

Scala Tutorial for yourself

Basic Syntax and introduction

Start your Scala in your command line tool(For windows) or terminal(Mac/Linux)

```
scala> println("Hello World!")
```

The result will be:

```
Hello World!
```

Also there is an another way to prompt Scala code.

First open your editor or IDE such as IntelliJ Idea and creat a **HelloWorld.scala** file and types in

```
Object HelloWorld{  
    def main(args:Array[String]){  
        println("Hello,World!")//scala do not care if there is a ; line  
    }  
}
```

Open your terminal and prompt:

```
>scalac HelloWorld.scala  
>scala HelloWorld  
>Hello,World!
```

In this tutorial, I suppose that you have already known how to compile java programs and run java classes.

As you can see from the **HelloWorld.scala** file:

The **def main(args:Array[String])** is the way that scala create it functions which is very similar to python language.(I have used python for a rather long time. So the first time I saw the definition, I was amazed by this declaration, and it is also similar to Java.)

How to name your Scala variables

Every language should have their variables and the rule to name those identifiers. Scala provide a standard identification rule as Java which means you can use either a letter or

underscore to start your identifiers.

Several examples are provided:

```
year, _length, mineCraft
```

Operator Identifiers

An operator identifier consists of one or more operator characters. Like

```
+ ++ ::: <?> :>
```

Some scala keywords

Keyword	keyword	Keyword
Abstract	Case	catch
False	Do	class

Still there are many keywords in Scala.

Declaring a variable

It is quiet common to declare several variables. In Scala, the interesting thing is that you may have several ways to define a variable or multi variables.

```
var myVar : String = "Foo"//var means this myVar can be changed
val myVal : String = "Bar"//val means this myVal cannot be changed
```

Also it is valid if you do not assign a value to the variable.

```
var myVar :Int;
val myVal :String;
```

Also it is valid to write your code so:

```
var myVar = 40
val myVal = "Hello SCALA!"
val {myVar,myVal} = Pair(40,"Hello SCALA!")
```

Data Access

The general case is that Scala also provide **private**, **public** and **protected** access modifiers which is just as Java, C++ as an object-oriented language. Some examples for Java programmers:

```
class Outer{
  class Inner{
    private def In() {println("I am in the loop!")}
    class InnerMost {
      In() //output: I am in the loop.
    }
  }
  (new Inner).In() //Because it is private you cannot call this function,
this line will give you an error.
}
```

```
class Super{
  //protected means public to subclass and private to unrelated class
  protected def s() {"I am in a super class."}
}

class Sub extends Super {
  s()//works
}

class other {
  (new Super).s()//won't work
}
```

when you do not give an exact **private** or **protected** before your declaration, the default one will be **public**.

However the interesting thing is that Scala do provide a system for programmers to define different access to classes.

A modifier of the form **private[X]** or **protected[X]** means that access is private or protected "up to" class, package or object type X.

```
package physics{
  package condensedPhysics{
    class SolidStatePhysics{
      private[physics] var commonTheorems;
      private[condensedPhysics] var scientists = null;
    }
  }
}
```

```

        private[this] var problems = null;

        def help( another : SolidStatePhysics) {
            println(another.commonTheorems)
            println(another.problems) // ERROR, the problems can only be
            // accessed by the this reference
        }
    }
}

```

Scala operations

Normal Syntax:

```

Object Test{
    def main(Args : Array[])
var a = 50
var b = 30
println("a + b = "+(a+b))
println("a - b = "+(a-b))
println("a / b = "+(a/b))
println("a * b = "+(a*b))
println("a % b = "+(a%b))

```

The result is as follows:

```

> scalac Test.scala
> scala Test
> 80
> 20
> 1
> 1500
> 20

```

Other operators like `==` `!=` `>` `<` `>=` `<=` are just as same as common language syntax.

Bitwise operators are as follows:

Operator	Description(bitwise)
&	Binary AND

!	Binary OR
^	Binary XOR
~	Binary Complement
<<	Binary Left Shift
>>	Binary Right Shift

Other operators are just the same like `+= -= *= /=`

Control flow

The **if else** style are just the same as Java.

Let's just see some examples

```
Object Test{
  def main(Args:Array[String]){
    var x = 20;

    if (x>25){
      println("x is bigger than 25")
    }else println("x is smaller than 25")
  }
}
```

It is worth to note that the **else if** statement is also available.

Loop

There are three kinds of loop in Scala : **while loop**, **do while loop** and **for loop**.

While loop

```
while (condition){
  statement(s)
}
```

Do while loop

```
do{
```

```
    statement(s)
  }while (condition)
```

For loop

This is kind of a interesting style. Let's just focus on the examples.

```
for ( var x <- Range ){
    statement(s)
}
```

Range here could be a range of numbers such as **i to j** or **i until j**. The left arrow **<-** is a generator because it is generating values from the range.

```
Object Test{
  def main(Args:Array[String]){
    var l = 0;

    for ( l <- 1 to 3 ){
      println("l value : "+ l )
    }
  }
}
```

The results will be seen as:

```
>1
>2
>3
```

And if you use **i until j**:

```
Object Test{
  def main(Args:Array[String]){
    var l = 0;

    for ( l <- 1 until 3 ){
      println("l value : "+ l )
    }
  }
}
```

The results will be as:

```
>1  
>2
```

Also we can use multiple ranges such as:

```
Object Test {  
  def main(Args : Array[String]){  
    var a = 0 ;  
    var b = 0;  
    for ( a <- 1 to 3  
          b <- 1 to 3){  
      println("a + b = " + (a + b))  
    }  
  }  
}
```

For loop is also valid in Collections:

```
for (var x<- List){  
  statement;  
}
```

The List here is a collection of objects with same type.

Scala also provide filters in the for loop:

```
for (var x<-List,  
     if condition1; if condition2;....){//may have more conditions  
  statement  
}
```

For loop with yield: Scala provide a return syntax for the for loop with keyword **yield** through which you can store the list of return values.

```
var retVar = for (x<-List,  
                  if condition1; if condition2;....)yield x
```

Scala does not support break and continue statement like JAVA, however, it provides another mechanism of break statement.

```
import scala.util.control._

val loop = new Breaks; //The scala.util.control package provides a special kind of object which is Breaks

loop.breakable{//make loop always breakable
  for (...){
    statement
    ...
    loop.break//break loop
  }
}
```

Let's see an example:

```
import scala.util.control._

object Test(Args:Array[String]){
  var i = 0;
  val list = List{1,2,3,4,5,6,7,8}

  var loop = new Breaks
  loop.breakable{
    for (i <- list){
      println(i)
      if (i==4){
        loop.break
      }
    }
  }
}
```

The result will be

```
>scalac Test.scala
>scala Test
>1
>2
>3
>4
```

For nested loops, we have to create several breakable loops.

Scala functions

Function definition format:

```
def functionName ([list of parameters]) : [return type] //method create this should be abstract  
def functionName2 ([list of parameters]) : [return type] = {  
    function body  
    return expression  
}
```

let's just create a adding method.

```
def addTwoInt (a: Int, b: Int) : Int = {  
    return a+b  
}
```

The **void** return type is **Unit**.

```
def printMe() : Unit = {  
    println("Hi, I am Scala!")  
}
```

Calling functions in Scala

```
functionName([list of parameters])  
[Instance].methodName([list of parameters])
```

Scala functions are the core part of Scala language, besides from the normal syntax above. Scala functions can be called through various ways.

Functions call by name

The normal functions calls parameters by value, which means that the input parameters' values are already fixed when the function is called. However, there is another way to call parameters. What if we can call an expression with no definite values into the function. The expression's value is only invoked when the function calls it.

A call-by-name mechanism passes a code block to the calling functions and each time it calls the code block passing through a parameter, the block will be evaluated and executed.

```
object Test{
  def main(Args : Array[String]) {
    delay(time())
  }
  def time(){
    println("Getting time in nanoseconds.")
    System.nanoTime
  }
  def delayed(t :=> long){// => means call by name
    println("In delayed method")
    println("param t: "+t)
    t
  }
}
```

The result is:

```
In delayed method
Getting time in nanoseconds
param t: //(long integer)
Getting time in nanoseconds
param t: x
Getting time in nanoseconds
```

Function with variable arguments

Scala provide a similar system as Python to allow parameters passing into the function to be repeated.

```
object Test{
  def main(args:Array[String]){
    printStrings("Hello","Scala","Python")
  }

  def printStrings(Args:String*) = {//Args:String* tells that this is a list of parameters
    var i : Int = 0
    for (arg <- Args){
      println("Args["+i+"]"+arg)
      i++
    }
  }
}
```

Scala also provide default parameters values.

```
def addTwo(x:Double = 5.0,y:Double = 3.0) = {  
  var Sum:Double = 0  
  Sum = a+b  
  
  return sum  
}
```

Scala is so-called functional programming language because it can pass function as a parameter into a function.

There are also other ways to declare the functions:

```
var mul = (x:Int,y:Int) => x*y  
var useDir = () => {System.getProperty("user.dir")}
```

Scala Closures

A closure is a function that return value depends on the value of one or more variables outside the function which is kind of easy.

Scala Strings

How to create strings?

```
val greeting = "Hello World!"  
val greeting2 : String = "Hello World!"
```

String type is very similar to the String Class in Java, so the method is a lot like the Java String.(Other methods can be

Concatenating the string can use either `+` or `String.concat(String)`

Format String

The `printf` method has been a default method in Scala, which makes it quite a convenient way to print format strings.

Scala Arrays

Scala also provides the general array data structure.

```
var z:Array[String] = new Array[String](3)
var z2 = new Array[String](3)
z(0) = "zara";z(1) = "nuha";z(2) = "Ayan"
var z3 = {"zara","nuha","Ayan"}
```

Multidimensional Array

```
var matrix=new Array[Int](3,3)//3*3 matrix
```

The interesting thing of array in Scala is that arrays can concatenate each other using the following command:

```
import Array._ //important
...
List3 = concat(List1,List2)
...
```

The keyword **range** in Array package also has the same syntax as python **range**.