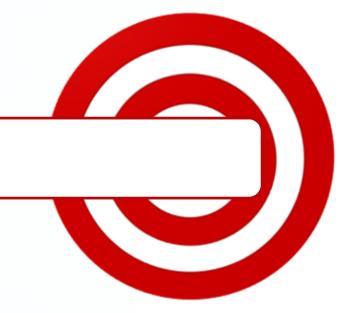


SCALA





Agenda

In this module, we are going to look at the following topics:

- ✓ Functional Programming
- ✓ Higher Order Functions





Functional Programming

• In a functional programming language functions are first-class citizens that can be passed around and manipulated just like any other data types.

```
import scala.math._
A. val fun = ceil _ // assigning a function to a variable
Array(1.23, 6.28, 4.0).map(fun)
```

Functions with no name are called Anonymous functions

```
( x : Long ) => x * x
Array(1, 6, 4).map((x:Long)=>x*x)
Array(1, 6, 4).map{(x:Long)=>x*x} // you can also use {} instead of ()
```



Functional Programming

Function vs Method:

 Anything defined with a def is a method. Methods can't be the final value of an expression, but function can be.



Higher Order Functions

- A higher order function is a function that takes a function as a parameter and/or produce a function as return value.
- When you pass an anonymous function to another function or method,
 Scala infers the type wherever possible.



Higher Order Functions

```
// passing a function as a method parameter
def atQuarter(f: (Double) => Double) = f(0.25)
// Scala type inference
atQuarter((x:Double) => 3*x)
atQuarter (x => 3*x) // scala can infer the type of x, so can omit
atQuarter(3 * ) ) // single param can be written like this
// method returning a function
def multiplyBy(factor : Int) = (x : Int) => factor * x
val half = multiplyBy(0.5)
half(20)
```





reduceLeft: takes a binary function (fn with two params) and applies it to all elements of a sequence, going from left to right (we have reduceRight also)

```
sortWith: takes a binary sorting function and applies it to all elements
"Mary had a little".split(" ").sortWith(_.length < _.length)</pre>
```



• flatMap: flattens the results of multiple collections into a single collection.

• **transform**: is in-place equivalent of map applicable to mutable collections, and replaces each element with the result of a function

```
val names = scala.collection.mutable.ArrayBuffer("Kanakaraju", "Veer")
names.transform( x => x.length().toString() ) // (10, 4)
```



• collect: works with functions that may not be defined for all values (aka *partial functions*) and yields a collection of all functions values of the arguments on which it is defined.

```
"-3+4".collect { case '+' => 1 ; case '-' => -1 } // (-1, 1)
```

• groupBy: method yields a map whose keys are the function values, and whose values are the collection of elements whose function values is the given key

```
val names = List("Raju", "Veer", "Harsha")
val map = names.groupBy(_subString(0,1).toUpper)
```



• **foldLeft & foldRight**: similar to reduce methods but provides an initial element other than the initial elem. of a collection.

```
val foldLeft(init)(op)
val foldLeft = List(1, 7, 2, 9).foldLeft(0)( - ) //(((0-1)-7)-2)-9) = -19
```

• scanLeft & scanRight: combines folding and mapping. You get a collection of all intermediate results.



zip: The zip method lets you combine two collections into a List of pairs.

```
val prices = List(15.0, 34.0, 19.5)
val quantities = List(14, 12, 51)
val zip1 = prices.zip(quantities)  //List((15.0, 14), (34.0, 12), (19.5, 51))
(prices zip quantities) map { p => p._1 * p._2 }  //List(210.0, 408.0, 994.5)
val zip4 = List(5.0, 20.0, 9.95) zip List(10, 2)  //List((5.0,10), (20.0,2))
```

• zipAll allows you to specify missing values for the elements in the list.

```
val zip5 = List(5.0,20.0,9.95).zipAll(List(10, 2), 0, 1)
```



THANK YOU

