Functional Programming and the Scala Language

Lectures 1-2

Eugene Zouev
Innopolis University
Spring Semester 2018

Who Is This Guy? ☺

- **Eugene Zouev**
- Have been working at Moscow Univ., Swiss Fed Inst of Technology (ETH Zürich), EPFL (Lausanne); PhD (1999, Moscow Univ).
- Prof. interests: compiler construction, language design, PL semantics.
- The author of the 1st Russian C++ compiler (Interstron Ltd., 1999-2000).
- Zonnon language implementation for .NET & Visual Studio (ETHZ, 2005).
- Swift prototype compiler for Tizen & Android (Samsung R&D Center, 2015)
- «Редкая профессия», ДМК Пресс, Москва 2014.

The Structure of the Course

- **The schedule**: one lecture + one seminar each Monday.
  Lecture: 17.20 – 18.50.
  Lab: 19.00 – 20.30.

Organizational
(***tentative; details will be given later today***)

- **Questions** for home thinking; **a few projects** of diff. complexity (each in a small team); presentations.

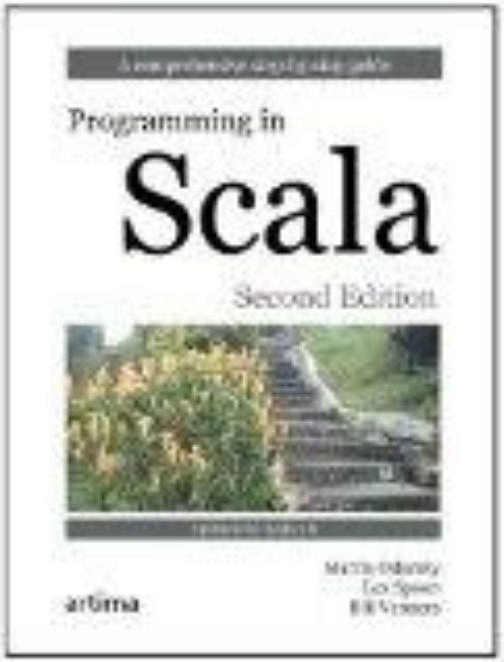
- **No** mid-term examination; **no** final examination.
  The final grade will be composed based on **your contribution** to projects and on the project **results**.

The Informal Remark

Pre-requisites & expectations

- I assume **you know** (at least basics of) **OOP** (something like Java/C#/C++/Eiffel). If you don't please help yourself studying it by your own ☺.
- I hope **you know a bit of** (basics of) **Java**. If you don't please read carefully any introductory Java textbook.

References (1)



Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition, 2011
by Martin Odersky, Lex Spoon, Bill Venners.
-- 2nd Edition, Artima Inc,
ISBN-10: 0981531644,
ISBN-13: 978-0981531649.

References (2)

References (3)

Scala for the Impatient
Cay S. Hostmann
Addison-Westley
ISBN 978-0-321-77409-5
2012

SCALA для нетерпеливых
Кей С.Хостманн,
Издательство ДМК-Пресс
ISBN 978-5-94074-920-2,
978-0-321-77409-5
2013

Initial Remark

Why Scala but not Lisp, Clojure, Haskell etc.

Smalltalk
the "pure" OOP language

C++
"real" OOP: engineered for practical use

Haskell
ML, OCaml, F#, Lisp etc.: "pure" FP languages

Scala
"Real" FP+OOP for easy & efficient use

Why Learn & Use Functional?

Functional is the Trend!

Almost every modern programming language has at least some "functional" features.

- **C++**: lambda expressions, function types, functions without side effects, type inference
- **C#**: function types ("delegates"), function literals, type inference
- **Java 8**: lambda expressions
- **Swift**: functions as values; closures; local functions
- **Rust**: functions as variables, as arguments, as return values; anonymous functions

Functional Programming

Two cornerstones of functional approach:

- Immutable objects
- Functions as "first-class entities"

Immutable Objects (1)

Data are immutable:
The operations of a program should **map** input values to output values rather than **change** data in place.

Or: Methods should not have any side effects. They should communicate with their environment only by taking arguments and returning results… For any given input the method call could be replaced by its result without affecting the program's semantics.

Immutable Objects (2)

Advantages:

- Programs are easy-for-testing.
- Programs are verifiable.
- Programs can be parallelized.

Disadvantages:

- Sometimes leads to intensive memory consumption.

Functions as First-Class Objects (1)

Functions are **values** – just as integers, arrays etc.

- Function is an abstraction of an **operation**.
- Define functions anywhere (just as other variables); **nested functions** are allowed.
- "Constant" (or unnamed) functions are allowed (just as integer constants): **literal functions**.
- New **control structures** can be defined as functions.

Functions as First-Class Objects (2)

Remarks:

- Existing control structures are actually "wrappers" for functions
- Not only functions are values – almost **any language construct** has (produces) a value.
  - For example, `if` and `for` constructs in fact produce values.

Scala: The Mix of OOP, GP & FP

- Strong typing with type inference.
- Type parametrization with type variance specs.
- Unified type system
  No separation between "embedded" and "user-defined" types: **all types are classes**.
- Classes with single inheritance, polymorphism, virtual functions & virtual members. Interfaces & traits (support for mixin technique).
- Pattern matching mechanism. Concurrency support using actors.
- Immutable variables and functions as first-class values.
- Deep interoperability with Java.

Scala Slogan

Object-oriented meets functional

http://www.scala-lang.org/

Scala: Conciseness (1)

- Conciseness: Scala programs ~2 times more compact than Java

```
class Point
{
  private int x, y;
  public Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}
```

C#, Java

⇒

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

Scala

"Hidden" public constructor with two parameters is assumed

```
val x: HashMap[Int,String] = new HashMap[Int,String]()
```

⇓

```
val x = new HashMap[Int,String]()
```

Type inference

Scala: Conciseness (2)

- Conciseness: one more example

```
boolean nameHasUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
  if (Character.isUpperCase(name.charAt(i)))
  {
    nameHasUpperCase = true;
    break;
  }
}
```

Java

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Scala

Scala: Conciseness (3)

M. Odersky:

Concise is nice!

☺ (However, there is some "dark size" of the feature…)

Scala & Standard Library

- Scala is very library-oriented: there is a great number of (standard) libraries with collections (containers) and control structures (e.g., **actors**).

  Many typical programming patterns are represented as library features.

> `for`-expression can be represented as a library function `foreach` which is defined for all Scala collections: arrays, lists, vectors etc.

Scala: Variables & Values (1)

var starts an object definition

Variable name

Variable type

Optional initializer

```scala
var x: Int = 777
```

Mutable variable

```scala
var x = 777
```

Short form of the definition; the type of x is deduced from the type of the initializer

This is why var keyword is necessary

Scala: Variables & Values (2)

`val` starts an immutable object definition

```
val x: Int = 777
```

Immutable variable ("value")

```
val x = 777
```

Short form of the definition; the type of `x` is deduced from the type of the initializer

This is why `var/val` keywords are necessary

Scala: Imperative vs Functional (1)

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

Conventional while-loop with **var** counter

```scala
def printArgs(args: Array[String]): Unit = {
  for (arg <- args) {
    println(arg)
  }
}
```

For-loop with library-based iteration over array elements; no **vars**

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

Functional style manipulation on collection

Scala: Imperative vs Functional (2)

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

Functional style manipulation on collection

Is it ***true*** functional style?
- No: it contains side effect (`println`)

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

```scala
println(formatArgs(args))
```

Side effect is minimized

Scala: Imperative vs Functional (3)

While loops, `var`s and recursion

```
def gcd(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

While loops are available but not recommended: they are often used together with `var`s...

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

As a better options, consider recursion:

Scala: Function Definition (1)

def starts a function definition

Function name

Parameter list in parentheses

Function's result type

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

Function body in curly braces

Return statement is not necessary here

Scala: Function Definition (2)

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

Some notes:

- `if` expression produces a value; this value becomes the result of the whole function (no `return` needed).
- Function body consists of the single `if`; omit curly braces.
- The result type can be deduced from the type of `if` expression; could be omitted.

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

Scala: Function Definition (3)

Some notes:

```
def greeting() = println("Hello, world!")
```

- The function `greeting` produces no result; the "type" `Unit` is used for specifying it (in full form). Similar to **`void`**.

```
def greeting(): Unit = { println("Hello, world!") }
```

- The function is used for its side effect only. The side effect is actually printing the message.

Scala: Local Functions (1)

The common philosophy in functional programming (and in Scala in particular):

> The program functionality is implemented as a (great) number of functions ("building blocks"), each of which performs relatively simple action on its arguments returning the result (without side effect).

There are three basic ways for programming functions:

- Define a function as a class member.
- Define function as an object member (~static)
- Define a function **local** to some other function.

Scala: Local Functions (2)

Example: reading string lines from a file selecting long ones:

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(filename, width, line)
  }

  private def processLine(filename: String,
                          width: Int, line: String) {
    if (line.length > width)
      println(filename +": "+ line.trim)
  }
}
```

`processLine()` is just a “helper” for `processFile()` and doesn’t make sense without the first one.

Scala: Local Functions (3)

Let's make `processLine()` **local**!

```
def processFile(filename: String, width: Int) {

  def processLine(filename: String, width: Int, line: String) {
    if (line.length > width)
      print(filename +": "+ line)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines) {
    processLine(filename, width, line)
  }
}
```

Simplifying...

```
def processFile(filename: String, width: Int) {
  def processLine(line: String) {
    if (line.length > width)
      print(filename +": "+ line)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines) {
    processLine(line)
  }
}
```

Tasks for your homework:

- Download Scala system from www.scala-lang.org and install it on your computer (it's easy and doesn't hurt ☺).
  Teach yourself how to work with Scala compiler and interpreter (it's easy, especially the interpreter).
- Take the following code (see next slide, Rational numbers) and perform some actions on it (see the slide after the next).
  (If you don't know what "rational number" is – read Wikipedia ☺.)

Rational Numbers

Tasks for your homework:

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational = new Rational(
                numer*that.denom+that.numer*denom,denom*that.denom)
  def + (i: Int): Rational = /* add the code */

  def - (that: Rational): Rational =
    new Rational(numer*that.denom-that.numer*denom,denom*that.denom)
  def - (i: Int): Rational = /* add the code */

  def * (that: Rational): Rational =
    new Rational(numer*that.numer,denom*that.denom)
  def * (i: Int): Rational = /* add the code */

  def / (that: Rational): Rational =
    new Rational(numer*that.denom,denom*that.numer)
  def / (i: Int): Rational = /* add the code */
  override def toString = numer + "/" + denom
  private def gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b,a%b)
}
```

Tasks for your homework:

1. Answer the following questions:
   - Where is the primary constructor of the class?
   - Why there is no "default" constructor?
   - What are `g`, `numer` and `denom` for?
   - Why `toString` method defined with `override`?
   - Why do we need second versions of `+`, `-`, `*` and `/`?
   - When does `require(d!=0)` gets executed?

   The answers could be given in a form of comments to your code.
2. Complete the class:
   - Write implementations instead of `/* comments */`.
3. Provide examples of use `Rational` class.

Type Systems & Operators

Heterogeneous C++, Java, C#

Two kinds of types:

- Language-defined: `int`, `float`, pointers…
- User-defined: records, classes, interfaces, …

If we want class instances to behave like instances of predefined types, we would also like to have **a means for specifying operators** on class instances!

Homogeneous (unified) Eiffel, Scala

- **All types are** (user-defined or library) **classes**

If we want to keep conventional notation for operators on class instances, we need **a means for specifying operators** on class instances!

For now, for both cases we have only **class methods**…

Functions & Operators (1)

`FunName(argument1,argument2)`

General form

What's the semantics of a function call?-

We invoke an algorithm specified by FunName calculating a result value on arguments passed to the function

`argument1 OpSign argument2`

Syntactic sugar!

© B.Liskov CLU

What's the semantics of an operation?-

We invoke an algorithm specified by the `OpSign`, calculating a result value on the operands

Functions & Operators (2)

Conclusions:

- Conventional (infix, prefix, postfix) operators are just another syntactic form of more general function/method calls.
- Therefore, we can treat each operator sign as an alias to the name of the corresponding function/method, and the conventional operator form as an alternative syntax to the function call.
- We can allow to introduce own versions of the predefined operators for user-defined types.

Functions & Operators: C++ (1)

- The type system is heterogeneous. There is a set of operators for predefined types, and it's allowed to introduce the versions of the operators for user-defined types (classes).
- Operator's arity and preference shouldn't change.
- It's not allowed to introduce **new** operators. The idea behind this restriction is to make language ***extendable*** but ***not modifiable***.
- All operators of "common use" can be redefined AS WELL AS operators like indexing `[]`, function calls `()`, **`new`** and **`delete`** operators.

Functions & Operators: C++ (2)

```
class C {
   int member;
   ...
public:
   C(int p) : member(p) { }
   C& operator+(C& c1) {
        return C(member+c1.member);
   }
   int operator()(int p) { return member+p; }
   int operator[](int p) { return member-p; }
};

C c1, c2;
C sum = c1+c2;     // ≡ c1.operator+(c2);

int inc = sum(1); // ≡ sum.operator()(1);
int dec = sum[3]; // ≡ sum.operator[](3);
```

Operator functions

Functions & operators: Scala (1)

All operators are (alternative forms of) **method calls**

'+', '-', '*' and many other "signs" are **identifiers**!

`1 + 2`

`Int` object with value `1`

Invoking on `1` a method named '+'

Passing the `Int` object `2` to the '+' method

`(1).+(2)`

Functions & Operators: Scala (2)

It's allowed to add **your own operator signs**!

```
class C {
   x : Int
   def +++(i: Int) = x+i+i+i
   ...
}
C c
```

```
 ... c.+++(1) ...
```

Function invocation form

```
 ... c +++ 1 ...
```

Conventional infix operator notation

Scala: Functions & operators (2)

…And any method can be invoked using operator notation:

Usual identifiers can be used as operator signs!

```
class C {
  def op(i: Int) = ...
}
C c;
```

```
... c.op(1) ...
```

Function invocation form

```
... c op 1 ...
```

Conventional infix operator notation

Scala: Functions & operators (3)

One more example: **for**

```
for (i <- 0 to 2)
  // Do something
```

What does `0 to 2` mean?- **the call**:

```
0 to 2
```

↔

```
(0).to(2)
```

```
0 until 2
```

↔

```
(0).until(2)
```

Scala: Function literals (1)

Like `1.23` is an unnamed constant, it's possible to define unnamed "constant" functions:

Parameters in parentheses

Right arrow

Function body

```
(x: Int, y: Int) => x + y
```

Function literals are treated as **usual values** of the corresponding type – and we can work with them exactly as with other "constants".

Scala: Function literals (2)

Example:

```
var increase = (x: Int) => x + 1
increase(10) // returns 11
...
increase = (x: Int) => x + 9999
increase(10) // returns 10009
```

Scala: Function literals (3)

Function literals: how to use?

Example: a `foreach` method is available for all collections (`List`s, `Set`s, `Array`s, `Map`s). It takes a **function** as an argument and invokes that function on each of its elements.

```
val someNumbers = List(-11,-10,-5,0,5,10)

someNumbers.foreach((x: Int) => println(x))
```

Instance of a collection type

Method name

Function literal as an argument of the method call

Scala: Placeholders (1)

Let's continue the previous example using another function literal:

```
val someNumbers = List(-11,-10,-5,0,5,10)

someNumbers.filter((x:Int)=>x>0) // List(5,10)
```

Actually, we do not need **type spec** here

Target typing

```
someNumbers.filter((x)=>x>0) // List(5,10)
```

Actually, we do not need even a **parameter name**!

```
someNumbers.filter(_ > 0) // List(5,10)
```

Scala: Placeholders (2)

```
val f = _ + _                            // Error

val f = (_: Int) + (_: Int) // Correct
f(5, 10)                                 // returns 15
```

Scala: Functions & Closures (1)

Coming back to function literals:

x is called **bound variable**

```
val addOne = (x: Int) => x + 1
addOne(5)  // returns 6
```

Here, the only variable used in the function – x – is its parameter…

What if we use **some other variable** in function literal?

```
val addMore = (x: Int) => x + more
addMore(5)
```

Compile-time error:
"not found: value more"

more is called **free variable**

Scala: Functions & Closures (2)

Now more is in the context…

…and this gets compiled

```
var more = 1
...
val addMore = (x: Int) => x + more
addMore(10)    // returns 11
```

Scala: Functions & Closures (2)

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

Function literal without free variables: **closed term**.

Strictly speaking, this is not a closure because it doesn't "capture" free variables.

```
val addMore = (x: Int) => x + more
```

Function literal with free variables: **open term**.

This is a **closure** because it "captures" free variables. The functional value `addMore` "closes" the open term while its creation.

Scala: Functions & Closures (3)

```scala
val addMore = (x: Int) => x + more
```

An important issue: what happens if `more` variable **changes its value** after the closure has created?

```scala
var more = 1
...
val addMore = (x: Int) => x + more
addMore(10)    // returns 11
...
more = 9999
addMore(10)    // returns...WHAT?
```

Scala: Functions & Closures (4)

The Scala's answer is that the closure is captured with the **variable** but not with its value.

```
var more = 1
...
val addMore = (x: Int) => x + more
addMore(10)    // returns 11
...
more = 9999
addMore(10)    // returns 10009, not 11
```

In the latest C++ standard **both options** are allowed.

Scala: Functions & Closures (5)

The opposite case is also possible:
changes to the captured variable made by the closure **are visible outside of the closure**.

```
val someNumbers = List(-11,-10,-5,0,5,10)
var sum = 0

someNumbers.foreach(sum += _)

// sum equals to -11 after the last call
```

Here, `sum += _` is the closure capturing the `sum` variable from the enclosing context and modifying it.

Scala: Functions & Closures (6)

The last case: suppose a closure accesses to a variable which has **several copies**?

```
def makeIncreaser(more: Int) =
                  (x: Int) => x + more
```

This function returns **new closure** as the result of every call!- and each closure captures ***different copies*** of the same parameter `more`.

```
val inc1 = makeIncreaser(1)
val inc9999 = makeIncreaser(9999)
```

Scala: Functions & Closures (7)

```scala
def makeIncreaser(more: Int) =
                    (x: Int) => x + more

val inc1 = makeIncreaser(1)
val inc9999 = makeIncreaser(9999)
```

`inc1` contains the closure which captures parameter `more` with the value 1.

`inc9999` contains the closure which captures parameter `more` with the value 9999.

```scala
inc1(10)      // returns 11
inc9999(10)   // returns 10009
```

Tasks for your homework:

1. Write the function `makeMultiplier` that returns **the closure** applying multiplication on an external variable by its parameter.
2. Write the function `cutter` returning its parameter if its value is less than the value of an external variable or the value of the external variable otherwise.
3. Write the function predicate `remover` that returns true if its parameter is less than the value of an external variable.
4. Create a list of 20+ integer values. Define the external variable `limit` that would play a role of the external variable.
5. Create a "multiplier" closure using `makeMultiplier` function. Apply multiplier and `cutter` to each list element, and then apply `remover` to the whole list. Show the resulting list.
6. Repeat step 5 for different values of `limit`.