Scala Introduction and Hands on

Why Scala?

- First, a developer can achieve a **significant productivity** jump by using Scala.
- Second, it helps developers write **robust code** with reduced bugs.
- Third, Spark is written in Scala, so Scala is a natural fit for **developing Spark applications**.

Functional Programming
What is Functional Programming?

- Functional programming makes it easier to write **concurrent or multithreaded** applications.
- Functional programming languages make it easier to write elegant code, which is **easy to read, understand, and reason** about.

Functions – Characteristics

First Class
FP treats functions as first-class citizens. A function has the same status as a variable or value. It allows a function to be used just like a variable. While in case of any imperative language such as C it treats function and variable differently.

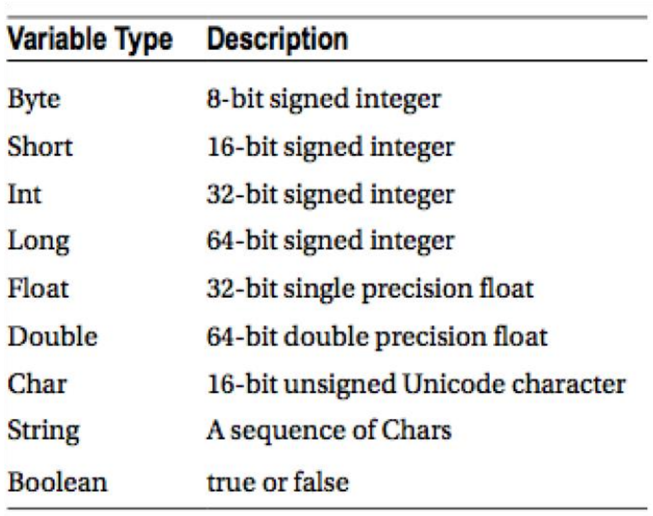
Composable
Functions in functional programming are composable. Function composition is a mathematical and computer science concept of **combining simple functions to create a complex one**.

No Side Effects
A function in functional programming does not have side effects. The result returned by a function depends only on the input arguments to the function**. The behavior of a function does not change with time. It returns the same output every time for a given input, no matter how many times it is called**. In other words, a function does not have a state. It does not depend on or update any global variable.

Simple
Functions in functional programming are **simple**. A function consists of a few lines of code and it does only one thing. A simple function is easy to reason about. Thus, it enables robustness and high quality.

Variables

Statically Defined and Dynamically Inferred
Mutable – Var
Immutable – Val
Lazy Values



Variable Type	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	true or false

Exercises:
The easiest way to get started with Scala is by using the Scala interpreter, which provides an interactive shell for writing Scala code. It is a **REPL** (read, evaluate, print, loop) tool

Variables

VALUES are immutable constants.
You can't change them once defined.

```
val name: String = "Inceptez!"
name =  "Inceptez Technologies"
println(name)
```

// Notice how Scala defines things backwards from other languages - you declare the //
name, then the type.

VARIABLES are mutable

```
var fullname: String = " Inceptez! "
fullname = "Inceptez!"  + " Technologies"
println(fullname)
```

// One key objective of functional programming is to use immutable objects as often as possible.
// Try to use operations that transform immutable objects into a new immutable object.
// For example, we could have done the same thing like this:

```
val immutablename = name + " Technologies"
println(immutablename)
```

Control Structures:

```
val x=10
val y = 5

if (x > y )  println("x is greater")  else if (y>x)
println("y is greater") else
println("x equals y")
```

For loops

```
for (x <- 1 to 4)
{
val squared = x * x
println(squared)
}
```

While loops:

```
var x = 10
while (x >= 0) {
println(x)
x -= 1
}


var sid = 4;
do {
println("Student Id is:"+sid);
sid = sid + 1
}while(sid < 10)
```

Expressions :

// "Returns" the final value in a block automatically

```
val x = 10;
x + 20

println  ({
val x = 10;
x + 20
})
```

Methods

How to Define Method in Scala?

A method in Scala is defined with the keyword def. A method definition starts with the name, which is followed by the comma-separated input parameters in parentheses along with their types. The closing parenthesis is followed by a colon, method output type, equal sign, and the method body in optional curly braces.

```
def highsal(p1: Int, p2: Int,p3:Int): Int = {
val salbonus = p1 + p2
if (salbonus > p3)
{
println("higher value is salary and bonus") ;
return salbonus;
}
else
{println("higher value is nettsal");
return p3
}
}

println(highsal (1,2,4))
println(highsal (1,4,4))
```

Scala allows a concise version of the same function, as shown next.

```
def add(firstInput: Int, secondInput: Int) = firstInput + secondInput

add(5,10)
```

The second version does the exact same thing as the first version. The type of the returned data is omitted since the compiler can infer it from the code. However, it is recommended not to omit the return type of a function.

The curly braces are also omitted in this version. They are required only if a function body consists of more than one statement.

Higher-Order Methods

A method that takes a function as an input parameter is called a *higher-order method*. Similarly, a highorder function is a function that takes another function as input. Higher-order methods and functions help reduce code duplication. In addition, they help you write concise code. The following example shows a simple higher-order function.

```
val salaries = Seq(20000, 70000, 40000)

def bonus(a:Int):Double = ((a*1.5))

val normalmethod = salaries.map (a=>(a*1.5) )

val higherordermethod = salaries.map (bonus)
```

Closures

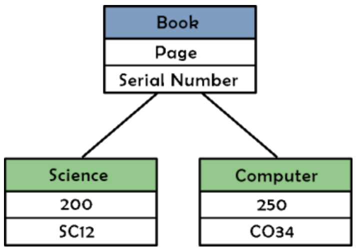
A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

```
var bonuspercent = .10
def bonus = (i:Int) => {i+(i * bonuspercent)}

println( "Bonus value is = " + bonus(100) )
```

Classes & Objects

- A class is a template or blueprint for creating objects at runtime.
- An object is an instance of a class.
- A class is defined in source code, whereas an object exists at runtime. A class is defined using the keyword class.
- A class definition starts with the class name, followed by comma-separated class parameters in parentheses, and then fields and methods enclosed in curly braces. An example is shown next.



An instance of a class is created using the keyword new.

```
class salary(val p1: Int, val p2: Int, val p3:Int) {
var hike: Int = p1
var penality: Int = p2
var incentive:Int = p3

def hike(id:String,sal: Int)
{
val netsal = hike + sal -penality
println ("nett salary for empid  " +id +" is "+ netsal);
}

def grosssalary(id:String,sal:Int)
{
    val grosssal = hike + sal +incentive-penality
    println ("gross salary for empid  " +id +" is  "+ grosssal);

  }
}

object Salcalc {
def main(args: Array[String]) {
val sal = new salary(10000,3000,args(0).toInt);
sal.hike("a1",30000);
sal.hike("a2",40000);
sal.grosssalary("a1",25000);
sal.grosssalary("a2",35000);
}
}

Salcalc.main(Array("5000"))
```

Case Classes

A case class is a class with a case modifier. An example is shown next.

All input parameters defined implicitly treated as Val, Useful for immutable objects and pattern matching
Creates Factory method automatically

```
case class emp(id: Int, name: String, address: String)

val request = emp(1, "Sam", "1, castle point blvd, nj")
println(request.name)
request.address
```

Traits

A *trait* represents an **interface** supported by a hierarchy of related classes. It is an **abstraction** mechanism that helps development of **modular, reusable, and extensible code**

A trait looks similar to an **abstract class**. Both can contain fields and methods. The key difference is that a class can inherit from only one class, but it can inherit from any number of traits. An example of a trait is shown next.

```
trait Shape {
def area(): Int ;
def plintharea(plintharea1:Int,reduced: Int): Int = {
val reducedarea = plintharea1 - reduced;
println ("Final plinth area calculated is " +reducedarea);
return reducedarea;
}
}

trait landarea {
def totarea(length:Int,breadth: Int): Int = {
val total = length*breadth;
return total;
}
}

class PlinthClass(length: Int,breadth:Int) extends Shape with landarea{
def area = length * length ;
val reducedarea = area/4;
def plinth = plintharea(area, reducedarea);
println("Total area value is " +area);
println("Reduced area value is " +reducedarea);
println("Plinth area value is " +plinth);
def ta = totarea(length,breadth);
println("Total area returned is : " + ta);
}

val plinthinstance = new PlinthClass(20,40)
val pli = plinthinstance.area
val pli = plinthinstance.ta
```

Tuples

A *tuple* is a container for storing two or more elements of different types. It is immutable, it cannot be modified after it has been created. It has a lightweight syntax, as shown next.

```
val twoElements = ("10", true)
val threeElements = (10, "harry", true)
```

An element in a tuple has a one-based index.
The following code sample shows the syntax for accessing elements in a tuple.

```
val first = threeElements._1
val second = threeElements._2
val clubbed = threeElements._3+ " "+ twoElements._1
```

Collections

A collection is a container data structure. It contains zero or more elements. Collections provide a higher- level abstraction for working with data. They enable declarative programming. With an easy-touse interface, they eliminate the need to manually iterate or loop through all the elements.
Categories:

- Sequences
- Maps
- Sets

Sequences

A *sequence* represents a collection of elements in a specific order. Since the elements have a defined order, they can be accessed by their position in a collection. For example, you can ask for the *nth* element in a sequence.

Array

An *Array* is an indexed sequence of elements. All the elements in an array are of the same type. It is a mutable data structure; you can update an element in an array. However, you cannot add an element to an array after it has been created. It has a fixed length.

```
var arr = Array(10, 20, 30, 40)
arr(0) = 50
arr(4) = 50
```

Map

Map is a collection of key-value pairs. In other languages, it known as a dictionary, associative array, or hash map. It is an efficient data structure for looking up a value by its key. It should not be confused with the map in Hadoop MapReduce. That map refers to an operation on a collection.

The following code snippet shows how to create and use a Map.

```
val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")

val indiaCapital = capitals("India")
```