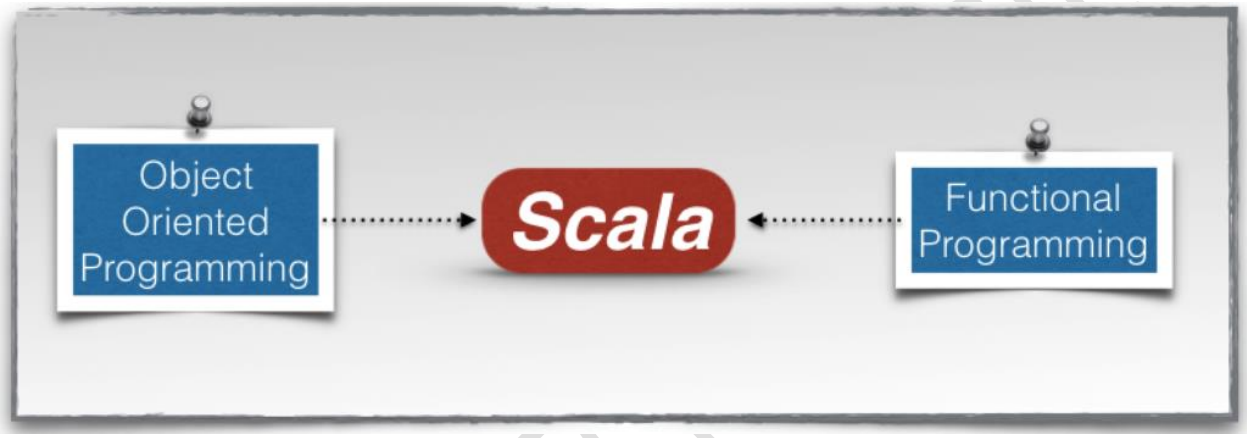
Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817

Scala Introduction and Hands on



By..

Martin Odersky

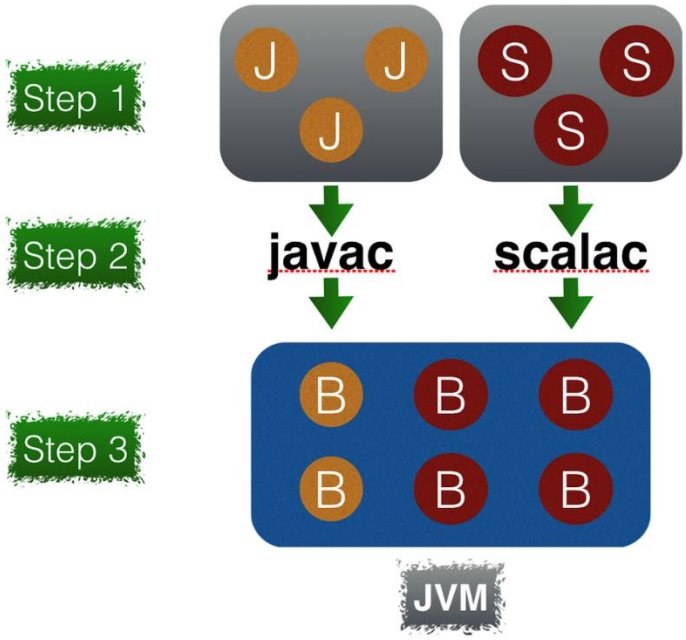
Why Scala?

- First, a developer can achieve a **significant productivity** jump by using Scala.
- Second, it helps developers write **robust code** with reduced bugs.
- Third, Spark is written in Scala, so Scala is a natural fit for **developing Spark applications**.

Scala is Compiler or Interpreted Language?

	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

Why do you need the JDK for writing Scala applications?



<u>Functional Programming</u>

What is Functional Programming?

- Functional programming makes it easier to write **concurrent or multithreaded** applications.
- Functional programming languages make it easier to write elegant code, which is **easy to read, understand, and reason** about.

Functions – Characteristics

First Class

FP treats functions as first-class citizens. A function has the same status as a variable or value. It allows a function to be used just like a variable. While in case of any imperative language such as C it treats function and variable differently.

Composable

Functions in functional programming are composable. Function composition is a mathematical and computer science concept of **combining simple functions to create a complex one**.

No Side Effects

A function in functional programming does not have side effects. The result returned by a function depends only on the input arguments to the function**. The behavior of a function does not change with time. It returns the same output every time for a given input, no matter how many times it is called**. In other words, a function does not have a state. It does not depend on or update any global variable.

Simple

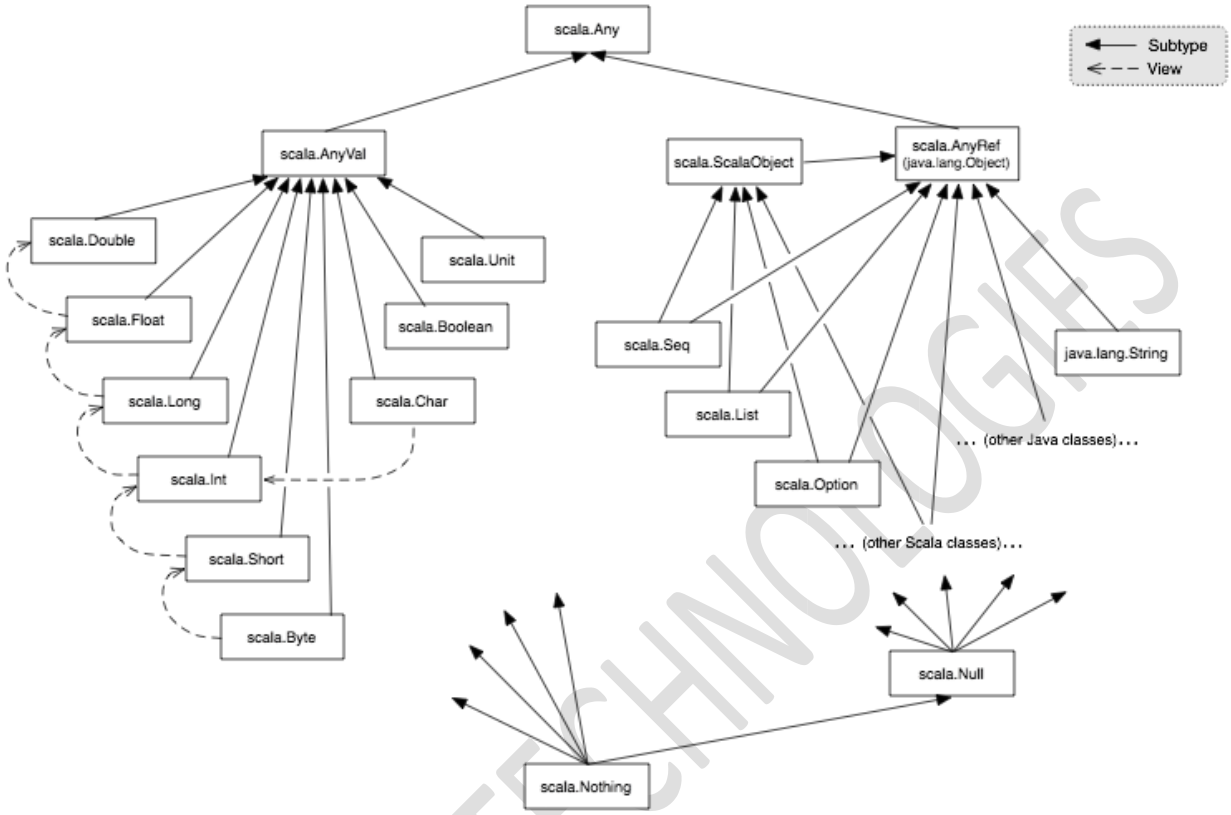
Functions in functional programming are **simple**. A function consists of a few lines of code and it does only one thing. A simple function is easy to reason about. Thus, it enables robustness and high quality.

Variables

Statically Defined and Dynamically Inferred
Mutable – Var
Immutable – Val
Lazy Values

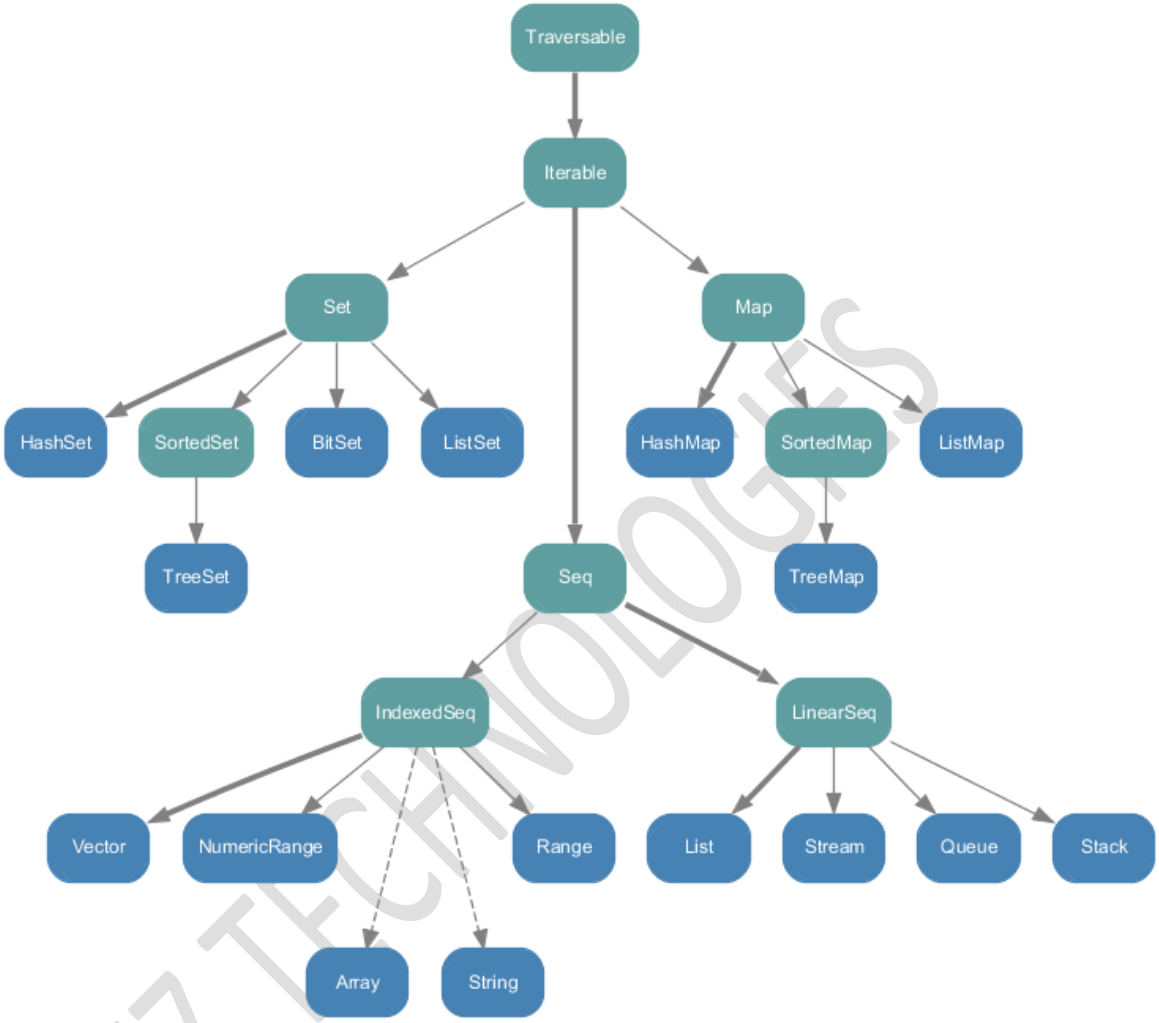
Scala DataTypes:

1. **Value Type**
2. **Reference**



Data Type	Width	Syntax	Data	Min Value	Max Value
int	32-bit	val aInt: Int = 50	whole numbers	-2147483648	2147483647
byte	8-bit	val aByte: Byte = 1	whole numbers	-128	127
short	16-bit	val aShort: Short = 4613	whole numbers	-32,768	32,767
long	64-bit	val aLong: Long = 46138612L	whole numbers	-9,223,372,036,854,770,000	9,223,372,036,854,770,000
float	32-bit	val aFloat: Float = 2.0f	single precision float	1.40E-45	3.40E+38
double	64-bit	val aDouble: Double = 2.34	double precision float	5.00E-324	1.80E+308
char	16-bit	val aChar: Char = 'a'	single character	1	1
boolean	64-bit	val aBoolean: Boolean = false	true or false / 1 or 0	/	/
String		val aString: String = "Hello World"	sequence of characters	/	2147483647

Scala Collection Types Hierarchy:



Let's Explore Scala:

Exercises:

The easiest way to get started with Scala is by using the Scala interpreter, which provides an interactive shell for writing Scala code. It is a **REPL** (read, evaluate, print, loop) tool

Variables

VALUES are immutable constants.

You can't change them once defined.

```
val name: String = "Inceptez!"
name =  "Inceptez Technologies"
println(name)
```

// Notice how Scala defines things backwards from other languages - you declare the // name, then the type.

VARIABLES are mutable

```
var fullname: String = " Inceptez! "
fullname = fullname  + " Technologies"
println(fullname)
```

// One key objective of functional programming is to use immutable objects as often as possible.
// Try to use operations that transform immutable objects into a new immutable object.
// For example, we could have done the same thing like this:

```
val immutablename = name + " Technologies"
println(immutablename)
```

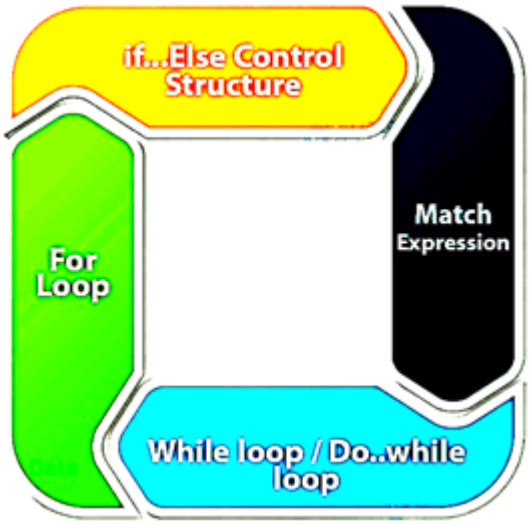
Operators:

- ✓ Arithmetic Operators
  - Used for performing functional/transformational operations on the variables.
  - Eg: +,-,*,/
- ✓ Relational Operators
  - Used for performing comparison operations on the variables.
  - Eg: ==, !=, >, <, <=, >=
- ✓ Logical Operators
  - Used to perform conditional operations on the variables.
  - Eg: &&, ||, !(var1&&var2)
- ✓ Bitwise Operators

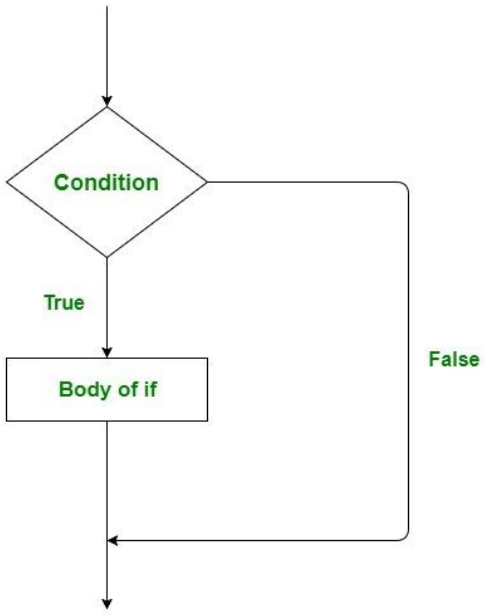
- Used to perform conditional operations by comparing bit by bit of the converted values on the variables.
- Eg: &, |, !

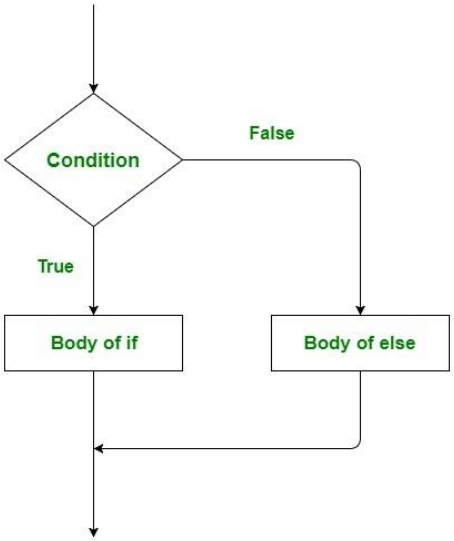
✓ Assignment Operators
- Used to assign or operate on the variable values.
- Eg: val x=10, var x=20; x=x+30 or x+=30

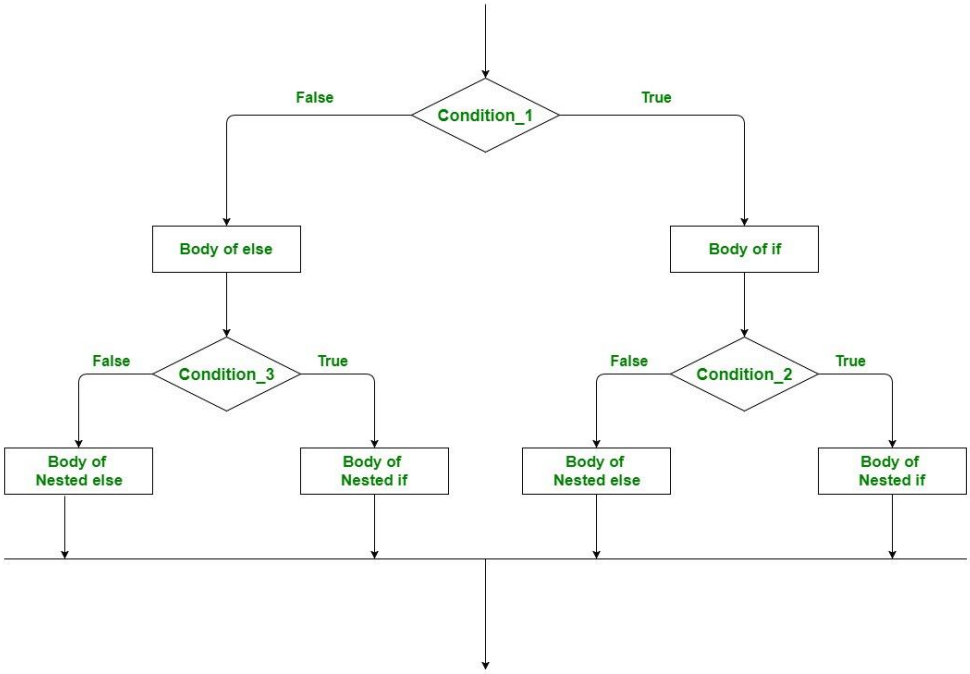
Control Structures:



If then else





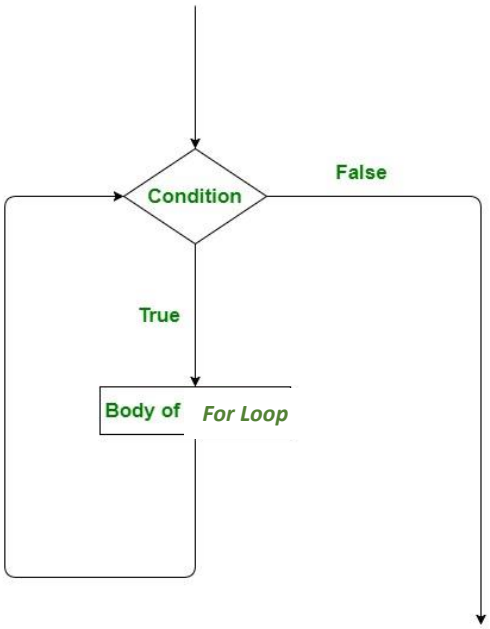



```
val x=10
val y = 5

if (x > y )  println("x is greater")  else if (y>x)
println("y is greater") else
println("x equals y")
```

Loops:

For loops:

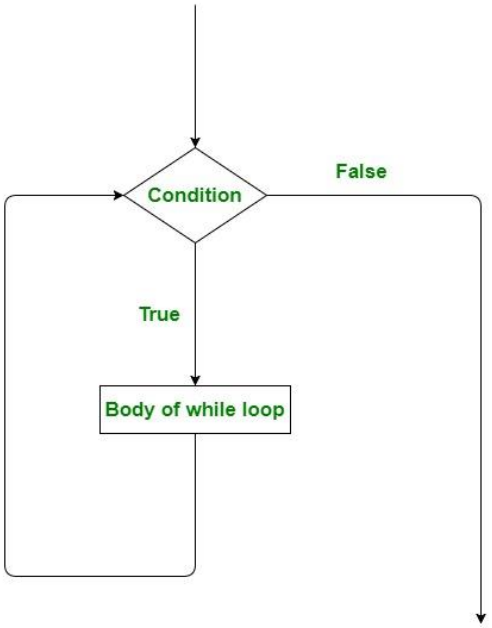
For loops are preferred when the number of times loop statements are to be executed is known beforehand




```
for (x <- 1 to 4)
{
val squared = x * x
println(squared)
}
```

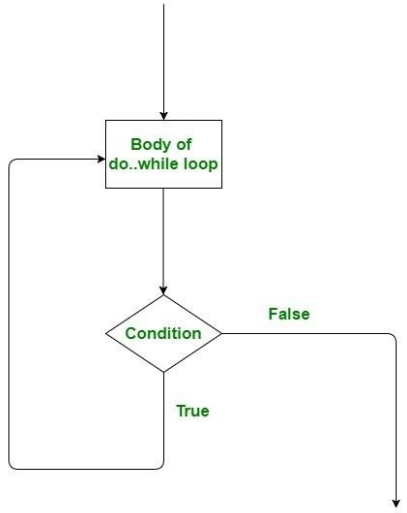
While loops:

*While loops are same as For loop, b*used when we don't know the number of times we want the loop to be executed, however we know the termination condition of the loop. It is also known as an ***entry controlled***




```
var x = 10
while (x >= 0) {
println(x)
x -= 1
}
```

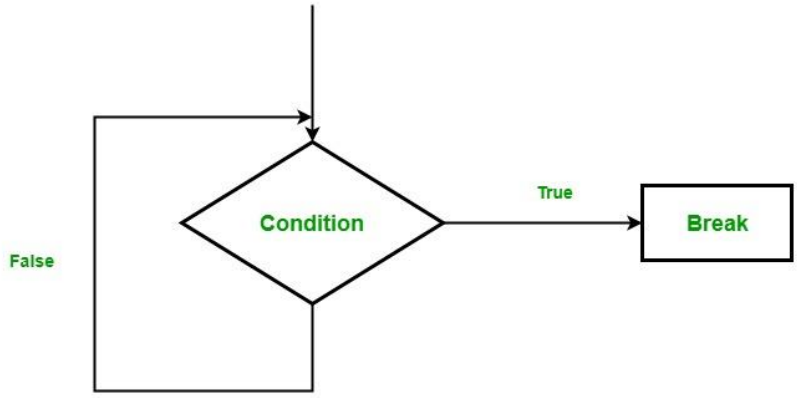
Do While




```
var sid = 4;
do {
println("Student Id is:"+sid);
sid = sid + 1
}while(sid < 10)
```

Break :

We use break statement to break the execution of the loop in the program




```
import scala.util.control.Breaks._
breakable
  {
    for (a <- 1 to 10)
    {
      if (a == 6)
```

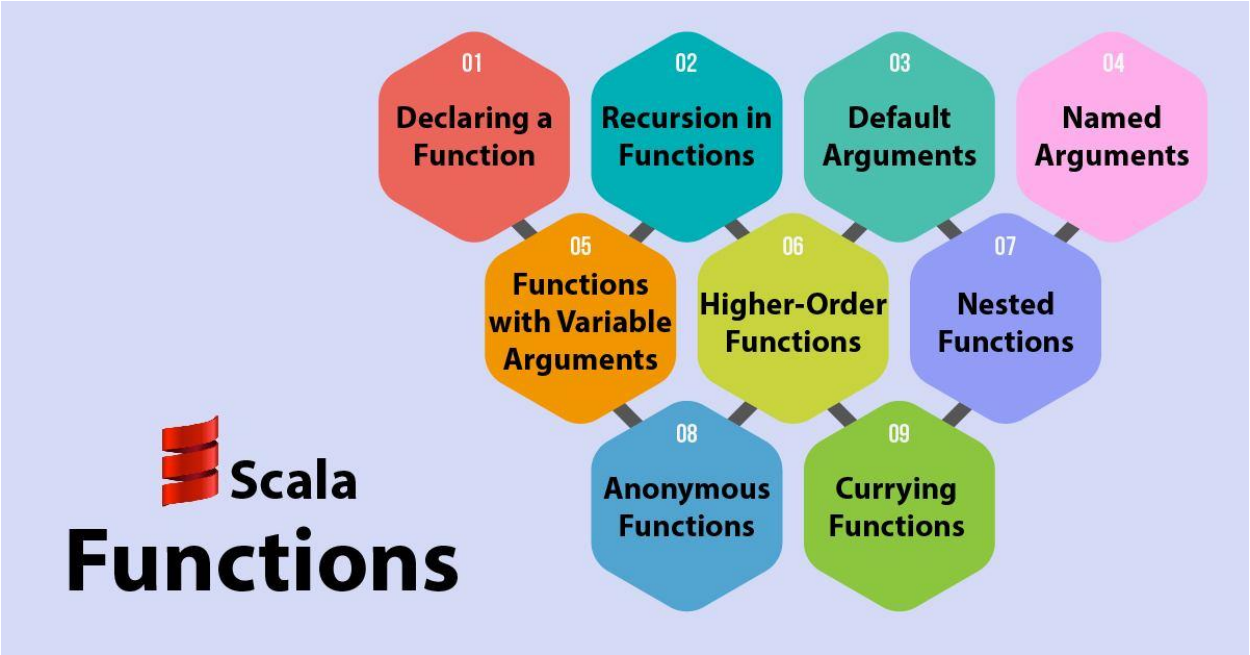
```
            // terminate the loop when
            // the value of a is equal to 6
            break
        else
         println(a);
    }
}
```

Expressions :

// "Returns" the final value in a block automatically

```
val x = 10;
x + 20

println  ({
val x = 10;
x + 20
})
```

Methods & Functions



- ✓ A function/method is a collection of statements that perform a certain task.
- ✓ Functions are used to put some common and repeated task into a single function, so instead of writing the same code again and again for different inputs, we can simply call the function.
- ✓ Scala is assumed as functional programming language so these play an important role. It makes easier to debug and modify the code.

Basic Difference between Scala Functions & Methods:

- ✓ Method always belongs to a class which has a name, signature bytecode etc.
- ✓ Function is a object

Method Declaration & Definition

method declaration & definition have 6 components:

- **def keyword:** "def" keyword is used to declare a method in *Scala*.
- **method_name:** It should be valid name in lower camel case and the name can have characters like +, ~, &, –, ++, \, / etc.
- **parameter_list:** In Scala, comma-separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis.
- **return_type:** User must mention return type of parameters while defining function and return type of a function is optional. If you don't specify any return type of a function, default return type is **Unit** which is equivalent to void in Java.

- = : In Scala, a user can create function with or without = (equal) operator. If the user uses it, the function will return the desired value. If he doesn't use it, the function will not return any value and will work like a subroutine.
- **Method body:** Method body is enclosed between braces { }. The code you need to be executed to perform your intended operations.

How to Define Method in Scala?

A method in Scala is defined with the keyword def. A method definition starts with the name, which is followed by the comma-separated input parameters in parentheses along with their types. The closing parenthesis is followed by a colon, method output type, equal sign, and the method body in optional curly braces.

```
def highsal(p1: Int, p2: Int,p3:Int): Int = {
val salbonus = p1 + p2
if (salbonus > p3)
{
println("higher value is salary and bonus") ;
return salbonus;
}
else
{println("higher value is nettsal");
return p3
}
}

println(highsal (1,2,4))
println(highsal (1,4,4))
```

Scala allows a concise version of the same function, as shown next.

```
def add(firstInput: Int, secondInput: Int) = firstInput + secondInput

add(5,10)
```

The second version does the exact same thing as the first version. The type of the returned data is omitted since the compiler can infer it from the code. However, it is recommended not to omit the return type of a function.

The curly braces are also omitted in this version. They are required only if a function body consists of more than one statement.

Higher-Order Methods

A method that takes a function as an input parameter is called a *higher-order method*. Similarly, a highorder function is a function that takes another function as input. Higher-order methods and functions help reduce code duplication. In addition, they help you write concise code. The following example shows a simple higher-order function.

```
val salaries = Seq(20000, 70000, 40000)

def bonus(a:Int):Double = ((a*1.5))

val normalmethod = salaries.map (a=>(a*1.5) )

val higherordermethod = salaries.map (bonus)
```

Closures

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

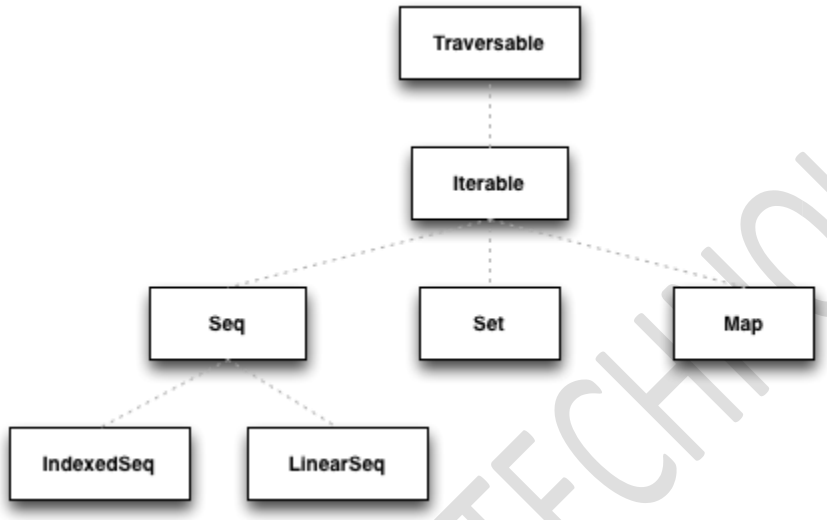
```
var bonuspercent = .10
def bonus = (i:Int) =>  {i+(i * bonuspercent)}

println( "Bonus value is = " +  bonus(100) )
```

Collections

A collection is a container data structure. It contains zero or more elements. Collections provide a higher- level abstraction for working with data. They enable declarative programming. With an easy-to use interface, they eliminate the need to manually iterate or loop through all the elements.

Categories:

- Sequences
- Maps
- Sets



Traversable:

The top of the Collection hierarchy.

The foreach method should traverse all the elements of the collection

Iterable:

A base for **iterable** collections.

This is a base trait for all **Scala** collections that define an iterator method to step through one-by-one the collection's elements. ...

Iterators are data structures that allow to iterate over a sequence of elements

Scala includes an elegant and powerful collection library. They are easy-to-use, concise, safe fast, and universal.

Scala collection is systematically distinguished between ***mutable and immutable***.

Mutable collection can be updated/added/deleted with the elements.

Immutable collection by contrast never changes, but still, can be added/deleted/updated by internally recreated.

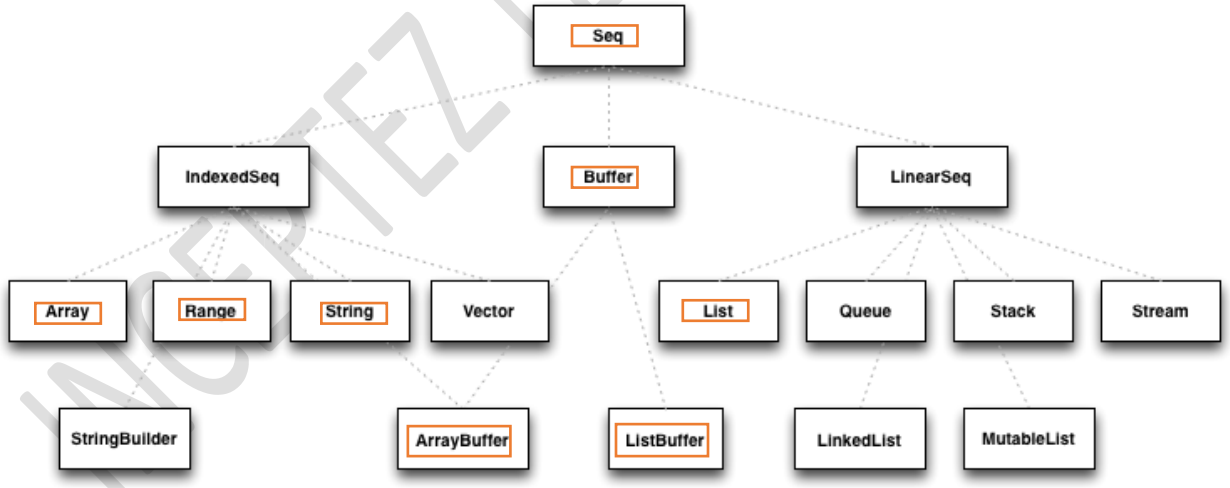
All collection classes are found in scala.collection, or in the sub package, **mutable, immutable, and generic**, ***by default, and collection always picks immutable.***

Category	Type	Access (select)	Mutable (update)	Resizable (Insert/Delete)	Application	Example for Mutation/Resize
Sequence	Seq (AnyVal)	Index starts with 0	List/Array	Non Resizable	Sequence of like elements such as salary, amount etc (use index starts from 0)	
Sequence	Range	Index starts with 0	Range -> List/Array	Non Resizable	Generation of data for sequence	
Sequence	Array	Index starts with 0	Mutable	Non Resizable	1. Array is a fixed size data structure that stores elements of the same data type. 2. The index of the first element of an array is zero and the last element is the total number of elements minus one. 3. It is a collection of mutable values Eg: Mutable Sequence of like elements such as salary, amount etc	Array(1,2,3) -> Array(1,3) -> Array(1,2,3,4,3) Array(1,2,3) -> Array(1,20,3)
Sequence	List	Index starts with 0	Immutable	Non Resizable	1. A list is a collection which contains immutable data 2. Linked list internally, where as Array is flat. Eg: Immutable Sequence of like elements such as id etc	List(1,2,3) -> List(1,20,3) List(1,2,3) -> List(1,3) -> List(1,2,3,4)
Sequence	ArrayBuffer	Index starts with 0	Mutable	Resizable	1. ArrayBuffer is also an Array that can be internally reconstructed and all the elements are copied to the new array if **resize of array happens**. Eg: **Mutable & Resizable** Sequence of like elements such as salary,	ArrayBuffer(1,2,3) -> ArrayBuffer(1,3) -> ArrayBuffer(1,2,3,4) ArrayBuffer(1,2,3) -> ArrayBuffer(1,20,3)
Sequence	ListBuffer	Index starts with 0	Mutable	Resizable	**1. Use a ListBuffer, If we need to create a list that is constantly changing.** **Mutable & Resizable** Sequence of like elements such as salary, amount, salary2 etc	Same as Arraybuffer
Map	Map	Key -> value	Both	Resizable	1. Map is a collection of key-value pairs 2. Keys are always unique while values need not be unique 3. Key-value pairs can have any data type. However, data type once used for any key and value must be consistent throughout Collecion of **key value pair** for lookup, reference and enrichment (hetrogeneous)	Map(1->9840800131)
Set	Set	Set of data (left union right)	Both	Resizable	Set is a Collection of **de duplicated data** Set supports most of the set operations	Set(1,2,3) -> Set(10,20,3)
Tuple	Tuple	Position/Name	Immutable	Non Resizable	Load and access the record / record set (hetrogeneous) Access using positional/named notation	(1,"Inceptez",6)

Sequences

A *sequence* represents a collection of elements in a specific order. Since the elements have a defined order, they can be accessed by their position in a collection.

For example, you can ask for the *nth* element in a sequence.



Array

- ✓ **An *Array* is an indexed sequence of elements.**
- ✓ All the elements in an array are of the same type.
- ✓ It is a mutable data structure;

- ✓ you can update an element in an array.
- ✓ However, you cannot add/remove an element to an array after it has been created.
- ✓ It has a fixed length.

```
var arr = Array(10, 20, 30, 40)
arr(0) = 50
arr+=50
```

List

- ✓ **A List is an indexed sequence of elements.**
- ✓ **All the elements in a list are of the same type.**
- ✓ **It is a Immutable data structure;**
- ✓ **You cannot update an element in an List.**
- ✓ **You cannot add/Remove an element to an array after it has been created.**
- ✓ **It has a fixed length.**

```
var lst = List(10, 20, 30, 40)
lst(0) = 50
lst -= 30
```

ArrayBuffer/ListBuffer

Scala's **Mutable** ArrayBuffer/ListBuffer is used to access elements at specific index, add, remove and update the elements and create an empty ArrayBuffer.

ArrayBuffer/ListBuffer
A ArrayBuffer is a mutable data structure which allows you to access and modify elements at specific index.
An ArrayBuffer is resizable while an Array is fixed in size.

```
var arrbuf=scala.collection.mutable.ArrayBuffer(10,20,30)
arrbuf(0)=100
arrbuf=arrbuf+=40
arrbuf=arrbuf-30

var lstbuf=scala.collection.mutable.ListBuffer(10,20,30)
lstbuf(0)=100
lstbuf=lstbuf+=40
lstbuf=lstbuf-30
```

Map

- ✓ *Map* is a collection of key-value pairs.
- ✓ By default immutable, but can be mutable also
- ✓ In other languages, it known as a dictionary, associative array, or hash map.
- ✓ It is an efficient data structure for looking up a value by its key.

- ✓ It should not be confused with the map in Hadoop MapReduce.
- ✓ That map refers to an operation on a collection.
- ✓ The following code snippet shows how to create and use a Map.

```
var capitals = Map("USA" -> "Washington D.C.", "England" -> "London", "India" -> "New

Delhi")

capitals("India")

capitals+=("New Zeland" -> "Wellington")

capitals-("New Zeland")
```

Set

- ✓ Set is a collection that contains no duplicate elements.
- ✓ There are two kinds of Sets, the immutable and the mutable.
- ✓ By default, Scala uses the immutable Set.
- ✓ We have to import scala.collection.mutable.Set class explicitly for mutable set.

```
var set1=scala.collection.mutable.Set("Washington D.C.", "England" ,"London", "India" )

set1.add("Delhi")

set1.update("Delhi",true)

set1.delete("England")
```

Tuples

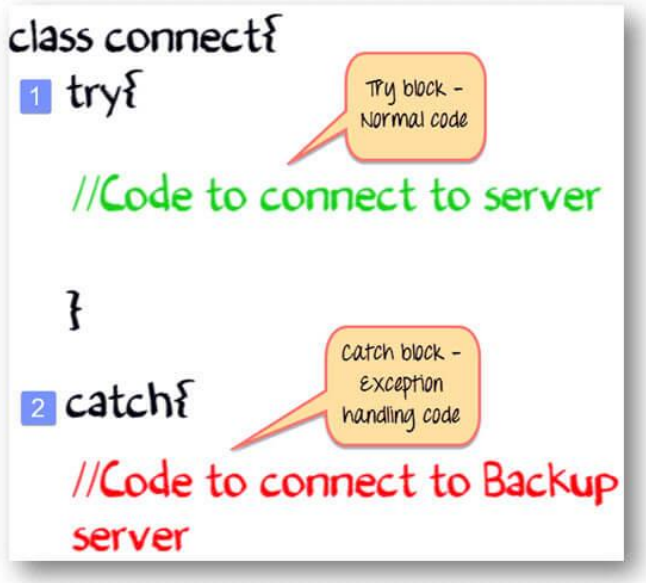
A *tuple* is a container for storing two or more elements of different types. It is immutable, it cannot be modified after it has been created. It has a lightweight syntax, as shown next.

```
val twoElements = ("10", true)
val threeElements = (10, "harry", true)
```

An element in a tuple has a one-based index.
The following code sample shows the syntax for accessing elements in a tuple.

```
val first = threeElements._1
val second = threeElements._2
val clubbed = threeElements._3+ " "+ twoElements._1
```

Exception Handling



finally{

//Code to close all open connections

Executed at Any Cost

}

What is Exception

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time. These events change the flow control of the program in execution.

Why Exception handler is needed

When an exception occurs, say an ArithmeticException as shown in the previous example the current operation is aborted, and the runtime system looks for an exception handler that can accept an ArithmeticException. Control resumes with the innermost such handler. If no such handler exists, the program terminates.

Exception handler blocks:

Throwing Exceptions

Throwing an exception. It looks same as in Java. we create an exception object and then we throw it by using throw keyword.

```
throw new ArithmeticException
```

The *try/catch* Construct

The try/catch construct is different in Scala than in Java, try/catch in Scala is an *expression*. The exception in Scala and that results in a value can be pattern matched in the catch block instead of

providing a separate catch clause for each different exception. Because try/catch in Scala is an expression. Here is an example of exception Handling using the conventional try-catch block in Scala

The finally Clause :
If we want some part of our code to execute irrespective of how the expression terminates we can use a finally block. .

```
try
    {
      var N = 5/0
     }
    catch
    {
      // Catch block contain cases.
      case i: IOException =>
      {
        println("IOException occurred.")
      }
      case a : ArithmeticException =>
      {
        println("Arithmetic Exception occurred.")
      }
finally
{
Println("I will be called at any case")
}
```

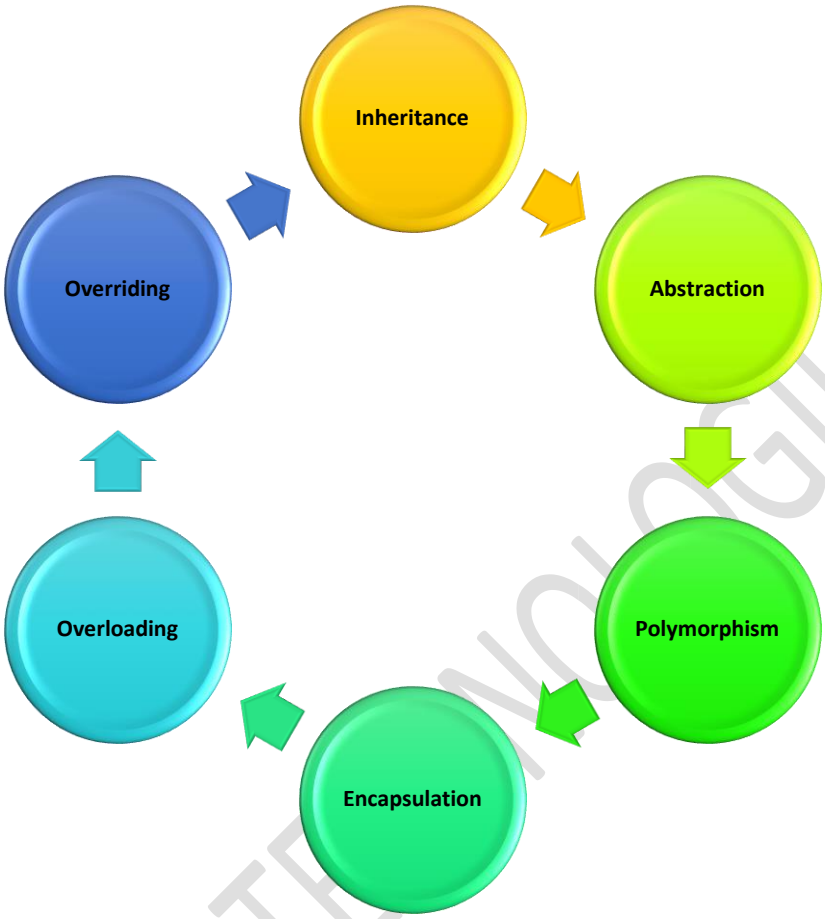
Scala Pattern Matching

```
expression match {
  case v         => expr_A
  case t: Type   => expr_B
  ...
  case _         => expr_Z
}
```

- ✓ Pattern matching is a way of checking the given sequence of tokens for the presence of the specific pattern.
- ✓ It is a technique for checking a value against a pattern.
- ✓ **match** can contain a sequence of alternatives.
- ✓ Each **case** statement includes a pattern and one or more expression which get evaluated if the specified pattern gets matched.
- ✓ To separate the pattern from the expressions, arrow symbol(=>) is used.
- ✓ Scala compiler generates far more efficient bytecode in the pattern matching case
- ✓ Having an extra level of indentation will warn you you're inside a scope

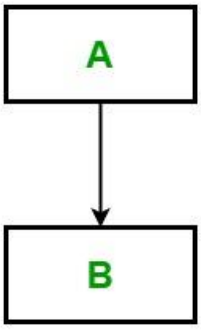
```
val x=1;

x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

OOPS Concepts



Inheritance:

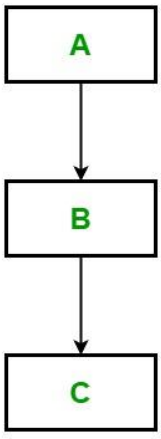
Single



Base class

Derived class

Multilevel

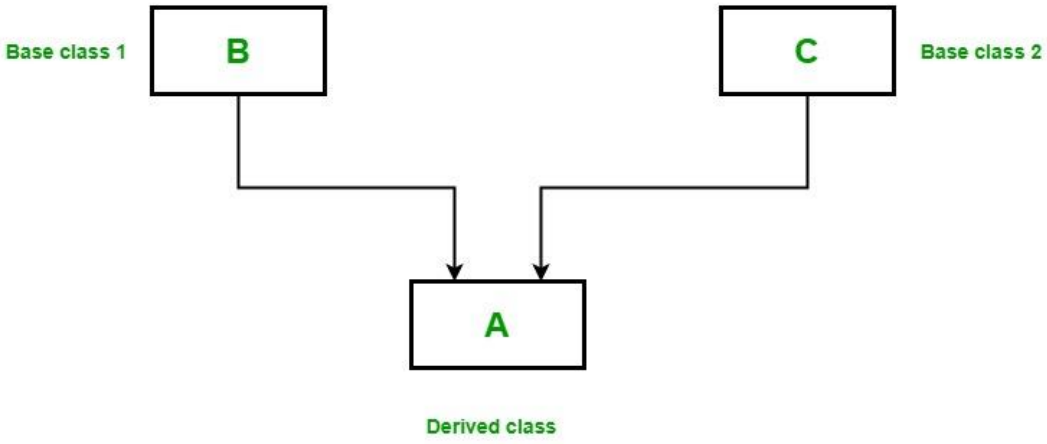


Base class

Derived from A and Base class for C

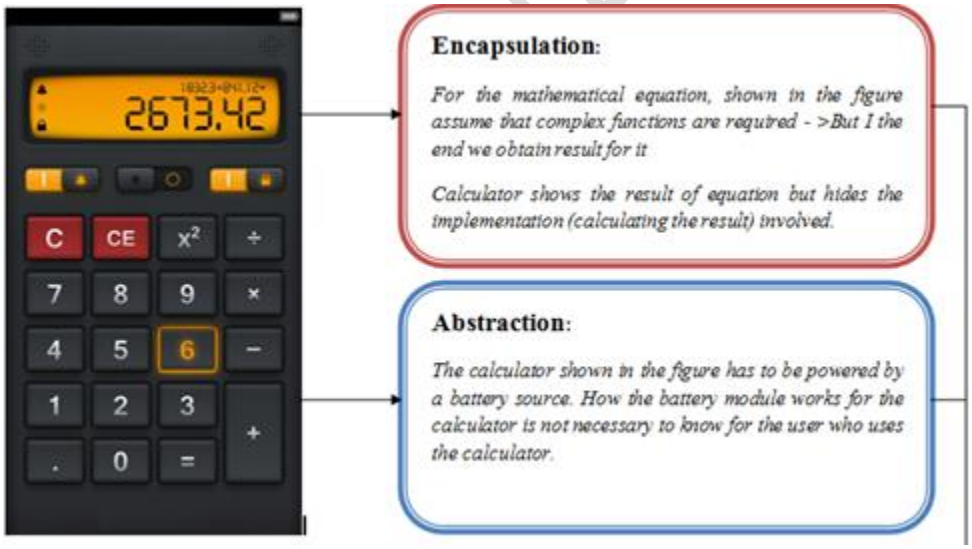
Derived from B

Multiple



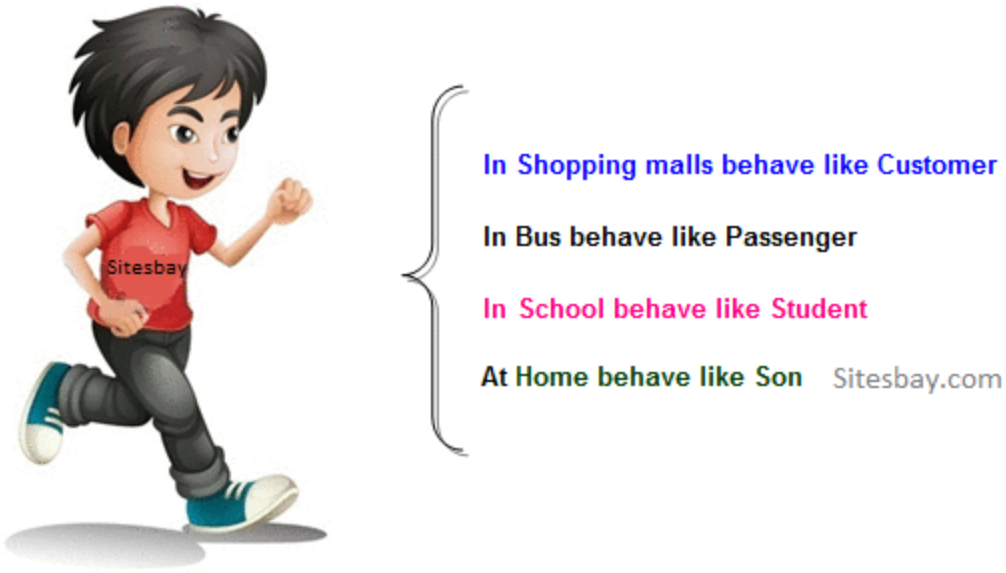
- Inheritance is the process of inheriting the features/members of the parent class
- Types: Single, Multilevel, Multiple, ~~Hybrid~~
- Multiple Inheritance: In Multiple inheritance , one class can have more than one superclass and inherit features from all parent classes
- Scala does not support multiple inheritance with classes, but it can be achieved by traits.
- Example - Abstract Class and Traits

Abstraction:



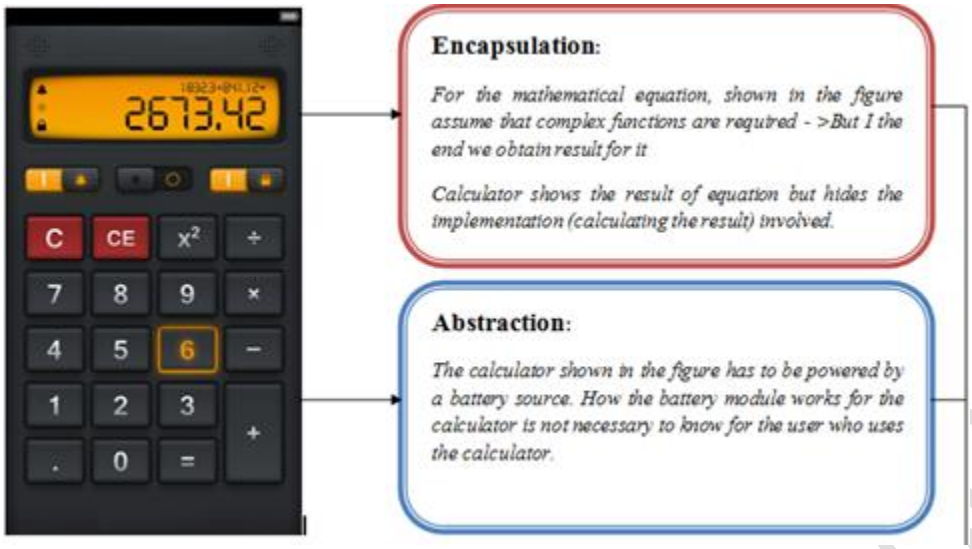
- Abstraction - Abstraction is the process to hide the internal details and showing only the functionality.
- Abstraction is achieved by using an abstract class.
- Example : Abstract class and Traits

Polymorphism/Overloading:



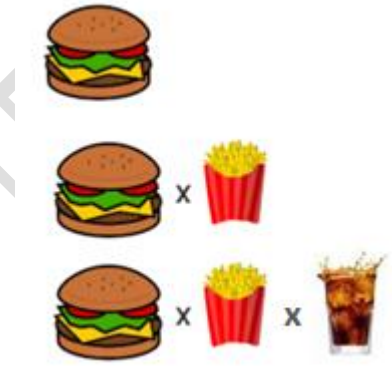
- Scala implements polymorphism through virtual functions, overloaded functions and overloaded operators. The word Polymorphism itself indicates the meaning as Poly means many and Morphism means types
- Polymorphism means that a function type comes "in many forms". The type can have instances of many types.
- Example - Method with different number/type of arguments

Encapsulation:



- Encapsulation is the method of hiding/restricting the access for certain members defined in a class.
- Specify access specified/modifier for providing access control to the objects or values
- Example : private var a=100;

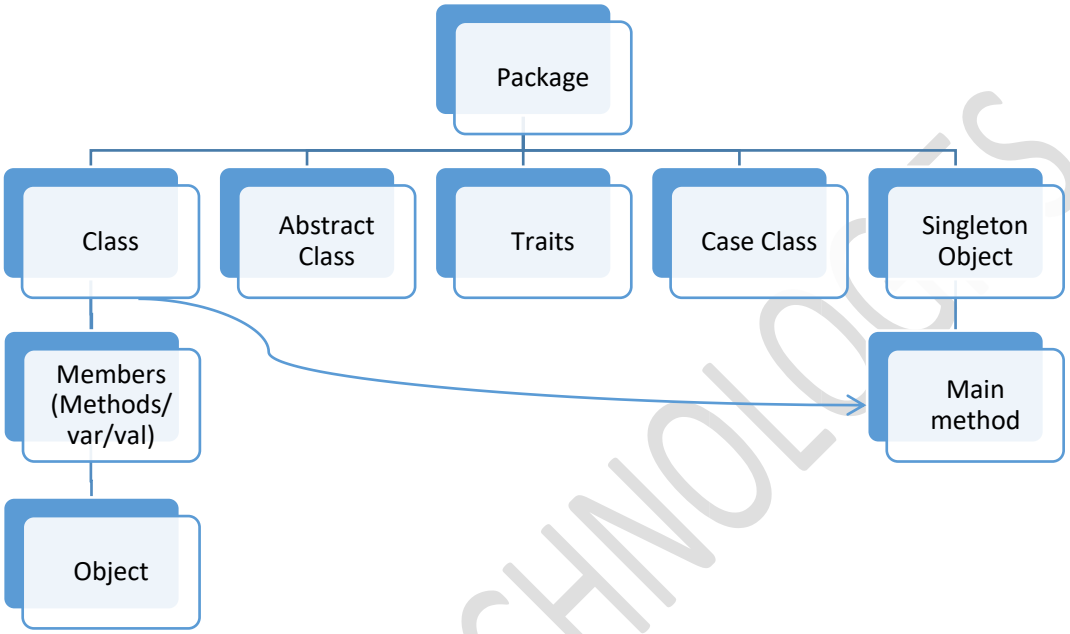
Overriding:



s

- Scala overriding method provides your own implementation of it.
- When a class inherits from another, it may want to modify the definition for a method of the superclass or provide a new version of it.
- We use the 'override' modifier to implement this.
- Example : Method or vals override with different implementations

Hierarchy of Programs in Scala:



Package:

- Package in Scala is a mechanism to encapsulate/collect/contains group of related classes/objects. Mainly used for segregation of the programs.
- Any definitions placed in a package object are considered members of the package itself
- Naming – Reverse of the domain name followed by project followed by modules

Eg:

```
:paste -raw

package com.inceptez.telecom
```

Classes & Objects

- A class is a template or blueprint for creating objects at runtime.
- A class is defined using the keyword class.
- A class is defined in source code.
- A class definition starts with the class name, followed by optional comma-separated class parameters in parentheses, and then fields and methods enclosed in curly braces.

Class:

Class

```
class telephone(model:String,costperhour:Int)
{
```

Encapsulation

```
 private val quality="High"
```

Member Function

```
  def installationcost(hoursforinstallation:Int):Int=
  {
   println("Total cost of installation for " + model + " model phone for hours of " + hoursforinstallation +
" with the quality " + quality + " " + hoursforinstallation*costperhour)
   return hoursforinstallation*costperhour
  }
}
```

Object:

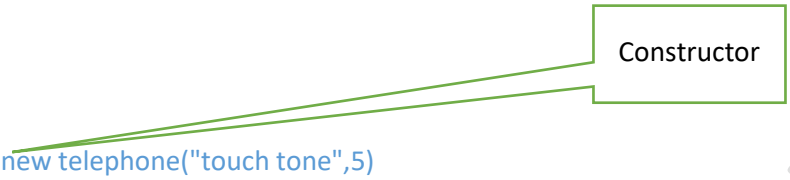
- An object is an instance of a class.
- An instance of a class is created using the keyword new.

Object

```
val rotaryobject=new telephone("rotary",10)
rotaryobject.installationcost(4)
```

Constructor:

- Constructor is used for initializing new objects in memory
- An instance of a class is created using the keyword new to mention construct a new memory area for this instance of the class.




```
val touchobject=new telephone("touch tone",5)
touchobject.installationcost(3)
```

Types of Classes:

Scala Object/Singleton Object:

- Singleton objects are pre-instantiated class.
- Unlike class, we don't required to instantiate to access the methods/variable declared inside singleton objects.
- Scala creates a singleton object to provide entry point for your program execution with the main method with out instantiating it again and again.
- Singleton object is created with UpperCamelCase with object keyword that can't accept arguments

```
object ObjPhone {
def main(args:Array[String])
{
val touchobject=new telephone("touch tone",5)
touchobject.installationcost(args(0).toInt)
}
}

ObjPhone.main(Array("4"))
```

// Following items are not much important

Abstract Class (Single Inheritance and Abstraction)

- Abstraction is the process to hide the internal details and showing only the functionality.
- In Scala, abstraction is achieved by using an abstract class.
- In Scala, an abstract class is constructed using the *abstract keyword*.
- It contains both abstract and non-abstract methods and cannot support multiple inheritances.
- A class can extend only one abstract class.
- All abstract methods/values are not supposed to be implemented.

```
abstract class absaddfeatures{
val volumeadjust=false;
def talktime(hrs:Int):Int={0};

}


class telephone(model:String,costperhour:Int) extends absaddfeatures
{
  def installationcost(hoursforinstallation:Int):Int=
  {
   println("Total cost of installation for " + model + " model phone for hours of " + hoursforinstallation +
" is " + hoursforinstallation*costperhour)
   return hoursforinstallation*costperhour
  }
  override val volumeadjust=true;
  override def talktime(hrs:Int):Int=
  {
  if(model=="touch tone")
  hrs;
  else
  hrs+100;
  }
}

val obj=new telephone("rotary",10);
obj.talktime(10);
```

Overriding

Traits (Multiple Inheritance)

- It is an abstraction mechanism that helps development of modular, reusable, and extensible code
- The key difference between trait and abstract class is that only once abstract class can be inherited, but traits can be inherited in multiple, hence it helps implementing multiple inheritance in scala.
- When a class extends a trait, each abstract method (un implemented methods) must be implemented.
- Use extends to extend the first trait, Use with to extend subsequent traits

```
trait traitaddfeatures1{
val loudspeaker=true;
def router:String;

}

trait traitaddfeatures2{
val antenna=true;
```

```
}

class telephone(model:String,costperhour:Int) extends traitaddfeatures1 with traitaddfeatures2
{
  def installationcost(hoursforinstallation:Int):Int=
  {
   println("Total cost of installation for " + model + " model phone for hours of " + hoursforinstallation +
" is " + hoursforinstallation*costperhour)
   return hoursforinstallation*costperhour
  }
  def router:String={"High Bandwidth"}
}

val obj=new telephone("rotary",10);
obj.router;
obj.antenna;
```

Case Classes (Important)

- A case class is a class with a case modifier.
- All input parameters defined implicitly treated as Val
- Useful for immutable objects and pattern matching

```
case class emp(id: Int, name: String, address: String)

val request = emp(1, "Sam", "1, castle point blvd, nj")
println(request.name)
request.address
```

Companion Object:

- A companion object is an object that's declared in the same **name** as a **class**
- A companion object and its **class** can access each other's private members