Creating your first Scala project

Scala - Basic Syntax

Variable Declaration

Scala has a different syntax for declaring variables. They can be defined as value, i.e., constant or a variable.

They can be defined as value, i.e., constant or a variable. Here, myVar is declared using the keyword var. It is a variable that can change value and this is called mutable variable. Following is the syntax to define a variable using var keyword −

```
Syntax

var myVar : String = "Foo"
```

Here, myVal is declared using the keyword val. This means that it is a variable that cannot be changed and this is called immutable variable. Following is the syntax to define a variable using val keyword −

```
Syntax

val myVal : String = "Foo"
```

Variable Type Inference

When you assign an initial value to a variable, the Scala compiler can figure out the type of the variable based on the value assigned to it. This is called variable type inference. Therefore, you could write these variable declarations like this −

```
var myVar = 10;

val myVal = "Hello, Scala!";
```

Here, by default, myVar will be Int type and myVal will become String type variable.

Multiple assignments

Scala supports multiple assignments. If a code block or method returns a Tuple (Tuple − Holds collection of Objects of different types), the Tuple can be assigned to a val variable.

```
val (myVar1: Int, myVar2: String) = Pair(40, "Foo")
```

```
       var myVar :Int = 10;
       val myVal :String = "Hello Scala with datatype declaration."
       var myVar1 = 20;
       val myVal1 = "Hello Scala new without datatype declaration."

       println(myVar); println(myVal); println(myVar1);
       println(myVal1);
```

Scala Functions

Function Declarations and Definitions

A function is a group of statements that perform a task. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically, the division usually is so that each function performs a specific task.

```
def functionName ([list of parameters]) : [return type] = {
   function body
   return [expr]
}
```

Very similar to Java, a return statement can be used along with an expression in case function returns a value. Following is the function which will add two integers and return their sum –

```
object add {
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b
      return sum
   }
}
```

A function, that does not return anything can return a Unit that is equivalent to void in Java and indicates that function does not return anything.

Calling Functions

Scala provides a number of syntactic variations for invoking methods. Following is the standard way to call a method –

```
functionName( list of parameters )
```

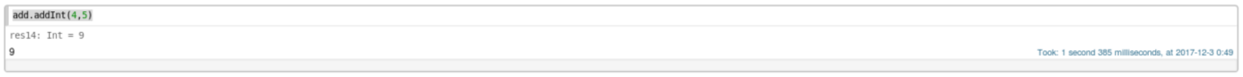
If a function is being called using an instance of the object, then we would use dot notation similar to Java as follows –

```
[instance.]functionName( list of parameters )
```

Let us call the defined function

add.addInt(4,5)

Output:



```
add.addInt(4,5)
res14: Int = 9
9                                    Took: 1 second 385 milliseconds, at 2017-12-3 0:49
```

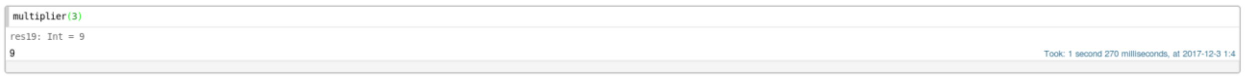
1. Scala – Closures

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

```
var factor = 3

val multiplier = (i:Int) => i * factor
```

Here the only variable used in the function body, i * 3 , is i, which is defined as a parameter to the function.

Output:



```
multiplier(3)
res19: Int = 9
9                                    Took: 1 second 270 milliseconds, at 2017-12-3 1:4
```

Try yourself:

```
var more = 1

val addMore = (x: Int) => x + more

addMore(10)
```

2. Scala Strings

A string is an immutable object, that is, an object that cannot be modified.

Creating a String

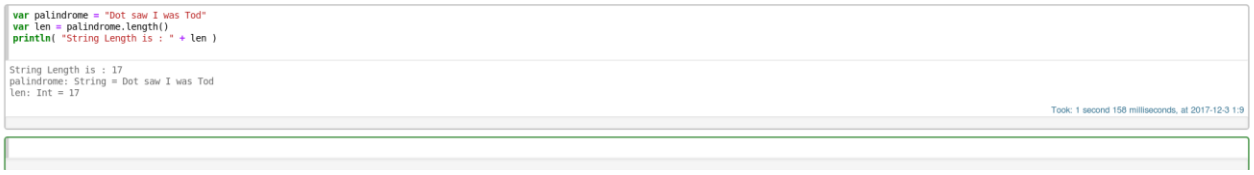
The following code can be used to create a String –

```
var greeting = "Hello world!";

or

var greeting:String = "Hello world!";
```

Let us try the example

```
var palindrome = "Dot saw I was Tod"

var len = palindrome.length()

println( "String Length is : " + len )
```

Output:



```
var palindrome = "Dot saw I was Tod"
var len = palindrome.length()
println( "String Length is : " + len )

String Length is : 17
palindrome: String = Dot saw I was Tod
len: Int = 17
Took: 1 second 158 milliseconds, at 2017-12-3 1:9
```

String Interpolation

String Interpolation is the new way to create Strings in Scala programming language. This feature supports the versions of Scala-2.10 and later. String Interpolation: The mechanism to embed variable references directly in process string literal.

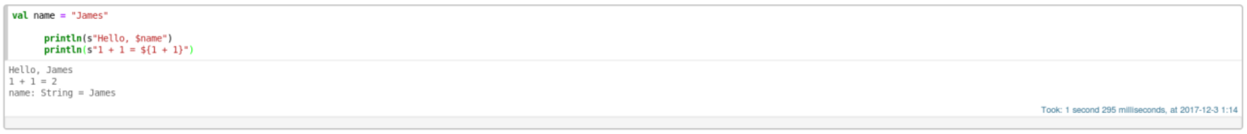
The ‘s’ String Interpolator

The literal ‘s’ allows the usage of variable directly in processing a string, when you prepend ‘s’ to it. Any String variable with in a scope that can be used with in a String. The following are the different usages of ‘s’ String interpolator.

The following example code snippet for the implementation of ‘s’ interpolator in appending String variable ($name) to a normal String (Hello) in println statement.

```
val name = "James"

      println(s"Hello, $name")

      println(s"1 + 1 = ${1 + 1}")
```

Output:



```
val name = "James"

      println(s"Hello, $name")
      println(s"1 + 1 = ${1 + 1}")
```

```
Hello, James
1 + 1 = 2
name: String = James
```

Took: 1 second 295 milliseconds, at 2017-12-3 1:14

3. Scala – Traits

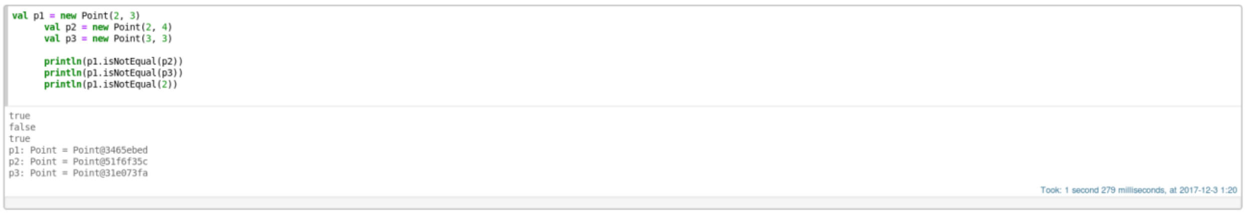
A trait encapsulates method and field definitions, which can then be reused by mixing them into classes.Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.

Try the following example program to implement traits.

```
trait Equal {
   def isEqual(x: Any): Boolean
   def isNotEqual(x: Any): Boolean = !isEqual(x)
}

class Point(xc: Int, yc: Int) extends Equal {
   var x: Int = xc
   var y: Int = yc

   def isEqual(obj: Any) = obj.isInstanceOf[Point] &&
obj.asInstanceOf[Point].x == y
}
```

```
      val p1 = new Point(2, 3)
      val p2 = new Point(2, 4)
      val p3 = new Point(3, 3)

      println(p1.isNotEqual(p2))
      println(p1.isNotEqual(p3))
      println(p1.isNotEqual(2))
```

Output:



```
val p1 = new Point(2, 3)
      val p2 = new Point(2, 4)
      val p3 = new Point(3, 3)

      println(p1.isNotEqual(p2))
      println(p1.isNotEqual(p3))
      println(p1.isNotEqual(2))
```

```
true
false
true
p1: Point = Point@3465ebed
p2: Point = Point@51f6f35c
p3: Point = Point@31e073fa
```

Took: 1 second 279 milliseconds, at 2017-12-3 1:20

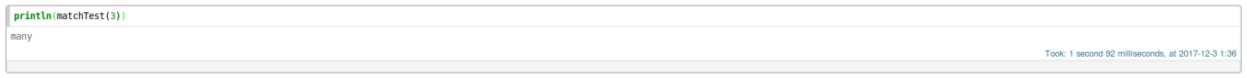
4. Scala - Pattern Matching

A pattern match includes a sequence of alternatives, each starting with the keyword case. Each alternative includes a pattern and one or more expressions, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions.

Try the following example program, which shows how to match against an integer value.

```
def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }
println(matchTest(3))
```

Output:



```
println(matchTest(3))
many
Took: 1 second 92 milliseconds, at 2017-12-3 1:36
```

Try the following, it is a simple pattern matching example using case class.

Let us first create a Case Class: The case classes are special classes that are used in pattern matching with case expressions. Syntactically, these are standard classes with a special modifier: case.

```
case class Person(name: String, age: Int)
```

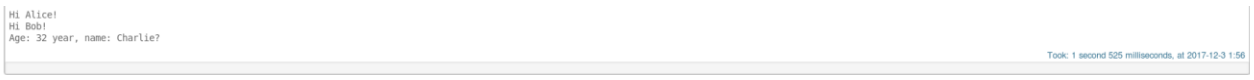
Let us create objects for the Case Class

```
val alice = new Person("Alice", 25)
val bob = new Person("Bob", 32)
val charlie = new Person("Charlie", 32)
```

Let us test the pattern matching passing different values

```
for (person <- List(alice, bob, charlie)) {
         person match {
            case Person("Alice", 25) => println("Hi Alice!")
            case Person("Bob", 32) => println("Hi Bob!")
            case Person(name, age) => println(
               "Age: " + age + " year, name: " + name + "?")
         }
      }
```

Output:



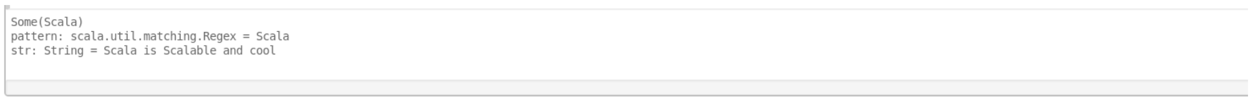
```
Hi Alice!
Hi Bob!
Age: 32 year, name: Charlie?
                                        Took: 1 second 525 milliseconds, at 2017-12-3 1:56
```

5. Scala - Regular Expressions

Try the following example program where we will try to find out word Scala from a statement.

```
val pattern = "Scala".r
val str = "Scala is Scalable and cool"

println(pattern findFirstIn str)
```

Output:



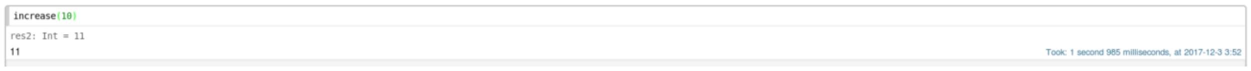
```
Some(Scala)
pattern: scala.util.matching.Regex = Scala
str: String = Scala is Scalable and cool
```

6. First-class functions

Example 1.

```
var increase = (x: Int) => x + 1
increase(10)
```

Output:

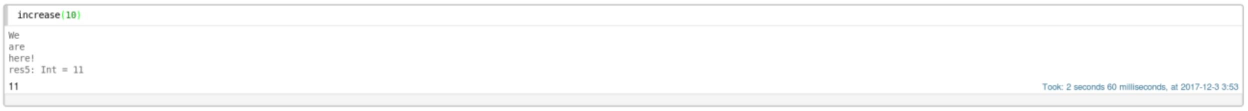


```
increase(10)
res2: Int = 11
11                                      Took: 1 second 965 milliseconds, at 2017-12-3 3:52
```

Example 2.

```
increase = (x: Int) => {
       |   println("We")
       |   println("are")
       |   println("here!")
       |   x + 1
       | }
increase(10)
```

Output:



```
increase(10)
We
are
here!
res5: Int = 11
11
```

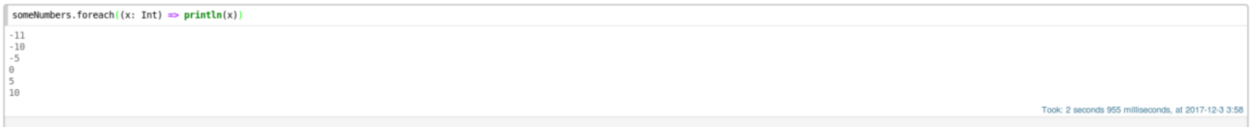
```
Example 3.
```

```
val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers.foreach((x: Int) => println(x))
```

```
This could also be called as
```

```
  someNumbers.foreach(println _)
```

```
Output:
```

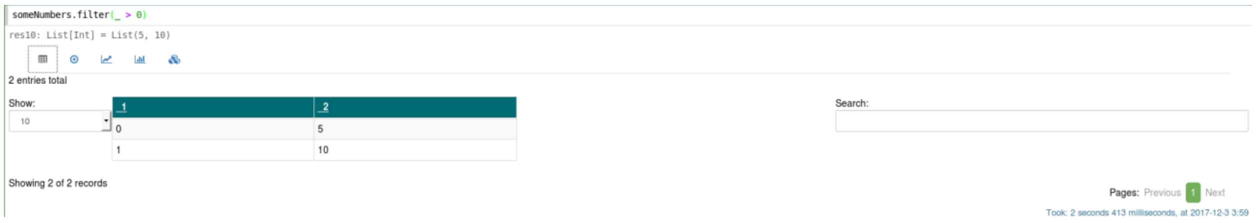


```
someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

Example 4.

```
someNumbers.filter(_ > 0)
```

Output:



```
someNumbers.filter(_ > 0)
res10: List[Int] = List(5, 10)
```

2 entries total

Show: 10

_1	_2
0	5
1	10

Showing 2 of 2 records

This is same as someNumbers.filter(x => x > 0).

7. HIGHER ORDER FUNCTIONS

Functional languages treat functions as first-class values.
This means that, like any other value, a function can be passed as a parameter and returned as a result. This provides a flexible way to compose programs. Functions that take other functions as parameters or that return functions as results are called *higher order functions*.

Let's define:

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

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)

def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)

def sumFactorials(a: Int, b: Int): Int =
  if (a > b) 0 else factorial(a) + sumFactorials(a + 1, b)
```

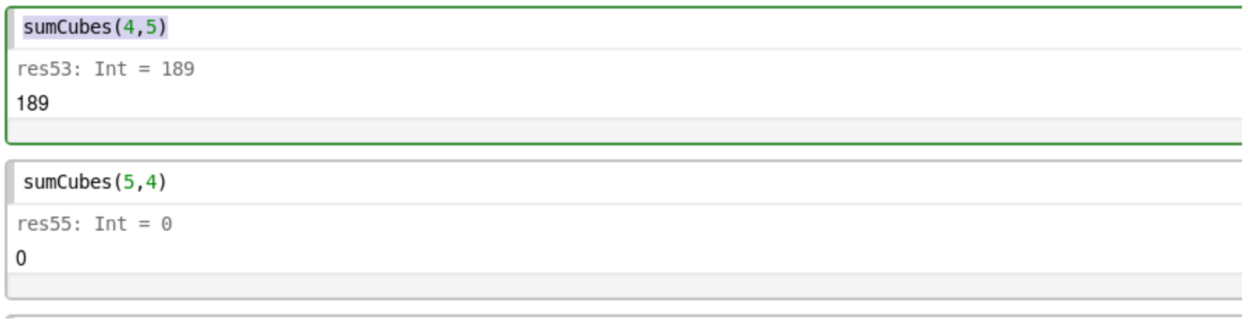
We can then write:

```
def id(x: Int): Int = x
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(factorial, a, b)
```

Output:

Try various combinations of calling the methods, like

sumCubes(4,5)



```
sumCubes(4,5)
res53: Int = 189
189
```

```
sumCubes(5,4)
res55: Int = 0
0
```