Functional Programing

cerenode.io

What is FP ??

"In computer science, functional programming is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data."

Scala Methods And Functions

- Scala has both function and method.
- The terms method and function interchangeably with a minor difference

```
// a simple method
scala> def double(x: Int) = 2*x
m: (x: Int)Int

// a simple function
scala> val triple = (x: Int) => 2*x
f: (Int) => Int = <function1>
```

Method Syntax

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

Function Syntax

There are two syntax available for function

First

```
val sum = (a:Int , b:Int) => a + b: Int
```

Second

```
val sum:(Int, Int) => Int = (a, b) => a + b
```

Methods vs Functions

- A method can appear in an expression but it can’t be the final value, while a function can
- A method can have no parameter list or have one, but a function must have one (parameter list can be empty)
- Method name means invocation while function name means the function itself
- Can pass function as arguments to method
    - Can pass method also like this but after converting the method to function (happens automatically)

Method to Function

In scala we explicitly convert a method to function by writing underscore (_) after the method name

```
scala> def double(x: Int) = 2 * x
double: (x: Int)Int

// explicitly convert the method into a function
scala> val doublefun = double _
doubleFun: (Int) => Int = <function1>
```

First Class Function

All Functions in Scala are first class , so you can do following things in scala

- Assign to variable
- Pass it as argument to other functions
- Return it as value from other functions

Properties of Pure Functions

- The output of a pure function depends only on
  - its input parameters
  - its internal algorithm
- A pure function has no side effects
- As a result of those first two statements, if a pure function is called with an input parameter x an infinite number of times, it will always return the same result y

Parameterization And Default Values

Methods have it , functions doesn’t

```
def count[A](list: List[A]) = list.size

def sum(a:Int ,b:Int = 10) = a + b
```

Evaluation Strategies

Evaluation strategies exists to determine when and how the arguments of a function call are evaluated. There are many evaluation strategies, but most of them end up in two categories:

- **Strict evaluation**, in which the arguments are evaluated before the function is applied.
  - Java, Scheme, JavaScript etc.
- **Non-strict evaluation**, which will defer the evaluation of the arguments until they are actually required/used in the function body.
  - Haskell

Call By Value

- Strict evaluation strategy
- Most commonly used by programming languages
- Expression is evaluated and bound to the corresponding parameter before the function body is evaluated
- Default evaluation strategy in Scala.

```
def sqrt(a :Int ) = { a * a }
```

Call by Name

- Non-strict evaluation strategy
- Defer the evaluation of the expression until the program needs it
- To make a parameter call-by-name just add `=>` before the type

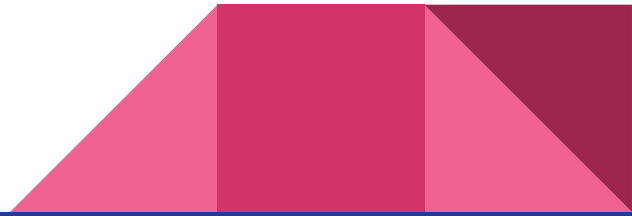
```
scala> val r = new util.Random()
r: scala.util.Random = scala.util.Random@5353dd09

scala> def randTuple(x: => Int) = { (x, x) }
randTuple: (x: => Int)(Int, Int)

scala> randTuple(r.nextInt)
res2: (Int, Int) = (858515317,-1695183353)
```

Lambdas

A **lambda** or, more specifically a **lambda expression** is a term used to refer to an expression that does not reference a value or variable, but instead references an anonymous function.



Closures

- Function whose return value depends on the value of one or more variable defined outside this function.
- A closure is an instance of a function, a value whose non-local variables have been bound either to values or to storage locations.

Nested Methods

```
def factorial(x: Int): Int = {
    def fact(x: Int, accumulator: Int): Int = {
        if (x <= 1) accumulator
        else fact(x - 1, x * accumulator)

    }
    fact(x, 1)
}
```

Higher Order Functions

- Higher order functions take other functions as parameters or return a function as a result
- This is possible because functions are first-class values in Scala
- The phrase “higher order function” is used for both methods and functions

Currying

- A function that takes multiple arguments can be translated into a series of function calls that each take a single argument.

```
def sum(a:Int)(b:int) = a + b
```

compose and **andThen**

```
val double = (x:Int) => x + x
val square = (y:Int) => y * y
double compose square apply 2  // returns 8
double andThen square apply 2  // returns 16
```

- The difference between compose and andThen is the order they execute the functions
- While the compose function executes the caller last and the parameter first, the andThen executes the caller first and the parameter last

Partial Functions

- As the name suggests, partial functions are only partial implementations.
- A partial function of type **PartialFunction[A, B]** is a unary function where the domain does not necessarily include all values of type **A**. The function **isDefinedAt** allows to test dynamically if a value is in the domain of the function

```
val squareRoot: PartialFunction[Double, Double] = {
    case x if x >= 0 => Math.sqrt(x)
}
squareRoot.isDefinedAt(4)  // returns true
squareRoot.isDefinedAt(-4)  // returns false
```

Partially Applied Functions

- Function that takes a function with multiple parameters and returns a function with fewer parameters

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

val increment = sum(1, _:Int) // partially applied function

increment(2)  // returns 3
```