Programming in Scala

based on the book
“Programming in Scala Third Edition”

Agenda

- Chapter I - A Scalable Language
  - Introduction to the Scala Programming Language
- Chapter II - First Steps in Scala
  - How to use the Scala Interpreter
  - How to write a Scala Script
  - Basic Code examples
- Chapter III - Next Steps in Scala
  - More complex examples
  - Using Collections

Chapter I - A scalable language

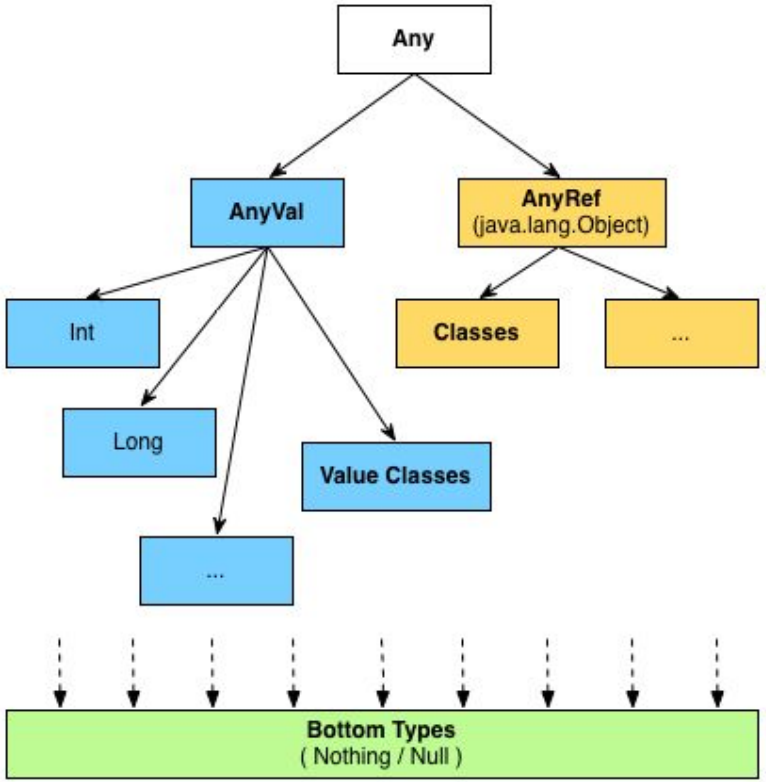
Scala = scalable langue

“The language is so named because it was designed to grow with the demands of its user”

- Hybrid paradigm language: Scala = OO + FP
- Runs on JVM
- Statically typed
- Cathedral vs Bazaar -> Scala is more like the latter

Growing new types

- Programmers can use JAVA built-in types
- Scala “type like” classes are in the Scala standard library - http://www.scala-lang.org/api/current/
- There are no primitive types
- Scala encourages the dev’s to define libraries *feel* like native language support and doesn’t try to give them everything



source: http://ktoso.github.io/scala-types-of-types/

Growing new control constructs

```
recipient ! message
```

- ! is not a built-in operator, it comes from the Akka library
- Actually this is a function defined in the recipient class
- In Scala it is easy to write a code that looks like and feels like native language support

Scala is Object Oriented

- Smalltalk was the first purely OO language
- Many languages follow OO principles, but mostly are not pure OO (Java!)
- Scala is fully OO like Smalltalk!

`2 + 3` vs. `2.+(3)`

- Traits (~Java interfaces) can contain implementations, fields...
- Objects are constructed with Mixin object composition mechanism (~Java multiple inheritance)

Scala is Functional

- FP is one of the oldest paradigm in computer science (1930s Church - Lambda calculus)
- Scala is also full-blown functional language
- Main ideas behind it:
  - Functions are first classes
  - Immutability (avoiding side effects) - referential transparency!!
- `val` vs `var`
- Scala libraries define many immutable data types
- Scala allows to use imperative style programming with mutability BUT it encourages you to use functional style programming!

Why Scala?

- Compatible
  - Scala compiles to JVM bytecode. Re-uses and enriches JAVA types and other elements
  - It can use and call JAVA methods, types, etc..., but It can be also invoked from JAVA
  - Scala.js - https://www.scala-js.org
- Concise
  - It can be both positive and negative :)
  - The avg. code length is 50% compared to JAVA code what resolves the same problem
- High-level
  - uses many higher level abstractions like streams, functions, predicates, etc...
- Statically typed
  - Compile time, strongly typed, support generics
  - Type inference!
  - Pattern matching

Chapter II - First steps in Scala

REPL - The Scala Interpreter

- This is a command line, interactive code interpreter
- Line by line interprets the code
- Keeps the context so you can define variable what you can re-use later
- Auto assigns expression to variables, like:

```
scala> 1 + 2
res0: Int = 3
```

- You can type multiline-expressions, just keep typing after hitting the `Enter` key, or `double Enter` to exit multiline mode

- Online REPL: http://www.scala-repl.org/code/welcome

Function definition

```
def functionName(param: Int) : Int = {
    // body
    return 7
}
```

Examples:

```
def max(x: Int, y: Int) = if (x > y) x else y

def greet() = println("hello")
```

- You can leave the curly braces if the body is just one single expression
- You can leave `return` keyword if the last expression has a value
- You can leave the return type if the function is NOT recursive

Scala Scripts

- Script is just a sequence of Scala statements in a file
- You can run it like:
  - A shell script on Mac
  - As a .bat file on Windows
  - from the terminal: `scala fileName.scala`
- You can pass command line arguments (0 based indexing!)

Loops

- `while` - this is not the best type of loop you can use in Scala, try to use for instead!

```
while (condition) { /*body*/ }
```

- `foreach` and `for`

```
args.forEach(println)
```

```
for (arg <- args)
    println(arg)
```

Chapter III - Next steps in Scala

Arrays

- you can instantiate objects, or class instances, using `new`
- you can parameterize it with values and types (configure them)
- **Important!** Get the nth element of an array is `array(n)` and not ~~`array[n]`~~

```
val big = new java.math.BigInteger("12345")

val greetStrings = new Array[String](3)

val greetStrings: Array[String] = new Array[String](3)
```

Factory method for creating arrays

```
val numNames = Array("zero", "one", "two")
```

is transformed to

```
val numNames2 = Array.apply("zero", "one", "two")
```

Array value assignments

When an assignment is made to a variable to which parentheses and one or more arguments have been applied, the compiler will transform that into an invocation of an update method that takes the arguments in parentheses as well as the object to the right of the equals sign

```
greetStrings(0) = "Hello"
```

is transformed to

```
greetStrings.update(0, "Hello")
```

Calling single parameter methods

- if a method takes only one parameter, you can call it without a dot or parentheses
- Scala doesn't technically have operator overloading, because it doesn't actually have operators in the traditional sense. **+, -, *, ÷** are all normal functions

`for (i <- 0 to 2)` vs. `for (i <- (0).to(2)`

`1 + 2` vs. `(1).+(2)`

Lists

- Lists are immutable unlike arrays, you can’t change a list element later
- Creating a list, you can use factory method:

```
val oneTwoThree = List(1, 2, 3)
```

- or, you can use `::` (“cons”) operator

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
```

- Concatenate lists

```
listA ::: listB
```

Tuples

- Immutable container object, but unlike Lists it can **contain different types** of elements

```
val pair = (99, "Luftballons")

val otherPair = new Tuple(99, "Luftballons")

println(pair._1)

println(pair._2)
```

- accessing an element you should use `_N` notation, where `N` uses 1 based indexing unlike Array/List!

Set

- Have both mutable and immutable versions, but with the same name! Only the package is different
  - `scala.collection.Set ->`
    - `scala.collection.immutable.Set;`
    - `scala.collection.mutable.Set`
- Working with sets

```
var jetSet = Set("Boeing", "Airbus")

jetSet += "Lear"

println(jetSet.contains("Cessna"))
```

- There are many set implementations other than the default, like HashSet

Map

- Like `Map`, it also has mutable and immutable version, immutable is the default version!

```
val treasureMap = mutable.Map[Int, String]()

treasureMap += (1 -> "Go to island.")

treasureMap += (2 -> "Find big X on ground.")

treasureMap += (3 -> "Dig.")

println(treasureMap(2))
```

Learn to recognize the functional style

- Scala is a hybrid language, it doesn't force you to use FP
- So you can program in Scala in an imperative OO style, but it is strongly advised to use the functional style!
- (In theory) coding in FP style results a
  - Clearer,
  - More concise,
  - Less error-prone,
  - And more understandable (???) code.
- So try to avoid
  - Using `var`
  - Defining functions with no return type (do not use `Unit` as a return type)
  - Having side-effects in your functions

Instead of having this

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

Try to have this

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

Or even more this (no side effects)

```
def formatArgs(args: Array[String]) = args.mkString("\n")


println(formatArgs(args))
```

Once again... Balance!

“Prefer vals, immutable objects, and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.”

Homework

Create the following Scala script:

- MUST: processing an HTML page from the internet and printing the 10 most frequently occuring words from it.
- OPTIONAL: filter out the HTML keywords from the result

HAVE FUN! :)

**you can find many useful tips how to do it at the end of the 3rd chapter ;)*