# Functional programming in Scala and further

**Prof. Nodar Momtselidze**

**Abstract**

Scala is one of the modern multi-paradigm programming language. It involves object oriented and functional programming paradigms possibilities. Scala has interpreter part and effective compiler which reduce it to JVMbytecode. In this article we bring the ideology of functional programming in Scala and some parts of its development. Scala has uniform and powerful abstraction concepts for different types and values;To be clear we compare some programs in Scala and Java.

## Keywords: Scala, functional programming, higher order functions, traits,

## monad, functor, futures, streams.

**Introduction**

**Scala** is an acronym for “*Scalable Language*”. It means that you can construct additional objects and functions and manipulate with them.

The main feature of the language - scalability is the result of a careful integration of object-oriented and functional language concepts.

Scala has interpreter part and effective compiler which reduce it to *JVM bytecode*. That's why you can use **Scala** and **Java** modules in one program. **Scala** compiler contains a subset of a **Java** compiler to make sense of such recursive dependencies.

**But what are object-oriented and functional programming concepts?**

***Object-oriented programming****(****OOP****)* is a [*programming paradigm*](http://en.wikipedia.org/wiki/Programming_paradigm) that represents concepts as "[*objects*](http://en.wikipedia.org/wiki/Object_(computer_science))" that have [*data fields*](http://en.wikipedia.org/wiki/Field_(computer_science)) (attributes that describe the object) and associated procedures known as [*methods*](http://en.wikipedia.org/wiki/Method_(computer_science))*.* Objects, which are usually [*instances*](http://en.wikipedia.org/wiki/Instance_(computer_science))*of*[*classes*](http://en.wikipedia.org/wiki/Class_(computer_science)), are used to interact with one another to design applications and computer programs. [***C++***](http://en.wikipedia.org/wiki/C%2B%2B)***,***[***ObjectiveC***](http://en.wikipedia.org/wiki/Objective-C)***,***[***Smalltalk***](http://en.wikipedia.org/wiki/Smalltalk)***,***[***Java***](http://en.wikipedia.org/wiki/Java_(programming_language)) and [***C#***](http://en.wikipedia.org/wiki/C_Sharp_(programming_language))

are examples of object-oriented programming languages.

The great idea of object-oriented programming is to make these containers fully general, so that they can contain operations as well as data, and that they are themselves values that can be stored in other containers, or passed as parameters to operations. Such containers are called objects.

But many languages admit values that are not objects, such as the primitive values. Or they allow static fields and methods that are not members of any object.

These deviations from the pure idea of object-oriented programming look quite harmless at first, but they have an annoying

tendency to complicate things and limit scalability.

**OOP** makes easy to adapt and extend complex systems, using

• subtyping and inheritance,

• dynamic configurations,

• classes as partial abstractions.

***Functional programming****(****FP****)* is

a programming paradigm that models computation as the evaluation of expressions. Expressions are built using functions that don’t have mutable state and side effects.

*Haskell,Curry,Idry,..* are examples of functional programming languages.

In math we have no assignment operation, loops, …, but we have [*designation*](http://www.lingvo-online.ru/en/Search/Translate/GlossaryItemExtraInfo?text=обозначение&translation=designation&srcLang=ru&destLang=en) x=sin(a),

A function relates every value of type X to exactly one value of Y. A type is associated with a set of values. Here type X represents the set of values (1, 2, 3) and Y represents the set of values (a, b, c).

In **Scala** you could write the signature of such a function as follows:

If function has input type **A** and output type **B** in Scala is written as: **A => B.**

**def f:** is definition of function **f**

**def f: X => Y**

1

**X** 2

3

a

b **Y**

c

The program in **FP** is constructed with *pure functions* which means that they have not side effects. In Scala everything are objects, including numbers and functions.

This makes pure functions easily testable and less bug prone.

It means that pure function can not:

* reassigning a variable;
* modify data structure in place;
* set a field on an object;
* throw an exception or halt with an error;
* read from or write to a file.

**FP** has restriction on writing programs, but not on programs what could be written.

FP makes easy to build interesting things from simple parts, using

• higher-order functions,

• algebraic types and pattern matching,

• parametric polymorphism.

Scala programs tend to be short. Scala programmers have reported reductions in number of lines of up to a factor of ten compared to Java.

*// this is Java*

**class MyClass {**

**private int index;**

**private String name;**

**public MyClass(int index, String name) {**

**this.index = index;**

**this.name = name;**

**}**

**}**

*//this is Scala*

**class MyClass(index: Int, name: String)**

/\* Given this code, the Scala compiler

will produce a class that has twoprivate

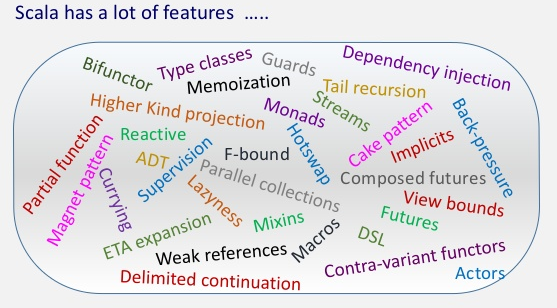
instance variables, an Int named **i**ndex

and a String named name, and a constructor.

*In short, you get essentially the same*

*functionality as the Java code*

*\*/*



* Scala has uniform and powerful abstraction concepts for different types and values;
* Scala has modular mixin-composition constructs for composing classes and traits.
* There is decomposition possibilities of objects by pattern matching

**Variables**

**Scala** has two types of variables **var and val.**

If we describein terming of **Java**

**var** isnon-final variables which can be reassigned, but **val** is similar to final variable in **Java.**

name; type; value

**var x: Double = 5.6**

//variable, could be reassigned (mutable)

**val y: Double = 5.6**

//value, can not be reassigned (immutable)

**val msg1: String = "Hello Scala!"**

//with String type

**val msg2: java.lang.String =**

**"Hello again Scala!"**

//with java.lang.String type

**val lst: List = List(1, 2, 3)**

//lst is immutable List

**val lst: List = 1 :: 2 :: 3 :: NULL**

//the same output

**val** **nms: Map = Map((1,"Nodar"), (2,"Maria"),(3,"Ann"))**

//nms is immutable Map value

**Functions**

Function definition starts with **def name(parameter:type,...):type = {body }**

"def" starts a function definition function name;

parameter list;

function's result type; function body

**def max(x: Int, y: Int): Int =**

**{ if (x > y) x else y }**

A function in Scala is a “first-class value”. You can pass it as parameter or return as result.

If functions take other functions as parameters or return as results are called **higher-order functions.**

**val double = (i: Int) => { i \* 2 }**

//think of symbol "=>" as transformer

***Scala is high-level***

Programmers are constantly grappling with complexity. Scala helps you manage complexity by letting you raise the level of abstraction in the interfaces you design

and use.

Assume you have String variable with name. You need to know if there are uppercase characters.

// this is Java

**boolean nameHasUpperCase = false;**

**for (int i = 0; i < name.length(); ++i) {**

**if(Character.isUpperCase(name.charAt(i)))**

**{**

**nameHasUpperCase = true;**

**break;**

**}**

**}**

// this is Scala

**val nameHasUpperCase =**

**name.exists(\_.isUpperCase)**

## *Evaluation Rules*

* Call by value: evaluates the function arguments before calling the function
* Call by name: evaluates the function first, and then evaluates the arguments .

**def example = 2**

// evaluated when called

**val example = 2**

// evaluated immediately

**lazy val example = 2**

// evaluated once when needed

**def square(x: Double)** // call by value

**def square(x: => Double)** // call by name

**def bxFct(bindings: Int\*) { ... }**

// bindings is a sequence of int, containing a // varying # of arguments

## *Higher order functions*

Functions in Scala are objects.

## So you can build functions that take function as a parameter or return functions.

## Such functions are called

## Higher order functions

Assume that we want build a function which calculates for different values of **f.**

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

Than for the functions:

**def id(k: Int): Int = k def square(k: Int): Int = k \* k def powerOfTwo(k: Int): Int =**

**if (k == 0) 1 else 2 \* powerOfTwo(k - 1)**

**def sumInts(a: Int, b: Int): Int =**

**sum(id, a, b) //**

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

**def sumPowersOfTwo(a: Int, b: Int): Int= sum(powerOfTwo, a, b) //**

and print.

**println( sumInts(1, 10) ) println( sumSquares(1, 10) ) println(sumPowersOfTwo (1, 10) )**

***Methods with collection***

Lets create collection of List type.

**val lst = List(1, 2, 3, 4, 5, 6, 7, 8)**

In Scala, there is a lot of methods working with List (map, filter, reduce,...)

**val sqr = lst.map(x => x \* x)**

//map applies function to all

//elements of List

**val flt = lst.filter(x => x < 6)**

//filters lst and creates flt as List(1,2, **val rds = lst.reduce((x,y) => x + y)**

//combines the elements of sequence into a //single element and creates rds as //List(3,7,11,15)

**Classes**

In Scala, classes are equivalent to classes in Java or C++. Every class has a primary constructor taken from class parameters. Class definition fields are generated into needed getters and setters automatically.

Auxiliary constructors are optional.

They are called as **this**.

**class Student(name: String,**

**scores: Int,**

**active: Boolean)**

Assume, we have class Student. Than, let us create Sequence (List) of Students in val st.

**val st = Seq(Student("Daviad", 38, true), Student("Mari", 95, true), Student("Gio", 51, false))**

Now, let us transform this Seq in functional style:

**val fst = st.filter(\_.score < 50) . filter(\_.active) . sortBy(\_.name) . map(\_.copy (active = false))**

In this one-liner we grabbed all students whose score is lower than 50 and who are still active; then we changed the active status of selected participants to false. **map** applies given function (**copy**) to every element.

The final output of the fds is List(**Student("David", 38, true)**.

There are dozens of situations where similar one-statements save functional programmers time and dramatically reduce the amount of code in the program.

**Traits**

Apart from inheriting code from super-class, Scala can import code from one or several traits. Comparing with Java, trait is interface which can contain code.

**trait Ord {**

**def < (that: Any): Boolean**

**def <= (that: Any): Boolean =**

**(this < that) || (this == that)**

**def > (that: Any): Boolean =**

**! (this <= that)**

**def >= (that: Any): Boolean =**

**! (this < that)**

**}**

**Closure**

A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

**val b = 10**

**val f = (x: Int) => x + b**

**println( f(5) ) //15**

**Monoid , Monad and Functor**

A *monoid* is defined as an algebraic structure (generally, a set) M with a binary operation (multiplication) • : M × M → M and an identity element (unit) η : 1 → M, following two axioms:  
  
i. **Associativity**  
∀ a, b, c ∈ M, (a • b) • c = a • (b • c)  
  
ii. **Identity**  
∃ e ∈ M ∀ a ∈ M, e • a = a • e = a

When specifying an endofunctor T : X → X (which is a functor that maps a category to itself) as the set M, the Cartesian product of two sets is just the composition of two endofunctors; what you get from here is a *monad*, with these two natural transformations:  
  
1. The binary operation is just a functor composition μ : T × T → T  
2. The identity element is just an identity endofunctor η : I → T  
  
Satisfied the monoid axioms (i. & ii.), a monad can be seen as a monoid which is an endofunctor together with two natural transformations. The name "monad" came from "monoid" and "triad", which indicated that it is a triple (1 functor + 2 trasformations), monoidic algebraic structure.  
  
In other words, *monoid* is a more general, abstract term. When applying it to the category of endofunctors, we have a *monad*.

A Functor accepts a function, A ⇒ B, and returns a new function M[A] ⇒ M[B], where M is any kind.

**trait Monoid[T] {**

**def Zero: T**

**def Op: (T,T) => T**

**}**

**trait Monad[T1] {**

**def map[T2] (f: T1 => T2): Monad[T2]**

**def flatMap[T2] (f: T1 => Monad[T2]): Monad[T2]**

**}**

**∀a,b ∈ C f:a ⇒b**

**F(a ⇒b) = F(a) ⇒F(b)**

**trait Functor[M[U]] {**

**def map[U,V](m:M[U])(f:U=>V): M[V]**

**}**

All these traits are useful for functions composition in functional programming.

**def compose(g: T => T, h: T => T) =**

**(x: T) => g(h(x))**

**Streams**

Sometimes, to solve problems, we need n numbers of an infinite sequence, but unfortunately it is impossible to determinate needed number of elements. Scala has data structure with infinite number of elements which are computed by demand. Such data structure are named streams. Streams are created using the constant ***empty*** and the constructor **cons. Streams reduce memory usage by relocated and releasing chunk of data allowing reuse of intermediating results.**

**def numsFrom (n :Int): Stream[Int]  =**

**Stream.cons(n,numsFrom (n+1))**

than calculate infinite stream using:

**lazy val N = numsFrom(0)**

**N take 10 print**

**The output will be:**

**0, 1, 2, 3, 4, 5, 6, 7, 8, 9, empty**

Stream is a List whose tail is a lazy val. The main benefit of Stream is working with infinite sequence, generally recursive created.

**Futures**

[Futures](http://docs.scala-lang.org/overviews/core/futures.html) are convenient abstraction for concurrent programming. Futures give us possibility to execute computations in parallel and receive result at some point of Futures.

**val fut = future {slowComputation}**

**fut.onComplete {**

**case Success => useSuccess(result)**

**case Failure => useError(exception)**

**}**

You can convert futures to list and back.

**List(Future(f1), Future(f2), ...Future(fn)**

**Futures(List(f1,f2,...fn))**

or reduce list of futures to new future

**Future.reduce(list)(f)**

**And many other technologies...**

Scala comes with a lot of different libraries, which give possibility to construct huge number of computation technologies.

**Conclusion**

This is the tour of some principal Scala features. I hope that the examples have given you some appreciation of the power of Scala and functional programming, and compare them with Java programs..

Later I hope to go deeper into Scala structures and features, asynchronous and parallel programming, and much more, to go into functional programming at a deeper level.

**References:**

Martin Odersky. Scala by Examples. (2014)

http://www.scala-lang.org/docu/files/ScalaByExample.pdf

Philipp Haller, Lukas Rytz, Martin Odersky. Scala: How to make best use of functions and objects. EPFL. (2010)

http://lampwww.epfl.ch/~phaller/doc/scala-tutorial-sac2010.pdf

Cay Hortsmann. Scala for the Impatient.

http://logic.cse.unt.edu/tarau/teaching/scala\_docs/scala-for-the-impatient.pdf

Learn Scala Programming.

http://www.tutorialspoint.com/scala/

# Scala School!

https://twitter.github.io/scala\_school/

[Vasily Vasinov](http://www.vasinov.com/).16-months-of-functional-programming.

http://www.vasinov.com/blog/16-months-of-functional-programming/#toc-immutable-state

Pramode C.E. Introduction to Functional Programming with Scala. (2013)

http://www.slideshare.net/pramode\_ce/introduction-to-functional-programming-with-scala?related=2 (ppg. 15, 16)

Pramode C.E. Advanced Functional programming in Scala. (2015)

http://www.slideshare.net/pnicolas/advanced-scala-concepts?related=1

(ppg. 3, 14, 15, 21)

# Bartosz Milevski. What is the difference between monoid and monad?

https://www.quora.com/What-is-the-difference-between-monoid-and-monad

Pedro Mateolo. Futures in Scala. (2013)

http://www.pmatiello.me/2013/10/futures-in-scala.html