UNIT 1

INTRODUCTION, BASICS AND WRITING PROGRAM

Introduction, need for scala, scala REPL, data types, scala variables and data types, option type, Type inference, Getting familier with Idea Intellij, how to install scala plugnins, build tools:Maven and SBT, download dependencies, write scala program, compiling , building and running program using console, Intellij setting and preferences, learn shortcuts, Debugging your code.

UNIT 2

SCALA LOOPS, CONDITIONAL STATEMENT, COLLECTIONS AND EXCEPTION HANDLING

Scala Loops:-scala for and scala while, conditional expressions, scala array and scala tuples etc. Scala Collections:-Immutable collections and mutable collections like list, set and map, break statement, scala comments, scala string, string methods, string interpolation**. Scala Exception Handling:-try-catch block, finally block, throw keyword, throws keyword, scala custom exception.**

UNIT 3

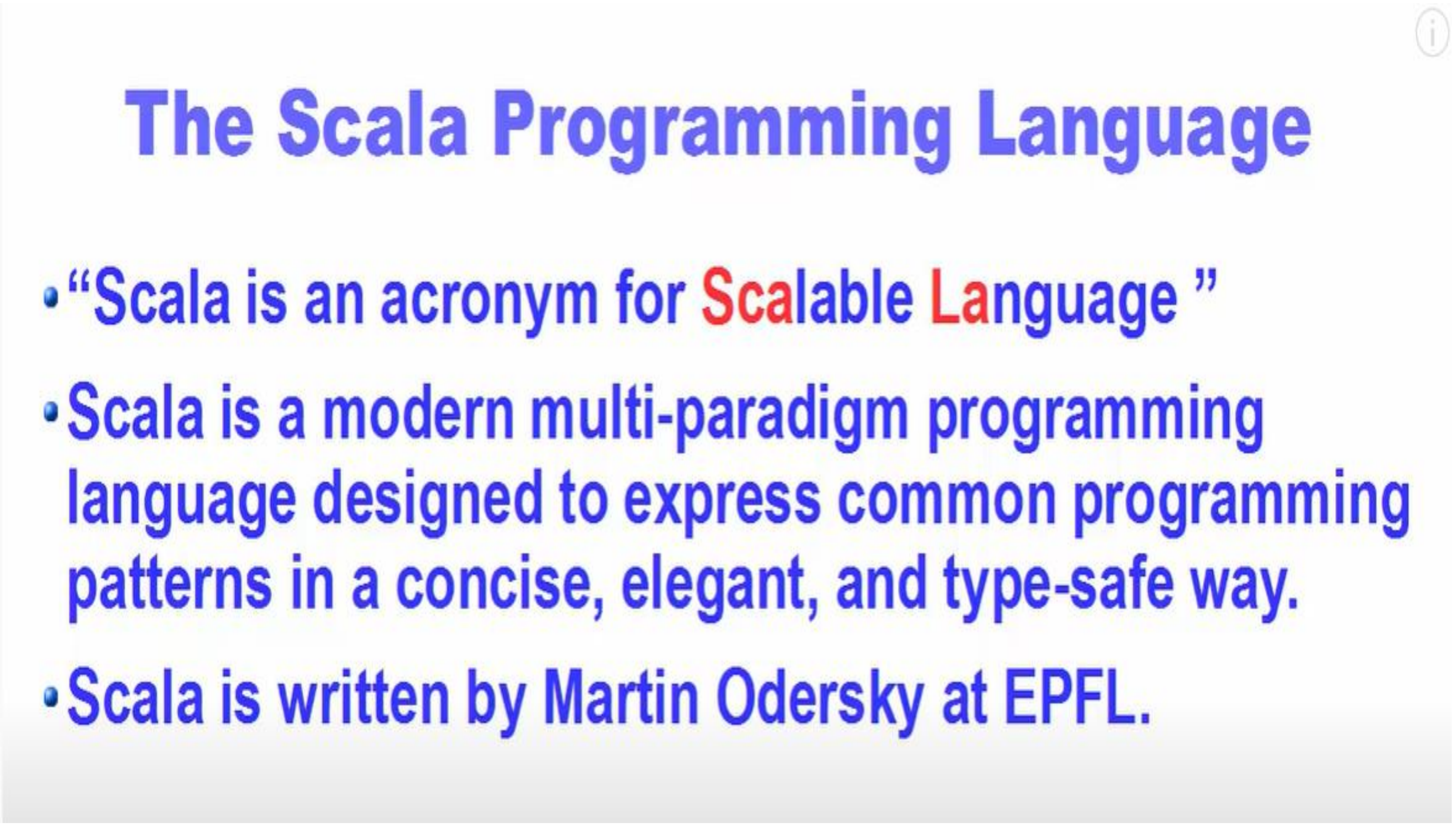
SCALA OBJECT ORIENTED PROGRAMING AND FILE HANDLING

Scala oops concepts:class and objects, singleton and companion objects, case class and objects, scala constructors, scala functions, scala method overloading and overriding, scala final. scala inheritance, scala Trait, scala trait mixins, scala complete program with case class and File handling in scala.

UNIT 4

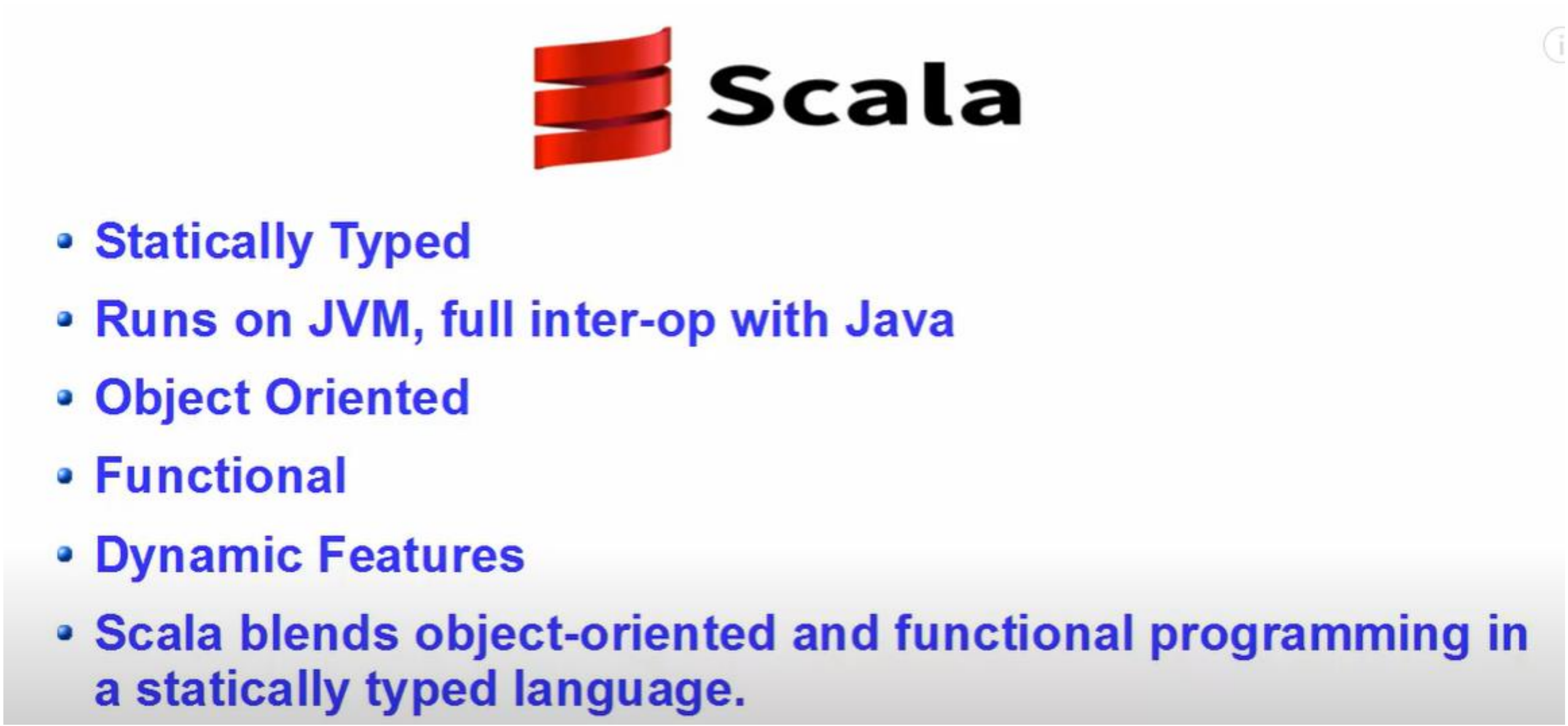
FUNCTIONAL PROGRAMMING, ASYNCHRONOUS PROGRAMING AND SOME IMPORTANT TOPICS ALONG WITH TESTING.

Functional and asynchronous programing in scala, Higher order functions, covariance, scala varrargs, pattern matching in scala, Implicit types, working with GitHub, Testing your functionalities with test cases.



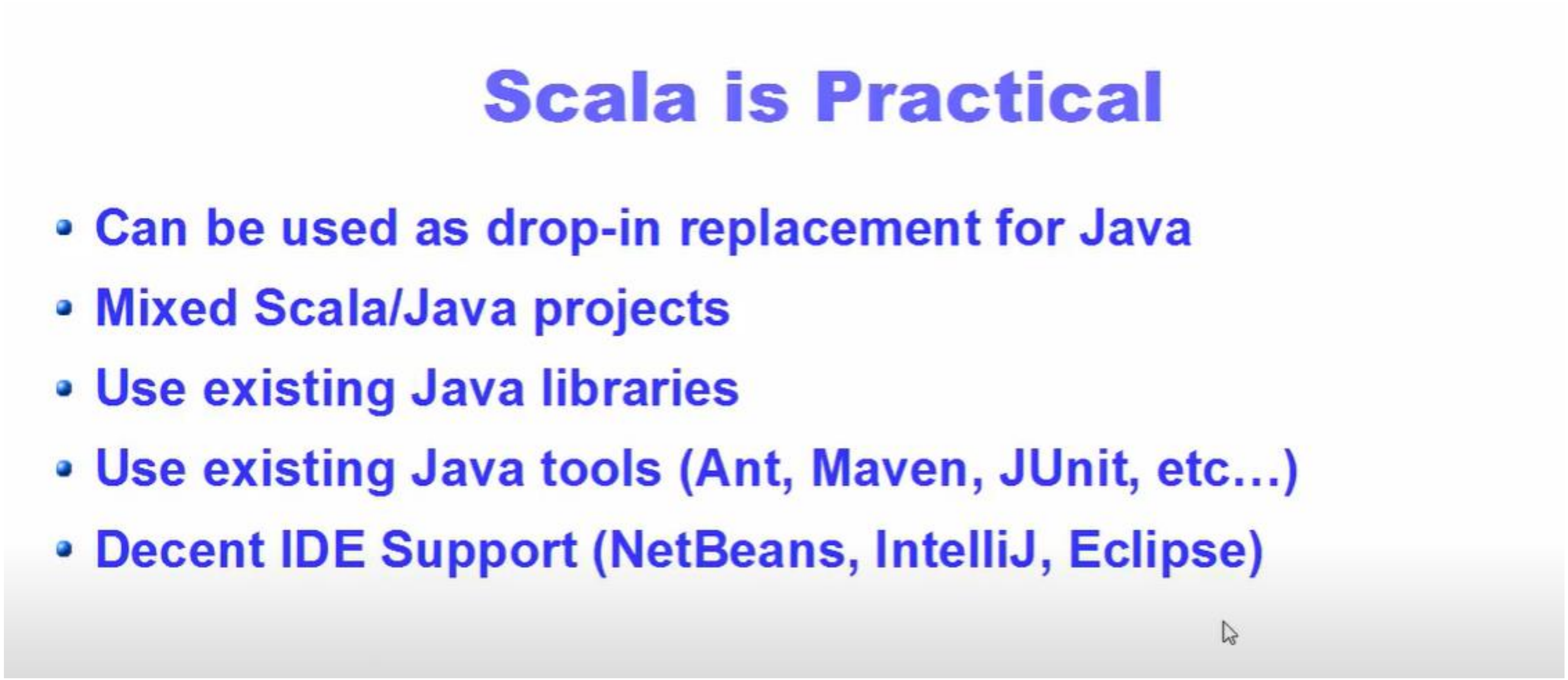
The Scala Programming Language

- “Scala is an acronym for **Sca**lable **La**nguage ”
- Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way.
- Scala is written by Martin Odersky at EPFL.



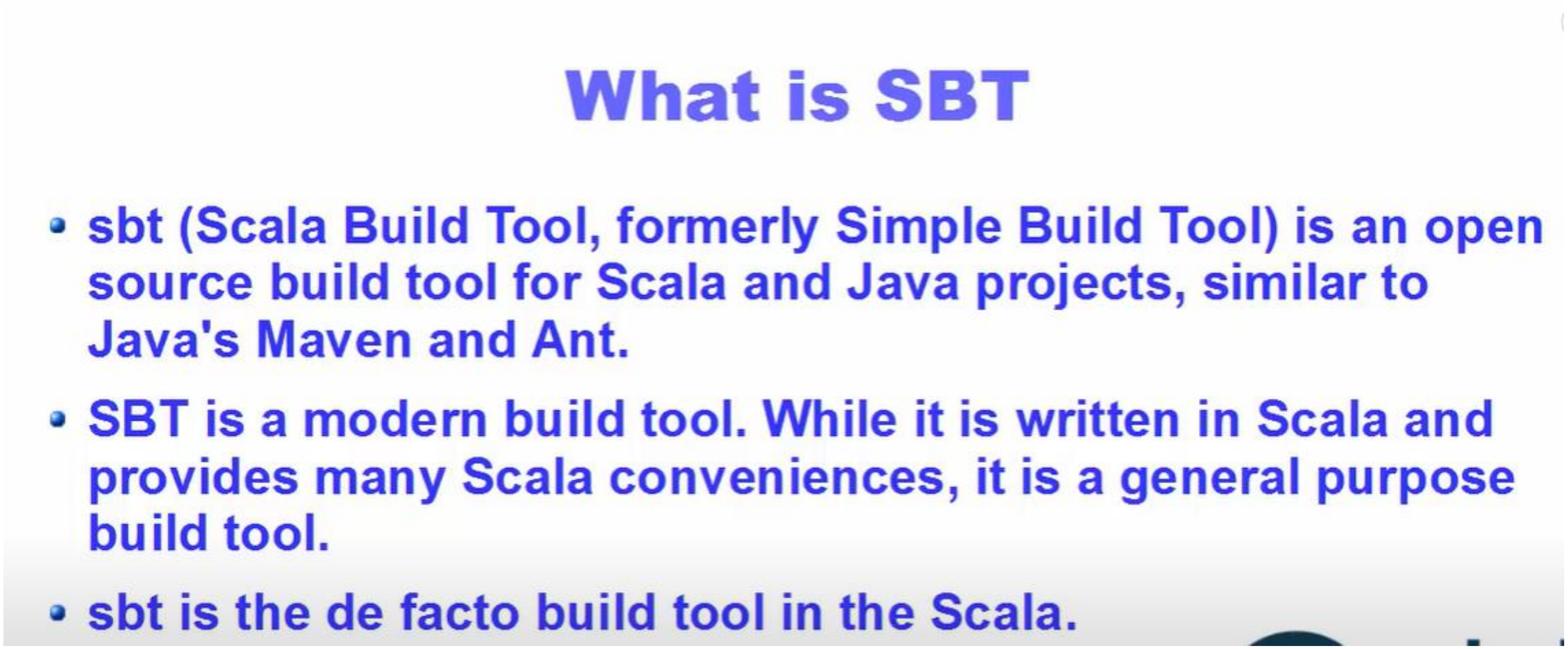
Scala

- Statically Typed
- Runs on JVM, full inter-op with Java
- Object Oriented
- Functional
- Dynamic Features
- Scala blends object-oriented and functional programming in a statically typed language.



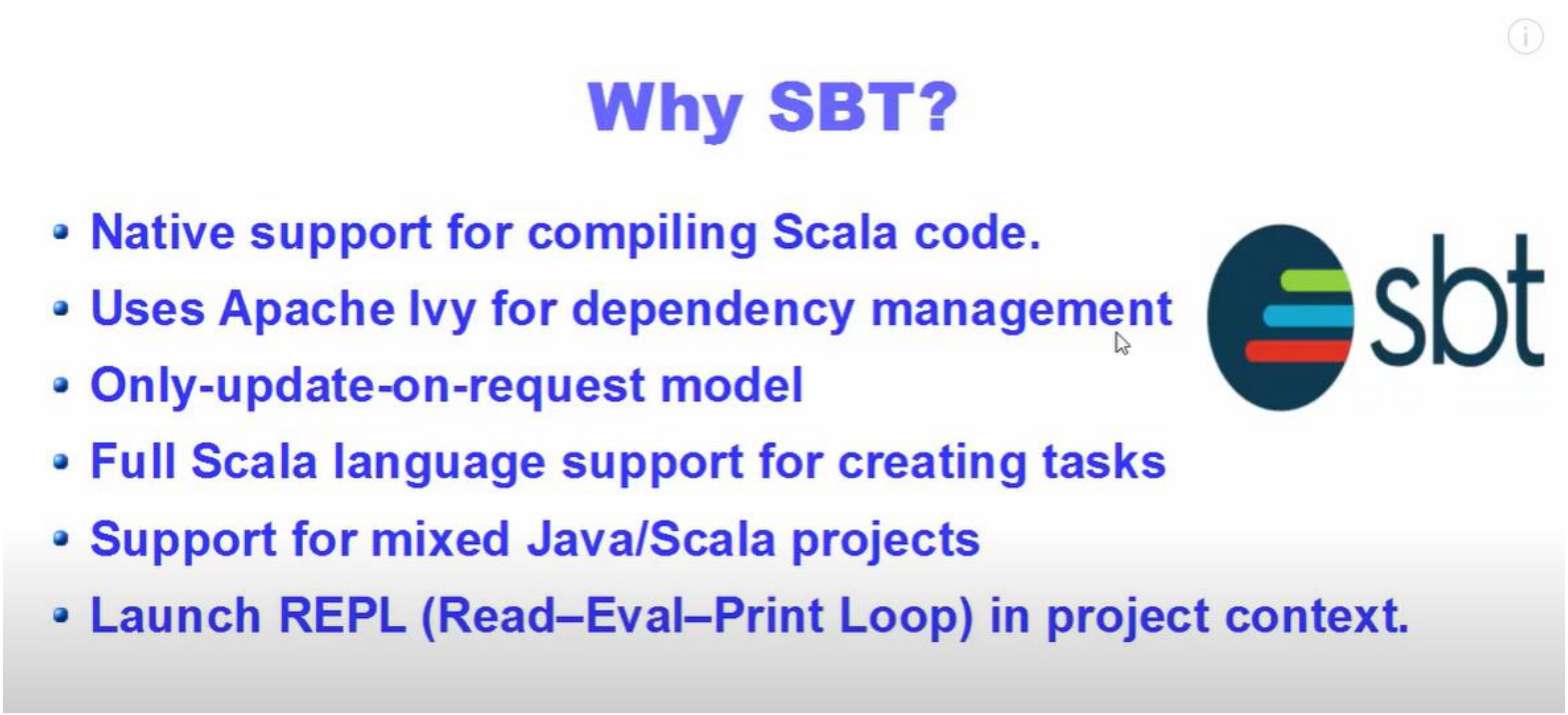
Scala is Practical

- Can be used as drop-in replacement for Java
- Mixed Scala/Java projects
- Use existing Java libraries
- Use existing Java tools (Ant, Maven, JUnit, etc…)
- Decent IDE Support (NetBeans, IntelliJ, Eclipse)



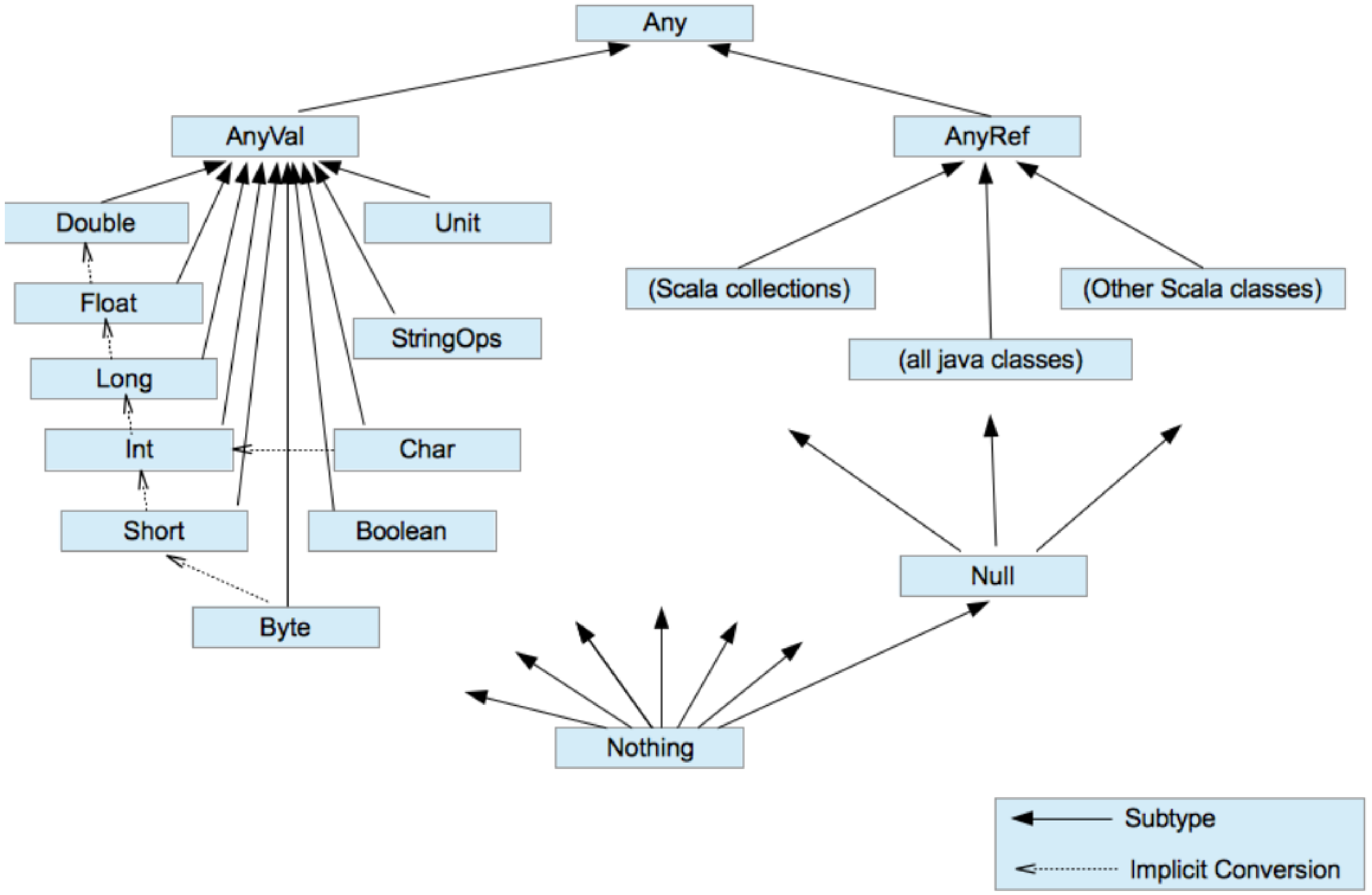
What is SBT

- sbt (Scala Build Tool, formerly Simple Build Tool) is an open source build tool for Scala and Java projects, similar to Java's Maven and Ant.
- SBT is a modern build tool. While it is written in Scala and provides many Scala conveniences, it is a general purpose build tool.
- sbt is the de facto build tool in the Scala.



Why SBT?

- Native support for compiling Scala code.
- Uses Apache Ivy for dependency management
- Only-update-on-request model
- Full Scala language support for creating tasks
- Support for mixed Java/Scala projects
- Launch REPL (Read–Eval–Print Loop) in project context.


Any
AnyVal
AnyRef
Double
Float
Long
Int
Short
Byte
Unit
StringOps
Char
Boolean
(Scala collections)
(all java classes)
(Other Scala classes)
Null
Nothing
Subtype
Implicit Conversion

1. The null reference is used as an absent value - when you access fields or methods you'll get the famous NullPointerException

2. The Null type is the type of the null reference.

3. Nil is the empty List.

4. None is the empty Option.

5. Unit represents "void" from other languages.

6. Nothing is the bottom of the type hierarchy.

Null and null

The null reference is used to represent an absent value, and Null with a capital 'N' is its type

Nothingness in Scala

There are 7 cases where you want to represent the concept of nothingness in Scala.

Nil - Used for representing empty lists, or collections of zero length. For sets you can use Set.empty

None - One of two subclasses of the Optional type, the other being Some. Very well supported by the Scala collections.

Unit - Equivalent to Java's void which is used for functions that don't have a return type

Nothing - A trait. It is a subtype of all other types, but supertype of nothing. It is like the leaf of a tree. No instances of Nothing.

Null - A trait. Not recommended to be used.

null - An instance of the trait, similarly used as the Java Null. Not recommended to be used.

Best way to handle NULL / Empty string in Scala

Technique 1: Checking the length to 0

```
val str:String = ""
if (str.length == 0){
  println(s"Length of Variable ${str} is 0")
}
```

This works perfectly when the value of str is empty. But when it contains NULL, it fails.

```
val str:String = null
if (str.length == 0){
  println(s"Length of Variable ${str} is 0")
}
```

Technique 2: Using isEmpty function

```
val str:String = ""
if (str.isEmpty()){
  println(s"Variable ${str} is empty")
}
```

This works perfectly when the value of str is empty. But when it contains NULL, this also fails.

```
val str:String = null
if (str.isEmpty()){
  println(s"Variable ${str} is empty")
}
```

Technique 3: Comparing it with double-quotes

```
val str:String = ""
if ("".equals(str)){
  println(s"Variable ${str} is empty")
}
```

This technique works well for empty and also handles NULL graciously without any error.

```
val str:String = null
if ("".equals(str)){
  println(s"Variable ${str} is empty")
}
```

Technique 4: Comparing it with double-quotes. What again same technique? Yes but comparing it another way around.

```
val str:String = ""
if (str.equals("")){
  println(s"Variable ${str} is empty")
}
```

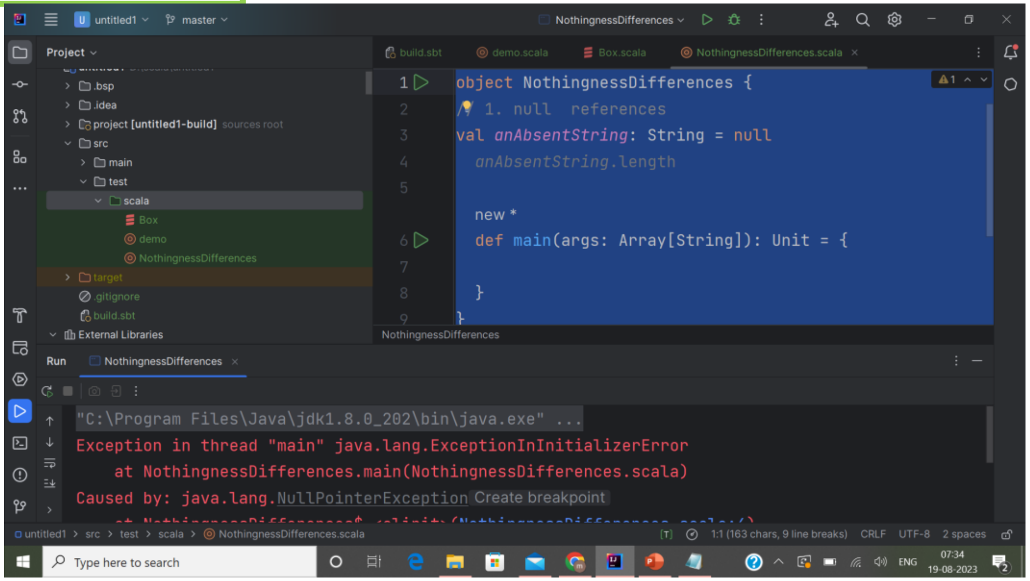
As this technique is similar to the previous one, you would expect it would work the same way. Unfortunately, it doesn’t handle Null values well.

```
val str:String = null
if (str.equals("")){
  println(s"Variable ${str} is empty")
}
```

1. The null reference is used as an absent value –

when you access fields or methods you'll get the famous NullPointerException

```
object NothingnessDifferences {
// 1. null  references
val anAbsentString: String = null
  anAbsentString.length

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

  }
}
```



```
object NothingnessDifferences {
// 1. null  references
val anAbsentString: String = null
  anAbsentString.length

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

  }
}
```

```
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
Exception in thread "main" java.lang.ExceptionInInitializerError
    at NothingnessDifferences.main(NothingnessDifferences.scala)
Caused by: java.lang.NullPointerException
```

2. The Null type is the type of the null reference.

1. Overview
Showed the differences between Nil, Null, Nothing, Unit, and None types in Scala.

Although it may seem like all these keywords are used for representing the absence of a value, each one has its own purpose.

We'll go through each of them and learn its characteristics using examples and use cases.

2. Null and null
The null reference is used to represent an absent value, and Null with a capital 'N' is its type

```
class abc
{
  def tryit(thing: Null): Unit = {
    println("That worked!");
  }

}
object Datatyes extends  App {
  var obj = new abc()
  obj.tryit("Mtech(DS)")
}
```

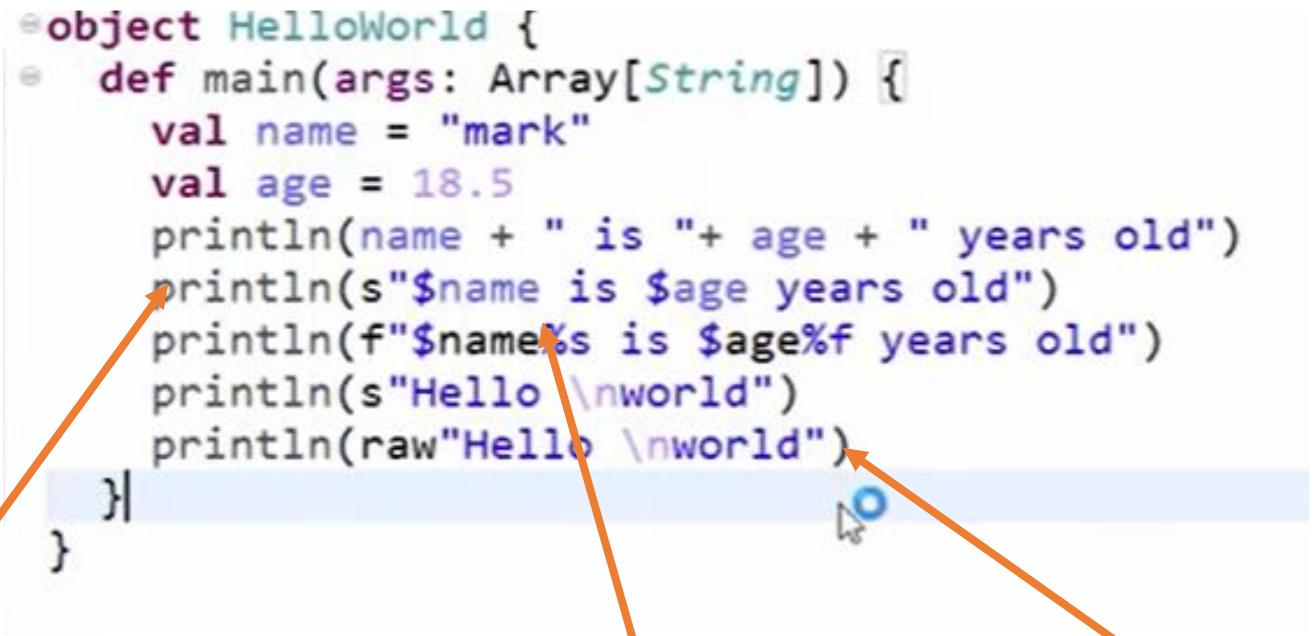
Error

```
D:\scala\untitled1\src\test\scala\Datatyes.scal
a:10:13
type mismatch;
 found   : String("Mtech(DS)")
 required: Null
  obj.tryit("Mtech(DS)")
```

```
class abc
{
  def tryit(thing: Null): Unit = {
    println("That worked!");
  }

}
object Datatyes extends  App {
  var obj = new abc()
  obj.tryit("Mtech(DS)")
}
```

```
type mismatch;
 found   : String("Mtech(DS)")
 required: Null
  obj.tryit("Mtech(DS)")
```

String Interpolation



```
object HelloWorld {
  def main(args: Array[String]) {
    val name = "mark"
    val age = 18.5
    println(name + " is "+ age + " years old")
    println(s"$name is $age years old")
    println(f"$name%s is $age%f years old")
    println(s"Hello \nworld")
    println(raw"Hello \nworld")
  }
}
```

String not safe interpolation

Type safe interpolation
Because we will provide data types itself.

Raw Interpolation

An **iterator** is a way to access elements of a collection one-by-one. It resembles to a collection in terms of syntax but works differently in terms of functionality. An iterator defined for any collection does not load the entire collection into the memory but loads elements one after the other. Therefore, iterators are useful when the data is too large for the memory. To access elements we can make use of **hasNext()** to check if there are elements available and **next()** to print the next element.

```
object Scala_Iterator {
  def main(args: Array[String]): Unit = {
    val v = Iterator(5, 1, 2, 3, 6, 4)

    //checking for availability of next element
    while(v.hasNext)

    //printing the element
      println(v.next)
  }
}
```

We can define an iterator for any collection(Arrays, Lists, etc) and can step through the elements of that particular collection.

```
object Scala_Iterator_for_any_collection {
  def main(args: Array[String]): Unit = {
    val v = Array(5,1,2,3,6,4)
    //val v = List(5,1,2,3,6,4)

    // defining an iterator
    // for a collection
    val i = v.iterator

    while (i.hasNext)
      print(i.next + " ")
  }
}
```

Scala Collection

Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable. we can use them according to your requirement. **Scala.collection.mutable** package contains all the mutable collections. You can add, remove and update data while using this package.
Scala.collection.immutable contains all the immutable collections. It does not allow you to modify data. Scala imports this package by default. If you want mutable collection, you must import **scala.collection.mutable** package in your code.

Scala Set

It is used to store unique elements in the set. It does not maintain any order for storing elements. You can apply various operations on them. It is defined in the Scala.collection.immutable package.

Scala Set Syntax

1. **val** variableName:Set[Type] = Set(element1, element2,... elementN) or
2. **val** variableName = Set(element1, element2,... elementN)

```
object Scala_Collections_Set {
  def main(args: Array[String]): Unit = {
    val set1 = Set()                         // An empty set
    val Mtech_Stu_Names = Set("Somesh","Kapil","Brijneder","Ritika","Tosseb")    //
Creating a set with elements
    val Mtech_Stu_Names1:Set[String] =Set("Somesh","Kapil","Brijneder","Ritika","Tosseb")
    println(set1)
    println("First way to Defined Set:="+Mtech_Stu_Names)
    println(s"Second way to Defined Set:=$Mtech_Stu_Names1")
//    println(Mtech_Stu_Names.head)            // Returns first element present in the set
//    println(Mtech_Stu_Names.tail)        // Returns all elements except first element.
//    println(Mtech_Stu_Names.isEmpty)        // Returns either true or false
  }
```

Scala Set Example: Merge two Set

we can merge two sets into a single set. Scala provides a predefined method to merge sets. In this example, ++ method is used to merge two sets.

```
object Scala_Collections_Set_Mearge_Two_Set
 {
  def main(args: Array[String]): Unit =
 {
    val name = Set("kapil","somesh","rohit","mohit")
    val rollnumber = Set("101","102","103","104","105")
    val mergeSet = name ++ rollnumber           // Merging two sets
    println("Elements in games set: "+name.size)   // Return size of collection
    println("Elements in alphabet set: "+rollnumber.size)
    println("Elements in mergeSet: "+mergeSet.size)
    println(mergeSet)
  }
}
```

Scala Set Example: Adding and Removing Elements

```
object Scala_Collection_Set_Add_Remove
{
  def main(args: Array[String]): Unit =
 {
    var Mtech_Stu_Name=Set("mk","jk","tk","np")
  //    Mtech_Stu_Name.foreach((name:String)=>println(name))
    Mtech_Stu_Name -= "makhan"
    Mtech_Stu_Name.foreach((name:String)=>println(name))

  }
}
```

Iterating Set Elements using foreach loop

```
object Scala_Collection_Iterating_elements_Foreach {
  def main(args: Array[String]): Unit = {
    var Mtech_Stu_names = Set("Somesh", "Rohit", "Kapil", "Mukesh")
    Mtech_Stu_names.foreach((element:String)=> println(element))
  }
}
```

Iterating Set Elements using for loop =?

Scala HashSet

HashSet is sealed class. It extends immutable Set and AbstractSet trait. Hash code is used to store elements. It neither sorts the elements nor maintains insertion order . The Set interface implemented by the HashSet class, backed by a hash table . In Scala, A concrete implementation of Set semantics is known HashSet.

Scala HashSet Example

In the following example, we have created a HashSet to store elements. Here, foreach is used to iterate elements.

```
import scala.collection.immutable.HashSet
object Scala_Collection_HashSet {
  def main(args: Array[String]): Unit = {
    var Mtech_Stu_Name = HashSet("kapil","somesh","rohit","mohit","xyz")
      Mtech_Stu_Name.foreach((name:String)=>println(name+" "))

//    Mtech_Stu_Name.foreach((element:String) => println(element+" "))
  }

}
```

Operations perform with HashSet

Adding an elements in HashSet : We can add an element in HashSet by using + sign. below is the example of adding an element in HashSet.

Adding more than one element in HashSet : We can add more than one element in HashSet by using ++ sign. below is the example of adding more than one elements in HashSet.

Remove element in HashSet : We can remove an element in HashSet by using – sign. below is the example of removing an element in HashSet.

Find the intersection between two HashSets : We can find intersection between two HashSets by using & sign. below is the example of finding intersection between two HashSets.

Scala Seq

Seq is a trait which represents indexed sequences that are guaranteed immutable. we can access elements by using their indexes. It maintains insertion order of elements. Sequences support a number of methods to find occurrences of elements or subsequences. It returns a list.

```
object Scala_Collection_Seq {
  def main(args: Array[String]): Unit = {
    var Roll_number:Seq[Int] = Seq(52,85,1,8,3,2,7)
    Roll_number.foreach((element:Int) => print(element+" "))
    println("\nAccessing element by using index")
    println(Roll_number(2))

  }
}
```

Commonly used Methods of Seq

Method	Description
def contains[A1 >: A](elem: A1): Boolean	Check whether the given element present in this sequence.
def copyToArray(xs: Array[A], start: Int, len: Int): Unit	It copies the seq elements to an array.
def endsWith[B](that: GenSeq[B]): Boolean	It tests whether this sequence ends with the given sequence or not.
def head: A	It selects the first element of this seq collection.
def indexOf(elem: A): Int	It finds index of first occurrence of a value in this immutable sequence.
def isEmpty: Boolean	It tests whether this sequence is empty or not.
def lastIndexOf(elem: A): Int	It finds index of last occurrence of a value in this immutable sequence.
def reverse: Seq[A]	It returns new sequence with elements in reversed order.

```
object Scala_Collection_Seq_Methods {
  def main(args: Array[String]): Unit = {
    var seq:Seq[Int] = Seq(52,85,1,8,3,2,7)
    seq.foreach((element:Int) => print(element+" "))
    println("\nis Empty: "+seq.isEmpty)
    println("Ends with (2,7): "+ seq.endsWith(Seq(2,7)))
    println("contains 8: "+ seq.contains(8))
    println("last index of 3 : "+seq.lastIndexOf(3))
    println("Reverse order of sequence: "+seq.reverse)
  }
}
```

Scala List

List is used to store ordered elements. It extends LinearSeq trait. It is a class for immutable linked lists. This class is good for last-in-first-out (LIFO), stack-like access patterns.
It maintains order of elements and can contain duplicates elements also.

```
object Scala_Collection_List {
 def main(args: Array[String]): Unit = {
  // List of Strings
  var list = List("mohit","Xyz")
  // List of Integer
  var list2:List[Int] = List(1,8,5,6,9,58,23,15,4)
  // Empty List.
  // Two dimensional list
  val dim: List[List[Int]] =
  List(
   List(1, 0, 0),
   List(0, 1, 0),
   List(0, 0, 1)
  )
  println(list)
  println(list2)
  println(dim)
 }
}
```

Scala Lists

•Lists are immutable whereas arrays are mutable in Scala.
•Lists represents a linked list whereas arrays are flat.

All lists can be defined using two fundamental building blocks, a tail **Nil** and **::**, which is pronounced **cons**. Nil also represents the empty list. All the above lists can be defined as follows.

```
object Scala_Collection_List1 {
  def main(args: Array[String]): Unit = {
    // List of Strings
    val Stu_name = "Kapil" :: ("Mohit" :: ("Shaifali" :: Nil))

    // List of Integers
    val Roll = 1 :: (2 :: (3 :: (4 :: Nil)))

    // Empty List.
    val empty = Nil

    // Two dimensional list
    val dim = (1 :: (0 :: (0 :: Nil))) ::
      (0 :: (1 :: (0 :: Nil))) ::
      (0 :: (0 :: (1 :: Nil))) :: Nil
    print(Stu_name)
    print(dim)
  }
}
```

Basic Operations on Lists

All operations on lists can be expressed in terms of the following three methods.

Sr. No	Methods & Description
1	**head** This method returns the first element of a list.
2	**tail** This method returns a list consisting of all elements except the first.
3	**isEmpty** This method returns true if the list is empty otherwise false.

```
object Scala_Collection_List_Operations {
  def main(args: Array[String]): Unit = {
    val Stu_name = "x" :: ("y" :: ("z" :: Nil))
    val Stu_nums = Nil

    println( "Head of fruit : " +
Stu_name.head )
    println( "Tail of fruit : " + Stu_name.tail
)
    println( "Check if fruit is empty : " +
Stu_name.isEmpty )
    println( "Check if nums is empty : " +
Stu_nums.isEmpty )
  }

}
```

Concatenating Lists

we can use either **:::** operator or **List.:::()** method or **List.concat()** method to add two or more lists. Please find the following example given below −

```
object Scala_Collection_List_Concat {
  def main(args: Array[String]): Unit = {
    val Stu_name = "x" :: ("y" :: ("z" :: Nil))
    val Stu_Stream = "Mtech" :: ("Msc" :: Nil)

    // use two or more lists with ::: operator
    var all_info = Stu_name ::: Stu_Stream
    println( "Stu_name ::: Stu_Stream : " + all_info )

    // use two lists with Set.:::() method
    all_info = Stu_name.:::(Stu_Stream)
    println( "Stu_name.:::(Stu_Stream) : " + all_info )

    // pass two or more lists as arguments
    all_info = List.concat(Stu_name, Stu_Stream)
    println( "List.concat(Stu_name, Stu_Stream) : " + all_info  )
  }
}
```

Creating Uniform Lists

we can use **List.fill()** method creates a list consisting of zero or more copies of the same element. Try the following example program.

```
object Scala_Collection_List_Uniform {
  def main(args: Array[String]): Unit = {
    val Stu_name = List.fill(3)("Rohit") // Repeats apples three times.
    println( "Stu_name : " + Stu_name  )

    val num = List.fill(10)(2)         // Repeats 2, 10 times.
    println( "num : " + num  )
  }
}
```

Tabulating a Function

we can use a function along with **List.tabulate()** method to apply on all the elements of the list before tabulating the list. Its arguments are just like those of List.fill: the first argument list gives the dimensions of the list to create, and the second describes the elements of the list. The only difference is that instead of the elements being fixed, they are computed from a function.

Try the following example program.

Example

Scala functions are first class values. Difference between Scala Functions & Methods: **Function is a object which can be stored in a variable. But a method always belongs to a class which has a name, signature bytecode etc**. Basically, we can say a method is a function which is a member of some object.

```
object Scala_Class_Function {
 def main(args: Array[String]): Unit = {
   val myGetAreaFn= (rad:Double) =>
   {
     val PI =3.14
     PI * rad*rad
   }
   println(myGetAreaFn(2.2))
 }
}
```

```
Class scala_class_methods {

  def mygetareamd(rad:double):double=
  {
    Val pi =3.14
    PI* rad* rad
  }
}
  Object  scala_class_methods_obj
  {
    Def main(args: array[string]): unit = {
      val m1= new scala_class_methods()
     Println(m1.Mygetareamd(2.2))
    }
  }
```

Scala Queue

Queue implements a data structure that allows inserting and retrieving elements in a first-in-first-out (FIFO) manner.

In scala, Queue is implemented as a pair of lists. One is used to insert the elements and second to contain deleted elements. Elements are added to the first list and removed from the second list.

```
import scala.collection.immutable._
object Scala_Collection_Queue {
 def main(args: Array[String]): Unit = {
   var queue = Queue(1,5,6,2,3,9,5,2,5)
   var queue2:Queue[Int] =
Queue(1,5,6,2,3,9,5,2,5)
   println(queue)
   println(queue2)
 }
}
```

```
import scala.collection.immutable._
object Scala_Collection_Queue_Methods {
 def main(args: Array[String]): Unit = {
   var queue = Queue(1,5,6,2,3,9,5,2,5)
   print("Queue Elements: ")
   queue.foreach((element:Int)=>print(element+" "))
   var firstElement = queue.front
   print("\nFirst element in the queue: "+ firstElement)
   var enqueueQueue = queue.enqueue(100)
   print("\nElement added in the queue: ")
   enqueueQueue.foreach((element:Int)=>print(element+" "))
   var dequeueQueue = queue.dequeue
   print("\nElement deleted from this queue: "+ dequeueQueue)
 }
}
```

Scala Set map() method

The **map()** method is utilized to build a new set by applying a function to all elements of this set.

Method Definition: *def map[B](f: (A) => B): immutable.Set[B]*

```
object Scala_Collection_Maps_new {
  def main(args: Array[String]): Unit = {
//    Creating a set
    val s1 = Set(5, 1, 3, 2, 4)
    // Applying map method
//    val result = s1.map(x => x*x)
    val result1 = s1.map(x => x+2)
   // Display output
//    println(result)
    println(result1)
  }
}
```

Scala Map

Map is a collection of key-value pairs. In other words, it is similar to dictionary. Keys are always unique while values need not be unique. Key-value pairs can have any data type. However, data type once used for any key and value must be consistent throughout. Maps are classified into two types: *mutable* and *immutable*. By default Scala uses immutable Map. In order to use mutable Map, we must import **scala.collection.mutable.Map** class explicitly.

How to create Scala Maps

```
object scala_collection_maps {
  def main(args: array[string]): unit = {
//    var name:vector[string] = vector("kapil","somesh","rohit","mohit") //or
//    var rollnumber = vector(101,102,103,104,105)
    var name =map((1,"kapil"),(2,"somesh"))
// another way
    var name2 = map(3->"somesh",4->"ball")
    var newmap = name2+(5->"xyz")                // adding a new element to map
    var removeelement = newmap-4            //  // removing an element from map
    println(name)
    println(name2)
    println(newmap)
    println(removeelement)
  }
}
```

Operations on a Scala Map

There are three basic operations we can carry out on a Map:

keys: In Scala Map, This method returns an iterable containing each key in the map.
values: Value method returns an iterable containing each value in the Scala map.
isEmpty: This Scala map method returns true if the map is empty otherwise this returns false.

```
object Scala_Collection_Map_Accessing {
  def main(args: Array[String]): Unit = {
    var name2 = Map(3->"somesh",4->"Ball")
    // Accessing score of Ajay
    val Three= name2(3)
    println(Three)
  }
}
```

If we try to access value associated with the key “Kapil”, we will get an error because no such key is present in the Map. Therefore, it is recommended to use contains() function while accessing any value using key.
This function checks for the key in the Map. If the key is present then it returns true, false otherwise.

Updating the values
If we try to update value of an immutable Map, Scala outputs an error. On the other hand, any changes made in value of any key in case of mutable Maps is accepted.
Example:
Updating immutable Map:

```
object Scala_Collections_Map_Updating {
  def main(args: Array[String]): Unit = {
    val mapIm = Map("Ajay" -> 30,"Bhavesh" -> 20,"Charlie" -> 50)
    println(mapIm)
    //Updating
    mapIm("Ajay") = 10
    println(mapIm)
  }
}
```

```
val mapMut = scala.collection.mutable.Map("Ajay" -> 30,
                                          "Bhavesh" -> 20,
                                          "Charlie" -> 50)
println("Before Updating: " + mapMut)

// Updating
mapMut("Ajay") = 10

println("After Updating: " + mapMut)
```

Scala Access Modifier

Access modifier is used to define accessibility of data and our code to the outside world. You can apply accessibly to classes, traits, data members, member methods and constructors etc. Scala provides least accessibility to access to all. You can apply any access modifier to your code according to your application requirement.
Scala provides only three types of access modifiers, which are given below:

1. No modifier
2. Protected
3. Private

Scala Example: Private Access Modifier

In scala, private access modifier is used to make data accessible only within class in which it is declared. It is most restricted and keeps your data in limited scope. Private data members does not inherit into subclasses.

Scala Example: Private Access Modifier

In scala, private access modifier is used to make data accessible only within class in which it is declared. It is most restricted and keeps your data in limited scope. Private data members does not inherit into subclasses.

```
class Scala_Access_Modifier_Private {
  private var a: Int = 10

  def show() {
    println(a)
  }
}
object  MainObject{
  def main(args:Array[String]){
    var p = new Scala_Access_Modifier_Private()
    p.a = 12                                        //variable a in class
Scala_Access_Modifier_Private cannot be accessed in Scala_Access_Modifier_Private
    p.show()
  }
}
```

Scala Example: Protected Access Modifier

Protected access modifier is accessible only within class, sub class and companion object. Data members declared as protected are inherited in subclass. Let's see an example.

```
class Scala_Procted_Access_Specifier {
  protected var a:Int = 10
}
class SubClass extends Scala_Procted_Access_Specifier{
  def display(){
    println("a = "+a)
  }


}

object MainObject3 {
  def main(args: Array[String]) {
    var s = new SubClass()
    s.display()
  }
}
```

Scala Example: No-Access-Modifier

In scala, when you don't mention any access modifier, it is treated as no-access-modifier. It is same as public in java. It is least restricted and can easily accessible from anywhere inside or outside the package.

```
class Scala_No_Access_Modifier
 {
  var a:Int = 10
  def show(){
    println(" a = "+a)
  }
}
object MainObject5{
  def main(args:Array[String]){
    var a = new Scala_No_Access_Modifier()
    a.show()
  }
}
```

Scala Pattern Matching

Pattern matching is a feature of scala. It works same as switch case in other programming languages. It matches best case available in the pattern.

Let's see an example.

```
object Scala_Pattern {
 def main(args: Array[String]): Unit = {
  var a = 1
  a match {
   case 1 => println("One")
   case 2 => println("Two")
   case _ => println("No")
  }
 }
}
```

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.

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.

```
object Scala_Pattern_Match_AntType {
 def main(args: Array[String]): Unit = {
  var result = search("Hello")
  print(result)
 }

 def search(a: Any): Any = a match {
  case 1 => println("One")
  case "Two" => println("Two")
  case "Hello" => println("Hello")
  case _ => println("No")
 }
```

How to take Input from Users

```
import scala.io.StdIn._
object Scala_how_to_take_input {
  def main(args: Array[String]): Unit = {
    print("Enter a number: a and b ")
    val a= readInt()
    var length =
readFloat(),readChar(),readBoolean(),readDouble(),readChar()
    val b= readInt()
    println("The value of a & b is=: "+ a,b)
  }
}
```

Scala Exception Handling

Exception handling is a mechanism which is used to handle abnormal conditions. we can also avoid termination of your program unexpectedly.

Scala makes "checked vs unchecked" very simple. It doesn't have checked exceptions. All exceptions are unchecked in Scala, even SQLException and IOException.

Scala Program Example without Exception Handling

```
class Scala_Exception{
    def divide(a:Int, b:Int) = {
            a/b            // Exception occurred here
        println("Rest of the code is executing...")
    }
}
object MainObject{
    def main(args:Array[String]){
        var e = new ExceptionExample()
        e.divide(100,0)

    }
}
```

Scala Try Catch

Scala provides try and catch block to handle exception. The try block is used to enclose suspect code. The catch block is used to handle exception occurred in try block. we can have any number of try catch block in your program according to need.

Scala Try Catch Example

In the following program, we have enclosed our suspect code inside try block. After try block we have used a catch handler to catch exception. If any exception occurs, catch handler will handle it and program will not terminate abnormally.

```
class Scala_try_catch {
  def divide(a:Int, b:Int) = {
    try{
      a/b
    }catch{
      case e: Exception=>println(e)
    }
    println("Rest of the code is executing...")
  }
}
object  Scala_try_obj
{

  def main(args: Array[String]): Unit = {
    var e = new Scala_try_catch()
    e.divide(100,0)

  }
}
```

Scala Try Catch Example 2

In this example, we have two cases in our catch handler. First case will handle only arithmetic type exception. Second case has Throwable class which is a super class in exception hierarchy. The second case is able to handle any type of exception in your program. Sometimes when you don't know about the type of exception, you can use super class.

```
class Scala_try_Multiple_catch {
  def divide(a:Int, b:Int) = {
    try{
      a/b
      var arr = Array(1,2)
      arr(10)
    }catch{
      case e: ArithmeticException => println(e)
      case ex: Throwable =>println("found a unknown exception"+ ex)
    }
    println("Rest of the code is executing...")
  }
}
object Scala_try_mul_obj
{
  def main(args: Array[String]): Unit = {
    var e = new Scala_try_Multiple_catch()
    e.divide(100,10)

  }

}
```

Scala Finally

The finally block is used to release resources during exception. Resources may be file, network connection, database connection etc. the finally block executes guaranteed. The following program illustrate the use of finally block.

Scala Finally Block Example

```
class ExceptionExample{
   def divide(a:Int, b:Int) = {
      try{
         a/b
         var arr = Array(1,2)
         arr(10)
      }catch{
         case e: ArithmeticException => println(e)
         case ex: Exception =>println(ex)
         case th: Throwable=>println("found a unknown exception"+th)
      }
      finally{
         println("Finaly block always executes")
      }
      println("Rest of the code is executing...")
   }
}


object MainObject{
   def main(args:Array[String]){
      var e = new ExceptionExample()
      e.divide(100,10)

   }
}
```

custom exceptions

In scala, we can create your own exception. It is also known as custom exceptions. You must extend Exception class while declaring custom exception class. we can create your own exception message in custom class.

```
class InvalidAgeException(s:String) extends Exception(s){}
class Scala_Custom_Exception {
  @throws(classOf[InvalidAgeException])
  def validate(age:Int){
    if(age<18){
      throw new InvalidAgeException("Not eligible")
    }else{
      println("You are eligible")
    }
  }
}

object MainObjects{
  def main(args:Array[String]){
    var e = new Scala_Custom_Exception()
    try{
      e.validate(5)
    }catch{
      case e : Exception => println("Exception Occured : "+e)
    }
  }
}
```

List of String Method in Scala with Example

1. char charAt(int index)
This method returns the character at the index we pass to it. Isn't it so much like Java?

```
object Scala_String_methods {
  def main(args: Array[String]): Unit = {
    var name = "Mtech"
    print(name.charAt(1))
  }
}
```

2. int compareTo(Object o)

except that it compares two strings lexicographically. If they match, it returns 0. Otherwise, it returns the difference between the two(the number of characters less in the shorter string, or the maximum ASCII difference between the two).

```
object Scala_String_methods {
  def main(args: Array[String]): Unit = {
    var name = "Mtech"
    var Stream="BDA"
    println(name.compareTo(Stream))
  }
}
```

String concat(String str)
This will concatenate the string in the parameter to the end of the string on which we call it.

```
object Scala_String_methods {
  def main(args: Array[String]): Unit = {
    var name = "Mtech"
    var Stream="BDA"
println(name.concat(Stream))
  }
}
```

Boolean contentEquals(StringBuffer sb)
contentEquals compares a string to a StringBuffer’s contents. If equal, it returns true; otherwise, false.

```
object Scala_String_methods {
  def main(args: Array[String]): Unit = {
val a=new StringBuffer(“Mtech")
print(“Mtech".contentEquals(a))
  }
}
```

Boolean endsWith(String suffix)
This Scala String Method returns true if the string ends with the suffix specified; otherwise, false.

```
object Scala_String_methods {
  def main(args: Array[String]): Unit = {
print("Mtech".endsWith("h"))
  }
}
```

Boolean equals(Object anObject)
This Scala String Method returns true if the string and the object are equal; otherwise, false.

hashCode()
This int hashCode method returns a hash code for the string.

length()
This Scala String Method method simply returns the length of a string.

Break statement in Scala

In Scala, we use a break statement to break the execution of the loop in the program. Scala programing language does not contain any concept of break statement(in above 2.8 versions), instead of break statement, it provides a break method, which is used to break the execution of a program or a loop. Break method is used by importing scala.util.control.breaks._ package.

```
import scala.util.control.Breaks.{break, breakable}
import scala.util.control.BreakControl
object Scala_Break {
  def main(args: Array[String]): Unit = {
    for(i<- 1 to 10 )
      {
        breakable{
        if(i == 2)
          {
            break

          }
        else
          {
            print(i)
          }
        }
      }

  }
}
```

Scala Array

Array is a collection of mutable values. It is an index based data structure which starts from 0 index to n-1 where n is length of array.

Scala arrays can be generic. It means, we can have an Array[T], where T is a type parameter or abstract type. Scala arrays are compatible with Scala sequences - we can pass an Array[T] where a Seq[T] is required. It also supports all the sequence operations.

Following image represents the structure of array where first index is 0, last index is 9 and array length is 10.

Scala Types of array

Single dimensional array

Multidimensional array

Scala Single Dimensional Array

Single dimensional array is used to store elements in linear order. Array elements are stored in contiguous memory space. So, if you have any index of an array, you can easily traverse all the elements of the array.

Syntax for Single Dimensional Array

1. **var** arrayName : Array[arrayType] = **new** Array[arrayType](arraySize);   or
2. **var** arrayName = **new** Array[arrayType](arraySize) or
3. **var** arrayName : Array[arrayType] = **new** Array(arraySize);   or
4. **var** arrayName = Array(element1, element2 ... elementN)

```
object Scala_Array_ID {
  def main(args: Array[String]): Unit = {
    var arr = new Array[Int](5)
    arr(0)=15
    arr(1)=25
    arr(2)=35
    arr(3)=45
    arr(4)=55
   for(a<-arr)
     {
       println(a)
     }
```

Scala Multidimensional Array

Multidimensional array is an array which store data in matrix form. You can create from two dimensional to three, four and many more dimensional array according to your need. Below we have mentioned array syntax. Scala provides an ofDim method to create multidimensional array.

Multidimensional Array Syntax

1.var arrayName = Array.ofDim[ArrayType](NoOfRows,NoOfColumns) or

2.var arrayName = Array(Array(element...), Array(element...), ...)

Explanation of Case Class

A Case Class is just like a regular class, which has a feature for modeling unchangeable data.
It is also constructive in pattern matching. It has been defined with a modifier case,
due to this case keyword, we can get some benefits to stop oneself from doing a sections of
codes that have to be included in many places with little or no alteration.
As we can see below a minimal case class needs the keyword case class, an identifier,
and a parameter list which may be vacant.
Syntax:

Case class className(parameters)
Note: The Case class has a default apply() method which manages the construction of object.

Explanation of Case Object

A **Case Object** is also like an object, which has more attributes than a regular Object. It is a blend of both case classes and object. A case object has some more features than a regular object.
Below two are important features of case object:
- It is serializable.
- It has a by default hashCode implementation.

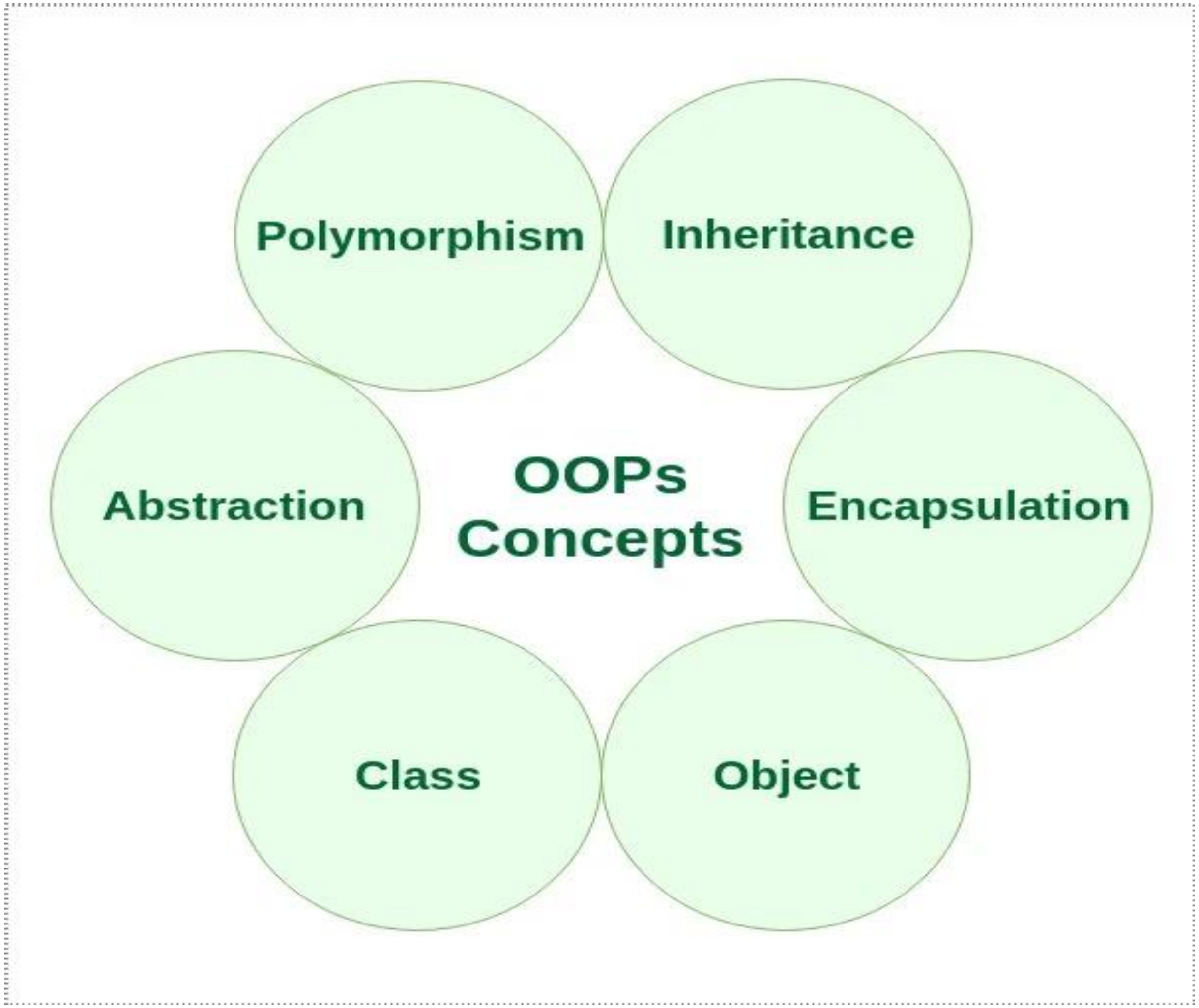
```
case class Scala_Case_Class(name:String, age:Int)
object  Scala_Case_CLass_Obj
{
  def main(args: Array[String]): Unit = {
    var c = Scala_Case_Class("Nidhi", 23)

    // Display both Parameter
    println("Name of the employee is " + c.name);
    println("Age of the employee is " + c.age);
  }
}
```

Build tools

Build tools. At the time of writing this article in April 2018 we have at our disposal:

- sbt
- cbt
- mill
- fury
- maven
- polyglot maven
- gradle
- ant (to be written)
- bazel (to be written)
- pants (to be written)
- make (to be written)

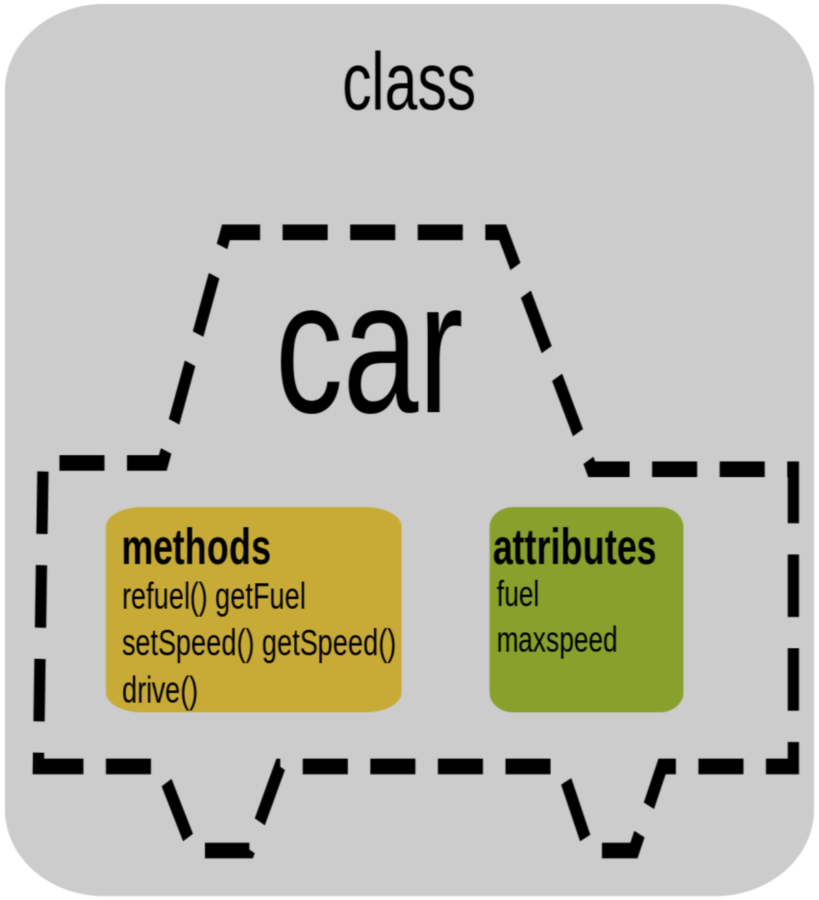
Introduction to OOPS



In general, Object-Oriented Programming (OOP) consists of classes and objects and aims to implement real-world entities like polymorphism, inheritance.

The class can be thought of as a representation or a design for objects. Classes will usually have their own **methods** (behavior) and **attributes**. Attributes are individual entities that differentiate each object from the other and determine various qualities of an object. Methods, on the other hand, are more like how a function usually operates in programming. They determine how the instance of the class works. It's mostly because of methods (behavior); objects have the power to be done something to them.

The above figure gives you more intuition about the flow of object-oriented programming or, to be more specific, what a class looks like. In the above picture, there is a class car which has attributes: fuel, max speed, and can have more attributes like the model, make, etc. It has different sets of methods like refuel(), getFuel(), setSpeed(), and some additional methods can be change gear, start the engine, stop the engine, etc.



Classes and Objects in Scala

Much like c++ and java, object-oriented programming in Scala follows pretty much the same conventions. It has the concept of defining classes and objects and within class constructors, and that is all there is to object-oriented programming in Scala.

Class Declaration

```
class Class_name{
// methods and attributes
}
```

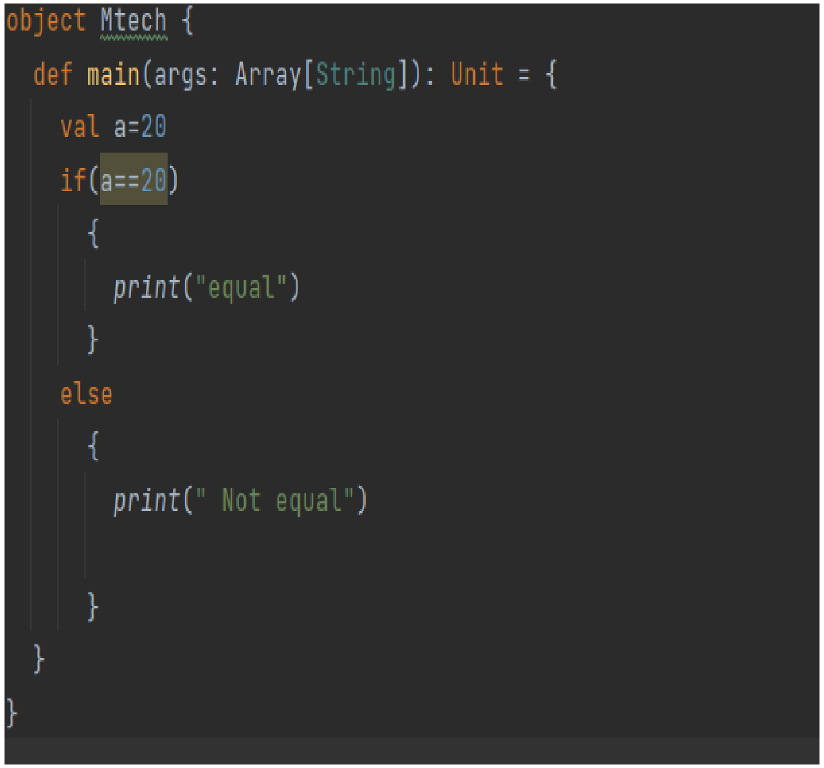
Class in Scala is defined by the keyword class followed by the name of the class, and generally, the class name starts with a capital letter. There are few keywords which are optional but can be used in Scala class declaration like: class-name, it should begin with a capital letter: superclass, the parent class name preceded by extend keyword: traits, it is a comma-separated list implemented by the class preceded by extend keyword.

A class can in Scala inherits only one parent class, which means Scala does not support multiple inheritances. However, it can be achieved with the use of Traits.

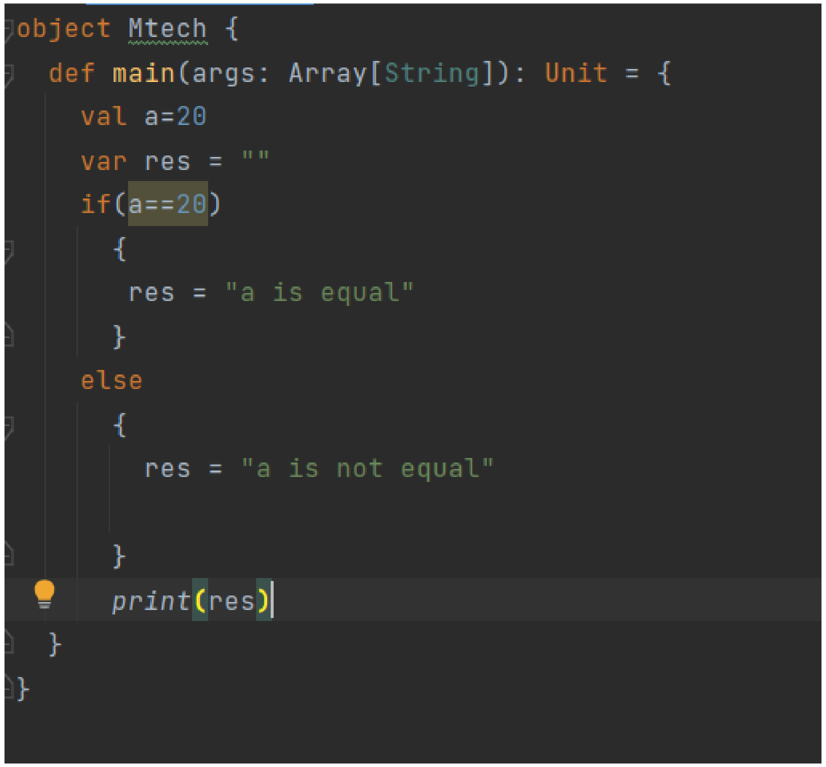
Finally, the body of a class in Scala is surrounded by curly braces {}

```
class Car {
  // Class variables
  var make: String = "BMW"
  var model: String = "X7"
  var fuel: Int = 40

  // Class method
  def Display()
  {
    println("Make of the Car : " + make);
    println("Model of the Car : " + model);
    println("Fuel capacity of the Car : " + fuel);
  }
}
object Main_Car_calss
{

  // Main method
  def main(args: Array[String])
  {

    // Class object
    var obj = new Car();
    obj.Display();
  }
}
```

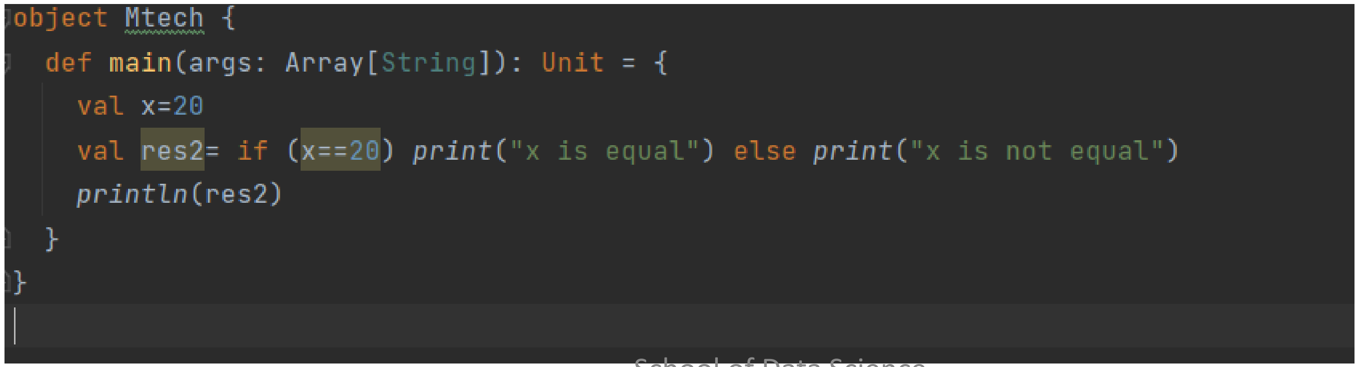
If Else Statement



```
object Mtech {
  def main(args: Array[String]): Unit = {
    val a=20
    if(a==20)
      {
        print("equal")
      }
    else
      {
        print(" Not equal")

      }
  }
}
```

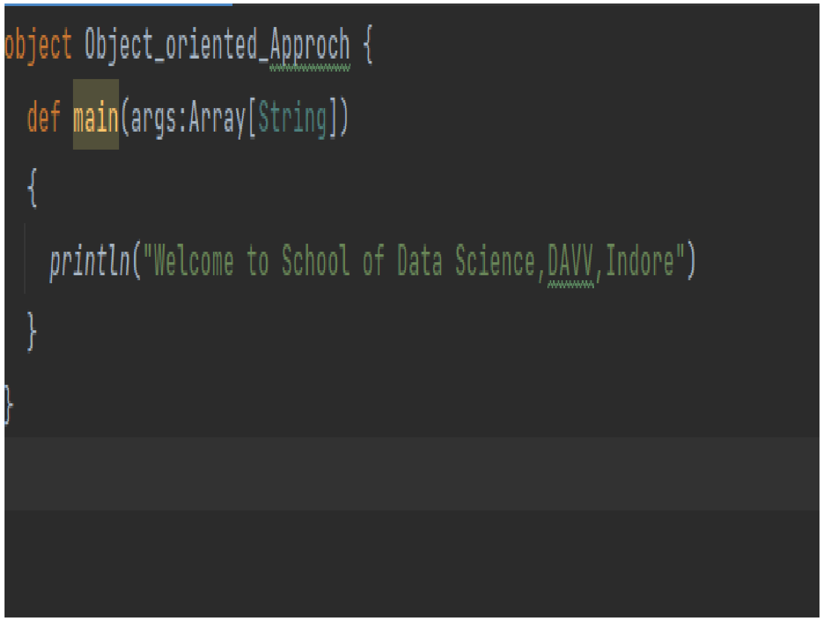


```
object Mtech {
  def main(args: Array[String]): Unit = {
    val a=20
    var res = ""
    if(a==20)
      {
        res = "a is equal"
      }
    else
      {
         res = "a is not equal"

      }
      print(res)
  }
}
```



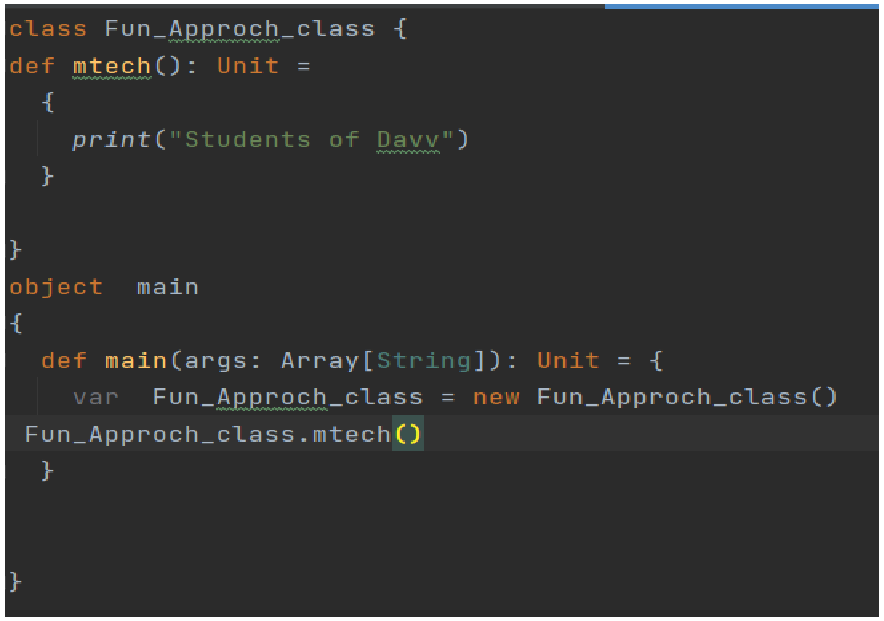
```
object Mtech {
  def main(args: Array[String]): Unit = {
    val x=20
    val res2= if (x==20) print("x is equal") else print("x is not equal")
    println(res2)
  }
}
```

Object Oriented Approach



```
object Object_oriented_Approch {
  def main(args:Array[String])
  {
    println("Welcome to School of Data Science,DAVV,Indore")
  }
}
```

Funtional Approach



```
class Fun_Approch_class {
def mtech(): Unit =
  {
    print("Students of Davv")
  }

}
object  main
{
  def main(args: Array[String]): Unit = {
    var  Fun_Approch_class = new Fun_Approch_class()
 Fun_Approch_class.mtech()
  }


}
```

Object Oriented Approach

Case Class

Scala case classes are like regular classes except for the fact that they are good for modeling immutable data and serve as useful in pattern matching. Case classes include public and immutable parameters by default. These classes support pattern matching, which makes it easier to write logical code. **The following are some of the characteristics of a Scala case class:**

- Instances of the class can be created without the new keyword.
- As part of the case classes, Scala automatically generates methods such as equals(), hashcode(), and toString().
- Scala generates accessor methods for all constructor arguments for a case class.

```
case class Case_Class(name:String, age:Int)
object MainObject
{
  def main(args:Array[String])
  {
    var c = Case_Class("Xyz", 23)
    println("Student name:" + c.name);
println("Student age: " + c.age);
  }
}
```

Closure in Scala

```scala
class Clousre_feature {
  var x = 20
  def function_name(y:Int)
  {
    println(x+y)
  }

}
object  main
{
  def main(args: Array[String]): Unit = {
    var obj = new Clousre_feature()
    obj.function_name(3)
  }

}
```

Scala closures are functions whose return value is dependent on one or more free variables declared outside the closure function. Neither of these free variables is defined in the function nor is it used as a parameter, nor is it bound to a function with valid values. Based on the values of the most recent free variables, the closing function is evaluated.

Traits in Scala

The concept of traits is similar to an interface in Java, but they are even more powerful since they let you implement members. It is composed of both abstract and non-abstract methods, and it features a wide range of fields as its members. Traits can either contain all abstract methods or a mixture of abstract and non-abstract methods. In computing, a trait is defined as a unit that encapsulates the method and its variables or fields. Furthermore, Scala allows partial implementation of traits, but no constructor parameters may be included in traits. To create traits, use the trait keyword.

```
trait MyCompany
{
def company
def position
}
 class MyClass extends MyCompany
{
def company()
{
println("Company: InterviewBit")
 }
def position()
{
println("Position: SoftwareDeveloper")
}
def employee() //Implementation of class method
{
println("Employee: Aarav")
}
}
object Main
{
def main(args: Array[String])
 {
 val obj = new MyClass(); obj.company();
 obj.position();
 Obj.employee();
}
}
```

Nil, Null, None and Nothing

Null, null, Nil, Nothing, None, and Unit are all used to represent empty values in Scala.

•Null refers to the absence of data, and its type is Null with a capital N.

•Null is considered to be a list that contains zero items. Essentially, it indicates the end of the list.

•Nothing is also a trait without instances.

•None represents a sensible return value.

•Unit is the return type for functions that return no value.

Inheritance in Scala

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in Scala by which one class is allowed to inherit the features(fields and methods) of another class.
Important terminology:

Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).
Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Scala
The keyword used for inheritance is extends.
Syntax:

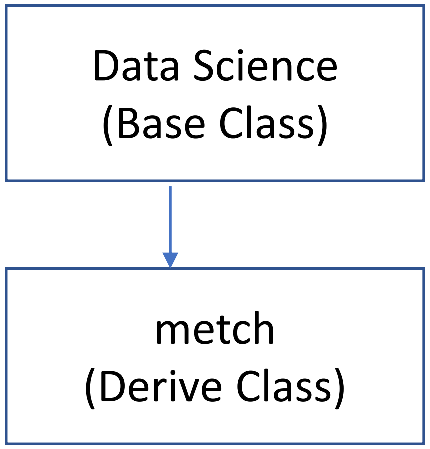
```
class child_class_name extends
parent_class_name {
// Methods and fields
}
```

```
class Data_Science {
 var Name: String = "Mtech"

}
class mtech extends Data_Science
{

 var Roll_No: Int = 130
 // Method// Method

 def details()
 {
  println("Stream name: " +Name);
  println("Roll number of student: " +Roll_No);
 }
}

object Main_Inheritance
{

 // Driver code
 def main(args: Array[String])
 {

  // Creating object of derived class
  val ob = new mtech();
  ob.details();
 }
}
```

Type of inheritance

Below are the different types of inheritance which are supported by Scala.

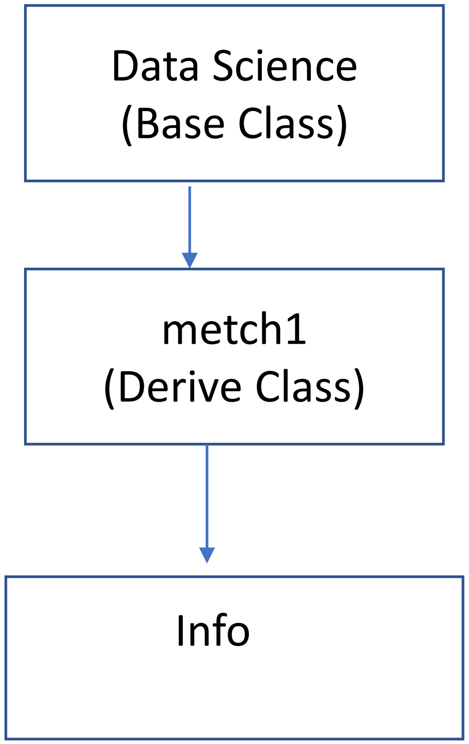
Single Inheritance: In single inheritance, derived class inherits the features of one base class. In the image below, class Data Science serves as a base class for the derived class metch.



```
class Data_Science {
  var Name: String = "Mtech"

}
class mtech extends Data_Science
{

  var Roll_No: Int = 130
  // Method// Method

  def details()
  {
    println("Stream name: " +Name);
    println("Roll number of student: " +Roll_No);
  }
}

object Main_Inheritance
{

  // Driver code
  def main(args: Array[String])
  {

    // Creating object of derived class
    val ob = new mtech();
    ob.details();
  }
}
```

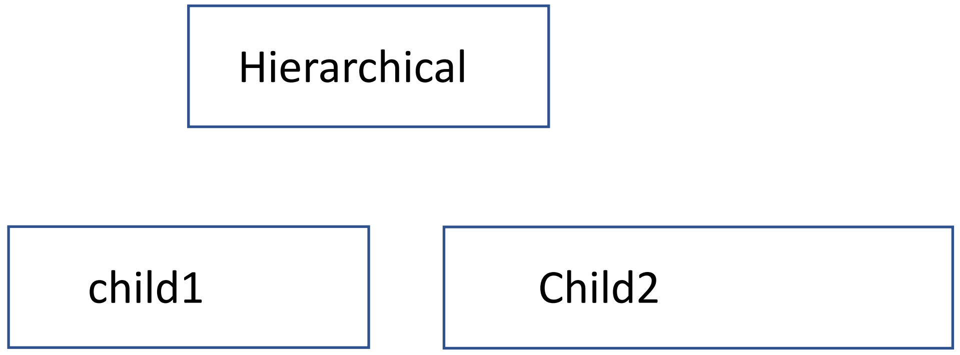
Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to another class. In the below image, the class Data Science serves as a base class for the derived class metch1 , which in turn serves as a base class for the derived class Info.



```
class Multilevel {
  var Name: String = "Mtech"
}
class mtech1 extends Multilevel {

  var Roll_No: Int = 130
}
class info extends mtech1
{
  // Method
  def details()
{
    println("Name: " +Name);
    println("Roll_number: " +Roll_No);
  }

}
object Multilevel_Inheritance
{
  def main(args: Array[String]): Unit = {
    val ob = new info();
    ob.details();
  }
}
```

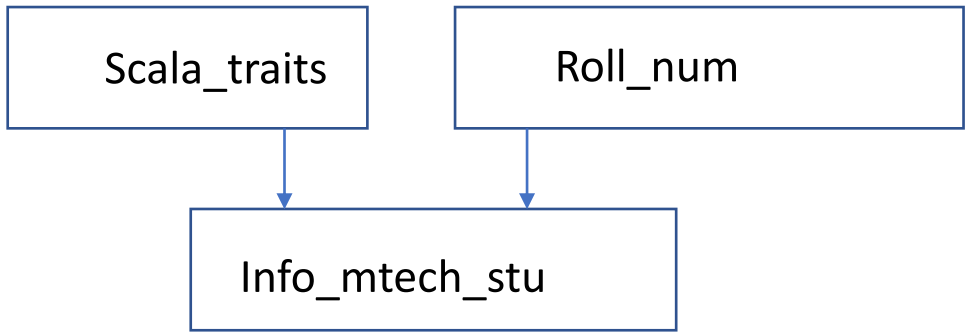
Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.In below image, class Hierarchical serves as a base class for the derived class child1 , child2.



```
class Hierarchical {
  var Name1: String = "Siya"
  var Name2: String = "Soniya"
}
class Child1 extends Hierarchical
{
  var Age: Int = 32
  def details1()
  {
    println(" Name: " +Name1);
    println(" Age: " +Age);
  }
}
// Derived from Parent class
class Child2 extends Hierarchical
{
  var Height: Int = 164

  // Method
  def details2()
  {
    println(" Name: " +Name2);
    println(" Height: " +Height);
  }
}

object Main_Hierarchical
{

  // Driver code
  def main(args: Array[String])
  {

    // Creating objects of both derived classes
    val ob1 = new Child1();
    val ob2 = new Child2();
    ob1.details1();
    ob2.details2();
  }
}
```

```
class Scala_Traits {
  var Name: String = "Mtech"
}
class  Roll_num {

  var Roll_No: Int = 130
}

class info_mtech_stu extends  Scala_Traits with Roll_num {

  def display()
  {
    println("Name: " +Name);
    println("Roll_number: " +Roll_No);

  }

}
object traits_scala
{
  def main(args: Array[String]): Unit = {
    val ob = new info_mtech_stu();
    ob.display();
  }
}
```

•**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.



```
trait Scala_Traits {
  var Name: String = "Mtech"
}
trait  Roll_num {

  var Roll_No: Int = 130
}

class info_mtech_stu extends  Scala_Traits with Roll_num{

  def display()
  {
    println("Name: " +Name);
    println("Roll_number: " +Roll_No);

  }

}
object traits_scala
{
  def main(args: Array[String]): Unit = {
    val ob = new info_mtech_stu();
    ob.display();
  }
}
```

Hybrid Inheritance: It is a mix of two or more of the above types of inheritance. Since Scala doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In Scala, we can achieve hybrid inheritance only through traits.

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[dr] & & C \arrow[dl] \\
 & D &
\end{tikzcd}$$

```
class A {  var numA: Int = 0;
  def setA(n: Int) {    numA = n;
  }
  def printA() { printf("numA: %d\n", numA);
  }
}
class B extends A {var numB: Int = 0;
  def setB(n: Int) {  numB = n;
  }
  def printB() {  printf("numB: %d\n", numB);
  }
}
class C extends B {  var numC: Int = 0;
  def setC(n: Int) {    numC = n;
  }
  def printC() {    printf("numC: %d\n", numC);
  }
}
class D extends A {  var numD: Int = 0;
  def setD(n: Int) {    numD = n;
  }
  def printD() {    printf("numD: %d\n", numD);
  }
}
object  hybrid_inheritance
{
  def main(args: Array[String]): Unit = {
    var obj1 = new C();
    var obj2 = new D();
    obj1.setA(10);    obj1.setB(20);    obj1.setC(30);    obj1.printA();    obj1.printB();
    obj1.printC();    obj2.setA(40);    obj2.setD(50);
    obj2.printA();    obj2.printD();
  }
}
```

Traits	Abstract Class
Traits support multiple inheritance.	Abstract class does not support multiple inheritance.
We are allowed to add a trait to an object instance.	We are not allowed to add an abstract class to an object instance.
Traits does not contain constructor parameters.	Abstract class contain constructor parameters.
Traits are completely interoperable only when they do not contain any implementation code.	Abstract class are completely interoperable with Java code.

Scala Polymorphism

Polymorphism is the ability of any data to be processed in more than one form. The word itself indicates the meaning as poly means many and morphism means types. Scala implements polymorphism through virtual functions, overloaded functions and overloaded operators.

Polymorphism is one of the most important concept of object oriented programming language. The most common use of polymorphism in object oriented programming occurs when a parent class reference is used to refer to a child class object. Here we will see how represent any function in many types and many forms. Real life example of polymorphism, **a person at the same time can have different roles to play in life. Like a woman at the same time is a mother, a wife, an employee and a daughter. So the same person has to have many features but has to implement each as per the situation and the condition**. Polymorphism is considered as one of the important features of Object Oriented Programming. In Scala the function can be applied to arguments of many types, or the type can have instances of many types.

Two to Achieve Polymorphism

Scala Method Overloading

Scala provides method overloading feature which allows us to define methods of same name but having different parameters or data types. It helps to optimize code.

Scala Method Overloading Example by using Different Parameters

In the following example, we have define two add methods with different number of parameters but having same data type.

```
class Arithmetic{
    def add(a:Int, b:Int){
        var sum = a+b
        println(sum)
    }
    def add(a:Int, b:Int, c:Int){
        var sum = a+b+c
        println(sum)
    }
}

object MainObject{
    def main(args:Array[String]){
        var a  = new Arithmetic();
        a.add(10,10);
        a.add(10,10,10);
    }
}
```

Method Overriding in Scala

```
class Scala_Overridding {
 def NumberOfStudents()=
 {   0 // Utilized for returning an Integer
 }
}
class class_1 extends Scala_Overridding {
 // Using Override keyword
 override def NumberOfStudents() = {
  30
 }
}
class class_2 extends Scala_Overridding
{
 // Using override keyword
 override def NumberOfStudents()=
 {
  32  }
}class class_3 extends Scala_Overridding
{  // Using override keyword
 override def NumberOfStudents()=
 {   29  }
}
object Scala_Overridding_Obj{
 def main(args: Array[String]): Unit = {
  var x=new class_1()
  var y=new class_2()
  var z=new class_3()
  // Displays number of students in class_1
  println("Number of students in class 1 : " +
   x.NumberOfStudents())
  // Displays number of students in class_2
  println("Number of students in class 2 : " +
   y.NumberOfStudents())

  // Displays number of students in class_3
  println("Number of students in class 3 : " +
   z.NumberOfStudents())
 }
}
```

Method Overriding in Scala is identical to the method overriding in Java but in Scala, the overriding features are further elaborated as here, both methods as well as var or val can be overridden. If a subclass has the method name identical to the method name defined in the parent class then it is known to be Method Overriding i.e, the sub-classes which are inherited by the declared super class, overrides the method defined in the super class utilizing the override keyword.

In the above example, we have a class named Scala_Overridding which defines a method NumberOfStudents() and we have three classes i.e, class_1, class_2 and class_3 which inherit from the super-class School and these sub-classes overrides the method defined in the super-class.

Overriding vs Overloading

In Scala, method overloading supplies us with a property which permits us to define methods of identical name but they have different parameters or data types whereas, method overriding permits us to redefine method body of the super class in the subclass of same name and same parameters or data types in order to alter the performance of the method.

In Scala, method overriding uses override modifier in order to override a method defined in the super class whereas, method overloading does not requires any keyword or modifier, we just need to change, the order of the parameters used or the number of the parameters of the method or the data types of the parameters for method overloading.

Lambda Expression in Scala

Lambda Expression refers to an expression that uses an anonymous function instead of variable or value. Lambda expressions are more convenient when we have a simple function to be used in one place. These expressions are faster and more expressive than defining a whole function. We can make our lambda expressions reusable for any kind of transformations. It can iterate over a collection of objects and perform some kind of transformation to them.

Syntax:

```
val lambda_exp = (variable:Type) => Transformation_Expression
```

Example:

```
// lambda expression to find double of x
val ex = (x:Int) => x + x
```

Working With Lambda Expressions

We can pass values to a lambda just like a normal function call.

Example :

```
object Scala_lambda {
  def main(args: Array[String]): Unit = {
    // lambda expression
    val ex1 = (x:Int) => x + 2

    // with multiple parameters
    val ex2 = (x:Int, y:Int) => x * y

    println(ex1(7))
    println(ex2(2, 3))
  }
}
```

We can see that the defined anonymous function to perform the square operation is not reusable.

We are passing it as an argument. However, we can make it reusable and may use it with different collections.

```scala
object Scala_Lambda_reuse {
  def main(args: Array[String]): Unit = {
    val l1 = List(1, 1, 2, 3, 5, 8)
    val l2 = List(13, 21, 34)

    // reusable lambda
    val func = (x:Int) => x * x

    // squaring each element of the lists
    val res1 = l1.map( func )
    val res2 = l2.map( func )

    println(res1)
    println(res2)
  }
}
```

A lambda can also be used as a parameter to a function.

```
object Scala_lambda_AsParameter {

  // transform function with integer x and
  // function f as parameter
  // f accepts Int and returns Double
  def transform( x:Int, f:Int => Double)
  =
    f(x)
  def main(args: Array[String]): Unit = {

    // lambda is passed to f:Int => Double
    val res = transform(2, r => 3.14 * r * r)

    println(res)
  }
}
```

In above example, transform function accepts integer x and function f, applies the transformation to x defined by f. Lambda passed as the parameter in function call returns Double type. Therefore, parameter f must obey the lambda definition.

We can perform the same task on any collection as well. In case of collections, the only change we need to make in transform function is using map function to apply transformation defined by f to every element of the list l.

```
object Scala_lambda_WithAnyCollection {
 def transform( l:List[Int], f:Int => Double)
 = {
   l.map(f)



   }
 def main(args: Array[String]): Unit = {
   // lambda is passed to f:Int => Double
   val res = transform(List(1, 2, 3), r => 3.14 * r * r)
   println(res)
 }
}
```

Anonymous Functions in Scala

In Scala, An anonymous function is also known as a function literal. A function which does not contain a name is known as an anonymous function. An anonymous function provides a lightweight function definition. It is useful when we want to create an inline function.

Syntax:

```
(z:Int, y:Int)=> z*y
Or
(_:Int)*(_Int)
```

In the above first syntax, => is known as a transformer. The transformer is used to transform the parameter-list of the left-hand side of the symbol into a new result using the expression present on the right-hand side.

In the above second syntax, _ character is known as a wildcard is a shorthand way to represent a parameter who appears only once in the anonymous function.

Anonymous Functions With Parameters

When a function literal is instantiated in an object is known as a function value. Or in other words, when an anonymous function is assigned to a variable then we can invoke that variable like a function call. We can define multiple arguments in the anonymous function.

```
object Scala_Anonymous_Fun {
  def main(args: Array[String]): Unit = {
    // Creating anonymous functions
    // with multiple parameters Assign
    // anonymous functions to variables
//    var myfc1 = (str1:String, str2:String) => str1
+ str2
    var myfc1 = (str1:Int, str2:Int) => str1 + str2
    // An anonymous function is created
    // using _ wildcard instead of
    // variable name because str1 and
    // str2 variable appear only once
    var myfc2 = (_:String) + (_:String)

    // Here, the variable invoke like a function
call
    println(myfc1(1, 2))
    println(myfc2("MBA", "BBA"))
```

Anonymous Functions Without Parameters

We are allowed to define an anonymous function without parameters.

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

    var myfun1 = () => {"Welcome to School of Data Science!!"}
    println(myfun1())

  }
}
```

Scala Closures

Scala Closures are functions which uses one or more free variables and the return value of this function is dependent of these variable. The free variables are defined outside of the Closure Function and is not included as a parameter of this function. So the difference between a closure function and a normal function is the free variable. A free variable is any kind of variable which is not defined within the function and not passed as the parameter of the function. A free variable is not bound to a function with a valid value. The function does not contain any values for the free variable.

```
class Clousre_feature
{
 var x = 20
 def function_name(y:Int)
 {
  var x=12;
  println(x+y)
 }

}
object  main
{
 def main(args: Array[String]): Unit = {
  var obj = new Clousre_feature()
  obj.function_name(3)
 }
```

Scala Final

Final is a keyword, which is used to prevent inheritance of super class members into derived class. You can declare final variables, methods and classes also.

Scala Final Variable Example

You can't override final variables in subclass. Let's see an example.

```
class Scala_Vechile {
  final val speed:Int = 60
}
class Bike extends Scala_Vechile{
  override val speed:Int = 100
  def show(){
    println(speed)
  }
}
object MainObject{
  def main(args:Array[String]){
    var b = new Bike()
    b.show()
  }
}
```

```
cannot override final member:
final val speed: Int (defined in class
Scala_Vechile)
  override val speed:Int = 100
```

Scala Final Member

```
class Scala_Vechile {
  final val speed:Int = 60
}
class Bike extends Scala_Vechile{
  override val speed:Int = 100
  def show(){
    println(speed)
  }
}
object MainObject{
  def main(args:Array[String]){
    var b = new Bike()
    b.show()
  }
}
```

Scala Final Method

```
class Scala_Vehicle{
    final def show(){
        println("vehicle is running")
    }
}
class Bike extends Scala_Vehicle{
   //override val speed:Int = 100
    override def show(){
        println("bike is running")
    }
}
object MainObject{
    def main(args:Array[String]){
        var b = new Bike()
        b.show()
    }
}
```

Scala Final class

```
final class Vehicle{
     def show(){
         println("vehicle is
running")
     }

}

class Bike extends Vehicle{
       override def show(){
        println("bike is running")
    }
}

object MainObject{
    def main(args:Array[String]){
        var b = new Bike()
        b.show()
    }
}
```

Cannot override final member,Final Methods, Final Class

Scala this

In scala, this is a keyword and used to refer current object. we can call instance variables, methods, constructors by using this keyword.

```
class Scala_This {
  var id:Int = 0
  var name: String = ""
  def this(id:Int, name:String){
    this()
    this.id = id
    this.name = name
  }
  def show(){
    println(id+" "+name)
  }
}
object this_main
{

  def main(args: Array[String]): Unit = {
    var t = new Scala_This(101,"Martin")
    t.show()
  }
}
```

Scala Constructor Calling by using this keyword

In the following example this is used to call constructor. It illustrates how we can call constructor from other constructor. we must make sure that this must be first statement in the constructor while calling to other constructor otherwise compiler throws an error.

Scala | yield Keyword

yield keyword will returns a result after completing of loop iterations. The for loop used buffer internally to store iterated result and when finishing all iterations it yields the ultimate result from that buffer. It doesn’t work like imperative loop. The type of the collection that is returned is the same type that we tend to were iterating over, Therefore a Map yields a Map, a List yields a List, and so on.

```
object Scala_Yield {
  def main(args: Array[String]): Unit = {
    // Using yield with for
    var print = for( i <- 1 to 10) yield i
    for(j<-print)
    {
      // Printing result
      println(j)
    }
  }
}
```

Scala Singleton and Companion Objects

Singleton Object

Scala is more object oriented language than Java so, Scala does not contain any concept of static keyword. Instead of static keyword Scala has singleton object. A Singleton object is an object which defines a single object of a class. A singleton object provides an entry point to our program execution. If we do not create a singleton object in our program, then our code compile successfully but does not give output. So we required a singleton object to get the output of our program. A singleton object is created by using object keyword.

Syntax:

```
object Name{
// code...
}
```

Important points about singleton object

The method in the singleton object is globally accessible.
You are not allowed to create an instance of singleton object.
You are not allowed to pass parameter in the primary constructor of singleton object.
In Scala, a singleton object can extend class and traits.
In Scala, a main method is always present in singleton object.
The method in the singleton object is accessed with the name of the object(just like calling static method in Java), so there is no need to create an object to access this method.

```
object Scala_Singleton {
  def hello(){
   print("hello Mtech students")
  }
  def main(args: Array[String]): Unit = {
    Scala_Singleton.hello()

  }
}
```

```
object Scala_Singleton {
  def hello():Int={
    var x=10
    return  x
  }
  def main(args: Array[String]): Unit = {
    var c=Scala_Singleton.hello()
    print(c)
  }
}
```

Scala Companion Object

In scala, when we have a class with same name as singleton object, it is called companion class and the singleton object is called companion object

The companion class and its companion object both must be defined in the same source file.

```
class Scala_ComapanionClass {
  def hello(){
    println("Hello, this is Companion Class.")
  }
}
object CompanoinObject{
  def main(args:Array[String]){
    new Scala_ComapanionClass().hello()
    println("And this is Companion Object.")
  }
}
```

Scala Case Classes and Case Object

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching.It uses equal method to compare instance structurally. It does not use new keyword to instantiate object.All the parameters listed in the case class are public and immutable by default.

Syntax
case class className(parameters)

```
case class Scala_Case_Class(name:String, 
age:Int)
case class Scala_Case_Class1(name:String, 
age:Int,roll:Int)
object  Scala_Case_CLass_Obj
{
  def main(args: Array[String]): Unit = {
//    var c = Scala_Case_Class("Nidhi", 23)
    var d = Scala_Case_Class1("Nidhi", 23, 34)

    // Display both Parameter
    println("Name of the employee is " + 
d.name);
    println("Age of the employee is " + d.roll);
  }
}
```

Constructors are used to initializing the object’s state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.
Scala supports two types of constructors:

Primary Constructor
When our Scala program contains only one constructor, then that constructor is known as a primary constructor. The primary constructor and the class share the same body, means we need not to create a constructor explicitly.

Syntax:

```
class class_name(Parameter_list){
// Statements...
}
```

Important points:

In the above syntax, the primary constructor and the class share the same body so, anything defined in the body of the class except method declaration is the part of the primary constructor.
Example:

```
class Scala_Primary_Constructor(id:Int, name:String)
 {
  def showDetails(){
    println(id+" "+name);
  }
}
object Scala_Primary_Constructor{

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

      var s = new Scala_Primary_Constructor(101,"Rama");
      s.showDetails()
    }
}
```

The primary constructor may contain zero or more parameters.
If we do not create a constructor in our Scala program, then the compiler will automatically create a primary constructor when we create an object of our class, this constructor is known as a default primary constructor. It does not contain any parameters.
Example:

```
class Scala_Default_Primary_Constructor {
  def show() {
    println("Hello from default constructor");

  }
}
object Scala_default{
  def main(args: Array[String]): Unit = {
    var obj= new
Scala_Default_Primary_Constructor()
    obj.show()
  }


}
```

Auxiliary Constructor

In a Scala program, the constructors other than the primary constructor are known as auxiliary constructors. we are allowed to create any number of auxiliary constructors in our program, but a program contains only one primary constructor.

Syntax:

```
def this(......)
```

Important points:

In a single program, we are allowed to create multiple auxiliary constructors, but they have different signatures or parameter-lists.
Every auxiliary constructor must call one of the previously defined constructors.
The invoke constructor may be a primary or another auxiliary constructor that comes textually before the calling constructor.
The first statement of the auxiliary constructor must contain the constructor call using this.

```
class Scala_Primary_Constructor(id:Int, name:String)
 {
  def showDetails(){
    println(id+" "+name);
  }
}
object Scala_Primary_Constructor{

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

      var s = new Scala_Primary_Constructor(101,"Rama");
      s.showDetails()
    }
}
```

```
class Scala_Auxiliary_Constructor(Aname: String, Cname: String) {
  var no: Int = 0;;
  def display()
  {
    println("Author name: " + Aname);
    println("Chapter name: " + Cname);
    println("Total number of articles: " + no);

  }

  // Auxiliary Constructor
  def this(Aname: String, Cname: String, no:Int)
  {

    // Invoking primary constructor
    this(Aname, Cname)
    this.no=no
  }
}
object Main_Auxiliary_Constructor
{
  def main(args: Array[String])
  {

    // Creating object of GFG class
    var obj = new Scala_Auxiliary_Constructor("Anya",
"Constructor", 34);
    obj.display();
  }
}
```

In Scala, we are allowed to make a primary constructor private by using a private keyword in between the class name and the constructor parameter-list.
Syntax:

```
// private constructor with two argument
class constructor private(name: String, class:Int){
// code..
}

// private constructor without argument
class constructor private{
// code...
}
```

In Scala, we are allowed to give default values in the constructor declaration.
Example:

Scala Trait Mixins

In scala, trait mixins means we can extend any number of traits with a class or abstract class. we can extend only traits or combination of traits and class or traits and abstract class.

It is necessary to maintain order of mixins otherwise compiler throws an error.

we can use mixins in scala like this:

Scala Trait Example: Mixins Order Not Maintained
In this example, we have extended a trait and an abstract class. Let's see what happen.

```
trait Print {
  def print()
}
abstract class PrintA4{
  def printA4()
}
class A6 extends Print with PrintA4{
  def print(){              // Trait print
    println("print sheet")
  }
  def printA4(){            // Abstract class printA4
    println("Print A4 Sheet")
  }
}
object MainObject_trait {
  def main(args: Array[String]) {
    var a = new A6()
    a.print()
    a.printA4()
  }
}
```

class PrintA4 needs to be a trait to be mixed in
class A6 extends Print with PrintA4{

Scala Mixins Order
The right mixins order of trait is that any class or abstract class which you want to extend, first extend this. All the traits will be extended after this class or abstract class.

Scala Trait Example: Mixins Order Maintained

```
trait Scala_Mixims_Traits {
  def print()
}
abstract class PrintA4{
  def printA4()
}

class A6 extends PrintA4 with Scala_Mixims_Traits{          // First one is abstract class
second one is trait
  def print(){                            // Trait print
    println("print sheet")
  }
  def printA4(){                          // Abstract class printA4
    println("Print A4 Sheet")
  }
}
object MainObject_Maxims_Traits{
  def main(args:Array[String]){
    var a = new A6()
    a.print()
    a.printA4()
  }
}
```

Currying Functions in Scala with Examples

Currying in Scala is simply a technique or a process of transforming a function. This function takes multiple arguments into a function that takes single argument. It is applied widely in multiple functional languages.

Syntax

```
def function name(argument1, argument2) = operation
```

Let’s understand with a simple example,

Example:

```
object Scala_Currying {
  // Define currying function
  def add(x: Int, y: Int) = x + y;
  def main(args: Array[String]): Unit = {
    println(add(20, 19));
  }
}
```

Here, we have define add function which takes two arguments (x and y) and the function simply adds x and y and gives us the result, calling it in the main function.

Another way to declare currying function
Suppose, we have to transform this add function into a Curried function, that is transforming the function that takes two(multiple) arguments into a function that takes one(single) argument.

Syntax

def function name(argument1) = (argument2) => operation
Example

```
object Scala_Currying_Another_Way {
  // transforming the function that
  // takes two(multiple) arguments into
  // a function that takes one(single) argument.
  def add2(a: Int) = (b: Int) => a + b;

  def main(args: Array[String]): Unit = {
    println(add2(20)(19));
  }
}
```

Here, we have define add2 function which takes only one argument a and we are going to return a second function which will have the value of add2. The second function will also take an argument let say b and this function when called in main, takes two parenthesis(add2()()), where the first parenthesis is of the function add2 and second parenthesis is of the second function. It will return the addition of two numbers, that is a+b. Therefore, we have curried the add function, which means we have transformed the function that takes two arguments into a function that takes one argument and the function itself returns the result.

Currying Function Using Partial Application

We have another way to use this Curried function and that is Partially Applied function. So, let’s take a simple example and understand. we have defined a variable sum in the main function
Example

```
object Scala_Currying_Partial_Application {
  def add2(a: Int) = (b: Int) => a + b;
  def main(args: Array[String]): Unit = {
    // Partially Applied function.
    val sum = add2(29);
    println(sum(5));
  }
}
```

Scala File handling

Scala provides predefined methods to deal with file. we can create, open, write and read file. Scala provides a complete package scala.io for file handling.

Scala Creating a File Example
Scala doesn't provide file writing methods. So, we will use the Java PrintWriter or FileWriter methods.

Creating a new file :

java.io.File defines classes and interfaces for the JVM access files, file systems and attributes.
File(String pathname) converts theparameter string to abstract path name, creating a new file instance.
Writing to the file

java.io.PrintWriter includes all the printing methods included in PrintStream.

```
// File handling program
import java.io.File
import java.io.PrintWriter
object Scala_File_Handling {
  // Main method
  def main(args:Array[String])
  {
    // Creating a file
    val file_Object = new File("C:\\Users\\Makhan\\IdeaProjects\\abc.txt1" )

    // Passing reference of file to the printwriter
    val print_Writer = new PrintWriter(file_Object)

    // Writing to the file
    print_Writer.write("Hello, This is  Mtech Sudents")

    // Closing printwriter
    print_Writer.close()
  }

}
```

A text file abc.txt is created and contains the string “Hello, This is Mtech Students”

Scala does not provide class to write a file but it provide a class to read the files. This is the class Source. We use its companion object to read files. To read the contents of this file, we call the fromFile() method of class Source for reading the contents of the file which includes filename as argument.

A text file abc.txt is created and contains the string “Hello, This is Mtech Students”

Scala does not provide class to write a file but it provide a class to read the files. This is the class Source. We use its companion object to read files. To read the contents of this file, we call the fromFile() method of class Source for reading the contents of the file which includes filename as argument.

Creating a file in Scala and writing content to it

```
import java.io._
object Scala_File_Writing_and_Reading {
  def main(args: Array[String]): Unit = {
    val file = new
File("C:\\Users\\Makhan\\IdeaProjects\\Mtech.txt" )

    //creating object of the PrintWrite
    //by passing the reference to the file
    val pw = new PrintWriter(file)

    //writing text to the file
    pw.write("Welcome to School of Data science")
    pw.write("writing text to the file\n")

    //closing the PrintWriter
    pw.close()
    println("PrintWriter saved and closed...")

  }
}
```

Reading Content line by line in Scala

```
import scala.io.Source
object Scala_File_Reading_line_by_line {
  def main(args: Array[String]): Unit = {
    val filename = "C:\\Users\\Makhan\\IdeaProjects\\Mtech.txt"

    //file reading - creating object name by passing
    //filename i.e. file object
    val filereader = Source.fromFile(filename)
    //printing characters
    for(line <-filereader.getLines())
        {
      println(line)
    }
    //closing
    filereader.close()
  }
}
```

Scala Reading File Example: Reading Each Charactar

```
import scala.io.Source
object
Scala_File_handling_Reading_Each_Char {
  def main(args: Array[String]): Unit = {
    val filename =
"C:\\Users\\Makhan\\IdeaProjects\\Mtech2.tx
t"
    val fileSource = Source.fromFile(filename)
    while(fileSource.hasNext){
      println(fileSource.next)
    }
    fileSource.close()
  }
}
```

Scala Multithreading

Multithreading is a process of executing multiple threads simultaneously. It allows we to perform multiple operations independently.

we can achieved multitasking by using Multithreading. Threads are lightweight sub-processes which occupy less memory. Multithreading are used to develop concurrent applications in Scala.

Scala does not provide any separate library for creating thread. If we are familiar with multithreading concept of Java, we will come to know that it is similar except the syntax of Scala language itself.

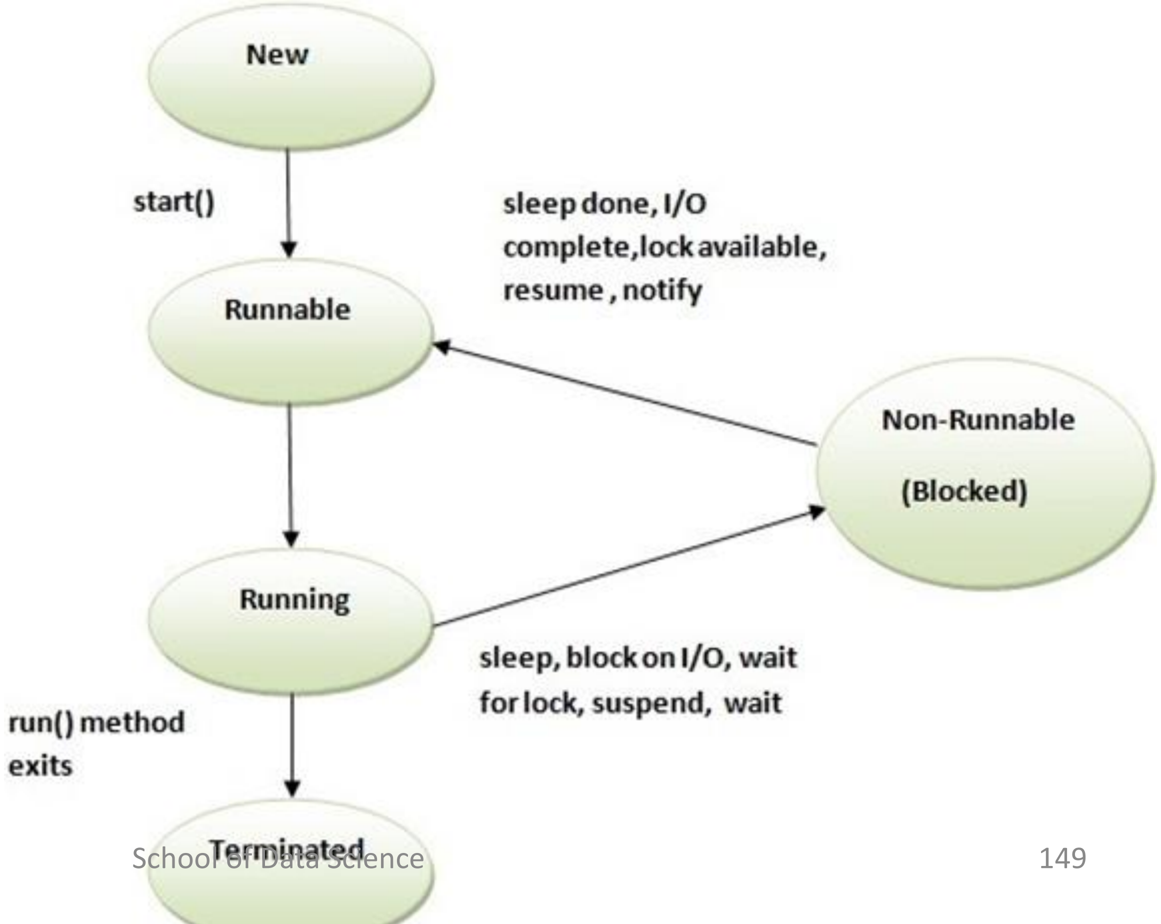
we can create thread either by extending Thread class or Runnable interface. Both provide a run method to provide specific implementation.

Scala Thread Life Cycle

Thread life cycle is a span of time in which thread starts and terminates. It has various phases like new, runnable, terminate, block etc. Thread class provides various methods to monitor thread's states.

The Scala thread states are as follows:

New
Runnable
Running
Non-Runnable (Blocked)
Terminated


New
start()
Runnable
sleep done, I/O complete, lock available, resume, notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

1) New
This is the first state of thread. It is just before starting of new thread.

2) Runnable
This is the state when thread has been started but the thread scheduler has not selected it to be the running thread.

3) Running
The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)
This is the state when the thread is still alive, but is currently not eligible to run due to waiting for input or resources.

5) Terminated
A thread is in terminated or dead state when its run() method exits.

Scala Thread
There are two ways to create a thread:

By extending Thread class
By implementing Runnable interface

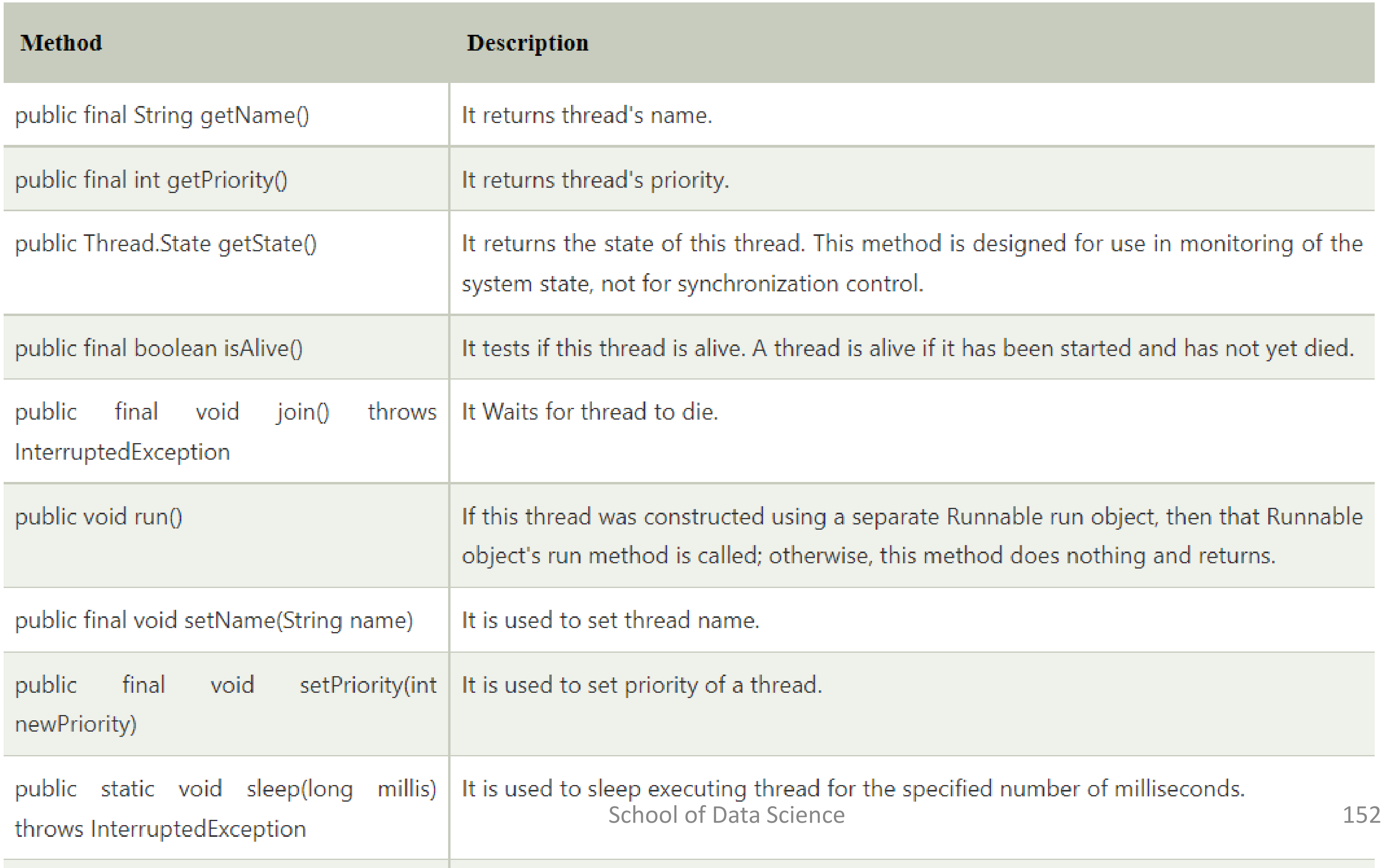
By extending Thread class

```
class Scala_Thread_Using_Extending_Thread
extends Thread {
  override def run() {
    println("Thread is running...");
  }
}
object MainObject_thread{
  def main(args:Array[String]){
    var t = new
Scala_Thread_Using_Extending_Thread()
    t.start()
  }
}
```

Scala Thread Methods

Thread class provides various methods to deals with thread's states.wecan use these methods to control the flow of thread.

The following table contains commonly used methods of Thread class.



Method	Description
public final String getName()	It returns thread's name.
public final int getPriority()	It returns thread's priority.
public Thread.State getState()	It returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control.
public final boolean isAlive()	It tests if this thread is alive. A thread is alive if it has been started and has not yet died.
public final void join() throws InterruptedException	It Waits for thread to die.
public void run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
public final void setName(String name)	It is used to set thread name.
public final void setPriority(int newPriority)	It is used to set priority of a thread.
public static void sleep(long millis) throws InterruptedException	It is used to sleep executing thread for the specified number of milliseconds.

Scala Thread sleep() Method

The sleep() method is used to sleep thread for the specified time. It takes time in milliseconds as an argument.

```
class Scala_Thread_Sleep extends Thread {
  override def run(){
    for(i<- 0 to 5){
      println(i)
      Thread.sleep(500)
    }
  }

}
object Scala_Thread_Sleep_obj{

  def main(args: Array[String]): Unit = {
    var t1 = new Scala_Thread_Sleep()
    var t2 = new Scala_Thread_Sleep()
    t1.start()
    t2.start()
  }
}
```

Scala Thread join() Method Example

The join() method waits for a thread to die. In other words, The join() method is used to hold the execution of currently running thread until the specified thread finished it's execution.

```
class Scala_Thread_join  extends Thread {
  override def run(){
    for(i<- 0 to 5){
      println(i)
      Thread.sleep(500)
    }
  }
}
object Scala_Thread_join_obj{

  def main(args: Array[String]): Unit = {
    var t1 = new Scala_Thread_join()
    var t2 = new Scala_Thread_join()
    var t3 = new Scala_Thread_join()
    t1.start()
    t1.join()
    t2.start()
    t3.start()
  }
}
```

Scala setName() Method Example

In the following example, we are setting and getting names of threads.

```
class Scala_Thread_setName_Method  extends Thread{
  override def run(){
    for(i<- 0 to 5){
      println(this.getName()+" - "+i)
      Thread.sleep(500)
    }
  }
}
object  Scala_Thread_setName_Method_obj{

  def main(args: Array[String]): Unit = {
    var t1 = new Scala_Thread_setName_Method()
    var t2 = new Scala_Thread_setName_Method()
    var t3 = new Scala_Thread_setName_Method()
    t1.setName("First Thread")
    t2.setName("Second Thread")
    t1.start()
    t2.start()
  }
}
```

Scala Thread Priority Example

We can set thread priority by using it's predefined method. The following example sets priority for the thread.

```
class Scala_Thread_Priority extends Thread {
  override def run() {
    for (i <- 0 to 5) {
      println(this.getName())
      println(this.getPriority())
      Thread.sleep(500)
    }
  }
}
object  Scala_Thread_Priority_obj
{
  def main(args: Array[String]): Unit = {
    var t1 = new Scala_Thread_Priority()
    var t2 = new Scala_Thread_Priority()
    t1.setName("First Thread")
    t2.setName("Second Thread")
    t1.setPriority(Thread.MIN_PRIORITY)
    t2.setPriority(Thread.MAX_PRIORITY)
    t1.start()
    t2.start()
  }
}
```

```
class Scala_Thread_Multitasking extends
Thread {
  override def run() {
    for (i <- 0 to 5) {
      println(i)
      Thread.sleep(500)
    }
  }

  def task() {
    for (i <- 0 to 5) {
      println(i)
      Thread.sleep(200)
    }
  }
}
object  Scala_Thread_Multitasking_obj{

  def main(args: Array[String]): Unit = {
    var t1 = new Scala_Thread_Multitasking()
    t1.start()
    t1.task()
```

What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What does Git do?

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Working with Git

- Initialize Git on a folder, making it a **Repository**
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered **modified**
- You select the modified files you want to **Stage**
- The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

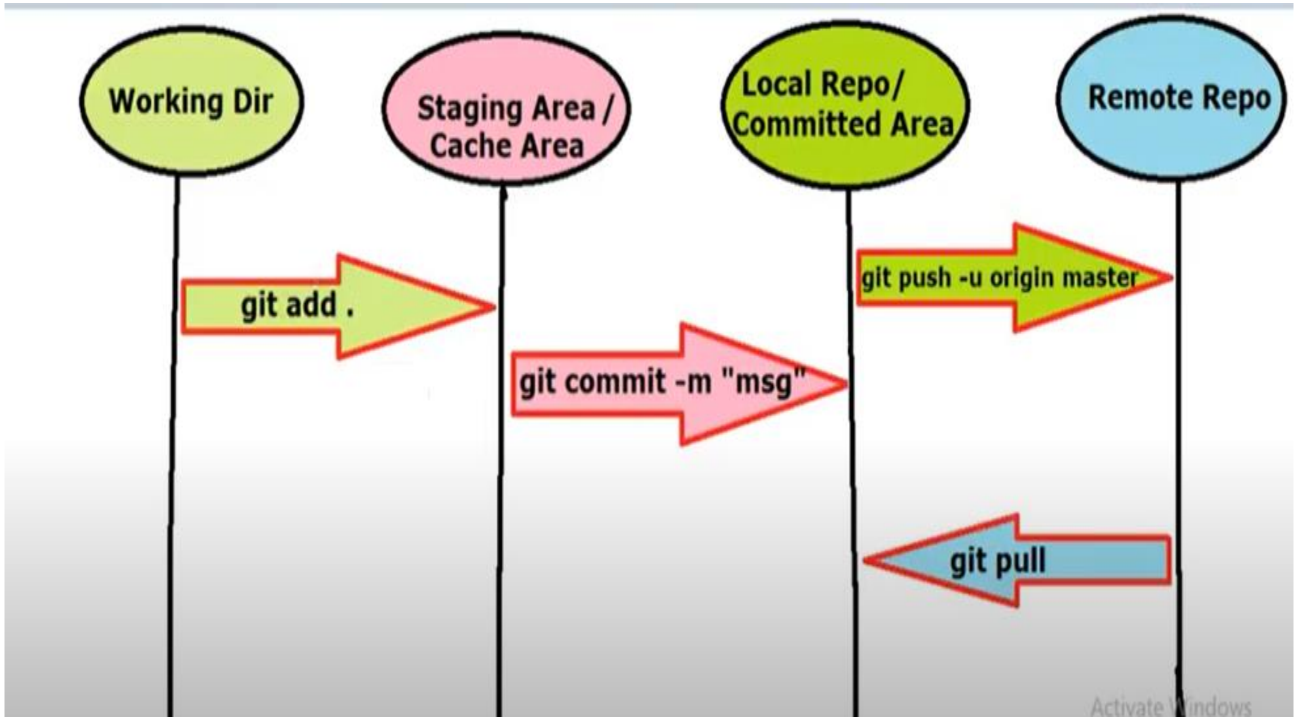
What is GitHub?

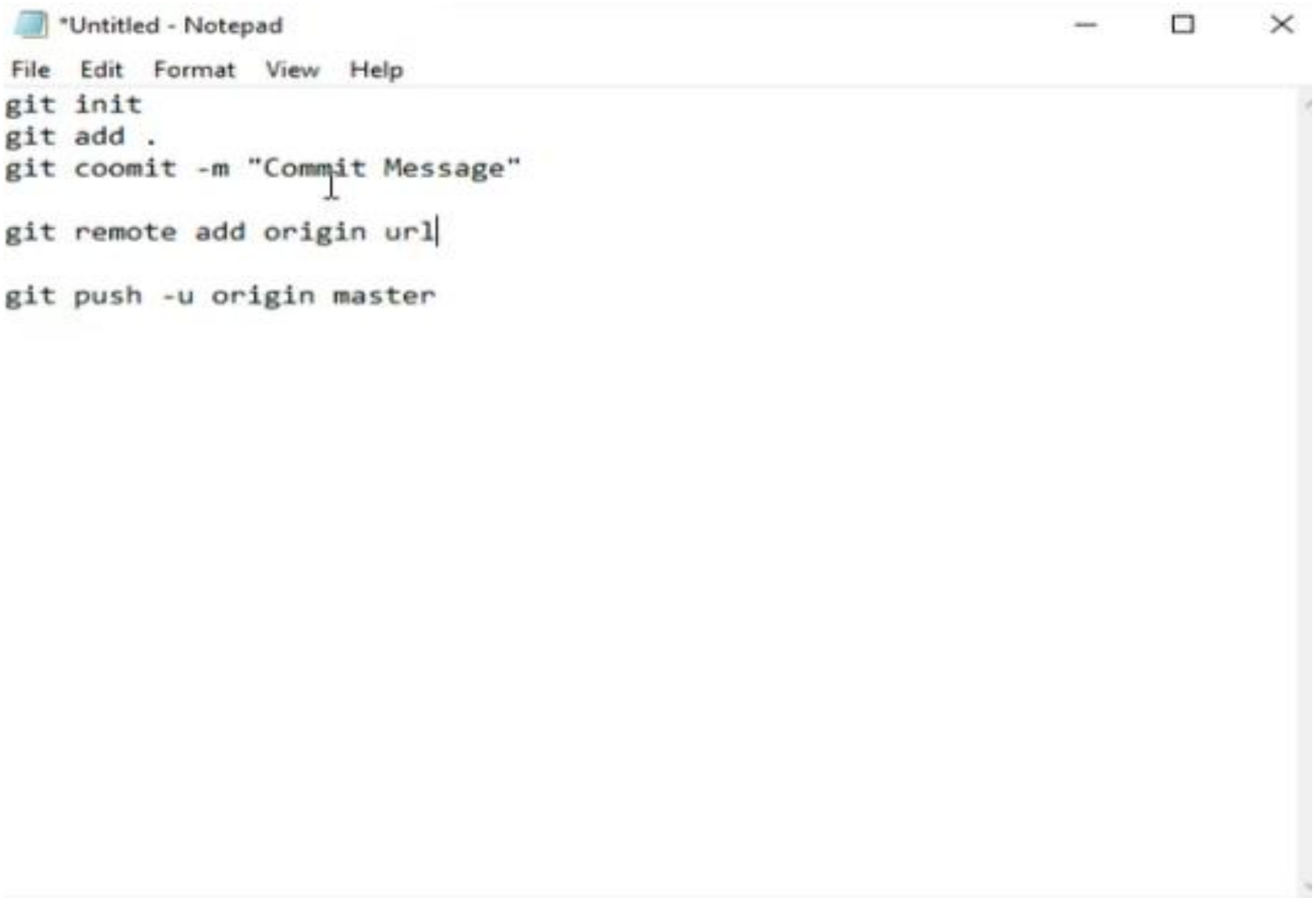
•Git is not the same as GitHub.
•GitHub makes tools that use Git.
•GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018.

Using Git with Command Line
To start using Git, we are first going to open up our Command shell.

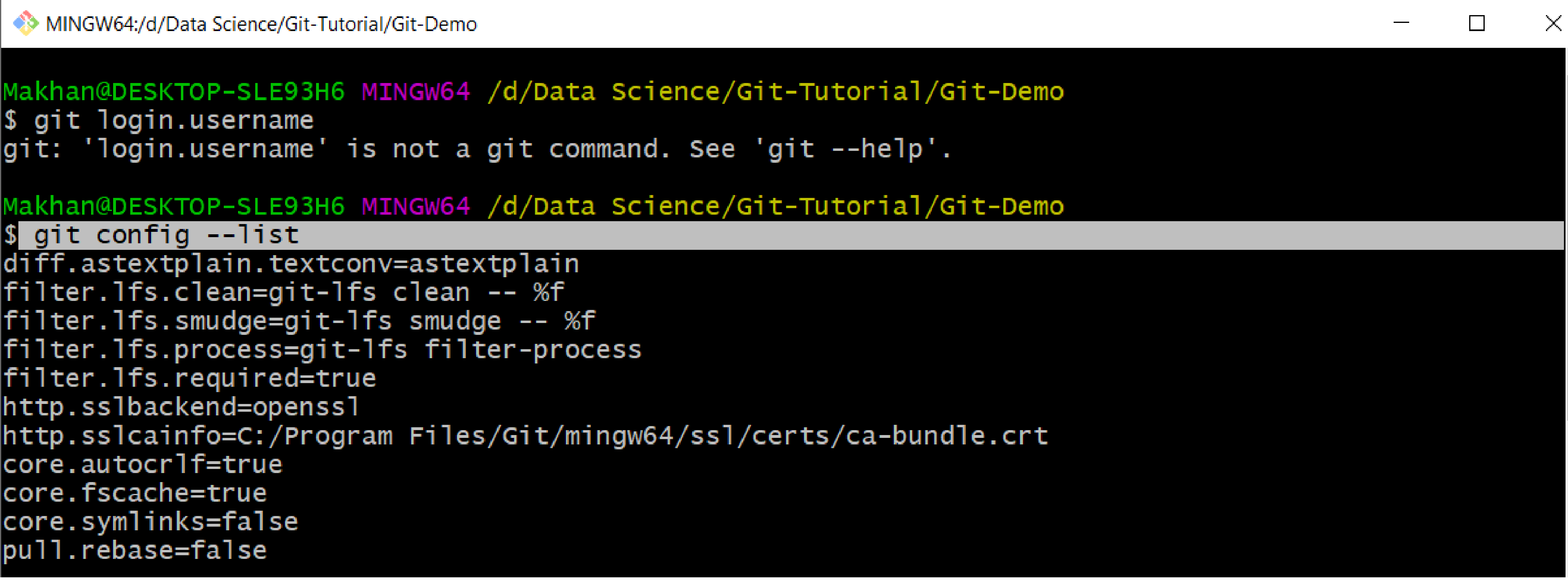
For Windows, you can use Git bash, which comes included in Git for Windows. For Mac and Linux you can use the built-in terminal.

The first thing we need to do, is to check if Git is properly installed:


Working Dir
Staging Area / Cache Area
Local Repo/ Committed Area
Remote Repo
git add .
git commit -m "msg"
git push -u origin master
git pull

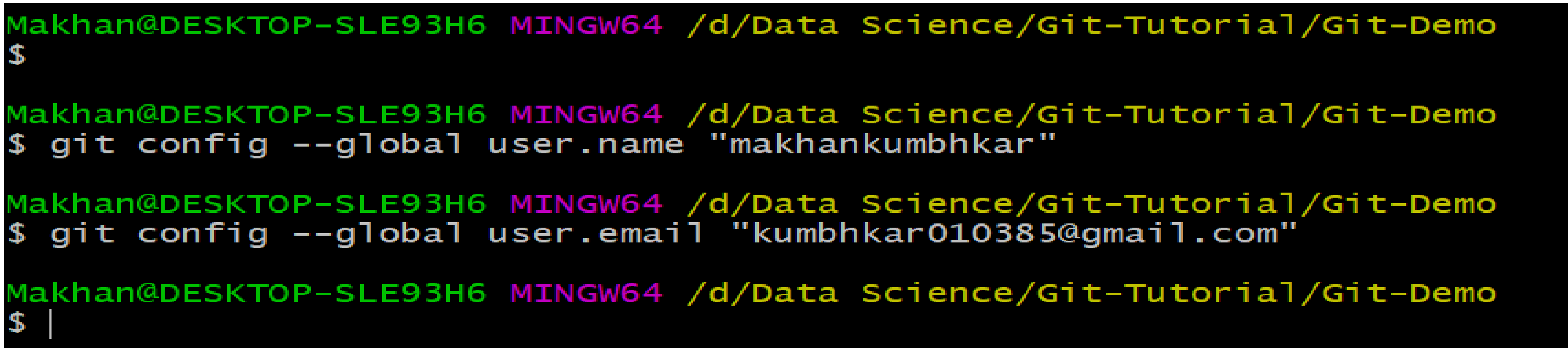


```
git init
git add .
git coomit -m "Commit Message"

git remote add origin url

git push -u origin master
```

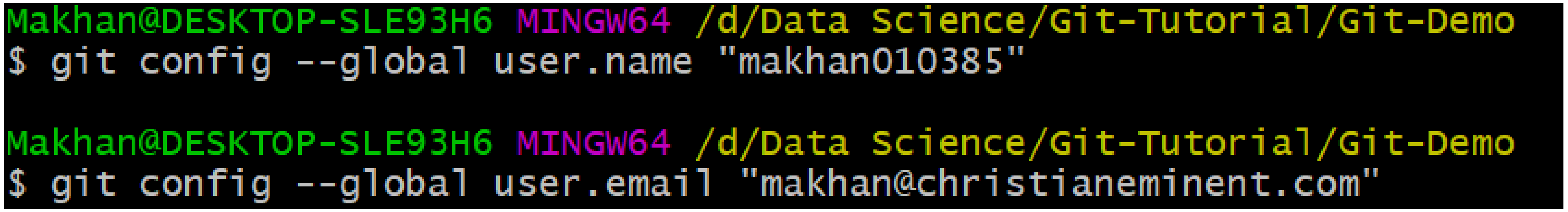


MINGW64:/d/Data Science/Git-Tutorial/Git-Demo

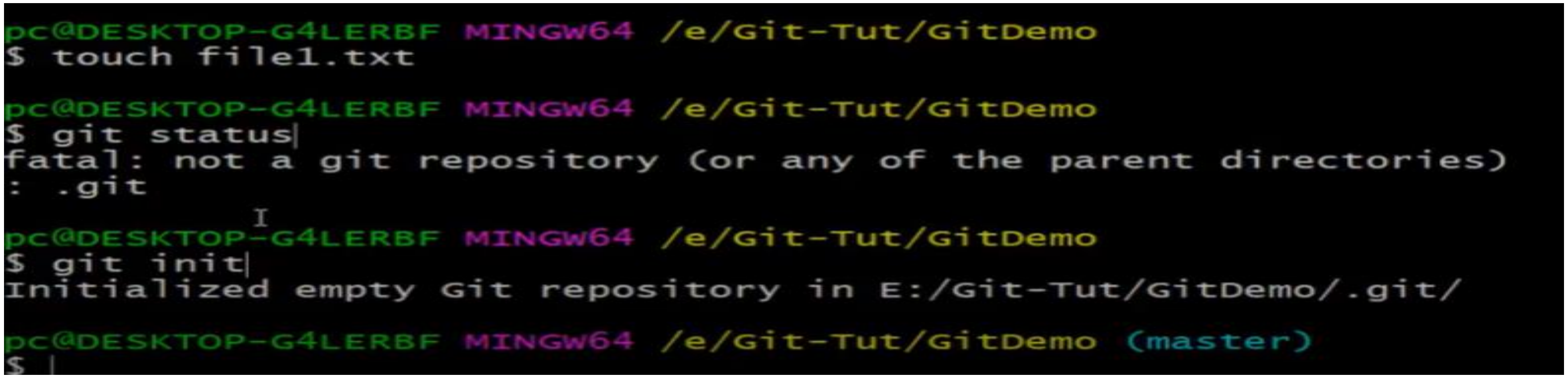
```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git login.username
git: 'login.username' is not a git command. See 'git --help'.

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
```



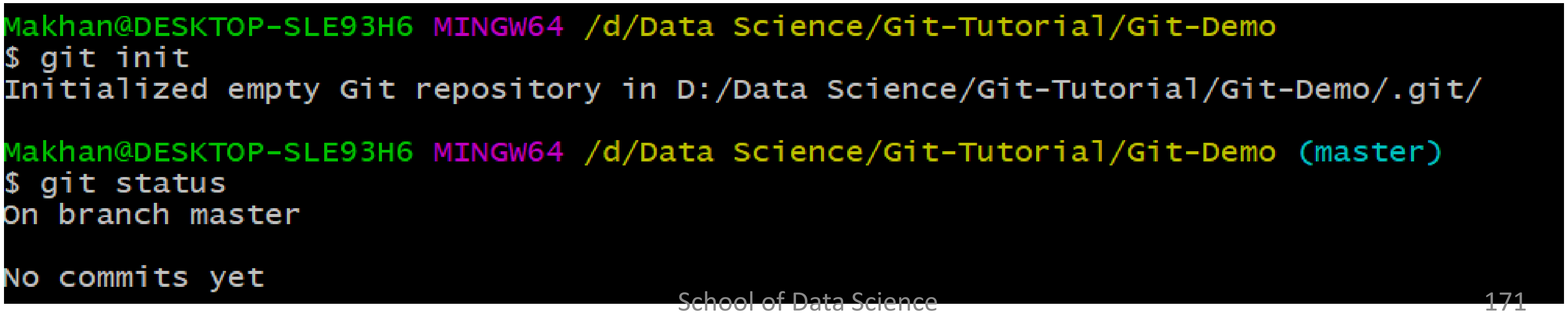
```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git config --global user.name "makhankumbhkar"

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git config --global user.email "kumbhkar010385@gmail.com"

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ |
```



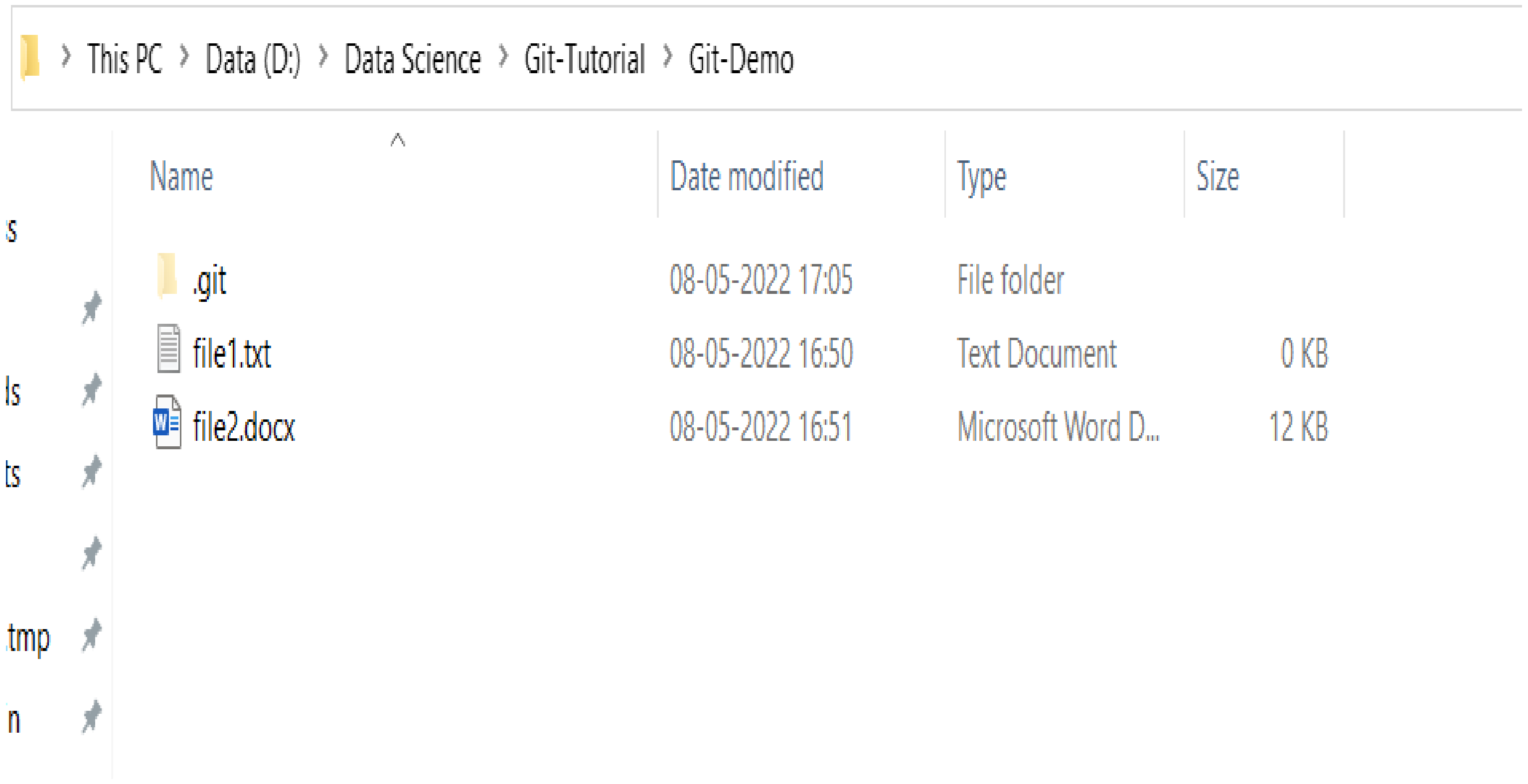
```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git config --global user.name "makhan010385"

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git config --global user.email "makhan@christianeminent.com"
```



```
pc@DESKTOP-G4LERBF MINGW64 /e/Git-Tut/GitDemo
$ touch file1.txt

pc@DESKTOP-G4LERBF MINGW64 /e/Git-Tut/GitDemo
$ git status
fatal: not a git repository (or any of the parent directories)
: .git

pc@DESKTOP-G4LERBF MINGW64 /e/Git-Tut/GitDemo
$ git init
Initialized empty Git repository in E:/Git-Tut/GitDemo/.git/

pc@DESKTOP-G4LERBF MINGW64 /e/Git-Tut/GitDemo (master)
$
```

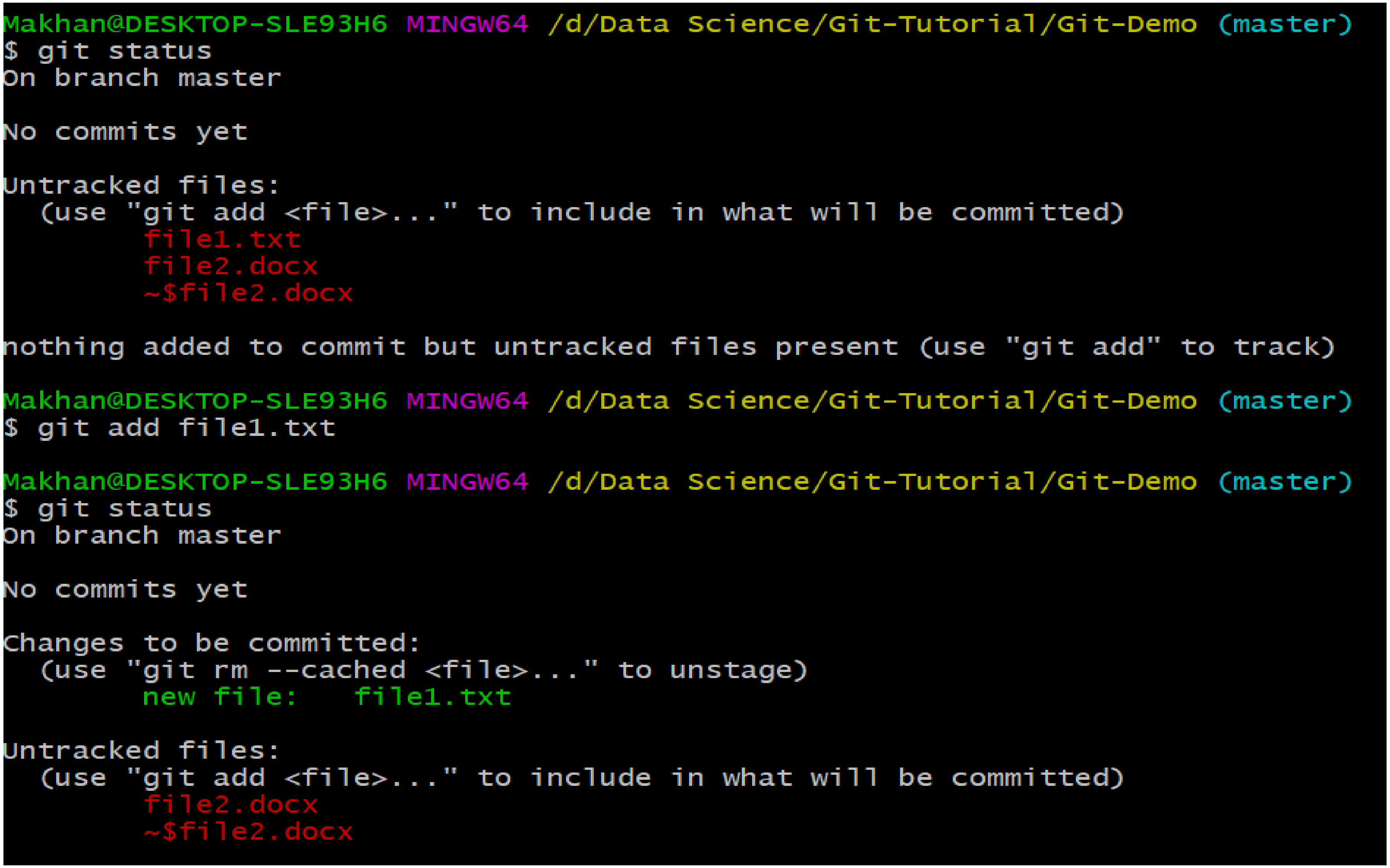


```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo
$ git init
Initialized empty Git repository in D:/Data Science/Git-Tutorial/Git-Demo/.git/

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master

No commits yet
```



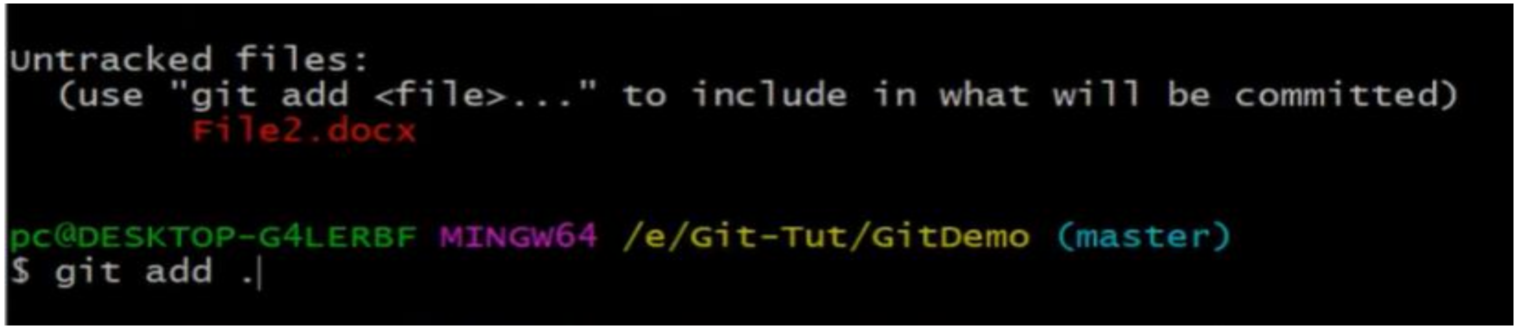
This PC › Data (D:) › Data Science › Git-Tutorial › Git-Demo

Name	Date modified	Type	Size
.git	08-05-2022 17:05	File folder	
file1.txt	08-05-2022 16:50	Text Document	0 KB
file2.docx	08-05-2022 16:51	Microsoft Word D...	12 KB

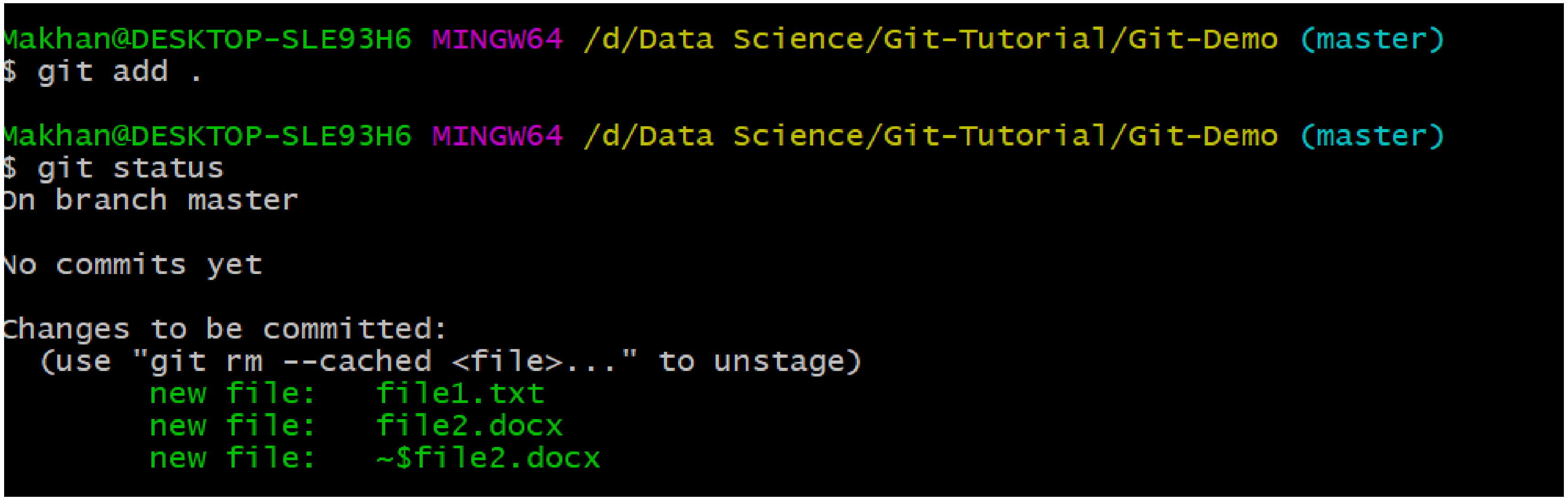


```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt
        file2.docx
        ~$file2.docx

nothing added to commit but untracked files present (use "git add" to track)

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git add file1.txt

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file2.docx
        ~$file2.docx
```

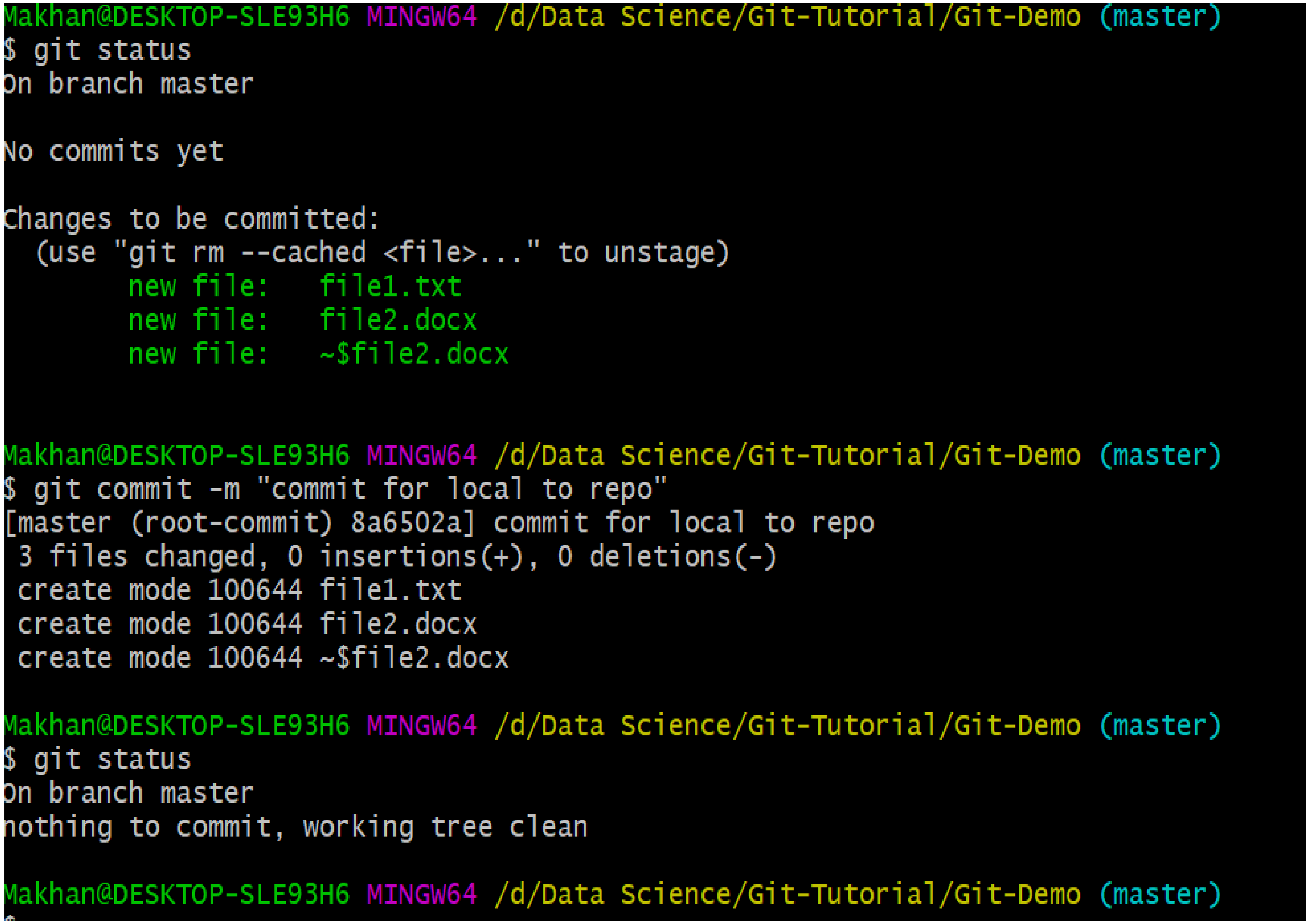
If you want to track all file with one command



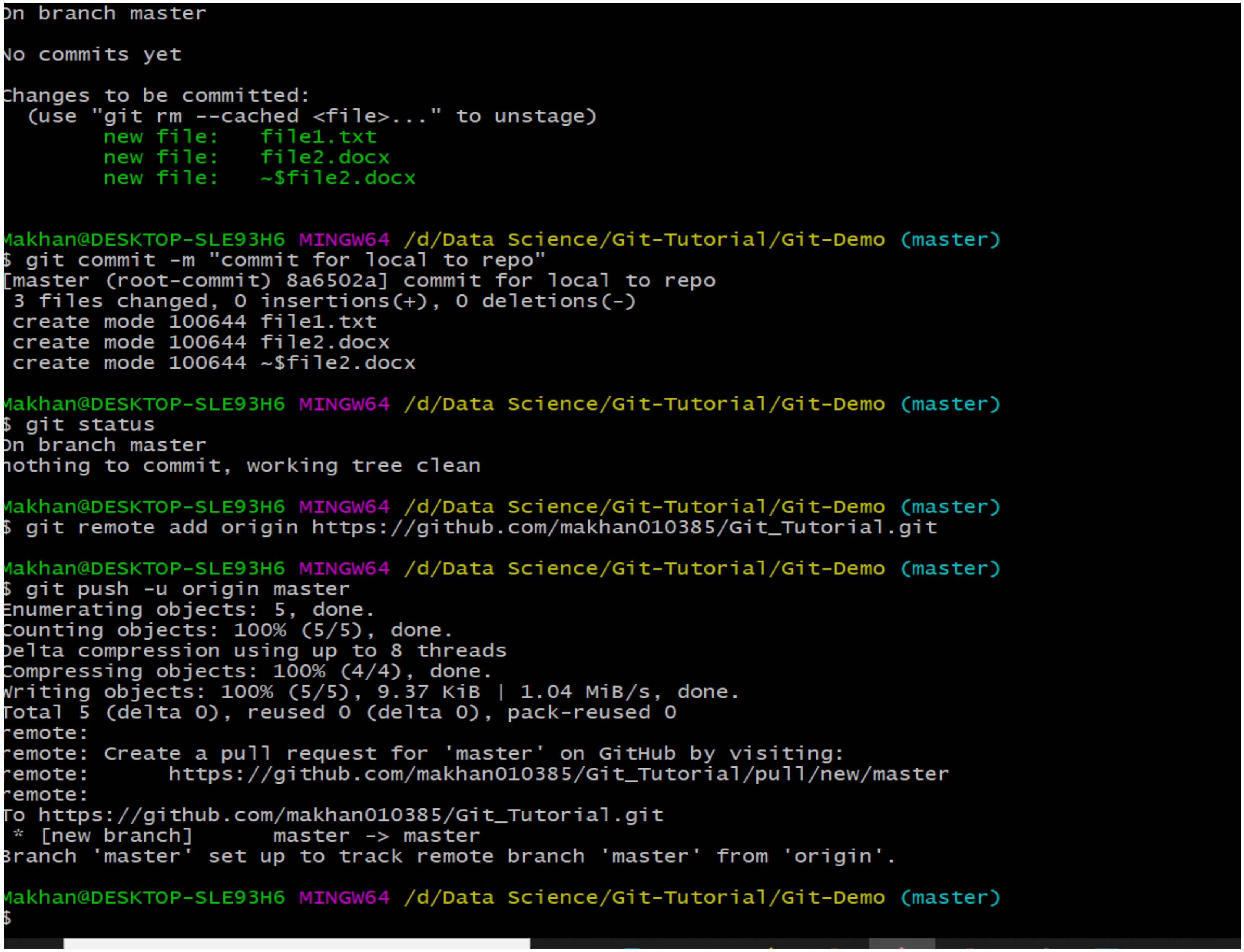
```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        File2.docx


pc@DESKTOP-G4LERBF MINGW64 /e/Git-Tut/GitDemo (master)
$ git add .
```



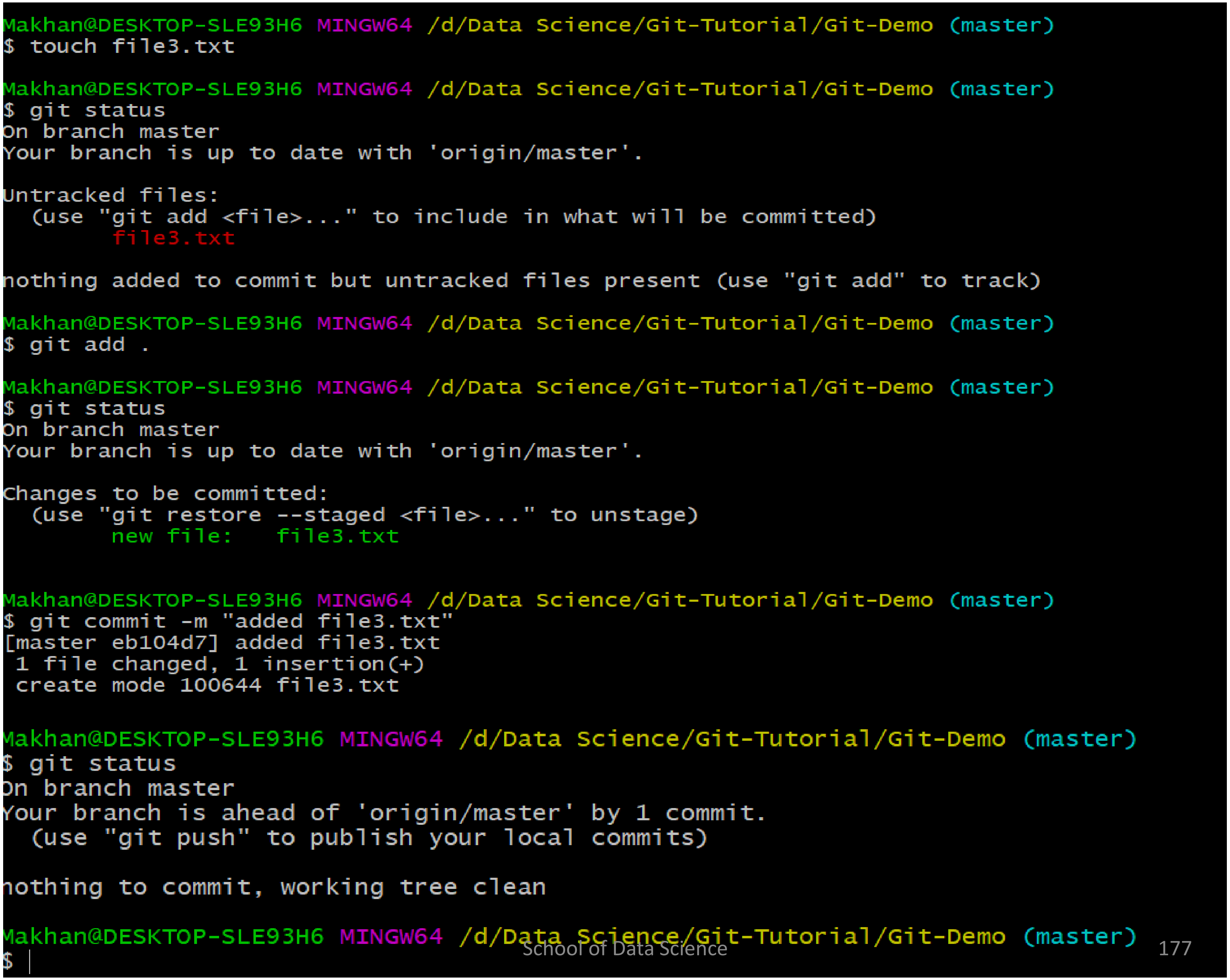
```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git add .

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.docx
        new file:   ~$file2.docx
```



```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.docx
        new file:   ~$file2.docx


Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git commit -m "commit for local to repo"
[master (root-commit) 8a6502a] commit for local to repo
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file2.docx
 create mode 100644 ~$file2.docx

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master
nothing to commit, working tree clean

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
```

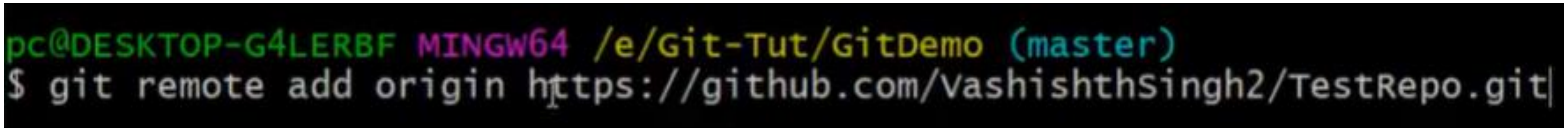


```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.docx
        new file:   ~$file2.docx


Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git commit -m "commit for local to repo"
[master (root-commit) 8a6502a] commit for local to repo
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file2.docx
 create mode 100644 ~$file2.docx

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master
nothing to commit, working tree clean

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git remote add origin https://github.com/makhan010385/Git_Tutorial.git

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git push -u origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 9.37 KiB | 1.04 MiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/makhan010385/Git_Tutorial/pull/new/master
remote:
To https://github.com/makhan010385/Git_Tutorial.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$
```



```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ touch file3.txt

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file3.txt

nothing added to commit but untracked files present (use "git add" to track)

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git add .

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file3.txt


Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git commit -m "added file3.txt"
[master eb104d7] added file3.txt
 1 file changed, 1 insertion(+)
 create mode 100644 file3.txt

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git-Tutorial/Git-Demo (master)
$ 
```

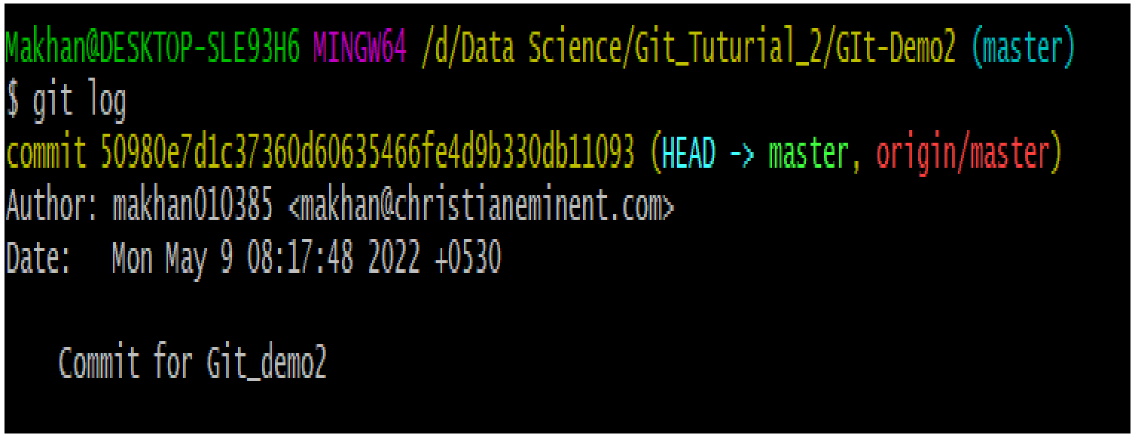
No need to fire remote command like below


```
pc@DESKTOP-G4LERBF MINGW64 /e/Git-Tut/GitDemo (master)
$ git remote add origin https://github.com/VashishthSingh2/TestRepo.git
```

Directly fire push command like

```
git push -u origin master
```

Git log show all Previous commits



```
Makhan@DESKTOP-SLE93H6 MINGW64 /d/Data Science/Git_Tuturial_2/GIt-Demo2 (master)
$ git log
commit 50980e7d1c37360d60635466fe4d9b330db11093 (HEAD -> master, origin/master)
Author: makhan010385 <makhan@christianeminent.com>
Date:   Mon May 9 08:17:48 2022 +0530

    Commit for Git_demo2
```

How to delete Branch

```
$ git branch -d bug1
Deleted branch bug1 (was 37c4b3b).
```

Higher order functions

Higher order functions take other functions as parameters or return a function as a result. This is possible because functions are first-class values in Scala. The terminology can get a bit confusing at this point, and we use the phrase “higher order function” for both methods and functions that take functions as parameters or that return a function.

In a pure Object Oriented world a good practice is to avoid exposing methods parameterized with functions that might leak object’s internal state. Leaking internal state might break the invariants of the object itself thus violating encapsulation.

One of the most common examples is the higher-order function map which is available for collections in Scala.

```
val salaries = Seq(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

doubleSalary is a function which takes a single Int, x, and returns x * 2. In general, the tuple on the left of the arrow => is a parameter list and the value of the expression on the right is what gets returned. On line 3, the function doubleSalary gets applied to each element in the list of salaries.

To shrink the code, we could make the function anonymous and pass it directly as an argument to map

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(x => x * 2) // List(40000, 140000, 80000)
```

Notice how x is not declared as an Int in the above example. That's because the compiler can infer the type based on the type of function map expects (see Currying). An even more idiomatic way to write the same piece of code would be:

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(_ * 2)
```

Since the Scala compiler already knows the type of the parameters (a single Int), you just need to provide the right side of the function. The only caveat is that you need to use _ in place of a parameter name (it was x in the previous example).

Coercing methods into functions

It is also possible to pass methods as arguments to higher-order functions because the Scala compiler will coerce the method into a function.

```
case class WeeklyWeatherForecast(temperatures: Seq[Double])
 {

  private def convertCtoF(temp: Double) = temp * 1.8 + 32

  def forecastInFahrenheit: Seq[Double] =
temperatures.map(convertCtoF) // <-- passing the method
convertCtoF
}
```

Here the method convertCtoF is passed to the higher order function map. This is possible because the compiler coerces convertCtoF to the function x => convertCtoF(x) (note: x will be a generated name which is guaranteed to be unique within its scope).

Functions that accept functions

One reason to use higher-order functions is to reduce redundant code. Let’s say you wanted some methods that could raise someone’s salaries by various factors. Without creating a higher-order function, it might look something like this:

```
object SalaryRaiser {

  def smallPromotion(salaries: List[Double]): List[Double] =
    salaries.map(salary => salary * 1.1)

  def greatPromotion(salaries: List[Double]): List[Double] =
    salaries.map(salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    salaries.map(salary => salary * salary)
}
```

Notice how each of the three methods vary only by the multiplication factor. To simplify, you can extract the repeated code into a higher-order function like so:

```
object SalaryRaiser {

  private def promotion(salaries: List[Double], promotionFunction: Double => Double):
List[Double] =
    salaries.map(promotionFunction)

  def smallPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * 1.1)

  def greatPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * salary)
}
```

The new method, promotion, takes the salaries plus a function of type Double => Double (i.e. a function that takes a Double and returns a Double) and returns the product. Methods and functions usually express behaviours or data transformations, therefore having functions that compose based on other functions can help building generic mechanisms. Those generic operations defer to lock down the entire operation behaviour giving clients a way to control or further customize parts of the operation itself.

Functions that return functions

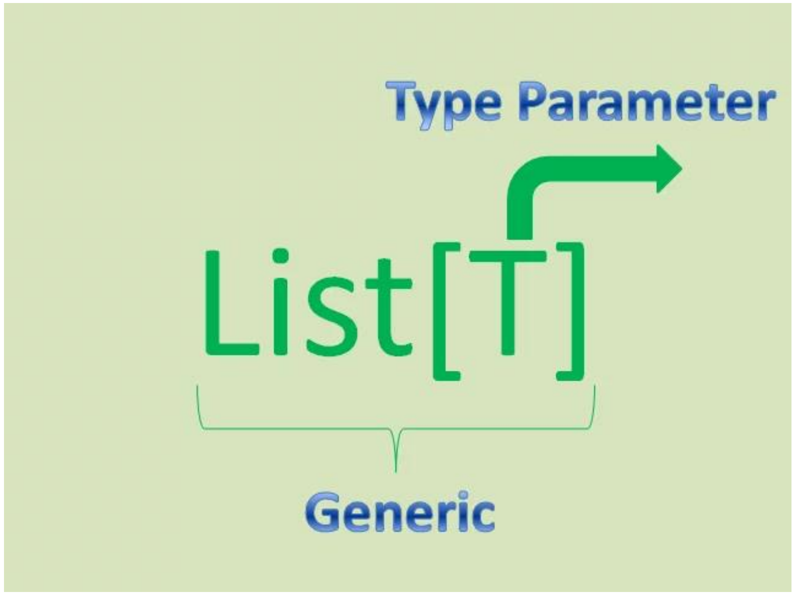
There are certain cases where we want to generate a function. Here’s an example of a method that returns a function.

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) =>
s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) // "https://www.example.com/users?id=1":
String
```

Notice the return type of urlBuilder (String, String) => String. This means that the returned anonymous function takes two Strings and returns a String.
In this case, the returned anonymous function is
(endpoint: String, query: String) => s"https://www.example.com/$endpoint?$query".

What is Variance?
Variance defines Inheritance relationships of Parameterized Types. Variance is all about Sub-Typing.

Please go through the following image to understand "What is Parameterized Type".



Here T is known as "Type Parameter" and List[T] is known as Generic.

For List[T], if we use List[Int], List[AnyVal], etc. then these List[Int] and List[AnyVal] are known as "Parameterized Types"

Variance defines Inheritance relationship between these Parameterized Types.

Advantage of Variance in Scala
The main advantage of Scala Variance is:

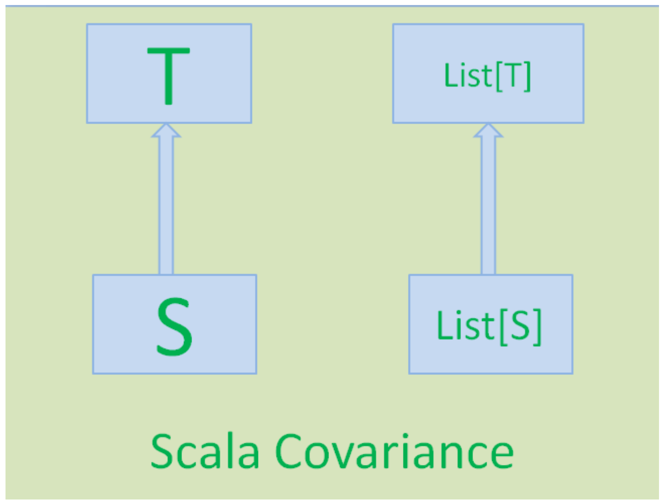
Variance makes Scala collections more Type-Safe.
Variance gives more flexible development.
Scala Variance gives us a technique to develop Reliable Applications.

Types of Variance in Scala
Scala supports the following three kinds of Variance.

Covariant
Invariant
Contravariant

We will discuss these three variances in detail in coming sections.

Covariant in Scala
If “S” is subtype of “T” then List[S] is is a subtype of List[T].



Scala Covariance Syntax:-
To represent Covariance relationship between two Parameterized Types, Scala uses the following syntax:
Prefixing Type Parameter with “+” symbol defines Covariance in Scala.

Example:-
Write a Scala program to demo Scala Covariant SubTyping technique.

```
class Animal[+T](val animial:T)

class Dog
class Puppy extends Dog

class AnimalCarer(val dog:Animal[Dog])

object ScalaCovarianceTest{
  def main(args: Array[String])
{
    val puppy = new Puppy
    val dog = new Dog

    val puppyAnimal:Animal[Puppy] = new Animal[Puppy](puppy)
    val dogAnimal:Animal[Dog] = new Animal[Dog](dog)

    val dogCarer = new AnimalCarer(dogAnimal)
    val puppyCarer = new AnimalCarer(puppyAnimal)

    println("Done.")
  }
}
```

Scala Varargs

Most of the programming languages provide us variable length argument mobility to a function, Scala is not a exception. it allows us to indicate that the last argument of the function is a variable length argument. it may be repeated multiple times. It allows us to indicate that last argument of a function is a variable length argument, so it may be repeated multiple times. we can pass as many argument as we want. This allows programmers to pass variable length argument lists to the function. Inside function, the type of args inside the declared are actually saved as a Array[Datatype] for example can be declared as type String* is actually Array[String].
Note :- We place * on the last argument to make it variable length.
Syntax : –

```
def Nameoffunction(args: Int *) : Int = { s foreach println. }
```

Below are some restrictions of varargs :

The last parameter in the list must be the repeating argument.

```
def sum(a :Int, b :Int, args: Int *)
```

No default values for any parameters in the method containing the varargs.
All values must be same data type otherwise error.

```
> sum(5, 3, 1000, 2000, 3000, "one")
> error: type mismatch;
found : String("one")
required: Int
```

Inside the body args is an array, so all values are packed into an array