SCALA

(References: Programming in Scala, 2nd edition , Martin Odersky, Lex Spoon, Bill Venners)

Every entity is an object and every action is a function/method
Scala implements the features of object/functional programming languages derived from ML languages (Caml, OCaml etc)
Type Inference : We don't need to declare types while declaring variables. They are inferred automatically based on literals by the runtime

val str = "hello"
var i = 2

Two types of variables:
val : Equivalent to java's final variable. Variables are to be defined at the time of declaration. Once defined, remain unchanged throughout the execution of program
var : Equivalent to non-final variables of java. Can be changed throughout the execution of the program

General programming tip(not specific to scala): Try to program exclusively with val and avoid var . Advantages of this can be seen in a very large program implemented in a single function and vars are modified at many places

Variables must be defined at the time of declaration

Inferred type variable declaration:
val x = 10
Explicit type variable declaration:
val x:Int =10

Primitive types of java are implemented by scala specific Classes
Datatypes:

Java	Scala
int	scala.Int
boolean	scala.Boolean
float	scala.Float
double	scala.Double
char	scala.Char
java.lang.String	java.lang.String
byte	scala.Byte
short	scala.Short
long	scala.Long
object	scala.Any
date	
-	scala.Symbol

Note: Strings in scala are implemented by java.lang.String
Byte,Short,Int,Long,Char are called integral types. Integral + Float,Double are called numeric types

scala.Any is the superclass of every other class and scala.Nothing is the subclass of every other class
scala.AnyVal is the superclass of all built-in value types(Byte,Short,Int,Long,Boolean,Char,Float,Double,Unit). All built-in value classes are declared as abstract, final.So cannot be instantiated.
scala.AnyRef is the super class of all reference classes.scala.AnyRef is just an alias for java.lang.Object.
Scala classes also inherit from special market trait called ScalaObject

scala.Nothing is the subtype of every other class. scala.Null is the subtype of all reference classes.

java.lang types are visible with their simple names in scala
i.e, we can simply write val s:String = “hi”
Scala implicitly imports packages java.lang , scala, Predef into every source file.
Scala compiles to java bytecodes and loads via class loader

Note:Statement termination with ‘;’ is optional in scala. However, multiple statements in a single line must be seperated by ;

$iw is the enclosing object of scala command line interpreter

Semicolon inference:
It is a set of rules to determine the end of line
Typically, a line ending is treated as a semi-colon, unless

i. It ends in a word that can’t be a legal end of line like period or infix operators
ii. Next line begins with a word that can’t be legal start of line
iii. Line ends in a parentheses() or square brackets[]

Functions:
Functions are declared with **def** keyword

```
def foo(s:String) : String = {
    return s.toUpperCase()
    }
```

return type in java is called as result type in scala

return keyword is optional. Last computed value in a function is automatically returned.
For a single line functions, curly braces are **optional**
return type declaration is **optional for non-recursive functions**
= sign before function body is **optional**
Function parameter type declaration is **mandatory**
Function parameters are vals. We can’t change its values inside the function.
Note: When a function returns nothing, it is suggested to omit = and put curly braces across function body as well to differentiate a function which returns something and not. A function which returns nothing can be called a procedure

function foo can be re-written as

```
def foo(s:String) = s.toUpperCase()
```

Note:If a method takes only one parameter, we can call it without a ‘.’ and parentheses() . E.g, 1 + 2 is transformed into a method call 1.+(2)
This works only if you explicitly specify the receiver of the method call
println 10 doesn’t work, but Console println 10 works

There are no increment/decrement unary operators (i++, ++i, i-- , --i) in scala.
Same can be ported in scala as short-hand increment/decrement operators
i+=1
i-=1

Scala's functions which returns nothing actually returns Unit type. Scala's Unit type is analogous to Java's void type

def foo(s:String):Unit = println(s)
A function delcared as returning Unit or procedure(with curly braces and no = sign) is treated as a Unit returning function by scala compiler irrespective of the function body returning any element.
Scala's compiler can convert any type to Unit

Note: Comments in scala : // -single line comments, /*....*/ - multi line comments

Identifiers:
Alphanumeric: Starts with a letter or underscore followed by letters,digits or underscore. $ charecter also counts as a letter, but shouldn't be used and it is used by scala compiler
Operator Identifiers: +, -, *, > -> etc Scala converts -> into $minus$greater
Mixed Identifiers: Alphanumeric followed by an underscore and an operator identifier e.g, unary_+ myident_*
Literal Identifiers: String enclosed in back ticks

```
E.g, val `x` =10
     `x` //prints 10
     x //prints 10
```

Any varialb/function name can be accessed using literal notation (enclosing in back ticks)

```
E.g, val x =10
     x //prints 10
     `x` //prints 10
```

Arrays:
Array's indexes are zero-based and are accessed by parentheses (0) rather than square brackets [0]
Arrays are mutable sequence of objects. i.e, though length of arrary can't be changed, its elements can be changed anytime

```
val arr:Array[Int]    = Array(1,2,3)
          Or
val arr    = Array(1,2,3) // equivalent to val arr = Array.apply(1,2,3)
          Or
val arr    = new Array[Int](3)
arr(0) =1 /* gets transformed to arr.apply(0). Not restricted to arrays but any application of objects to
          parentheses will result in apply method call. When an assignment is made, it is
          transformed to update method call*/
arr(1) =2
arr(2) = 3
println(arr(0))
println(arr(1))
```

So, val x = arr(i) will be transformed to val x = arr.apply(i)
arr(i) =10 will be transformed to arr.update(i,10)

Control Structures:
Built-in control structures: if, while,do-while for, try, match and function calls
All control structures return some value which can be used for further processing
Note: There's no ternary operator in scala. Ternary operation can be easily implemented by if-else block
max = (x > y) ? x : y (Not available in scala)

Same can be ported in scala as
max = If(x > y) x else y

if-else: returns some value
val max = if(x>y) x else y //returns either x or y based on condition
While:don't return any value. Best to avoid them.

```
val arr:Array[Int]    = Array(1,2,3)
var i =0
while(i<arr.length) {
   println(arr(i))
   i+=1
}
```

foreach

arr.foreach(**function literal**)
A function literal is like a lambda expression in c#.net
A function literal can also be said as an anonymous function. The portion to the left of => contains parameter declaration and to the right contains function body

```
val arr = Array(1,2,3)
arr.foreach(a =>println(a))
```

Or

```
arr.foreach((a:Int) => println(a))
```

Or

```
arr.foreach((a:Int) => {println(a)})
```

Or

arr.foreach(println) (this is called a partially applied function)

for

```
val arr = Array(1,2,3)
for(a <- arr) println(a)    //a <-arr syntax is called generator
```

Or

for(i <- 0 **to** arr.length -1) println(arr(i)) (note that 0 to n is **inclusive** set of both 0 and n)

Or

```
for(i <- 0.to(arr.length-1)) println(arr(i))
```

Or

for(i<- 0 **until** arr.length) println(println(i)) (**exclusive set ; 0 to arr.length -1**)
Entity to the left of <- represents a val for each element in arr .<- can be called as 'in'

-filtering

```
val arr = Array(1,2,3)
for(a<-arr if a==1) println(a)
For(a <-arr
      if a==1
      if a==2
      ) println(a)
```

-Nested Iteration

```
val arr1 = Array(1,2,3)
val arr2 = Array(4,5,6)
for(a1 <- arr1 if a1 == 1)
for(a2 <-arr2 if a2 == 4)
println(a1 + " " + a2)
```

```
val arr1 = Array(1,2,3)
val arr2 = Array(4,5,6)
for(
      a1 <- arr1 if a1 == 1;
      a2 <- arr2 if a2 == 4
     ) println(a1 + " " +a2)
```

Or

```
for {
      a1 <- arr1 if a1 ==1
      a2 <- arr2 if a2 ==4
     }println(a1 + " " +a2)
```

-MidStream variable binding

```
val arr1 = Array(1,2,3)
val arr2 = Array(4,5,6)
for(
     a1 <- arr1 if a1 ==1;
     a2 <-arr2
     a3 = a2 +1 if a3 = 5
     ) println(a1 + " "+ a2 +" " +a3)
```

-for yield

```
val arr = Array(1,2,3)
val arrInt = for(a <-arr) yield a

println(arrInt) //yield returns Array[Int] for
                //above exaple
```

A variable when declared with val can't be re-assigned, but the object it is pointing to can potentially be changed over time, i.e, individual elements of an array/list can be changed
E.g val arr = Array(1,2,3)
arr(0) = 10 //valid
arr(3) =30 //invalid ArrayIndexOutOfBoundsException

Exception Handling:
Concept remains same as in Java except the syntax

```
try {
   a = 1
} catch {
   case ex:Exception => { println(ex); val x =10}
} finally {
    Dosomething()
}
```

try-catch can return values:

```
val x = try{1} catch{2} finally{3}
println(x) // 1. Java returns 3
```

Match:
Same as switch but few differences. Can result in a value

```
val a = 10
val b = match {
  case 1 => {10}
  case 2 => {20}
  - => {100} //underscore(_) implies default case
// - =>    cases can be empty as well
}
```

breaks are implied
no follow-through from one case to another

There're no break and continue statements in scala. Replace continue with an if statement and break with a boolean variable
scala.util.control.Breaks._ provides break to exit from breakable code

E.g breakable{
    while(true) break;
    }

Variable scope:

It remains same as in java, except that a variable in inner block can have same name as that of outer block and outer block requies semi-colon

```
a =1;
{
    a = 2
}
```

Lists:

Lists are immutable , bounds-free sequence of same type of objects
Java's lists are mutable
ListBuffer is a mutable list

```
val lst = List(1,2,3)
        Or
val lst:List[Int] = List(1,2,3)

val lst:List[Int] = new List[Int](3) //error. List is an abstract type
```

Concatenation:

```
val oneTwo = List(1,2)
val threeFour = List(3,4)
val oneTwoThreeFour = oneTwo ::: threeFour
```

:: is called as cons. It prepends a new element to the beginning of the list

```
val lst =List(1,2)
val newList = 0 :: lst // 0,1,2. Transformed to lst.::(0)
```

Note: If a method is used in operator notation viz., a + b, method is invoked on left operand i.e, a.+(b) .
If a method name ends in a colon(:), it is invoked on right operand
Empty list can be specified with a Nil val lst = Nil
To add any element to an empty list, we can prepend to it. i.e, lst = 1::Nil

Tuples:

Tuples are immutable sequence of different types of objects
To return, multiple objects from a function, we can use tuples
Individual elements in a tuple can be accessed with a dot,underscore ._
Tuples' indexes are one-based

```
val tup = (1,"str") //Tuple2[Int,String]
tup._1 // 1
tup._2 // str

val tup =(1,"str",'c') //Tuple3[Int,String,Char]
```

We can define upto Tuple22

Sets And Maps

Scala provides both mutable and immutable sets and maps.Default is immutable

Sets are immutable ,bounds-free sequence of objects

var s = Set("str1","str2")
s+="str3" // += is error-free only when s is declared as var and not val. In a mutable set += adds the element to itself, whereas in an immutable set += creates a new set and returns

For mutable set, we need to import
import scala.collection.mutable.Set
val s = Set("str1","str2")
s+="str3"

Maps:
Maps are key/value pair sequence of objects
Scala provides both mutable & immutable maps. Immutable is the default map

import scala.collection.mutable.Map
val m = Map[Int,String]()
m +=(1 -> "One")
m+=(2->"Two")
println(m(2))

Or

val m = Map(1->"One",2->"Two")

Note: Prefer vals, immutable objects and methods without side effects

Classes & Objects
A class is blueprint for objects which contains data to hold state and method to compute/perform actions on that data

class Test { var x =10 def sum(n:int) = x+n; }	val t = new Test t.x =20 t = new Test //compilation error //since val	class Test //valid class

Members of a class can either be public/private/protected.Default access level is public. **To declare public variables, we mustn't put any access modifiers**
Unlike java, filenames needn't be named after classes
Scala classes can't have any static members.Static members are compensated by Singleton Objects
Constructors:

class Test(n:Int) { .... } //n is called **class parameter** //compiler creates a primary constructor with n change its values inside the class //as parameter	val t = new Test //error val t = new Test(2) // no error t.n **//error class parameter is not a member of class** **class parameters are vals. So, we can't change its values inside the class**

To access class parameters in objects other than current, we need to make them into fields
this refers to the current, self-referenced object on which field functions are invoked
Class parameters are accessible only in the class where they are defined and not accessible outside the definition of class. To access class parameters we can use one of below

```
class Test(n:Int) {
   val n1 =n
   .....
   .....
}

val t = new Test
t.n //error
t.n1 // no error
```

Parametric fields (shorthand for left)

```
class Test(val n:Int) {
   .....
   .....
}

val t = new Test
t.n //no error
```

Parametric fields can be either val or var and can also have access modifiers(private|protected|override)

```
class Test(private var n:Int) {
   .....
   .....
}

class Test(override var n:Int) {
   .....
   .....
}
```

Note:

Any statements written inside the class body are invoked at the time of object creation

```
class Test {
println("object created")
}
```

```
val t = new Test // prints object created
```

Precondition is a constraint put on the values passed to the constructor of the object. They are defined in Predef package

```
class Test(n:Int) {
      require(n!=0)
      def foo = n
}
```

require is defined in scala.Predef package

Auxilary constructor:

Every constructor other than primary constructor are called auxilary constructors.

Every auxilary constructor must call either primary constructor or other auxilary constructors that comes texually before as its first statement

Every constructor invocation ends up calling primary constructor.Thus, **primary constructor is the single point of entry of a class**

Auxilay constructor can be defined using **def this**

```
class Foo(m:Int, n:Int) {
def this(m:Int) = this(m,1) //calling primary
def this() = this(1) //calling first auxilary
}
```

```
val f = new Foo(1,2)
val f1 = new Foo(3)
val f2 = new Foo()
```

Method Overloading:

```
class Foo {
def Add(m:Int, n:Int) = m +n
def Add(m:String n:String) = m+n
}
```

```
val f = new Foo
      f.Add(1,2) //3
f.Add("str1","str2") //str1str2
```

Abstract Classes

Classes which contains atleast one method whose definition is not provided and must be provided by classes extending it

Abstract classes can't be instantiated

Methods whose definitions are not provided are considered to be abstract implicitly and shouldn't be preceded by abstract keyword where such classes must be declared abstract

abstract class Figure { def name = "Figure" def area():Float def perimeter():Float } class square(val side:Float) extends Figure { override def name ="Square" override def area() = side * side override def perimeter() = 4* side }	val f = new Figure //error val f:Figure = new Square(3) f.name //Square f.area() // 9.0 f perimeter //12

Traits

√ Traits have significant differences compared to interfaces in java.However, they get compiled to interfaces.

√ Traits can have definitions for few or all of methods declared(traits can't be considered to be abstract)

√ Traits can't be instantiated

√ Traits are mixed in by classes using **extends, with**

trait Figure { def name = "Figure" def area():Float def perimeter():Float } trait Quadrilateral { def name = "Quadrilateral" def noOfSides = 4 } class Square(val side:Float) extends Figure with Quadrilateral { override def name = "Square" override def area() = side * side override def perimeter = 4 * side }	val f = new Figure //error val f:Figure = new Square(3) f.name //Square f.area() //9.0 f.perimeter() //12 f.noOfSides() //error val q:Quadrilateral = new Square(3) q.name //Square q.area() //error q.perimeter() //error q.noOfSides //4 val s = new Square(4) s name //Square s area /16 s perimeter //16 s noOfSides //4
trait Figure { def name ="Figure" def area():Float def perimeter():Float }	val f = new Figure //error val f:Figure = new Square(3) f.name //Square f.area() //9.0 f.perimeter() //12

trait Quadrilateral extends Figure { override def name ="Quadrilateral" def noOfSides = 4 } class Square(val side:Float) extends Quadrilateral { override def name ="Square" override def area() = side * side override def perimeter() = 4 * side }	f.noOfSides() //error val q:Quadrilateral = new Square(3) q.name //Square q.area() q.perimeter() q.noOfSides //4 val s = new Square(4) s name //Square s area /16 s perimeter //16 s noOfSides //4

√ **Mixin traits:** Traits that help to easily modify the behaviour of a single class are called mixin traits

Traits can themselves extend concrete classes, abstract classes, other traits. So

(i) traits can only be mixed into a class that also extends those classes|traits extended by this trait

(ii)methods in such traits can call parent's methods with super keyword. Such methods are to be declared abstract override. These are called **stackable modifications**

In case of stackable modifications, lineraizaion rules define the order of super call

Implicit Conversions:

Singleton Objects:

Singleton Objects can be declared with keyword object.

Singleton Objects can have variables or methods as its members.

Singleton Objects can have both private and public fields

Public fields in a single objects compensates for static fields

object Test { private val xpri =10 val xpub =20 private FooPri() {println("Foo Private")} private FooPub() {println("Foo Public")}	Test.xpub // 20 Test.xpri //error Test.FooPub() // Foo Public Test.FooPri() // error

Note:Singleton objects can exist individually called stand-alone objects or co-exist with a class

Singleton objects extend a superclass and can mix-in traits

When co-exist with a class:

a) Both the class and object should be defined in one source file

b) Object is said to be **comanion object** of class and class is said to be **companion class** of object

c) **Companion classes|Companion objects can access both private and public fields of each other**

d) Singleton objects are initialized when first time some code accesses it(Just like java static)

e) Each singleton object is implemented as an instance of a **synthetic class(object name followed by $ sign, for above example synthetic class generated is Test$)**

f) Standalone objects can be used as an entry point to a scala application

g) Standalone objects with a main method that takes one parameter Array[String] and result type as Unit can be used as an entry point to a scala applciation or we can simple extend Application

```
object ScalaApp {
    def main (args:Array[String]) {
        .....
        .....
    }
```

```
object scalaApp extends Application {
  .....
  .....
}
```

Drawbacks:

1. Can't use command line arguments
2. Can't be used for multi-threaded apps

Types,Literals,Operators:

Integer Literals:

Preceded by

:0x or 0X represents hex numbers

val a = 0x1AB

:0 represents octal numbers

val o = 023

:nothing represents decimal numbers

val d = 23

Floating point Literals:

Succeeded by

:f or F represents float

val v1 = 2.3f

val v2 = 2.5F

val v3 = 2.34e1

val v4 = 2.456E1

:nothing or :d or D represents double

val d1 = 2.4

val d2 = 2.3345E1D

Charecter Literals:

Preceeded by

:nothing represents unicode charecter

val c = 'A'

:\ represents octal/hex charecter

val c1 = '\101'

:\u 4-digit unicode charecter code in hex

val c = '\u0041'

String Literals:

val s ="string"

Raw strings(scala specific):

val s = “””Anything written between three quote’s
is taken as is””” //prints spaces as well
To avaoid spaces use pipe | on each line and call stripMargin at the end
val s = “””|Anything written between three quote’s
|is taken as is”””.stripMargin

Symbol Literals:
Declared by ‘ident where ident can be any alphanumeric identifier
val s =’symbol // Invokes Symbol(“symbol”)
s.name // symbol
println(s + “New”) // ‘symbolNew
A symbol has no properties apart from name

Boolean Literals:
Can either be true or false

Operators:
All operators are methods
Infix : a+b, a-b, a*b, a/b, a%b etc. Invokes a.+(b)
Prefix: -a, +a, !a, ~a Ivokes a.Unary_-
Postfix: s toLowerCase etc
() after methods are required only in case of methods executed for side-effects(no result type) e.g, println()

Arithmetic Operaations:
Logical Operations:
Bitwise Operations:

Equality of Objects:
Unlike java, which compares equality of reference types, scala’s == can be used to compare objects and its elements
1 == 1 //true
1 == 2 //false
1 == 1.0 //true
List(1,2,3) == List(1,2,3) //true
1 == null //false
null == 1 //false
null == null //true

Operator precedence and associativity:

For each basic types, there are also additional operations possible provided by its corresponding rich wrapper.

Rich Wrapper Classes:

Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Long	scala.runtime.RichLong
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble

Boolean	scala.runtime.RichBoolean
String	scala.collection.immutable.StringOps

Functional Objects:
Objects that are immutable (do not have any mutable state) are called functional objects

Functions & Closures:
Different types of functions:

1. **Methods:** Functions which are defined as members of an object/class

2. **Local Functions:** Functions within functions. They are visible only in their enclosing blocks

```
def PrintIfOdd(n:Int) = {
   def Print() {println(n)}
   if(n%2 !=0) Print()
}
```

```
PrintIfOdd(3) //3
//Local functions can access the
//parameters of enclosing function
```

3. **First class functions:** Functions defined as literals and can be passed around as values
   Function literals exist in source files and **function values** are the corresponding runtime objects

val f = (x:Int, y:Int) => x+y	f(2,3) //5 f(1,2) //3
val f = (x:Int,y:Int) => { println(x) println(y) x + y }	f(2,3) // 2 // 3 // 5
def f = (x:Int,y:Int) => x+y	f(2,3) //5
val arr = Array(1,2,3) arr.foreach((a:Int) =>println(a)) Or arr.foreach(a => println(a)) //short form Or arr.foreach(println _) //placeholder notation arr.foreach(println) //partially applied functions	val arr = Array(1,2,3) arr.filter((x:Int) => x==1) Or arr.filter(x => x==1) //shorthand notation Or arr.filter(_==1) //placeholder notation **Note: In placeholder notation, left part of => is omitted and only right part is mentioned**
Placeholder noation: val f = (_:Int) + (_:Int) //Equivalent of writing val f = (x:Int,y:Int) =>x+y //Parameter type declaration with parentheses is //must **Note: In placeholder notation, left part of => is omitted and only right part is mentioned** **Underscores(_) simply mean that they are taken from arguments in the order they are passed** val f = (_:Array[Int]).foreach(println) f(Array(1.2,3))	**Partially applied functions:** Function value created from an existing function/function literal without specifying all or some parameters val s = (x:Int,y:Int,z:Int) => x+y+z val pf =s(2,_:Int,3)// not specifying one param pf(1) //6 val pf = sum _ //not specifying any param pf(1,3,4) //8 **Note:Missing parameter in partially applied function must match with the corresponding parameter of function from which it is created** **val pf =s(1,_:Float,3) //error**

Closures:
val f = (x:Int) => x + somevariable

Above expression is called a closure where a function literal uses a value(somevariable) not defined as a parameter. Such variables are called free variables.

x -bound variable
somevariable - free variable

For above expression to work, all free variables must be within the scope of the function literal.

4.Higher Order Functions:

Functions that take functions as parameters

```
val f = (x:Int, func:(Int =>Int)) => x + func(x)
val g = (_:Int) *2
f(3,g) //9
Now I've a requirement that g has to be multiplied by 3 instead of 2 for few cases
Then I can simply define new function g1 as below and f remains unchanged
val g1 = (_:Int) *3
f(3,g1) //12
Or simply
f(2, (_:Int)*4) //10
        Or
f(2, _ *4) //10
```

Currying:

Splitting a list of parameters in a function into individual parameters is called currying.
When curried, each function call with a parameter returns a function formed with remaining parameters

```
def sum(x:Int,y:Int) = x+y
Above function when curried
def sum(x:Int)(y:Int) = x+y
sum(1,2) //3
sum(1)_ // (Int => Int) = <function1>
Which is equivalent to writing
def sum(x:Int) = (y:Int) => x+y
sum(1) //(Int =>Int) =<function1>
```

Special function calls:

Repeated parameters:

The last parameter of a function may be repated using repeated parameters .Like paramArray in C#.net

```
def foo(arr:String *) {
   arr.foreach(println);
}
```

```
foo("str1") //str1
foo("str1","str2") //str1 str2

val a = Array("str1","str2")
foo(a:_*) //str1 str2
```

Note: To pass array, we have to use above notation

Named arguments:

We can shuffle the arguments passed to functions by specifying name when passing

def foo(bar1:Int ,bar2:Int) = bar1 +bar2	foo(bar2= 2, bar1=3) //5 foo(1,2) //3

Default paramters:

While defining functions, we can give default values to parameters.So when calling such functions, if explicit arguments are not passed takes default values.Default parameters should be the last parameters

def greet(greeting:String, name:String=”Tom”) ={ println(greeting + name) }	greet(“hi”) // hi Tom greet(“hi”,”Sawyer”) // hi Sawyer

Tail recursion:

Tail call optimization occurs only when a method or nested function calls itself directly as its last operation. Tail optimization helps recursive functions run as fast as non-recursive loop functions

def fact(n:Int):Int ={ if(n==0) 1 else fact(n-1) } Tail optimization occurs	def fact(n:Int):Int = { if(n==0) 1 else fact(n-1) +1 } Tail optimization don’t occur
def isOdd(n:Int) : Boolean={ if(n%2!=0) true else isEven(n) } def isEven(n:Int) : Boolean ={ if(n%2 == 0) true else false } Tail optimization don’t occur	val funValue = (n:Int) => fact(n) def fact(n:Int):Int = { if(n==0) 1 else funValue(n-1) } Tail optimization don’t occur

Packages and Access modifiers

Classes traits, and singleton objects(or companion objects) can be placed into groups for modular access called packages

package test class X { .... }	package test { class X { .... } }
package test { package TestInner { class X { .... } } }	package test package TestInner class X { .... }

root is the root package of every scala package

Package objects

Package's top level definitions include classes ,traits, objects(companion or standalone)
If we need to define utilitiy methods which are accessible throughout the package, we can use package objects.
A package can have only one package object

package object FooPackage { def foo = 10; }	package X.FooPackage class FooClass { println(foo) }

Imports:

Individual packages and its elements can be imported using import statement
Scala has following properties for import

1)Flexible

2)May import objects & its public members in addition to classes

3)lets you rename|hide some of the imported members

Flexible imports class Figure { def foo =10 } class Foo {	May import objects as well import StandaloneObject.myStandalone class Figure { def foo =10; println(myStandalone) }

def foo(f: Figure) { import f._ // imports all public members of //Figure println(foo) // f.foo } }	object StandaloneObject{ def myStandalone = 100 }
Lets you rename|hide imported members import org.x is equivalent to import org.{x} import org.{x,y} //multiple imports import org.x._ //(imports all members of org.x) import org.x.{sdb => db} //(imports org.x.db as //sdb import org.x.{db => _} //imports all members of //org.x excluding db import org.x.{db=>_ ,_} **{originalObject => renamedObject}** //syntax	

By default scala implicitly imports following packages into every source file

import java.lang._ import scala._ import Predef._	In case of same names in these packages, scala._ shadows java.lang._ Predef._ shadows scala._ and java.lang._

Access Modifiers:

Package members(clapsses, objects , traits) can have private or protected access modifiers
Each individual members of class or object can have acess modifiers as private, public or protected.
Note: When no private or protected precedes member definitions, it is public by default

Foreach individual members of classes or objects, we have variations of above modifiers to give specific meaning

accessModifier[accessibleArea]

accessibleArea should only be **enclosing package or class or this

accessModifier[package] visible within package and accessModifier rules apply outside package //package can be outer or same package where a member is defined accessModifier[class] visible within class and accessModifier rules apply outside class

Note: In case of nested classes, an outer class can access even private variables of inner class in java, where as its not possible in scala

Case classes & Pattern Matching