Scala programming language

Inceptez Techologies

What is Scala?

- Concise, Statically Typed
- Runs on JVM, full inter-op with java
- Object Oriented
- Functional
- Martin Odersky started developing in 2001 and released in 2004

Object Oriented Paradigm

- Fully Supports OOP
- Everything is object in scala
- Unlike Java scala doesn’t support primitive
- Support static object via singleton class
- Improved support to OOP by Traits

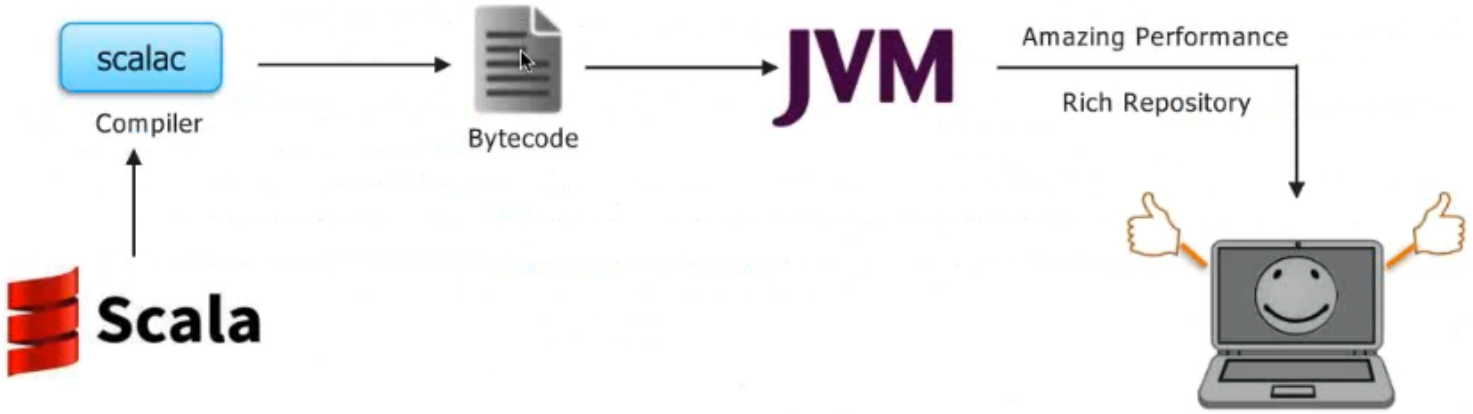
Functional Programming Paradigm

- Support both mutable & immutable
- Functions are first class citizens
  - Function can be assigned to a variable
  - Function can be stored in a data structure
  - Function can be passed around as an argument to other functions
  - Function can be returned from the functions

  In functional programming languages, it is possible to do the above mentioned things.
- Functions are also objects

Scala on JVM

- scalac compiles Scala to Java bytecode
  - (regular .class files)
- **Any Java class can be used from Scala**



Variables

- Statically Typed Language
- Type Inference
- Mutable – Var
- Immutable – Val

Basic Types

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

Variables & values, type inference

```
var msg = "Hello“      // msg is mutable
msg += " world"
msg = 5;               // compiler error
```

Variables & values, type inference

```
val msg = "Hello world“     // msg is immutable
msg += " world“             // compiler error
```

```
val n : Int = 3     // explicit type declaration
var n2 : Int = 3
```

Immutability

- Why?
  - Immutable objects are automatically thread-safe

    (you don't have to worry about object being changed by another thread)
  - Compiler can reason better about immutable values -> optimization

  - Steve Jenson from Twitter: *"Start with immutability, then use mutability where you find appropriate."*

If..else Condition

```
object Test
{
   def main(args: Array[String])
   {
         var x = 10;
         if( x < 20 )
         {
                println("This is if statement");
         }
         else
         {
                 println("This is else statement");
          }
    }
}
```

While loop

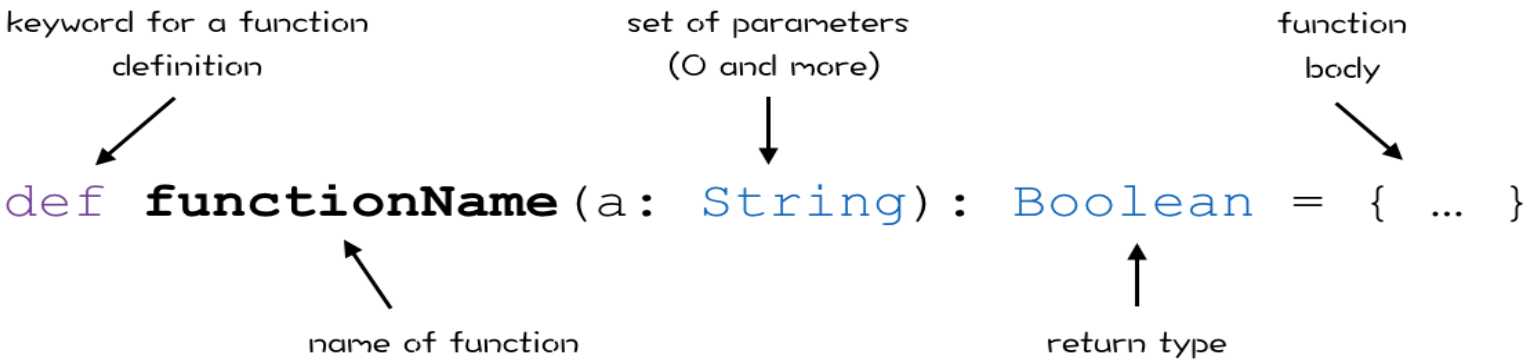
```
object Test
 {
    def main(args: Array[String])
   {
      // Local variable declaration:
      var a = 10;
      // while loop execution
      while( a < 20 )
     {
       println( "Value of a: " + a );
       a = a + 1;
      }
    }
}
```

for loop

```
object Test
 {
    def main(args: Array[String])
   {
      var a = 0;
      // for loop execution with a range
      for( a <- 1 to 10)
     {
        println( "Value of a: " + a );
      }
    }
}
```

Methods

```
def max(x : Int, y : Int) = if (x > y) x else y
// equivalent:
def neg(x : Int) : Int = -x
def neg(x : Int) : Int = { return -x; }
```



Methods..cont

- With or without equal operator
- Nested functions
- Parameter with Default value
  def add(a:Int = 0,b:Int = 0)
- Named Parameter
  add(a = 15, b = 2)
- Return keyword is optional

```
object Test
 {
 def main(args: Array[String])
{
                println( "Returned Value : " + addInt(5,7)
);
 }
 def addInt( a:Int, b:Int ) : Int =
{
                var sum:Int = 0
                sum = a + b
                return sum
 }
}
```

Anonymous functions

Anonymous function is a function that has no name but works as a function.
It is good to create an anonymous function when you don't want to reuse it latter.

```
object Test
{
      def main(args: Array[String])
     {
         println( "Returned Value : " + addInt(5,7) );
         def addInt( a:Int, b:Int ) : Int =
         {
             var sum:Int = 0
             sum = a + b
             return sum
         }
}
```

```
//equivalent Anonymous function
val addInt = (a:Int,b:Int) => a + b
```

Pattern Matching

Pattern matching is a feature of scala. It works same as switch case in other programming languages. It matches best case available in the pattern.

```
object MainObject {
  def main(args: Array[String]) {
     var a = 1
     a match{
        case 1 => println("One")
        case 2 => println("Two")
        case _ => println("No")
     }
     }
}
```

```
object MainObject {
  def main(args: Array[String]) {
     var result = search ("Hello")
     print(result)
   }
   def search (a:Any):Any =
     a match{
     case 1  => println("One")
     case "Two" => println("Two")
     case "Hello" => println("Hello")
     case _ => println("No")
     }
}
```

Higher Order Functions

A Higher Order Function is a function which takes another function as its parameters. It can be used as callback function

```
object Demo {
   def main(args: Array[String]) {
      println( apply( layout, 10) )
   }
   def apply(f: String => String, v: Int) = f(v)
   def layout(x: String) = "[" + x + "]"
}
```

```
// functions as parameters
def call(f: Int => Int) = f(1)

call(plusOne)     → 2
call(x => x + 1)  → 2
call(_ + 1)       → 2

def plusOne (x:Int)
{
   x + 10
}
```

Scala Collections

Collections are the container of things which contains random number of elements. All **collection**classes are found in the package **scala**.**collection**

- Arrays
- Lists
- Sets
- Tuple
- Maps
- Option

Scala Collections : Array

- Indexed sequence of elements
- All elements are of same type
- Mutable
- Fixed Length

```
scala> val numbers = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: Array[Int] = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

```
scala> numbers(3) = 10
```

Scala Collections : Array

- Indexed sequence of elements
- All elements are of same type
- Mutable
- Fixed Length

```
scala> val numbers = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: Array[Int] = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

```
scala> numbers(3) = 10
```

Scala Collections : Lists

Lists preserve order, can contain duplicates, and are immutable.

```
scala> val numbers = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: List[Int] = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
scala> numbers(3) = 10
<console>:9: error: value update is not a member of List[Int] numbers(3) = 10
```

Scala Collections : Sets

Sets do not preserve order and have no duplicates

```
scala> val numbers = Set(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Scala Collections : Tuple

Generalised form of Pair.More than two values of different types

```
scala> val emp= (1, ”aaa”, 9000.00)
emp: (Int, String,Float) = (1,aaa,9000.00)
```

Tuple don’t have named accessors, instead they have accessors that are named by their position and is 1-based rather than 0-based.

```
scala> emp._1
res0: Int= 1

scala> emp._2
res1: String = aaa
```

Scala Collections : Maps

Map is a collection of pair. Pair is a group of two values

Map is immutable but mutable type available with scala.collections.mutable.Map

```
scala> val numbers = Map("one" -> 1, "two" -> 2)
numbers:
scala.collection.immutable.Map[java.lang.String,Int] =
Map(one -> 1, two -> 2)

scala> numbers.get("two")
res0: Option[Int] = Some(2)

 scala> numbers.get("three")
res1: Option[Int] = None
```

Options

Option class when returning a value from a function that can be null. instead of returning one object when a function succeeds and null when it fails, your function should instead return an instance of an Option, where the Option object is either:

An instance of the Scala Some class.

An instance of the Scala None class

```
def getName(value: Int): Option[String] = {
if (value >= 1) {
        return Option("Oxford")
    }
    else {
        return None
    }
}

// Accepted to Oxford.
println(getName(10))
// Rejected.
println(getName(0))
```

Higher-Order Methods on Collection Classes

- Map
- foreach
- flatMap
- Filter

Map

Map – apply a function on the sequence collection and return another colleciton

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> numbers.map((i: Int) => i * 2)
res0: List[Int] = List(2, 4, 6, 8)
```

or pass in a function (the Scala compiler automatically converts our method to a function)

```
scala> def timesTwo(i: Int): Int = i * 2
timesTwo: (i: Int)Int
```

Foreach

foreach is like map but returns nothing.

```
scala> numbers.foreach((i: Int) => i * 2)
```

returns nothing.

You can try to store the return in a value but it'll be of type Unit (i.e. void)

```
scala> val doubled = numbers.foreach((i: Int) => i * 2)
doubled: Unit = ()
```

Filter

Removes any elements where the function you pass in evaluates to false. Functions that return a Boolean are often called predicate functions.

```
scala> numbers.filter((i: Int) => i % 2 == 0)
res0: List[Int] = List(2, 4)

scala> def isEven(i: Int): Boolean = i % 2 == 0
isEven: (i: Int)Boolean

scala> numbers.filter(isEven)
res2: List[Int] = List(2, 4)
```

FlatMap

FlatMap takes a function that works on the nested lists and then concatenates the results back together.

```
scala> val nestedNumbers = List(List(1, 2), List(3, 4))
nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

scala> nestedNumbers.flatMap(x => x.map(_ * 2))
res0: List[Int] = List(2, 4, 6, 8)
```

A class ...

... in Java:

```
public class Person {
  public final String name;
  public final int age;
  Person(String name, int age) {
      this.name = name;
      this.age = age;
  }
}
```

... in Scala:

```
class Person(val name: String,
             val age: Int) {}
```

Objects

Objects are used to hold single instances of a class.

```
object Timer
{
	var count = 0
	def currentCount(): Long =
	{
		count += 1
		count
	 }
}
```

How to use

```
scala> Timer.currentCount()
res0: Long = 1
```

Case classes

Case classes are used to conveniently store and match on the contents of a class. You can construct them without using new.

case class Book(title: String, pages: Int)

What is hidden behind this line of code?

1) Class with 2 immutable fields
2) Getters for the fields
3) Constructor
4) Useful methods

```
val b1 = Book("Scala book", 150)
b1.title //Scala book
b1.pages //150
val b2 = b1.copy(pages = 220)
b2.title //Scala book
b2.pages //220

b1.eq(b2) //false
b1.eq(Book("Scala book", 150)) //false
b1.equals(Book("Scala book", 150)) //true
b1 == b2 //false
b1 == Book("Scala book", 150) //true
```

Exceptions

```
object MainObject{
   def main(args:Array[String]){
      var e = new ExceptionExample()
      e.divide(100,10)

   }
}
```

```
class ExceptionExample{
   def divide(a:Int, b:Int) = {
      try{
         a/b
         var arr = Array(1,2)
         arr(10)
      }catch{
         case e: ArithmeticException => println(e)
         case ex: Exception =>println(ex)
       }
      finally{
         println("Finaly block always executes")
      }
      println("Rest of the code is executing...")
   }
}
```

Work-Outs