**Scala Document**

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.

Open the Scala worksheet

**Scalculator**

2+ 5;

34/2;

**Operators are methods**

(1).+(2) //> 3

**Traits**

Similar to interfaces in Java, traits are used to define object types by specifying the signature of the supported methods. Like in Java 8, Scala allows traits to be partially implemented; i.e. it is possible to define default implementations for some methods. In contrast to classes, traits may not have constructor parameters

**Notes :**

**Difference between using App trait and main method in scala**

What is the difference between

object Application extends App {

println("Hello World")

}

and

object Application {

def main(args: Array[String]): Unit = {

println("Hello World");

}

The App trait is a convenient way of creating an executable scala program. The difference to the main method altenative is (apart from the obvious syntactic differences) that the App trait uses the delayed initalization feature.

From the release notes for 2.9 (see <http://www.scala-lang.org/old/node/9483> )

**Main method in Scala**

class MyScalaClass{

def main(args: Array[String]): Unit = {

println("Hello from main of class")

}enter code here

}

Compiled from "MyScalaClass.scala"

public class MyScalaClass {

public void main(java.lang.String[]);

public MyScalaClass();

}

So it means that when the scala class is converted to java class then the main method of the scala class which in turn being converted to the main method in java class is **not static**. And hence we would not be able to run the program because JVM is not able to find the starting point in the program

But if the same code is done by using the **'object'** keyword then:

Compiling the following:

object MyScalaClass{

def main(args: Array[String]): Unit = {

println("Hello from main of object")

}

}

Decompiling the following:

javap MyScalaClass$.class

Compiled from "MyScalaClass.scala"

public final class MyScalaClass$ {

public static final MyScalaClass$ MODULE$;

public static {};

public void main(java.lang.String[]);

}

Decompiling the following

javap MyScalaClass.class

Compiled from "MyScalaClass.scala"

public final class MyScalaClass {

public static void main(java.lang.String[]);

}

So here we got**public static void main in MyScalaClass.class** therefore the main method can be executed directly by the JVM here.

**Anonymous Functions**

Anonymous functions in Scala are of the following forms:

(x: Int) => x \* x //type is: Int => Int, e.g. gets an Int and returns an Int

(x: Int, y: Int) => x + y //type is: (Int, Int) => Int, e.g. gets 2 Ints and returns an Int

Which is basically a "syntactic sugar" for this form:

new Function1[Int ,Int] {

def apply(x: Int): Int = x \* x

}

new Function2[Int ,Int ,Int] {

def apply(x: Int, y: Int): Int = x + y

}

**Type**

The type of the 2nd anonymous function is (Int, Int) => Int and reads: "A function that map from two integers (Int, Int) to (=>) an integer (Int)"

The method doWithOneAndTwo expects a parameter of that type, so we can pass (x ,y) => x + y as a parameter.

**Parameter type inference**

Note that we were able to drop the type declarations for x and y here, becase the compiler already "knows" that doWithOneAndTwo expects a function that gets 2 Int parameters, therefore we can omit the type information for the parameters x and y in the second call in the example on the left.

//a method that requires a function as a parameter

//the function's type is (Int,Int) => Int

//e.g. maps from 2 Ints to an Int

def doWithOneAndTwo(f: (Int, Int) => Int) = {

f(1, 2)

}

//Explicit type declaration

val call1 = doWithOneAndTwo((x: Int, y: Int) => x + y)

//The compiler expects 2 ints so x and y types are inferred

val call2 = doWithOneAndTwo((x, y) => x + y)

//Even more concise syntax

val call3 = doWithOneAndTwo(\_ + \_)

println(call1, call2, call3)

* The first example is a method definition as we've seen before
* The second, is like the previous slide, only assigned to a val, this is very, very roughly like the difference between

function foo(x, y) {

return x + y;

}

and

var foo = function(x, y) {

return x + y;

}

in JavaScript. - The third, was briefly demonstrated in the previous slide, uses the shorter \_ placeholder syntax. However the usage on the left is rare in Scala, the \_ notation for anonymous functions is mostly useful when passing them as parameter to higher order functions (functions that recieve or return other functions)

def add1(x:Int, y:Int) = x + y //method

val add2 = (x:Int, y:Int) => x + y //anonymous function

val add3:(Int,Int)=>Int = \_ + \_ //alternate way

val add4 = (\_ + \_):(Int,Int)=>Int //alternate way, rare

println(add1(42,13))

println(add2(42,13))

println(add3(42,13))

println(add4(42,13))

http://scalatutorials.com/tour/interactive\_tour\_of\_scala\_anonymous\_functions.html

**Apply Function in Scala**

Mathematicians have their own little funny ways, so instead of saying "then we call function fpassing it x as a parameter" as we programmers would say, they talk about "applying function f to its argument x".

In mathematics and computer science, Apply is a function that applies functions to arguments.  
[*Wikipedia*](http://en.wikipedia.org/wiki/Apply)

apply serves the purpose of closing the gap between Object-Oriented and Functional paradigms in Scala. Every function in Scala can be represented as an object. Every function also has an OO type: for instance, a function that takes an Int parameter and returns an Int will have OO type of Function1[Int,Int].

// define a function in scala

(x:Int) => x + 1

// assign an object representing the function to a variable

val f = (x:Int) => x + 1

Since everything is an object in Scala f can now be treated as a reference to Function1[Int,Int]object. For example, we can call toString method inherited from Any, that would have been impossible for a pure function, because functions don't have methods:

f.toString

Or we could define another Function1[Int,Int] object by calling compose method on f and chaining two different functions together:

val f2 = f.compose((x:Int) => x - 1)

Now if we want to actually execute the function, or as mathematician say "apply a function to its arguments" we would call the apply method on the Function1[Int,Int] object:

f2.apply(2)

Writing f.apply(args) every time you want to execute a function represented as an object is the Object-Oriented way, but would add a lot of clutter to the code without adding much additional information and it would be nice to be able to use more standard notation, such as f(args). That's where Scala compiler steps in and whenever we have a reference f to a function object and write f (args) to apply arguments to the represented function the compiler silently expands f (args) to the object method call f.apply (args).

Every function in Scala can be treated as an object and it works the other way too - every object can be treated as a function, provided it has the apply method. Such objects can be used in the function notation:

// we will be able to use this object as a function, as well as an object

object Foo {

var y = 5

def apply (x: Int) = x + y

}

Foo (1) // using Foo object in function notation

There are many usage cases when we would want to treat an object as a function. The most common scenario is a [factory pattern](http://en.wikipedia.org/wiki/Factory_method_pattern). Instead of adding clutter to the code using a factory method we can apply object to a set of arguments to create a new instance of an associated class:

List(1,2,3) // same as List.apply(1,2,3) but less clutter, functional notation

// the way the factory method invocation would have looked

// in other languages with OO notation - needless clutter

List.instanceOf(1,2,3)

So apply method is just a handy way of closing the gap between functions and objects in Scala.

**++++++++++++++++++++++++++++++++++++++++++++++++++++++++**

Worksheet Programs

Asserts:

Scala Test makes three assertions available by default in any style trait. You can use:

* assert for general assertions;
* assertResult to differentiate expected from actual values;
* intercept to ensure a bit of code throws an expected exception.

In any Scala program, you can write assertions by invoking assert and passing in a Boolean expression:

val left = 2

val right = 1

assert(left == right)

java.lang.AssertionError: assertion failed

at scala.Predef$.assert(Predef.scala:151)

... 53 elided

val left = 2

val right = 1

assert(left == right)

If the passed expression is true, assert will return normally. If false, Scala's assert will complete abruptly with an AssertionError. This behavior is provided by the assert method defined in object Predef, whose members are implicitly imported into every Scala source file.

Val and Var :

Scala allows one to decide whether a variable is immutable or mutable. Immutable is read-only whereas mutable is read-write. Immutable variables are declared with the keyword val.

val age:Int = 22

age = 23 // throws error => error: reassignment to val

age = 22 // throws error => error: reassignment to val

Mutable variables are declared with the keyword var. Unlike val, var can be reassigned to different values or point to different objects. But they have to be initialised at the time of declaration

var age:Int = 22

age = 35 //age will be 35

Remember that val does not lock down the internal state of the variable, only its assignment. Let us consider an Array being declared as val.

val stringArray:Array[String] = new Array(6)

The stringArray can be modified, but the reference cannot be modified to point to another Array.

For example,

stringArray = new Array(33) // will result in a reassignment to val error, but

stringArray(3) = "foo" //will not result in any error.

Classes:

Classes in Scala are static templates that can be instantiated into many objects at runtime. Classes in Scala are parameterized with constructor arguments. Classes are instantiated with the *new* primitive.

You can define class with var or val parameters. val parameters in class definition **define getter**:

class ClassWithValParameter(val name: String)

val aClass = new ClassWithValParameter("Gandalf")

aClass.name // Gandalf will be printed

var parameters in class definition **define getter and setter**:

class ClassWithVarParameter(var description: String)

val aClass = new ClassWithVarParameter("Flying character")

aClass.description = "Flying white character"

aClass.name // Flying white character will be printed

You can define class with private fields:

class ClassWithPrivateFields(name: String)

val aClass = new ClassWithPrivateFields("name")

aClass.name // Compilation error . name cannot be accessed