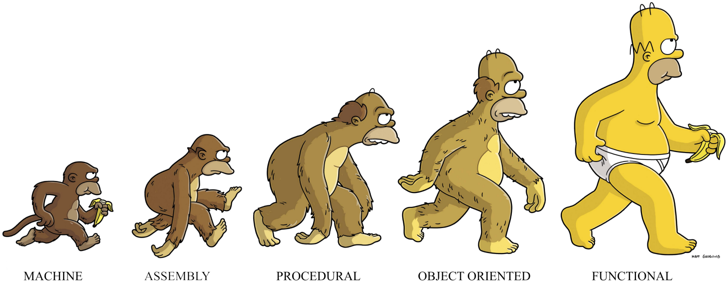
https://tinyurl.com/bdenpwc5

Introduction

- Scala: scalable language
- A blend of object-oriented and functional programming.
- Runs on the Java Virtual Machine.
- Designed by Martin Odersky at EPFL.

Functional Programming Languages

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



[https://medium.com/@cscalfani/so-you-want-to-be-a-functional-programmer-part-1-1f15e387e536]

HELLO
WORLD!

The “Hello, world!” Program

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

Compile and Execute It!

```
// Compile it!
> scalac HelloWorld.scala

// Execute it!
> scala HelloWorld
```

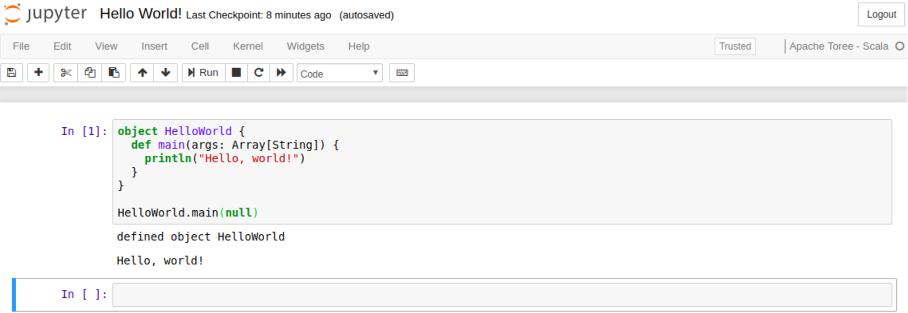
- It is always better to separate sources and build products.

```
// Compile it!
> scalac -d classes HelloWorld.scala

// Execute it!
> scala -cp classes HelloWorld
```

Run in Jupyter-Notebook

- Apache toree



In [1]:
```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}

HelloWorld.main(null)
```

```
defined object HelloWorld

Hello, world!
```

In [ ]:

Outline

- Scala basics
- FunctionsFunctions
- CollectionsCollections
- Classes and objectsClasses and objects
- SBTSBT

Scala Variables

- Values: immutable
- Variables: mutable
- Always use immutable values by default, unless you know for certain they need to be 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
```

If ... Else

```
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");
}
```

Loops (1/3)

```scala
var a = 10

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

// while loop execution
while(a < 20) {
  println(s"Value of a: $a")
  a = a + 1
}
```

Loops (2/3)

```
var a = 0
var b = 0

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

/* output
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
*/
```

Loops (3/3)

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

```
// for loop with multiple filters
for (a <- numList if a != 3; if a < 5) {
  println(s"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)
```

WESTCLOX

Hands-on Exercises

- Define an `Int` mutable variable, increment and print it out.

```
var a: Int = 5
a = a + 1
println(a)
```

- Define two immutable variables `Double` and `String`, and print them out.

```
val a: Double = 6.2
val b: String = "Hi!"
println(a, b)
```

- Print out even numbers between 1 and 10.

```
for (a <- 1 to 10; if a % 2 == 0)
    println(s"Value of a: $a")
```

Outline

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

Functions - Definition

```
// def [function name]([list of parameters]): [return type] = [expr]
// the expression may be a {}-block
```

```
def addInt(a: Int, b: Int): Int = a + b

println("Returned Value: " + addInt(5, 7))
// Returned Value: 12
```

- You can also specify default values for all or some parameters.

```
def addInt(a: Int = 5, b: Int = 7): Int = a + b

// and then invoke with named parameters
println("Returned Value:" + addInt(a = 10))
// Returned Value: 17
```

Functions - Variable Arguments

```scala
def printStrings(args: String*) = {
  var i : Int = 0;
  for (arg <- args) {
    println(s"Arg value[$i] = $arg")
    i = i + 1;
  }
}

printStrings("SICS", "Scala", "BigData")
```

Functions - Nested Functions

```
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))
```

Functions - Anonymous Functions

- Lightweight syntax for defining anonymous functions.

```
var inc = (x: Int) => x + 1
var x = inc(7) - 1
```

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

Functions - Higher-Order Functions

```scala
def apply(f: Int => String, v: Int) = f(v)

def layout[A](x: A) = s"[$x]"

println(apply(layout, 10))
// [10]
```

Functions - Call-by-Value

- Call-by-Value: the value of the parameter is determined before it is passed to the function.

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

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

delayed(time())

/* output
Getting time in nano seconds
In delayed method
Param: 2532847321861830
*/
```

Functions - Call-by-Name

- 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(s"Param: $t")
}

delayed2(time())

/* output
In delayed method
Getting time in nano seconds
Param: 2532875587194574
*/
```

WESTCLOX

Hands-on Exercises (1/2)

- Write a function to sum all integers between two numbers a and b.

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

- Write a function to sum the squares of all integers between two numbers a and b.

```
def square(x: Int): Int = x * x

def sumSquares(a: Int, b: Int): Int =
    if (a > b) 0 else square(a) + sumSquares(a + 1, b)
```

Hands-on Exercises (2/2)

- Assume the following methods

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f, a + 1, b)

def id(x: Int): Int = x

def square(x: Int): Int = x * x
```

- Reimplement the previous methods using higher-order functions.

```
def sumInts(a: Int, b: Int): Int = sum(id, a, b)

def sumSquares(a: Int, b: Int): Int = sum(square, a, b)
```

Outline

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

Collections

- 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

Collections - Arrays

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

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

```
// Assign values or get access to individual elements
t(0) = "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
```

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

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

```scala
// 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.

```
// 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)
```

Collections - Maps

- 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
```

Colletion Methods

- map
- foreach
- filter
- zip
- partition
- find
- drop and dropWhile
- foldRight and foldLeft
- flatten
- flatMap

Functional Combinators - map

- Evaluates a function over each element in the list, returning a list with the same number of elements.

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

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

numbers.map(timesTwo _)
// or
numbers.map(timesTwo)
// res1: List[Int] = List(2, 4, 6, 8)
```

Functional Combinators - foreach

- It is like map, but returns nothing.

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

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

numbers.foreach(print)
// 1234
```

Functional Combinators - filter

- Removes any elements where the function you pass in evaluates to false.

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

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

```
def isEven(i: Int): Boolean = i % 2 == 0
// isEven: (i: Int)Boolean

numbers.filter(isEven)
// res2: List[Int] = List(2, 4)
```

Functional Combinators - zip

- Aggregates the contents of two lists into a single list of pairs.

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

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

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.

```
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)

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
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)

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.

```
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)

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

numbers.dropWhile(_ % 3 != 0)
// res1: List[Int] = List(3, 4, 5, 6, 7, 8, 9, 10)
```

Functional Combinators - foldLeft

- Takes an associative binary operator function and uses it to collapse elements from the collection.
- It goes through the whole List, from head (left) to tail (right).

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

numbers.foldLeft(0) { (acc, i) =>
  println("i: " + i + " acc: " + acc)
  i + acc
}

/* output
i: 1 acc: 0
i: 2 acc: 1
i: 3 acc: 3
i: 4 acc: 6
i: 5 acc: 10
15 */
```

Functional Combinators - foldRight

- It is the same as `foldLeft` except it runs in the opposite direction.

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

numbers.foldRight(0) { (i, acc) =>
  println("i: " + i + " acc: " + acc)
  i + acc
}

/* output
i: 5 acc: 0
i: 4 acc: 5
i: 3 acc: 9
i: 2 acc: 12
i: 1 acc: 14
15 */
```

Functional Combinators - flatten

- It collapses one level of nested structure.

```
List(List(1, 2), List(3, 4)).flatten
// res0: List[Int] = List(1, 2, 3, 4)

List(Some(1), None, Some(3)).flatten
// res0: List[Int] = List(1, 3)
```

Functional Combinators - flatMap

- It takes a function that works on the nested lists and then concatenates the results back together.

```
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)
```

WESTCLOX

Hands-on Exercises (1/3)

- Declare a list of integers as a variable called `myNumbers`.

```scala
val myNumbers = List(1, 2, 5, 4, 7, 3)
```

- Declare a function, `pow`, that computes the second power of an int.

```scala
def pow(a: Int): Int = a * a
```

Hands-on Exercises (2/3)

- Apply the function to myNumbers using the map function.

```
myNumbers.map(x => pow(x))
// or
myNumbers.map(pow(_))
// or
myNumbers.map(pow)
```

- Write the pow function inline in a map call, using closure notation.

```
myNumbers.map(x => x * x)
```

- Iterate through myNumbers and print out its items.

```
myNumbers.foreach(println)
```

Hands-on Exercises (3/3)

- Declare a list of pair of string and integers as a variable called `myList`.

```
val myList = List[(String, Int)](("a", 1), ("b", 2), ("c", 3))
```

- Write an inline function to increment the integer values of the list `myList`.

```
val x = v.map { case (name, age) =>  age + 1 }
// or
val x = v.map(i => i._2 + 1)
// or
val x = v.map(_._2 + 1)
```

Common Other Types

- Tuples
- Option
- Either

Common Data Types - Tuples

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

```
// Tuple definition
val t2 = (1 -> "hello") // special pair constructor
val t3 = (1, "hello", Console)
val t3 = new Tuple3(1, "hello", 20)

// Tuple getters
t3._1
t3._2
t3._3
```

Common Data Types - Option (1/2)

- 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.

Common Data Types - Option (2/2)

```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
```

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

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

Common Data Types - Either

- Sometimes you might definitely have a value, but it can be one of two different types.
- Scala provides the `Either` type for these cases.

```scala
def getNum(s: String): Either[Int, String] = try {
  Left(s.toInt)
} catch {
  case _ => Right(s)
}

getNum("5")
// Left(5)
```

Outline

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

Everything is an Object

- Scala is a pure object-oriented language.

- Everything is an object, including numbers.

```
1 + 2 * 3 / x
(1).+(((2).*(3))./(x))
```

- 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.

Classes and Objects

```scala
// constructor parameters can be declared as fields and can have default values
class Calculator(val brand = "HP") {
  // an instance method
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
// HP
```

Inheritance and Overloading Methods

- 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))
}
```

Singleton Objects

- A singleton is a class that can have only one instance.

```
class Point(val x: Int, val y: Int) {
  def printPoint {
    println(s"Point x location: $x");
    println(s"Point y location: $y");
  }
}

object SpecialPoint extends Point(10, 20)

SpecialPoint.printPoint
/* output
Point x location: 10
Point y location: 20
*/
```

Abstract Classes

```
abstract class Shape {
  // subclass should define this
  def getArea(): Int
}

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

val s = new Shape // error: class Shape is abstract
val c = new Circle(2)
c.getArea
// 12
```

Traits

- A class can mix in any number of traits.

```
trait Car {
  val brand: String
}

trait Shiny {
  val shineRefraction: Int
}

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

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
case class Calculator(brand: String, model: String)
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"
}

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.

SBT - Hello World!

```
$ 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.

build.sbt

- 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 := "2.12.8"
```

Add Dependencies

- Add in `build.sbt`.
- Module ID format:
  `"groupID" %% "artifact" % "version" % "configuration"`

```
libraryDependencies += "org.apache.spark" %% "spark-sql" % "3.3.0"

// multiple dependencies
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-sql" % "3.3.0",
  "org.apache.spark" % "spark-streaming_2.12" % "3.3.0",
  "org.apache.spark" % "spark-sql-kafka-0-10_2.12" % "3.3.0",
  "org.apache.spark" % "spark-streaming-kafka-0-10_2.12" % "3.3.0"
)
```

Summary

Summary

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

References

- M. Odersky, Scala by example, 2011.

Questions?