**Scala Overview**

* **Scala is programing language**
* **Spark is framework which has been develop on top the Scala**

**Scala** is a modern multi-paradigm programming language which is a combination of object-oriented programming and functional programming. It is highly scalable which is why it is called **Scala**. ... **Scala**language is a strong static type of programming language and is influenced by the Java programming language.

Thus, the **Scala** became the key to success for managing the huge amount of **big-data.**

Now that we know the importance of Scala, let us now understand why actually it is the most preferred language in the present trends.

**Why we need Scala?**

* Scala is capable to work with the data which is stored in a **Distributed** fashion. It accesses all the available resources and supports parallel data processing.
* Scala supports **Immutable** data and it has support to the higher order functions.
* Scala is an**upgraded version of Java** which was designed to eliminate unnecessary code. It supports multiple Libraries and APIs which will allow the programmer to achieve **Less Down Time**.
* Scala supports multiple type **Constructs** which enables the programmer to work with wrappers/container types with ease.

Var a =11; //mutable (data can be changed)

Val b =44; //imutable (readonly)

## ****Frameworks of Scala****

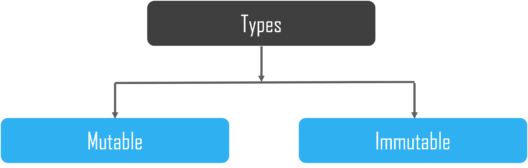
Akka, Spark, Play, Neo4j, Scalding are some of the major frameworks that Scala can support.

* **Akka**is a toolkit on runtime for building highly concurrent, distributed, and fault-tolerant applications on the JVM. **Akka** is written in Scala, with language bindings provided for both Scala and Java.
* **Spark** Framework is designed to handle, and process big-data and it solely supports **Scala**.
* **Play** framework is designed to create web applications and it uses Scala in the process in order to obtain the best in class performance.
* **Scalding** is a**domain-specific language** (**DSL**)in the Scala programming language, which integrates **Cascading**. It is a functional programming paradigm used in Scala which is much closer than Java to the original model for **MapReduce** functions.
* **Neo4j** is a **java spring framework** supported by Scala with domain-specific functionality, analytical capabilities, graph algorithms, and many more.

These were the popular Frameworks supported by Scala, Now let us understand the variables and data types in Scala.

## ****Variables in Scala****

Variables can be defined as the reserved memory locations used to store the values. Similarly, we do have variables in Scala Programming Language as well. The Variables in Scala are divided into two types.

****

**Mutable Variables**

These variables allow us to **change** a value after the declaration of a variable. **Mutable** variables are defined by using the **var** keyword. The first letter of data type should be in capital letter because in Scala data type is treated as an object.

|  |  |
| --- | --- |
| 1  2 | var b = "Test"  b = "Test 2" |

**Immutable Variable**

These variables do not allow you to change a value after the declaration of a variable. **Immutable variables** are defined by using the **val** keyword. The first letter of data type should be in capital letter because in the **Scala** data type is treated as objects.

|  |  |
| --- | --- |
| 1  2 | val a = "hello world"  a = "how are you" //error |

#### ****Lazy Evaluation****

|  |  |
| --- | --- |
| 1  2 | lazy val x = 100  x\*2 |

**output:**

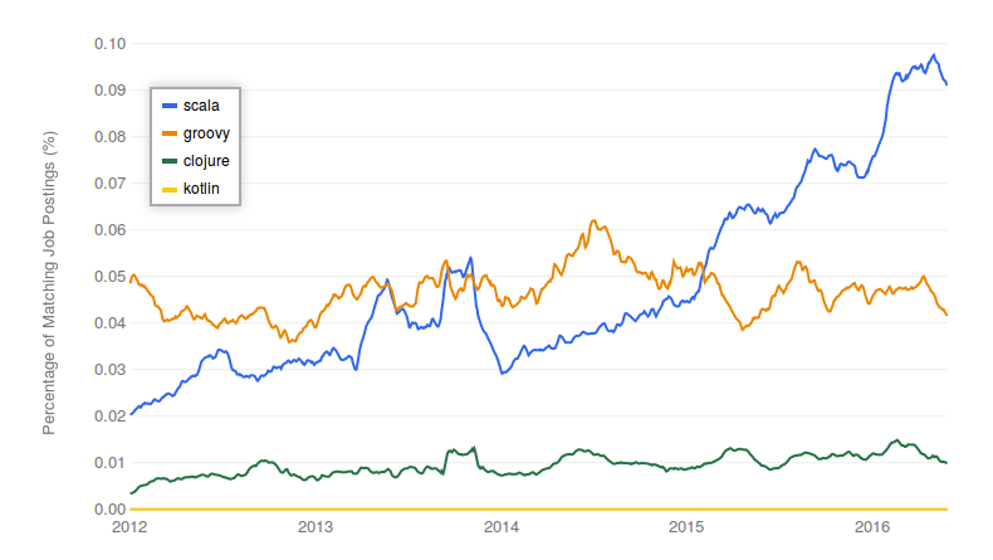
x: Int = <lazy>  
res: Int = 200

**Lazy** **Evaluation** is the primary feature of Scala which bought it the dignity of a whole new level. Here, the declared variable **will not be accessed** or **any** **operation** **is** **not** **performed** on to the variable unless the programmer particularly **accesses** it and**performs an operation** on to it.

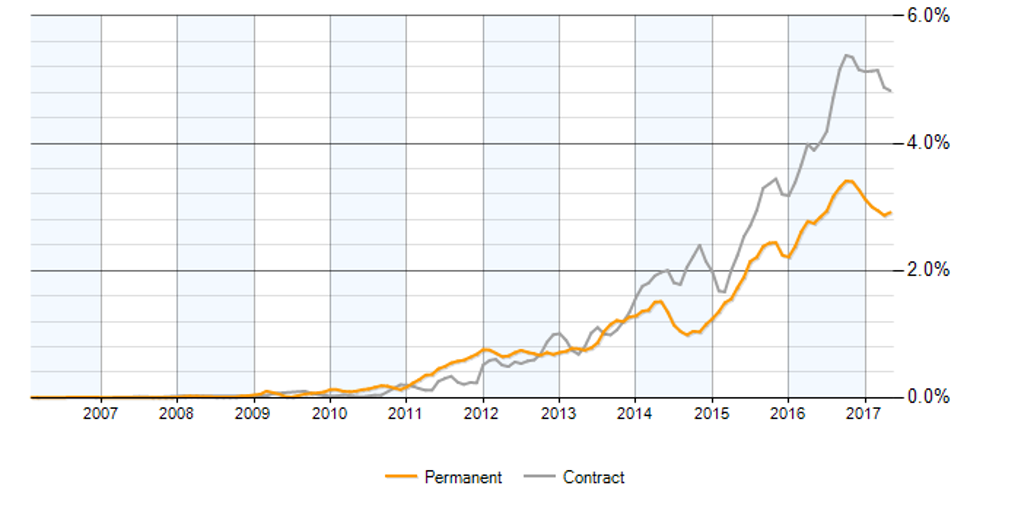
In simple words, it is an**On-Demand execution** of an operation which saves a lot of **memory** and**processing resources in real-time.**

## ****Scope for Scala****

**Scala** is the miracle of the 20th century in **multiple** **streams**. It has seen astounding growth since day one and it is for sure it is one of the programming languages which is in **higher** **demand**. The stats below explain more about the **scope of Scala** in the near future.



The **chart** below describes the**permanent jobs** and **Contract based jobs** available based on the knowledge of Scala Programming Language.

So, with this, we come to an end of this What is Scala article. I hope we sparked a little light upon your knowledge about Scala, Its features and the various types of operations that can be performed using Scala.

# Installing Scala and Spark on Ubuntu

**Step 1: Installing Java**

Check to see if Java is already installed by typing:

java -version

First, update the package index by in your terminal typing:

sudo apt-get update

Now you can install the JDK with the following command:

sudo apt-get install default-jdk

Export JAVA\_HOME=/usr/local/java-current

Export PATH=$PATH:$JAVA\_HOME/bin/

**Step 2: Install Scala**

sudo apt-get install scala

Type scala into your terminal:

scala

You should see the scala REPL running. Test it with:

println(“Hello World”)

You can then quit the Scala REPL with

:q

**Step 3: Install Spark**

Next its time to install Spark. We need git for this, so in your terminal type:

sudo apt-get install git

Next, go to https://spark.apache.org/downloads.html and download a pre-built for Hadoop 2.7 version of Spark (preferably Spark 2.0 or later). Then download the .tgz file and remember where you save it on your computer.

Then in your terminal change directory to where you saved that .tgz file (or just move the file to your home folder), then use

tar xvf spark-2.0.2-bin-hadoop2.7.tgz

(Your version numbers may differ).

Then once its done extracting the Spark folder, use:

cd spark-2.0.2-bin-hadoop2.7.tgz

then use

cd bin

and then type

./spark-shell

and you should see the spark shell pop up

this is where you can load .scala scripts. You can confirm that this is working by typing something like:

println(“Spark shell is running”)

## Using the interactive interpreter

The easiest way to execute a single line of Scala code is to use the interactive interpreter with which statements and definitions are evaluated on the fly. We start the interactive interpreter using the Scala command line tool “scala” which is located in the bin folder in the folder where Scala is installed. With the proper environment variable in place you can just type “scala” to start it. Using the interactive interpreter we can do our first Hello world by using the println method.

C:\Users\Joel>scala   
Welcome to Scala version 2.8.0.RC7 (Java HotSpot(TM) Client VM, Java 1.6.0\_20).   
Type in expressions to have them evaluated.   
Type :help for more information.

scala> println("Hello world")   
Hello world

## Executing Scala code as a script

Another way to execute Scala code is to type it into a text file and save it with a name ending with “.scala”. We can then execute that code by typing “scala filename”. For instance, we can create a file named hello.scala with our Hello world in it:

*// Inside hello.scala*

println("Hello world!")

To execute it we specify the filename as a parameter to the Scala command line tool.

Source code(.scala) -> compile -> byte code(.class) -> run

c:\learningscala>scala hello.scala   
Hello world!

c:\learningscala>

In the above script we saw an example of a comment. Just like in Java and C# single line comments begin with two forward slashes (//). Comments that span several line begins with /\* and ends with \*/.

## Compiling Scala code

Finally, we can also execute Scala code by first compiling it using the scalac command line tool. Then the code will need to be executed in the context of an application so we’ll need to add an object with a main method.

object Greeting {

def main(args: Array[String]) = println("Hello world!")

}

A = []

A: Array[Sting]

Wow, that’s more than a one line statement that writes to the console. We’ll discuss what this code does in a second. First, let’s look at how we can execute it though.

To compile it we type “scalac Greeting” which will produce two new files, Greeting$.class and Greeting.class.

c:\learningscala>scalac helloClass.scala

c:\learningscala>dir

Directory of c:\learningscala   
Greeting$.class   
Greeting.class   
hello.scala   
helloClass.scala

To execute the main method with it’s Hello world message we type “scala Greeting”.

c:\learningscala>scala Greeting   
Hello world!

c:\learningscala>

First Program

1. **object** ScalaExample{
2. **def** main(args:Array[String]){
3. println "Hello Scala"
4. }
5. }

Data Types in Scala

Data types in scala are much similar to java in terms of their storage, length, except that in scala there is no concept of primitive data types every type is an object and starts with capital letter. A table of data types is given below. You will see their uses further.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Size** |
| Boolean | False | True or false |
| Byte | 0 | 8 bit signed value (-27 to 27-1) |
| Short | 0 | 16 bit signed value(-215 to 215-1) |
| Char | '\u0000' | 16 bit unsigned Unicode character(0 to 216-1) |
| Int | 0 | 32 bit signed value(-231 to 231-1) |
| Long | 0L | 64 bit signed value(-263 to 263-1) |
| Float | 0.0F | 32 bit IEEE 754 single-precision float |
| Double | 0.0D | 64 bit IEEE 754 double-precision float |
| String | Null | A sequence of characters |

# Scala Conditional Expressions

Scala provides if statement to test the conditional expressions. It tests boolean conditional expression which can be either true or false. Scala use various types of if else statements.

* If statement
* If-else statement
* Nested if-else statement
* If-else-if ladder statement

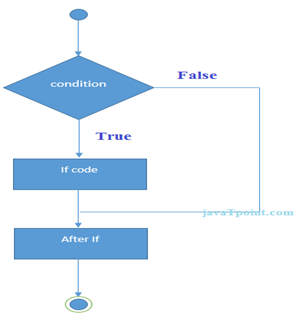
## Scala if statement

The scala if statement is used to test condition in scala. If block executes only when condition is true otherwise execution of if block is skipped.

**Syntax**

1. **if**(condition){
2. // Statements to be executed
3. }

### Flowchart



## Scala Example: If Statement

1. **var** age:**Int** = 20;
2. **if**(age > 18){
3. println ("Age is greate than 18")
4. }

Output:

Age is greate than 18

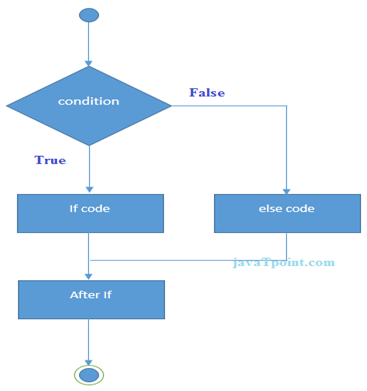
## Scala If-Else Statement

The scala if-else statement tests the condition. If the condition is true, if block executes otherwise else block executes.

**Syntax**

1. **if**(condition){
2. // If block statements to be executed
3. } **else** {
4. // Else bock statements to be executed
5. }

### Flowchart



## Scala if-else example

1. **var** number:**Int** = 21
2. **if**(number%2==0){
3. println("Even number")
4. }**else**{
5. println("Odd number")
6. }

Output:

Odd number

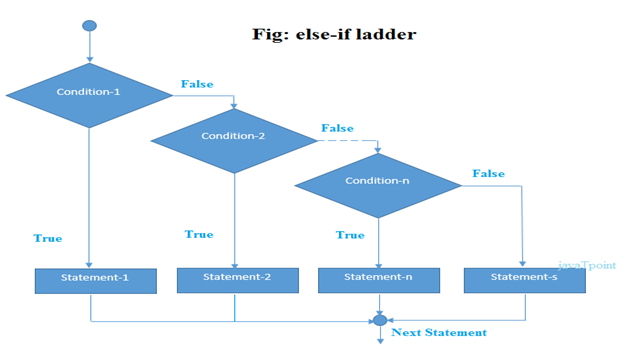
## Scala If-Else-If Ladder Statement

The scala if-else-if ladder executes one condition among the multiple conditional statements.

**Syntax**

1. **if** (condition1){
2. //Code to be executed if condition1 is true
3. } **else** **if** (condition2){
4. //Code to be executed if condition2 is true
5. } **else** **if** (condition3){
6. //Code to be executed if condition3 is true
7. }
8. ...
9. **else** {
10. //Code to be executed if all the conditions are false
11. }

### Flowchart



## Scala If-Else-If Ladder Example

1. **var** number:**Int** = 85
2. **if**(number>=0 && number<50){
3. println ("fail")
4. }
5. **else** **if**(number>=50 && number<60){
6. println("D Grade")
7. }
8. **else** **if**(number>=60 && number<70){
9. println("C Grade")
10. }
11. **else** **if**(number>=70 && number<80){
12. println("B Grade")
13. }
14. **else** **if**(number>=80 && number<90){
15. println("A Grade")
16. }
17. **else** **if**(number>=90 && number<=100){
18. println("A+ Grade")
19. }
20. **else** println ("Invalid")

Output:

A Grade

## Scala If Statement as better alternative of Ternary Operators

In scala, you can assign if statement result to a function. Scala does not have ternary operator concept like C/C++ but provides more powerful *if*which can return value. Let's see an example

**Example**

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **val** result = checkIt(-10)
4. println (result)
5. }
6. **def** checkIt (a:**Int**)  =  **if** (a >= 0) 1 **else** -1    // Passing a if expression value to function
7. }

## Scala Pattern Matching Example

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 1
4. a **match**{
5. **case** 1 => println("One")
6. **case** 2 => println("Two")
7. **case** \_ => println("No")
8. }
9. }
10. }

In the above example, we have implemented a pattern matching.

Here, match using a variable named *a*. This variable matches with best available case and prints output. Underscore (\_) is used in the last case for making it default case.

Output:

One

Match expression can return case value also. In next example, we are defining method having a match with cases for any type of data. Any is a class in scala which is a super class of all data types and deals with all type of data. Let's see an example.

## Scala Pattern Matching Example2

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = search ("Hello")
4. print(result)
5. }
6. **def** search (a:Any):Any = a **match**{
7. **case** 1  => println("One")
8. **case** "Two" => println("Two")
9. **case** "Hello" => println("Hello")
10. **case** \_ => println("No")
12. }
13. }

Scala Loop

## Scala while loop Example

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 10;                       // Initialization
4. **while**( a<=20 ){                // Condition
5. println(a);
6. a = a+2                        // Incrementation
7. }
8. }
9. }

Output:

10

12

14

16

18

20

## Scala Infinite While Loop Example

You can also create an infinite while loop. In the below program, we just passed *true* in while loop. Be careful, while using infinite loop.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 10;           // Initialization
4. **while**( **true** ){        // Condition
5. println(a);
6. a = a+2            // Incrementation
7. }
8. }
9. }

Output:

10

12

14

16

...

Ctr+Z // To stop execution

## Scala do-while loop example

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 10;         // Initialization
4. **do** {
5. println( a );
6. a = a + 2;      // Increment
7. }
8. **while**( a <= 20 )     // Condition
9. }
10. }

Output:

10

12

14

16

18

20

## Scala Infinite do-while loop

In scala, you can create infinite do-while loop. To create infinite loop just pass *true* literal in loop condition.

Let's see an example.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 10;                         // Initialization
4. **do** {
5. println( a );
6. a = a + 2;                      // Increment
7. }
8. **while**( **true**)                         // Condition
9. }
10. }

## Scala for-loop example by using *to* keyword

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **for**( a <- 1 **to** 10 ){
4. println(a);
5. }
6. }
7. }

## Scala for-loop Example by using *until* keyword

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **for**( a <- 1 **until** 10 ){
4. println(a);
5. }
6. }
7. }

## Scala for-loop filtering Example

You can use *for* to filter your data. In the below example, we are filtering our data by passing a conditional expression. This program prints only even values in the given range.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **for**( a <- 1 **to** 10 **if** a%2==0 ){
4. println(a);
5. }
6. }
7. }

## Scala for-loop Example by using *yield* keyword

In the above example, we have used *yield* keyword which returns a result after completing of loop iterations. The for use buffer internally to store iterated result and after finishing all iterations it yields the final result from that buffer. It does not work like imperative loop.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = **for**( a <- 1 **to** 10) yield a
4. **for**(i<-result){
5. println(i)
6. }
7. }
8. }

## Scala for- loop Example for Iterating Collection

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** list = List(1,2,3,4,5,6,7,8,9)          // Creating a list
4. **for**( i <- list){                         // Iterating the list
5. println(i)
6. }
8. }
9. }

## Scala for-each loop Example for Iterating Collection

In the below code we have use three approaches of for-each loop. You can implement any of them according to your need.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** list = List(1,2,3,4,5,6,7,8,9)  // Creating a list
4. list.foreach{
5. println     // Print each element
6. }
7. list.foreach(print)
8. println
9. list.foreach((element:**Int**)=>print(element+" "))      // Explicitly mentioning type of elements
10. }
11. }

## Scala for-loop Example using *by* keyword

In the above example, we have used *by* keyword. The *by* keyword is used to skip the iteration. When you code like: by 2 it means, this loop will skip all even iterations of loop.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **for**(i<-1 **to** 10 by 2){
4. println(i)
5. }
6. }
7. }

## Scala Break Example: Break inner Loop

You can use break statement to terminate execution of inner loop in nested loop.

Let's see an example.

1. **import** scala.util.control.Breaks.\_
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. **for**(i <- 1 **to** 3){
5. breakable {
6. **for**(j <- 1 **to** 3){
7. **if**(i == 2 & j == 2 )
8. break
9. println(i+" "+j)
10. }
11. }
12. }
13. }
14. }

## Scala Single Line Comment Example

Single line comment is used to comment single line of code.

1. // Example of single line comment.
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. **var** a = 1           // Here, a is a variable
5. println(a)
6. }
7. }

## Scala Multiline Comment

Multiline comment is used to comment multiple lines of code in the program.

1. // Example of multi line comment.
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. **var** a = 1
5. println(a)
6. }
7. /\*
8. In the main method, we have created a variable named a
9. and printed it
10. \*/
11. }

# Scala Functions

In scala, functions are first class values. You can store function value, pass function as an argument and return function as a value from other function. You can create function by using **def** keyword. You must mention return type of parameters while defining function and return type of a function is optional. If you don't specify return type of a function, default return type is Unit.

## Scala Function Declaration Syntax

1. **def** functionName(parameters : typeofparameters) : returntypeoffunction = {
2. // statements to be executed
3. }

## Scala Function Example without using = Operator

The function defined below is also known as non parameterized function.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. functionExample()           // Calling function
4. }
5. **def** functionExample()  {        // Defining a function
6. println("This is a simple function")
7. }
8. }

## Scala Function Example with = Operator

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = functionExample()          // Calling function
4. println(result)
5. }
6. **def** functionExample() = {       // Defining a function
7. **var** a = 10
8. a
9. }
10. }

# Scala Functions

Scala supports functional programming approach. It provides rich set of built-in functions and allows you to create user defined functions also.

In scala, functions are first class values. You can store function value, pass function as an argument and return function as a value from other function. You can create function by using **def** keyword. You must mention return type of parameters while defining function and return type of a function is optional. If you don't specify return type of a function, default return type is Unit.

## Scala Function Declaration Syntax

1. **def** functionName(parameters : typeofparameters) : returntypeoffunction = {
2. // statements to be executed
3. }

In the above syntax, = (equal) operator is looking strange but don't worry scala has defined it as:

You can create function with or without = (equal) operator. If you use it, function will return value. If you don't use it, your function will not return anything and will work like subroutine.

Scala functions don?t use return statement. Return type infers by compiler from the last expression or statement present in the function.

## Scala Function Example without using = Operator

The function defined below is also known as non parameterized function.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. functionExample()           // Calling function
4. }
5. **def** functionExample()  {        // Defining a function
6. println("This is a simple function")
7. }
8. }

Output:

This is a simple function

## Scala Function Example with = Operator

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = functionExample()          // Calling function
4. println(result)
5. }
6. **def** functionExample() = {       // Defining a function
7. **var** a = 10
8. a
9. }
10. }

Output:

10

## Scala Parameterized Function Example

when using parameterized function you must mention type of parameters explicitly otherwise compiler throws an error and your code fails to compile.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. functionExample(10,20)
4. }
5. **def** functionExample(a:**Int**, b:**Int**) = {
6. **var** c = a+b
7. println(c)
8. }
9. }

## Scala Recursion Function

In the program given below, we are multiplying two numbers by using recursive function.

In scala, you can create recursive functions also. Be careful while using recursive function. There must be a base condition to terminate program safely.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result = functionExample(15,2)
4. println(result)
5. }
6. **def** functionExample(a:**Int**, b:**Int**):**Int** = {
7. **if**(b == 0)          // Base condition
8. 0
9. **else**
10. a+functionExample(a,b-1)
11. }
12. }

## Scala Function Parameter example with default value

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result1 = functionExample(15,2)     // Calling with two values
4. **var** result2 = functionExample(15)   // Calling with one value
5. **var** result3 = functionExample()     // Calling without any value
6. println(result1+"\n"+result2+"\n"+result3)
7. }
8. **def** functionExample(a:**Int** = 0, b:**Int** = 0):**Int** = {   // Parameters with default values as 0
9. a+b
10. }
11. }

## Scala Function Named Parameter Example

In scala function, you can specify the names of parameters during calling the function. In the given example, you can notice that parameter names are passing during calling. You can pass named parameters in any order and can also pass values only.

Let's see an example.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result1 = functionExample(a = 15, b = 2)    // Parameters names are passed during call
4. **var** result2 = functionExample(b = 15, a = 2)    // Parameters order have changed during call
5. **var** result3 = functionExample(15,2)             // Only values are passed during call
6. println(result1+"\n"+result2+"\n"+result3)
7. }
8. **def** functionExample(a:**Int**, b:**Int**):**Int** = {
9. a+b
10. }
11. }

# Higher Order Functions

## Scala Example: Passing a Function as Parameter in a Function

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. functionExample(25, multiplyBy2)                   // Passing a function as parameter
4. }
5. **def** functionExample(a:**Int**, f:**Int**=>AnyVal):Unit = {
6. println(f(a))                                   // Calling that function
7. }
8. **def** multiplyBy2(a:**Int**):**Int** = {
9. a\*2
10. }
11. }

## Scala Example: Function Composition

In scala, functions can be composed from other functions. It is a process of composing in which a function represents the application of two composed functions.

Let's see an example.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result = multiplyBy2(add2(10))      // Function composition
4. println(result)
5. }
6. **def** add2(a:**Int**):**Int** = {
7. a+2
8. }
10. **def** multiplyBy2(a:**Int**):**Int** = {
11. a\*2
12. }
13. }

## Scala Anonymous (lambda) Function

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.

You can create anonymous function either by using => (rocket) or \_ (underscore) wild card in scala.

Let's see an example.

## Scala Anonymous function Example

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result1 = (a:**Int**, b:**Int**) => a+b        // Anonymous function by using => (rocket)
4. **var** result2 = (\_:**Int**)+(\_:**Int**)              // Anonymous function by using \_ (underscore) wild card
5. println(result1(10,10))
6. println(result2(10,10))
7. }
8. }

## Scala Multiline Expression

Expressions those are written in multiple lines are called multiline expression. In scala, be carefull while using multiline expressions.

The following program explains about if we break an expression into multiline, the scala compiler throw a warning message.

## Scala Multiline Expression Example

1. **def** add1(a:**Int**, b:**Int**) = {
2. a
3. +b
4. }

## Scala Example Multiline Expression

1. **object** MainObject {
2. **def** add2(a:**Int**, b:**Int**) = {
3. a+
4. b
5. }
6. **def** add3(a:**Int**, b:**Int**) = {
7. (a
8. +b)
9. }
10. **def** main(args: Array[String]) = {
11. **var** result2 = add2(10,10)
12. **var** result3 = add3(10,10)
13. println(result2+"\n"+result3)
14. }
15. }

## Scala Function Currying

In scala, method may have multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

In other words it is a technique of transforming a function that takes multiple arguments into a function that takes a single argument.

## Scala Function Currying Example

1. **object** MainObject {
2. **def** add(a:**Int**)(b:**Int**) = {
3. a+b
4. }
5. **def** main(args: Array[String]) = {
6. **var** result = add(10)(10)
7. println("10 + 10 = "+result)
8. **var** addIt = add(10)\_
9. **var** result2 = addIt(3)
10. println("10 + 3 = "+result2)
11. }
12. }

## Scala Nested Functions Example

1. **object** MainObject {
2. **def** add(a:**Int**, b:**Int**, c:**Int**) = {
3. **def** add2(x:**Int**,y:**Int**) = {
4. x+y
5. }
6. add2(a,add2(b,c))
7. }
8. **def** main(args: Array[String]) = {
9. **var** result = add(10,10,10)
10. println(result)
11. }
12. }

## Scala Example: Function with Variable Length Parameters

1. **def** add(args: **Int**\*) = {
2. **var** sum = 0;
3. **for**(a <- args) sum+=a
4. sum
5. }
6. **var** sum = add(1,2,3,4,5,6,7,8,9);
7. println(sum);