Scala Points:-

1.Paradigm:-functional and object oriented,Imperative

2.Designed By:-Martin Odersky

3.Platform:-JVM,JavaScript

4.Filename extensions:- .scala and .sc

5.Stable Release:- 2.12.1 date - 5 December 2016

Scala Overview:-

1.Scala runs on the Java Virtual Machine:-Scala is compiled into Java Byte Code which is executed by the Java Virtual Machine (JVM).

This means that Scala and Java have a common runtime platform.

2.Scala can Execute Java Code:-you can use all Java classes in your Scala code.

3.Scala Has a Compiler, Interpreter and Runtime:-Scala has both a compiler and an interpreter which can execute Scala code.

The Scala compiler compiles your Scala code into Java Byte Code which can then be executed by the scala command.

The scala command is similar to the java command, in that it executes your compiled Scala code.

Scala Features:-

Scala has a set of features which differ from Java. Some of these are:

1.All types are objects.

2.Type inference:-Scala's type inference can figure out the type of a variable, based on the value assigned to it.

Therefore, you could actually omit the type in the field declaration like this

var myField : Int = 0; (Not define type inference)

var myField = 0;(Since 0 is by default assumed to be an Int, the Scala compiler can infer the type of the myField based on the 0 assigned to it)

3.Functions are objects.

4.Domain specific language (DSL) support.

5.Traits.

6.Closures.

7.Concurrency support inspired by Erlang.

Scala Classes:-A Scala class is a template for Scala objects.A class can contain information about Fields(variables that is local to class)

,method,constructors,Superclasses etc.

The Basic Class Definition

Here is a simple class definition in Scala:

```
class MyClass {


}
```

Fields:-A field is a variable that is accessible inside the whole object.

```
class MyClass {
  var myField : Int = 0;
}
```

Constructors:-n scala constructors are declared like this.

```
lass MyClass {
    var myField : Int = 0;



        def this(value : Int) = {
        this();
        this.myField = value;

        }


  }
```

This example defines a constructor which takes a single parameter, and assigns its value to the field myField.

Methods:-In Scala methods in a class are defined like this.

```
  class MyClass {

  var myField = 0;


  def getMyField() : Int = {

    return this.myField;

  }


}
```

The above example defines a method called getMyField. The return type, Int, is declared after the method

name. Inside the { and } the method body is declared.

The method currently just returns the myField field. Notice the = sign between the Int and {. Methods that

return a value should have this equals sign there.

Scala Singleton and Companion Objects:-Scala classes cannot have static variables or methods.

Instead a Scala class can have what is called a singleton object, or sometime a companion object.

A singleton object is declared using the object keyword. Here is an example.

In scala, there is no static concept. So scala creates a singleton object to provide entry point for your program execution.

```
object Main {
    def sayHi() {
        println("Hi!");
    }
}
```

This example defines a singleton object called Main. You can call the method sayHi() like this:

```
Main.sayHi();
```

Scala Singleton Object Example:-

```
object Singleton{
    def main(args:Array[String]){
        SingletonObject.hello()        // No need to create object.
    }
}


object SingletonObject{
    def hello(){
        println("Hello, This is Singleton Object")
    }
}
```

Scala Companion Object:-When a singleton object is named the same as a class, it is called a companion object.

A companion object must be defined inside the same source file as the class.

```
class Main {
    def sayHelloWorld() {
        println("Hello World");
    }
}
```

```
object Main {
   def sayHi() {
      println("Hi!");
   }
}
```

In this class you can both instantiate Main and call sayHelloWorld() or call the sayHi() method on the companion object directly, like this:

```
var aMain : Main = new Main();
aMain.sayHelloWorld();


Main.sayHi();
```

Scala Variables:-

var myVar : Int = 0; // mutable var This means that it is a variable that can change value.

val myVal : Int = 1; // immutable var This means that it is a value, a constant, that cannot change value once assigned.

Type Inference:-When you assign an initial value to a variable, the Scala compiler can figure out the type of

the varible based on the value assigned to it.

This is called type inference. Therefore, you could write these variable declarations like this:

Before type Inference :-

```
var myVar : Int = 0;
val myVal : Int = 1;
```

After type Inference:-

```
var myVar = 0;
val myVal = 1;
```

Scala Data Types:-

Type	Value Space
Boolean	true or false
Byte	8 bit signed value
Short	16 bit signed value
Char	16 bit unsigned Unicode character
Int	32 bit signed value
Long	64 bit signed value
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
String	A sequence of characters

Note:-All Data Types are Objects.ere are no primitive types like in Java.

Scala Arrays:-Scala arrays are immutable objects. You create an array like this-

var myArray : Array[String] = new Array[String](10); / Once created, you cannot change the length of an array

Types of array:-Single Dimentional,multidimentional

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.

Accessing Array Elements:-You access the elements of an array by using the elements index in the array. Element indexes

go from 0 to the length of the array minus 1.

So, if an array has 10 elements, you can access these 10 elements using index 0 to 9.

You can access an element in an array like this:

var aString : String = myArray(0);

To assign a value to an array element, you write this:

myArray(0) = "some value";

Iterating Array Elements:-You can iterate the elements of an array in two ways. You can iterate the element indexes, or iterate the elements themselves.

Iterate Indexes:-he first way is to use a for loop, and iterate through the index numbers from 0 until the length of the array. Here is how:

```
for(i <- 0 until myArray.length){

   println("i is: " + i);

   println("i'th element is: " + myArray(i));

}
```

The until keyword makes sure to only iterate until myArray.length - 1. Since array element indexes go from 0 to the array

length - 1,

this is the appropriate way to iterate

the array.

Iterate Elements:-The second way to iterate an array is to ignore the indexes, and just iterate the elements themselves. Here is how

```
for(myString <- myArray) {

   println(myString);

}
```

if - statements as Functions:-In Scala if-statements can be used as functions. That is, they can return a value. Here is an example:

```
var myInt : Int = 1;


var myText : String =

  if(myInt == 0) "myInt == 0";

  else         "myInt != 0";


println(myText);
```

Scala Array Example: Single Dimensional

```
class ArrayExample{
    var arr = Array(1,2,3,4,5)      // Creating single dimensional array
    def show(){
        for(a<-arr)                      // Traversing array elements
            println(a)
        println("Third Element  = "+ arr(2))       // Accessing elements by using index
    }
}

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

Output:

```
1
2
3
4
5
Third Element  = 3
```

Scala Example 2: Single Dimensional

In this example, we have created an array by using new keyword which is used to initialize memory for array. The entire array elements are set to default value, you can assign that later in your code.

```
class ArrayExample{
    var arr = new Array[Int](5)        // Creating single dimensional array
    def show(){
        for(a<-arr){                    // Traversing array elements
            println(a)
        }
        println("Third Element before assignment = "+ arr(2))       // Accessing elements by using index
        arr(2) = 10                                                  // Assigning new element at 2 index
        println("Third Element after assignment = "+ arr(2))
    }
}

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

Output:

```
0
0
0
0
0
Third Element before assignment = 0
Third Element after assignment = 10
```

Scala Passing Array into Function

You can pass array as an argument to function during function call. Following example illustrate the process how we can pass an array to the function.

```
class ArrayExample{
    def show(arr:Array[Int]){
        for(a<-arr)                // Traversing array elements
            println(a)
        println("Third Element = "+ arr(2))       // Accessing elements by using index
    }
}

object MainObject{
    def main(args:Array[String]){
        var arr = Array(1,2,3,4,5,6)    // creating single dimensional array
        var a = new ArrayExample()
        a.show(arr)                     // passing array as an argument in the function
    }
}
```

Output:

```
1
2
3
4
5
6
Third Element = 3
```

Scala Array Example: Iterating By using Foreach Loop

```
}
```

You can also iterate array elements by using foreach loop. Let's see an example.

```
class ArrayExample{
   var arr = Array(1,2,3,4,5)      // Creating single dimensional array
   arr.foreach((element:Int)=>println(element))       // Iterating by using foreach loop
}

object MainObject{
   def main(args:Array[String]){
      new ArrayExample()
   }
}
```

Output:

```
1
2
3
4
5
```

to vs. until:-You can use either the keyword to or until when creating a Range object. The difference is, that to includes

the last value in the range,

whereas until leaves it out. Here are two examples:

```
for(i <- 1 to 10) {
   println("i is " + i);
}
```

```
for(i <- 1 until 10) {
    println("i is " + i);
}
```

The first loop iterates 10 times, from 1 to 10 including 10.

The second loop iterates 9 times, from 1 to 9, excluding the upper boundary value 10

Iterating Collections and Arrays:-You can iterate a collection or array using the for loop, like this:

```
var myArray : Array[String] = new Array[String](10);

for(i <- 0 until myArray.length){
    myArray(i) = "value is: " + i;
}

for(value : String <- myArray ) {
    println(value);
}
```

Scala Multidimensional Array:-

Multidimensional Array Syntax

```
var arrayName = Array.ofDim[ArrayType](NoOfRows,NoOfColumns) or
var arrayName = Array(Array(element?), Array(element?), ?)
```

Scala Multidimensional Array Example by using ofDim

In This example, we have created array by using ofDim method.

```
class ArrayExample{
    var arr = Array.ofDim[Int](2,2)         // Creating multidimensional array
    arr(1)(0) = 15                          // Assigning value
    def show(){
        for(i<- 0 to 1){                       // Traversing elements by using loop
           for(j<- 0 to 1){
                print(" "+arr(i)(j))
            }
            println()
        }
        println("Third Element = "+ arr(1)(1))       // Accessing elements by using index
    }
}

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

Output:

```
0 0
15 0
Third Element = 0
```

Scala Functions:-you can create function by using def keyword. You must mention return type of parameters while

defining function and return type of a function

is optional. If you don't specify return type of a function, default return type is Unit.

Scala Function Declaration Syntax:-

```
def functionName(parameters : typeofparameters) : returntypeoffunction = {
// statements to be executed
}
```

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

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

anything and will work like subroutine.

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

Scala Function Example without using = Operator:-

```
object MainObject {
   def main(args: Array[String]) {
        functionExample()           // Calling function
    }
    def functionExample()  {        // Defining a function
          println("This is a simple function")
    }
}
```

Scala Function Example with = Operator:-

```
object MainObject {
   def main(args: Array[String]) {
        var result = functionExample()          // Calling function
        println(result)
    }
    def functionExample() = {       // Defining a function
        var a = 10
        a
    }
}    output=10
```

Scala Parameterized Function Example:-when using parameterized function you must mention type of parameters explicitly otherwise compiler

throws an error and your code fails to compile.

```
 object MainObject {
   def main(args: Array[String]) = {
        functionExample(10,20)
    }
    def functionExample(a:Int, b:Int) = {
         var c = a+b
         println(c)
    }
}
```

Scala Higher Order Functions:-Higher order function is a function that either takes a function as argument or returns a function. In other words we

can say a function which works with function is called higher order function.

Higher order function allows you to create function composition, lambda function or anonymous function etc.Let's see an example.

```
object MainObject {
   def main(args: Array[String]) = {
     functionExample(25, multiplyBy2)                  // Passing a function as parameter
    }
    def functionExample(a:Int, f:Int=>AnyVal):Unit = {
        println(f(a))                                // Calling that function
    }
    def multiplyBy2(a:Int):Int = {
        a*2
    }
}   op=60
```

Scala Example: Function Composition:-In scala, functions can be composed from other functions.

It is a process of composing in which a function represents

the application of two composed functions.

Let's see an example.

```
object MainObject {
   def main(args: Array[String]) = {
     var result = multiplyBy2(add2(10))      // Function composition
     println(result)
    }
    def add2(a:Int):Int = {
        a+2
    }

    def multiplyBy2(a:Int):Int = {
        a*2
    }
}  op=24
```

Scala Anonymous (lambda) Function:-Anonymous function is a function that has no name but works as a function. It is good to create an anonymous function
when you don't want to reuse it latter.
You can create anonymous function either by using => (rocket) or _ (underscore) wild card in scala.
Let's see an example.
Scala Anonymous function Example:-

```
object MainObject {
  def main(args: Array[String]) = {
    var result1 = (a:Int, b:Int) => a+b        // Anonymous function by using => (rocket)
    var result2 = (_:Int)+(_:Int)            // Anonymous function by using _ (underscore) wild card
    println(result1(10,10))
    println(result2(10,10))
  }
}    Output: 20 20
```

Scala Multiline Expression:-Expressions those are written in multiple lines are called multiline expression. In scala, be carefull while using multiline expressions.
The following program explains about if we break an expression into multiline, the scala compiler throw a warning message.
Scala Multiline Expression Example

```
def add1(a:Int, b:Int) = {
      a
      +b
  }
```

The above program does not evaluate complete expression and just return b here. So, be careful while using multiline expressions.

Scala Example Multiline Expression:-

```
object MainObject {
    def add2(a:Int, b:Int) = {
        a+
        b
    }
    def add3(a:Int, b:Int) = {
        (a
        +b)
    }
    def main(args: Array[String]) = {
        var result2 = add2(10,10)
        var result3 = add3(10,10)
        println(result2+"\n"+result3)
    }
}
```

```
Output;- 20 20
```

Scala Function Currying:-

In scala, method may have multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a
function taking the missing parameter lists as its arguments.
In other words it is a technique of transforming a function that takes multiple arguments into a function that takes a single argument.

ex:-

```
object MainObject {
    def add(a:Int)(b:Int) = {
        a+b
    }
    def main(args: Array[String]) = {
        var result = add(10)(10)

        println("10 + 10 = "+result)

        var addIt = add(10)_

        var result2 = addIt(3)

        println("10 + 3 = "+result2)

    }

}
```

output:- 20

Scala Nested Functions

Scala is a first class function language which means it allows you to passing function, returning function, composing function, nested function etc.

An example below explain about how to define and call nested functions.

Scala Nested Functions Example

```
object MainObject {

    def add(a:Int, b:Int, c:Int) = {

        def add2(x:Int,y:Int) = {

            x+y

        }

        add2(a,add2(b,c))

    }

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

        var result = add(10,10,10)

        println(result)

    }

}
```

Output:

```
30
```

Scala Function with Variable Length Parameters

In scala, you can define function of variable length parameters. It allows you to pass any number of arguments at the time of calling the function.

Let's see an example.

Scala Example: Function with Variable Length Parameters

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

Output:

```
45
```

Scala Object and Classes:-

Class:-Class is a template or a blueprint. It is also known as collection of objects of similar type.

In scala, a class can contain:Data Member,Member function,constructor etc.

Note:-You must initialize all instance variables in the class. There is no default scope. If you don't specify access scope, it is public.

ex 1:-

```
class Student{
   var id:Int = 0;                    // All fields must be initialized
   var name:String = null;
}
object MainObject{
   def main(args:Array[String]){
      var s = new Student()             // Creating an object
      println(s.id+" "+s.name);
   }
}
```

Output:- 0 null

Ex2:-

In scala, you can create class like this also. Here, constructor is created in class definition.

This is called primary constructor.

```
class Student(id:Int, name:String){     // Primary constructor
   def show(){
      println(id+" "+name)
   }
}
object MainObject{
   def main(args:Array[String]){
      var s = new Student(100,"Martin")   // Passing values to constructor
      s.show()               // Calling a function by using an object
   }
}
```

Output:

```
100 Martin
```

Scala Example of class that maintains the records of students

```
class Student(id:Int, name:String){
    def getRecord(){
        println(id+" "+name);
    }
}

object MainObject{
    def main(args: Array[String]){
        var student1 = new Student(101,"Raju");
        var student2 = new Student(102,"Martin");
        student1.getRecord();
        student2.getRecord();
    }
}
```

Output:

```
101 Raju
102 Martin
```

Scala Anonymous object:-

In scala, you can create anonymous object. An object which has no reference name is called anonymous object. It is good

to create anonymous object when you don't want to reuse it further.

Scala Anonymous object Example

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

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


    }
}
```

Output:

```
Sum = 20
```

Scala Constructor:-

In scala, constructor is not special method. Scala provides primary and any number of auxiliary constructors. We have explained

each in details in the following example.

Scala Default Primary Constructor

In scala, if you don't specify primary constructor, compiler creates a constructor which is known as primary constructor. All the

statements of class body treated as part of constructor. It is also known as default constructor.

Scala Default Primary Constructor Example

```
class Student{
println("Hello from default constructor");
}
```

Output:

```
Hello from default constructor
```

Scala Primary Constructor

Scala provides a concept of primary constructor with the definition of class. You don't need to define explicitly constructor if

your code has only one constructor. It helps to optimize code. You can create primary constructor with zero or more parameters.

Scala Primary Constructor Example

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

object MainObject{
  def main(args:Array[String]){
    var s = new Student(101,"Rama");
    s.showDetails()
  }
}
```

Output:

```
101 Rama
```

Scala Secondary (auxiliary) Constructor

You can create any number of auxiliary constructors in a class. You must call primary constructor from inside the auxiliary constructor.

this keyword is used to call constructor from other constructor. When calling other constructor make it first line in your constructor.

Scala Secondary Constructor Example

```
class Student(id:Int, name:String){
    var age:Int = 0
    def showDetails(){
        println(id+" "+name+" "+age)
    }
    def this(id:Int, name:String,age:Int){
        this(id,name)       // Calling primary constructor, and it is first line
        this.age = age
    }
}

object MainObject{
    def main(args:Array[String]){
        var s = new Student(101,"Rama",20);
        s.showDetails()
    }
}
```

Output:

```
101 Rama 20
```

Scala Example: Constructor Overloading

In scala, you can overload constructor. Let's see an example.

```
class Student(id:Int){
    def this(id:Int, name:String)={
        this(id)
        println(id+" "+name)
    }
    println(id)
}

object MainObject{
    def main(args:Array[String]){
        new Student(101)
        new Student(100,"India")
    }
}
```

Output:

```
101
100
100 India
```

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);
   }
}
```

Output:

```
20

30
```

Scala Method Overloading Example by using Different Data Type

In the following example, we have created two add method having same number of parameters but different data types.

```
class Arithmetic{
   def add(a:Int, b:Int){
      var sum = a+b
      println(sum)
   }
   def add(a:Double, b:Double){
      var sum = a+b
      println(sum)
   }
}
object MainObject{
   def main(args:Array[String]){
      var b = new Arithmetic()
      b.add(10,10)
      b.add(10.0,20.0)

   }
}
```

Output:

```
20
30.0
```

Scala this

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

Scala this Example

In the following example, this is used to call instance variables and primary constructotr.

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

Output:

```
101 Martin
```

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.

You must make sure that this

must be first statement in the constructor while calling to other constructor otherwise compiler throws an error.

```
class Student(name:String){
    def this(name:String, age:Int){
        this(name)
        println(name+" "+age)
    }
}

object MainObject{
    def main(args:Array[String]){
        var s = new Student("Rama",100)
    }
}
```

Output:

Rama 100

Scala Inheritance

Inheritance is an object oriented concept which is used to reusability of code. You can achieve inheritance by using extends

keyword. To achieve inheritance

a class must extend to other class. A class which is extended called super or parent class. a class which extends class

is called derived or base class.

Syntax

```
class SubClassName extends SuperClassName(){

/* Write your code

*  methods and fields etc.

 */

 }
```

Scala Single Inheritance Example

```
class Employee{

   var salary:Float = 10000

}


class Programmer extends Employee{

   var bonus:Int = 5000

   println("Salary = "+salary)

   println("Bonus = "+bonus)

}
```

```
object MainObject{
   def main(args:Array[String]){
      new Programmer()
   }
}
```

Output:

```
Salary = 10000.0
Bonus = 5000
```

Types of Inheritance in Scala

Scala supports various types of inheritance including single, multilevel, multiple, and hybrid. You can use single,

multilevel and hierarchal in your class.

Multiple and hybrid can only be achieved by using traits. Here, we are representing all types of inheritance by using

pictorial form.

Scala Multilevel Inheritance Example

```
class A{
   var salary1 = 10000
}

class B extends A{
   var salary2 = 20000
}
```

```
class C extends B{
  def show(){
    println("salary1 = "+salary1)
    println("salary2 = "+salary2)
  }
}

object MainObject{
  def main(args:Array[String]){{
    var c = new C()
    c.show()

  }
}
```

Output:

```
salary1 = 10000
salary2 = 20000
```

Scala Method Overriding

When a subclass has the same name method as defined in the parent class, it is known as method overriding. When subclass wants

to provide a specific implementation for the method defined in the parent class, it overrides method from parent class.

In scala, you must use either override keyword or override annotation to override methods from parent class.

Scala Method Overriding Example 1

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

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

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

Output:

```
Bike is running
```

Scala Method Overriding Example 2

This example shows how subclasses override the method of parent class.

```
class Bank{
        def getRateOfInterest()={
```

```
        0
    }
}

class SBI extends Bank{
    override def getRateOfInterest()={
     8
    }
}

class ICICI extends Bank{
    override def getRateOfInterest()={
        7
    }
}

class AXIS extends Bank{
    override def getRateOfInterest()={
        9
    }
}

object MainObject{
    def main(args:Array[String]){
        var s=new SBI();
        var i=new ICICI();
        var a=new AXIS();
        println("SBI Rate of Interest: "+s.getRateOfInterest());
        println("ICICI Rate of Interest: "+i.getRateOfInterest());
```

```
            println("AXIS Rate of Interest: "+a.getRateOfInterest());
        }
    }
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Scala Field Overriding

In scala, you can override fields also but it has some rules that need to be followed. Below are some examples that illustrate how to override fields.

Scala Field Overriding Example1

```
class Vehicle{
    var speed:Int = 60

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

Output:

```
Error - variable speed needs 'override' modifier
```

In scala, you must use either override keyword or override annotation when you are overriding methods or fields of super class.

If you don't do this, compiler reports an error and stops execution of program.

Scala Field Overriding Example2

```
class Vehicle{
     val speed:Int = 60


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

Output:

```
100
```

In scala, you can override only those variables which are declared by using val keyword in both classes. Below are some

examples which demonstrate the whole process.

Scala Field Overriding Example3

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

Output:

```
variable speed cannot override a mutable variable
```

Scala Field Overriding Example4

```
class Vehicle{
     val speed:Int = 60

}

class Bike extends Vehicle{
   override var speed:Int = 100
    def show(){
        println(speed)
    }
}

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

Output:

```
Error - variable speed needs to be a stable, immutable value
```

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 Vehicle{
     final val speed:Int = 60
}
class Bike extends Vehicle{
   override val speed:Int = 100
    def show(){
        println(speed)
    }
}

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

Output:

```
Error - value speed cannot override final member
```

Scala Final Method

Final method declare in the parent class can't be override. You can make any method to final if you don't want to get it overridden.

Attempt to override final method will cause to a compile time error.

Scala Final Method Example

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

Output:

```
method show cannot override final member
    override def show(){
                 ^
one error found
```

Scala Final Class Example

You can also make final class. Final class can't be inherited. If you make a class final, it can't be extended further.

```
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()
    }
}
```

Output:

```
error: illegal inheritance from final class Vehicle
class Bike extends Vehicle{
                   ^
one error found
```

Scala Abstract Class

A class which is declared with abstract keyword is known as abstract class. An abstract class can have abstract methods and non-abstract methods as well.

Abstract class is used to achieve abstraction. Abstraction is a process in which we hide complex implementation details and show only functionality to the user.

In scala, we can achieve abstraction by using abstract class and trait. We have discussed about these in detail here.

Scala Abstract Class Example

In this example, we have created a Bike abstract class. It contains an abstract method. A class Hero extends it and provides implementation of its run method.

A class that extends an abstract class must provide implementation of its all abstract methods. You can't create object of an abstract class.

```
abstract class Bike{
  def run()
}

class Hero extends Bike{
  def run(){
    println("running fine...")
  }
}

object MainObject{
  def main(args: Array[String]){
    var h = new Hero()
```

```
        h.run()
    }
}
```

Output:

```
running fine...
```

Scala Abstract Class Example: Having Constructor, Variables and Abstract Methods

```
abstract class Bike(a:Int){             // Creating constructor
    var b:Int = 20                      // Creating variables
    var c:Int = 25
    def run()                           // Abstract method
    def performance(){                  // Non-abstract method
        println("Performance awesome")
    }
}

class Hero(a:Int) extends Bike(a){
    c = 30
    def run(){
        println("Running fine...")
        println("a = "+a)
        println("b = "+b)
        println("c = "+c)
    }
}

object MainObject{
    def main(args: Array[String]){
        var h = new Hero(10)
        h.run()
        h.performance()
    }
}
```

Output:

```
Running fine...
a = 10
b = 20
c = 30
Performance awesome
```

Scala Abstract Class Example: Abstract Method is not implemented

In this example, we didn't implement abstract method run(). Compiler reports an error during compilation of this program.

Error message is given below in output section.

```
abstract class Bike{
    def run()               // Abstract method
}

class Hero extends Bike{        // Not implemented in this class
    def runHero(){
        println("Running fine...")
    }
}

object MainObject{
    def main(args: Array[String]){
        var h = new Hero()
        h.runHero()
    }
}
```

Output:

```
error: class Hero needs to be abstract, since method run in class Bike of type ()Unit is not defined
class Hero extends Bike{
      ^
one error found
```

To avoid this problem either you must implement all abstract members of abstract class or make your class abstract too.

Scala Trait

A trait is like an interface with a partial implementation. In scala, trait is a collection of abstract and non-abstract methods.

You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

A variable that is declared either by using val or var keyword in a trait get internally implemented in the class that implements

the trait. Any variable which is declared by using val or var but not initialized is considered abstract.

Traits are compiled into Java interfaces with corresponding implementation classes that hold any methods implemented in the traits.

Scala Trait Example

```
trait Printable{
   def print()
}

class A4 extends Printable{
   def print(){
      println("Hello")
   }
}

object MainObject{
   def main(args:Array[String]){
      var a = new A4()
      a.print()
   }
}
```

Output:

```
Hello
```

If a class extends a trait but does not implement the members declared in that trait, it must be declared abstract. Let's see an example.

Scala Trait Example

```
trait Printable{
   def print()
}

abstract class A4 extends Printable{            // Must declared as abstract class
```

```
  def printA4(){
      println("Hello, this is A4 Sheet")
  }
}
```

Scala Trait Example: Implementing Multiple Traits in a Class

If a class implements multiple traits, it will extend the first trait, class, abstract class. with keyword is used to extend rest of the traits.

You can achieve multiple inheritances by using trait.

```
trait Printable{
   def print()
}

trait Showable{
  def show()
}

class A6 extends Printable with Showable{
   def print(){
       println("This is printable")
   }
   def show(){
       println("This is showable");
   }
}

object MainObject{
   def main(args:Array[String]){
       var a = new A6()
       a.print()
       a.show()
   }
}
```

Output:

```
This is printable
This is showable
```

Scala Trait having abstract and non-abstract methods

You can also define method in trait as like in abstract class. I.e. you can treat trait as abstract class also. In scala, trait is

almost same as abstract class except that it can't have constructor. You can't extend multiple abstract classes but can extend multiple traits.

Scala Trait Example

```
trait Printable{
    def print()         // Abstract method
    def show(){         // Non-abstract method
        println("This is show method")
    }
}

class A6 extends Printable{
    def print(){
        println("This is print method")
    }
}

object MainObject{
    def main(args:Array[String]){
        var a = new A6()
        a.print()
        a.show()
    }
}
```

Output:

```
This is print method
This is show method
```

Scala Trait Mixins

In scala, trait mixins means you can extend any number of traits with a class or abstract class. You 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.

You 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{
  def main(args:Array[String]){
    var a = new A6()
    a.print()
    a.printA4()
  }
}
```

Output:

```
error: class PrintA4 needs to be a trait to be mixed in
class A6 extends Print with PrintA4{
                            ^
```

one error found

The above program throws a compile time error, because we didn't maintain mixins order.

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 Print{
  def print()
}

abstract class PrintA4{
  def printA4()
}

class A6 extends PrintA4 with Print{          // 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{
    def main(args:Array[String]){
        var a = new A6()
        a.print()
        a.printA4()
    }
}
```

Output:

```
print sheet
Print A4 Sheet
```

Another Example of Scala Trait

Here, we have used one more approach to extend trait in our program. In this approach, we extend trait during object creation.

Let's see an example.

```
trait Print{
    def print()
}

abstract class PrintA4{
    def printA4()
}

class A6 extends PrintA4 {
    def print(){                        // Trait print
        println("print sheet")
    }
    def printA4(){                          // Abstract class printA4
```

```
        println("Print A4 Sheet")
    }
}

object MainObject{
    def main(args:Array[String]){
        var a = new A6() with Print            // You can also extend trait during object creation
        a.print()
        a.printA4()
    }
}
```

Output:

```
print sheet
Print A4 Sheet
```

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:

No modifier

Protected

Private

In scala, if you don't mention any access modifier, it is treated as no modifier.

NOTE:-In scala, if you don't mention any access modifier, it is treated as no modifier

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 AccessExample{
    private var a:Int = 10
    def show(){
        println(a)
    }
}
object MainObject{
   def main(args:Array[String]){
      var p = new AccessExample()
      p.a = 12
      p.show()
   }
}
```

Output:

```
error: variable a in class AccessExample cannot be accessed in AccessExample
      p.a = 12
```

```
        ^
one error found
```

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 AccessExample{
    protected var a:Int = 10
}
class SubClass extends AccessExample{
   def display(){
      println("a = "+a)
   }
}
object MainObject{
   def main(args:Array[String]){
      var s = new SubClass()
      s.display()
   }
}
```

Output:

```
a = 10
```

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 AccessExample{
    var a:Int = 10
    def show(){
        println(" a = "+a)
    }
}

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

Output:

```
a = 10
```

Scala String Interpolation:-

Starting in Scala 2.10.0, Scala offers a new mechanism to create strings from your data. It is called string interpolation.

String interpolation allows users to embed variable references directly in processed string literals. Scala provides three string interpolation methods: s, f and raw.

Scala Program Example: Without using s Method

This is simple example which does not use s method in string.

```
class StringExample{
    var pi = 3.14
    def show(){
        println("value of pi = "+pi)
    }
}
object MainObject{
    def main(args:Array[String]){
        var s = new StringExample()
        s.show()
    }
}
```

Output:

```
value of pi = 3.14
```

Scala String Interpolation Example

This program use string interpolation in print function. You can see the advantage of interpolation. Here, we did not use + operator

to concatenate string objects.

```
class StringExample{
    var pi = 3.14
    def show(){
        println(s"value of pi = $pi")
    }
}
```

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

Output:

```
value of pi = 3.14
```

Scala String Interpolation Example By using s Method

The s method of string interpolation allows us to pass variable in string object. You don't need to use + operator to format your output

string. In the following example, a string variable is passed to string in the print function. This variable is evaluated by compiler

and variable is replaced by value.

```
class StringExample{
    var s1 = "Scala string example"
    def show(){
        println(s"This is $s1")
    }
}



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

Output:

```
This is Scala string example
```

Scala String Interpolation Example By using f Method

The f method is used to format your string output. It is like printf function of c language which is used to produce formatted output.

You can pass your variables of any type in the print function.

```
class StringExample{
    var s1 = "Scala string example"
    var version = 2.12
    def show(){
        println(f"This is $s1%s, scala version is $version%2.2f")
    }
}



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

Output:

This is Scala string example, scala version is 2.12

Scala Exception Handling

Exception handling is a mechanism which is used to handle abnormal conditions. You 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 ExceptionExample{
  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)

  }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
```

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. You 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 ExceptionExample{
    def divide(a:Int, b:Int) = {
        try{
            a/b
        }catch{
            case e: ArithmeticException => println(e)
        }
        println("Rest of the code is executing...")
    }
}
object MainObject{
    def main(args:Array[String]){
        var e = new ExceptionExample()
        e.divide(100,0)
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
Rest of the code is executing...
```

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)

    }
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
Finally block always executes
Rest of the code is executing...
```

Scala Throw keyword

You can throw exception explicitly in you code. Scala provides throw keyword to throw exception. The throw keyword mainly used to throw

custom exception. An example is given below of using scala throw exception keyword.

Scala Throw Example

```
class ExceptionExample2{
    def validate(age:Int)={
        if(age<18)
           throw new ArithmeticException("You are not eligible")
        else println("You are eligible")
    }
}

object MainObject{
    def main(args:Array[String]){
        var e = new ExceptionExample2()
        e.validate(10)

    }
}
```

Output:

```
java.lang.ArithmeticException: You are not eligible
```

Scala Throws Keyword

Scala provides throws keyword to declare exception. You can declare exception with method definition. It provides information to

the caller function that this method may throw this exception. It helps to caller function to handle and enclose that code in try-catch

block to avoid abnormal termination of program. In scala, you can either use throws keyword or throws annotation to declare exception.

Scala Throws Example

```
class ExceptionExample4{
    @throws(classOf[NumberFormatException])
    def validate()={
        "abc".toInt
    }
}

object MainObject{
    def main(args:Array[String]){
        var e = new ExceptionExample4()
        try{
            e.validate()
        }catch{
            case ex : NumberFormatException => println("Exception handeled here")
        }
        println("Rest of the code executing...")
    }
}
```

Output:

```
Exception handeled here
Rest of the code executing...
```

Scala Custom Exception

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

You can create your own exception message in custom class. Let's see an example.

Scala Custom Exception Example

```
class InvalidAgeException(s:String) extends Exception(s){}
class ExceptionExample{
  @throws(classOf[InvalidAgeException])
  def validate(age:Int){
    if(age<18){
      throw new InvalidAgeException("Not eligible")
    }else{
      println("You are eligible")
    }
  }
}
object MainObject{
  def main(args:Array[String]){
    var e = new ExceptionExample()
    try{
      e.validate(5)
    }catch{
      case e : Exception => println("Exception Occured : "+e)
    }
  }
}
```

Output:

```
Exception Occured : InvalidAgeException: Not eligible
```

Scala Collection

Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable.

You 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 Traversable

It is a trait and used to traverse collection elements. It is a base trait for all scala collections.

It implements the methods which are common to all collections.

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

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

val variableName = Set(element1, element2,... elementN)

Scala Set Example

In this example, we have created a set. You can create an empty set also. Let's see how to create a set.

```
import scala.collection.immutable._
object MainObject{
   def main(args:Array[String]){
      val set1 = Set()                         // An empty set
      val games = Set("Cricket","Football","Hocky","Golf")    // Creating a set with elements
      println(set1)
      println(games)
   }
}
```

Output:

```
Set()      // an empty set
Set(Cricket,Football,Hocky,Golf)
```

Scala Set Example 2

In Scala, Set provides some predefined properties to get information about set. You can get first or last element of Set and many more. Let's see an example.

```
import scala.collection.immutable._
object MainObject{
      def main(args:Array[String]){
         val games = Set("Cricket","Football","Hocky","Golf")
         println(games.head)            // Returns first element present in the set
         println(games.tail)        // Returns all elements except first element.
         println(games.isEmpty)         // Returns either true or false
      }
   }
```

Output:

```
Cricket
Set(Football, Hocky, Golf)
false
```

Scala Set Example: Merge two Set

You 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.

```
import scala.collection.immutable._
object MainObject{
    def main(args:Array[String]){
        val games = Set("Cricket","Football","Hocky","Golf")
        val alphabet = Set("A","B","C","D","E")
        val mergeSet = games ++ alphabet           // Merging two sets
        println("Elements in games set: "+games.size)   // Return size of collection
        println("Elements in alphabet set: "+alphabet.size)
        println("Elements in mergeSet: "+mergeSet.size)
        println(mergeSet)
    }
  }
```

Output:

```
Elements in games set: 4
Elements in alphabet set: 5
Elements in mergeSet: 9
Set(E, Football, Golf, Hocky, A, B, C, Cricket, D)
```

This example also proves that the merge set does not maintain order to store elements.

Scala Set Example 2

You can check whether element is present in the set or not. The following example describe the use of contains() method.

```
import scala.collection.immutable._
object MainObject{
    def main(args:Array[String]){
      val games = Set("Cricket","Football","Hocky","Golf")
      println(games)
      println("Elements in set: "+games.size)
      println("Golf exists in the set : "+games.contains("Golf"))
      println("Racing exists in the set : "+games.contains("Racing"))

    }
  }
```

Output:

```
Set(Cricket, Football, Hocky, Golf)
Elements in set: 4
Golf exists in the set : true
Racing exists in the set : false
```

Scala Set Example: Adding and Removing Elements

You can add or remove elements from the set. You can add only when your code is mutable. In this example, we are adding and removing elements of the set.

```
import scala.collection.immutable._
object MainObject{
    def main(args:Array[String]){
          var games = Set("Cricket","Football","Hocky","Golf")
          println(games)
          games += "Racing"              // Adding new element
          println(games)
          games += "Cricket"             // Adding new element, it does not allow duplicacy.
          println(games)
          games -= "Golf"            // Removing element
          println(games)
        }
   }
```

Output:

```
Set(Cricket, Football, Hocky, Golf)
Set(Football, Golf, Hocky, Cricket, Racing)
Set(Football, Golf, Hocky, Cricket, Racing)
Set(Football, Hocky, Cricket, Racing)
```

Scala Set Example: Iterating Set Elements using for loop

You can iterate set elements either by using for loop or foreach loop. You can also filter elements during iteration. In this example have used

for loop to iterate set elements.

```
import scala.collection.immutable._
object MainObject{
        def main(args:Array[String]){
          var games = Set("Cricket","Football","Hocky","Golf")
          for(game <- games){
             println(game)
          }
        }
    }
```

Output:

```
Cricket
Football
Hocky
Golf
```

Scala Set Example Iterating Elements using foreach loop

In this example, we are using foreach loop to iterate set elements.

```
import scala.collection.immutable._
    object MainObject{
        def main(args:Array[String]){
            var games = Set("Cricket","Football","Hocky","Golf")
            games.foreach((element:String)=> println(element))
        }
}
```

Output:

```
Cricket
Football
Hocky
Golf
```

Scala Set Example: Set Operations

In scala Set, you can also use typical math operations like: intersection and union. In the following example we have used predefined methods to perform set operations.

```
import scala.collection.immutable._
object MainObject{
   def main(args:Array[String]){
      var games = Set("Cricket","Football","Hocky","Golf","C")
      var alphabet = Set("A","B","C","D","E","Golf")
      var setIntersection = games.intersect(alphabet)
      println("Intersection by using intersect method: "+setIntersection)
      println("Intersection by using & operator: "+(games & alphabet))
      var setUnion = games.union(alphabet)
      println(setUnion)
   }
}
```

Output:

```
Intersection by using intersect method: Set(Golf, C)
Intersection by using & operator: Set(Golf, C)
Set(E, Football, Golf, Hocky, A, B, C, Cricket, D)
```

Scala SortedSet

In scala, SortedSet extends Set trait and provides sorted set elements. It is useful when you want sorted elements in the Set collection.

You can sort integer values and string as well.

It is a trait and you can apply all the methods defined in the traversable trait and Set trait.

Scala SortedSet Example

In the following example, we have used SortedSet to store integer elements. It returns a Set after sorting elements.

```
import scala.collection.immutable.SortedSet
object MainObject{
   def main(args:Array[String]){
      var numbers: SortedSet[Int] = SortedSet(5,8,1,2,9,6,4,7,2)
      numbers.foreach((element:Int)=> println(element))
   }
}
```

Output:

```
1
2
4
5
6
7
8
9
```

Scala HashSet

HashSet is a sealed class. It extends AbstractSet and immutable Set trait. It uses hash code to store elements.

It neither maintains insertion order nor sorts the elements.

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 MainObject{
    def main(args:Array[String]){
        var hashset = HashSet(4,2,8,0,6,3,45)
        hashset.foreach((element:Int) => println(element+" "))
   }
}
```

Output:

```
0
6
2
45
3
8
4
```

Scala BitSet

Bitsets are sets of non-negative integers which are represented as variable-size arrays of bits packed into 64-bit words. The memory footprint of

a bitset is determined by the largest number stored in it. It extends Set trait.

Scala BitSet Example

```
import scala.collection.immutable._

object MainObject{

    def main(args:Array[String]){

        var numbers = BitSet(1,5,8,6,9,0)

        numbers.foreach((element:Int) => println(element))

    }

}
```

Output:

```
0

1

5

6

8

9
```

Scala BitSet Example: Adding and Removing Elements

You can perform basic operations like adding and deleting in the bitset. In the following example, we have applied these operations.

```
import scala.collection.immutable._

object MainObject{

    def main(args:Array[String]){

        var numbers = BitSet(1,5,8,6,9,0)

        numbers.foreach((element:Int) => print(element+" "))

        numbers += 20               // Adding an element

        print("\nAfter adding 20: ")
        numbers.foreach((element:Int) => print(element+" "))
        numbers-=0              // Deleting an element
        print("\nAfter deleting 0: ")
        numbers.foreach((element:Int) => print(element+" "))
    }
}
```

Output:

```
0 1 5 6 8 9
After adding 20: 0 1 5 6 8 9 20
After deleting 0: 1 5 6 8 9 20
```

Scala ListSet

In scala, ListSet class implements immutable sets using a list-based data structure. Elements are stored internally in reversed insertion order,

which means the newest element is at the head of the list. It maintains insertion order.

This collection is suitable only for a small number of elements. You can create empty ListSet either by calling the constructor or by applying

the function ListSet.empty. Its iterate and traversal methods visit elements in the same order in which they were first inserted.

Scala ListSet Example

```
import scala.collection.immutable._
object MainObject{
    def main(args:Array[String]){
        var listset = ListSet(4,2,8,0,6,3,45)
        listset.foreach((element:Int) => println(element+" "))
    }
}
```

Output:

```
4
2
8
0
6
3
45
```

Scala ListSet Example: Creating ListSet and Adding Elements

```
import scala.collection.immutable._
object MainObject{
    def main(args:Array[String]){
        var listset:ListSet[String] = new ListSet()                // Creating empty ListSet by using constructor
        var listset2:ListSet[String] = ListSet.empty               // Creating an empty listset
        println("listset: "+listset)
        println("listset2: "+listset2)
        println("After adding new elements:")
        listset+="India"            // Adding new element
        listset2+="Russia"          // Adding new element
        println("listset: "+listset)
        println("listset2: "+listset2)
    }
}
```

Output:

```
listset: ListSet()

listset2: ListSet()

After adding new elements:

listset: ListSet(India)

listset2: ListSet(Russia)
```

Scala Seq

Seq is a trait which represents indexed sequences that are guaranteed immutable. You 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.

Scala Seq Example

In the following example, we are creating Seq and accessing elements from Seq.

```
import scala.collection.immutable._

object MainObject{

   def main(args:Array[String]){

      var seq:Seq[Int] = Seq(52,85,1,8,3,2,7)

      seq.foreach((element:Int) => print(element+" "))

      println("\nAccessing element by using index")

      println(seq(2))

   }
}
```

Output:

```
52 85 1 8 3 2 7
Accessing element by using index
1
```

You can also access elements in reverse order by using reverse method. Below we have listed some commonly used method and their description.

Scala Seq Example

In this example, we have applied some predefined methods of Seq trait.

```
import scala.collection.immutable._
object MainObject{
  def main(args:Array[String]){
    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)
  }
}
```

Output:

```
52 85 1 8 3 2 7
is Empty: false
```

```
Ends with (2,7): true
contains 8: true
last index of 3 : 4
Reverse order of sequence: List(7, 2, 3, 8, 1, 85, 52)
```

Scala Vector

Vector is a general-purpose, immutable data structure. It provides random access of elements. It is good for large collection of elements.

It extends an abstract class AbstractSeq and IndexedSeq trait.

Scala Vector Example

```
import scala.collection.immutable._
object MainObject{
  def main(args:Array[String]){
    var vector:Vector[Int] = Vector(5,8,3,6,9,4) //Or
    var vector2 = Vector(5,2,6,3)
    var vector3 = Vector.empty
    println(vector)
    println(vector2)
    println(vector3)
  }
}
```

Output:

```
Vector(5, 8, 3, 6, 9, 4)
Vector(5, 2, 6, 3)
```

Vector(

Scala Vector Example

In the following example, we have created a vector. You can also add new element and merge two vectors.

```
import scala.collection.immutable._
object MainObject{
   def main(args:Array[String]){
      var vector = Vector("Hocky","Cricket","Golf")
      var vector2 = Vector("Swimming")
      print("Vector Elements: ")
      vector.foreach((element:String) => print(element+" "))
      var newVector  = vector :+ "Racing"                          // Adding a new element into vector
      print("\nVector Elements after adding: ")
      newVector.foreach((element:String) => print(element+" "))
      var mergeTwoVector = newVector ++ vector2                    // Merging two vector
      print("\nVector Elements after merging: ")
      mergeTwoVector.foreach((element:String) => print(element+" "))
      var reverse = mergeTwoVector.reverse                         // Reversing vector elements
      print("\nVector Elements after reversing: ")
      reverse.foreach((element:String) => print(element+" "))
      var sortedVector = mergeTwoVector.sorted                     // Sorting vector elements
      print("\nVector Elements after sorting: ")
      sortedVector.foreach((element:String) => print(element+" "))
   }
}
```

Output:

```
Vector Elements: Hocky Cricket Golf

Vector Elements after adding: Hocky Cricket Golf Racing

Vector Elements after merging: Hocky Cricket Golf Racing Swimming

Vector Elements after reversing: Swimming Racing Golf Cricket Hocky

Vector Elements after sorting: Cricket Golf Hocky Racing Swimming
```

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.

Scala List Example

In this example, we have created two lists. Here, both lists have different syntax to create list.

```
import scala.collection.immutable._

object MainObject{

   def main(args:Array[String]){

      var list = List(1,8,5,6,9,58,23,15,4)

       var list2:List[Int] = List(1,8,5,6,9,58,23,15,4)

       println(list)

       println(list2)

   }

}
```

Output:

```
List(1, 8, 5, 6, 9, 58, 23, 15, 4)
List(1, 8, 5, 6, 9, 58, 23, 15, 4)
```

Scala List Example: Applying Predefined Methods

```
import scala.collection.immutable._
object MainObject{
  def main(args:Array[String]){
    var list = List(1,8,5,6,9,58,23,15,4)
    var list2 = List(88,100)
    print("Elements: ")
    list.foreach((element:Int) => print(element+" "))       // Iterating using foreach loop
    print("\nElement at 2 index: "+list(2))           // Accessing element of 2 index
    var list3 = list ++ list2                          // Merging two list
    print("\nElement after merging list and list2: ")
    list3.foreach((element:Int)=>print(element+" "))
    var list4 = list3.sorted                         // Sorting list
    print("\nElement after sorting list3: ")
    list4.foreach((element:Int)=>print(element+" "))
    var list5 = list3.reverse                        // Reversing list elements
    print("\nElements in reverse order of list5: ")
    list5.foreach((element:Int)=>print(element+" "))

  }
}
```

Output:

```
Elements: 1 8 5 6 9 58 23 15 4
Element at 2 index: 5
Element after merging list and list2: 1 8 5 6 9 58 23 15 4 88 100

Element after sorting list3: 1 4 5 6 8 9 15 23 58 88 100

Elements in reverse order of list5: 100 88 4 15 23 58 9 6 5 8 1
```

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.

Scala Queue Example

```
import scala.collection.immutable._

object MainObject{

  def main(args:Array[String]){

    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)

  }

}
```

Output:

```
Queue(1, 5, 6, 2, 3, 9, 5, 2, 5)

Queue(1, 5, 6, 2, 3, 9, 5, 2, 5)
```

Scala Queue Example 2

```
import scala.collection.immutable._
object MainObject{
  def main(args:Array[String]){
    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)
  }
}
```

Output:

```
Queue Elements: 1 5 6 2 3 9 5 2 5
First element in the queue: 1
Element added in the queue: 1 5 6 2 3 9 5 2 5 100
Element deleted from this queue: (1,Queue(5, 6, 2, 3, 9, 5, 2, 5))
```

Scala Stream

Stream is a lazy list. It evaluates elements only when they are required. This is a feature of scala. Scala supports lazy computation.

It increases performance of your program.

Scala Stream Example

In the following program, we have created a stream.

```
object MainObject{
    def main(args:Array[String]){
        val stream = 100 #:: 200 #:: 85 #:: Stream.empty
        println(stream)
    }
}
```

Output:

```
Stream(100, ?)
```

In the output, you can see that second element is not evaluated. Here, a question mark is displayed in place of element. Scala does not evaluate list

until it is required.

Scala Stream Example: Applying Predefined Methods

In the following example, we have used some predefined methods like toStream, which is used to iterate stream elements.

```
import scala.collection.immutable._
object MainObject{
    def main(args:Array[String]){
        var stream = 100 #:: 200 #:: 85 #:: Stream.empty
        println(stream)
        var stream2 = (1 to 10).toStream
        println(stream2)
        var firstElement = stream2.head
        println(firstElement)
        println(stream2.take(10))
        println(stream.map{_*2})
    }
}
```

Output:

```
Stream(100, ?)
Stream(1, ?)
1
Stream(1, ?)
Stream(200, ?)
```

Scala Maps

Map is used to store elements. It stores elements in pairs of key and values. In scala, you can create map by using two ways either by using comma

separated pairs or by using rocket operator.

Scala maps Example

In the following example, we have both approaches to create map.

```
object MainObject{
    def main(args:Array[String]){
        var map = Map(("A","Apple"),("B","Ball"))
        var map2 = Map("A"->"Aple","B"->"Ball")
        var emptyMap:Map[String,String] = Map.empty[String,String]
        println(map)
        println(map2)
        println("Empty Map: "+emptyMap)
    }
}
```

Output:

```
Map(A -> Apple, B -> Ball)
Map(A -> Aple, B -> Ball)
Empty Map: Map()
```

Scala Map Example: Adding and Removing Elements

You can add and remove new elements in maps. Scala provides you lots of predefined method. You can use them to perform operations on the Maps.

In the following example, we have created a new Map.

```
object MainObject{
    def main(args:Array[String]){
        var map = Map("A"->"Apple","B"->"Ball")          // Creating map
        println(map("A"))                            // Accessing value by using key
        var newMap = map+("C"->"Cat")                // Adding a new element to map
        println(newMap)
        var removeElement = newMap - ("B")           // Removing an element from map
        println(removeElement)
    }
}
```

Output:

```
Apple
Map(A -> Apple, B -> Ball, C -> Cat)
Map(A -> Apple, C -> Cat)
```

Scala HashMap

HashMap is used to store element. It use hash code to store elements and return a map.

HashMap Example

In this example, we have created a HashMap.

```
import scala.collection.immutable._
object MainObject{
   def main(args:Array[String]){
      var hashMap = new HashMap()
      var hashMap2 = HashMap("A"->"Apple","B"->"Ball","C"->"Cat")
      println(hashMap)
      println(hashMap2)
   }
}
```

Output:

```
Map()
Map(A -> Apple, B -> Ball, C -> Cat)
```

Scala HashMap Example: Adding and Accessing Elements

In the following example, we have created a HashMap. this program add elements and access elements as well.

```
import scala.collection.immutable._
object MainObject{
  def main(args:Array[String]){
    var hashMap = HashMap("A"->"Apple","B"->"Ball","C"->"Cat")
    hashMap.foreach {
      case (key, value) => println (key + " -> " + value)      // Iterating elements
    }
    println(hashMap("B"))              // Accessing value by using key
    var newHashMap = hashMap+("D"->"Doll")
    newHashMap.foreach {
      case (key, value) => println (key + " -> " + value)
    }

  }
}
```

Output:

```
A -> Apple
B -> Ball
C -> Cat
Ball
A -> Apple
B -> Ball
C -> Cat
D -> Doll
```

Scala ListMap

This class implements immutable maps by using a list-based data structure. It maintains insertion order and returns ListMap. This collection

is suitable for small elements.

You can create empty ListMap either by calling its constructor or using ListMap.empty method.

Scala ListMap Example

In this example, we have created an empty ListMap and non-empty ListMap as well.

```
import scala.collection.immutable._
object MainObject{
   def main(args:Array[String]){
       var listMap = ListMap("Rice"->"100","Wheat"->"50","Gram"->"500")    // Creating listmap with
elements
       var emptyListMap = new ListMap()           // Creating an empty list map
       var emptyListMap2 = ListMap.empty          // Creating an empty list map
       println(listMap)
       println(emptyListMap)
       println(emptyListMap2)
   }
}
```

Output:

```
ListMap(Rice -> 100, Wheat -> 50, Gram -> 500)
ListMap()
```

ListMap()

Scala ListMap Example: Applying Basic Operations

```
import scala.collection.immutable._
object MainObject{
   def main(args:Array[String]){
      var listMap = ListMap("Rice"->"100","Wheat"->"50","Gram"->"500")    // Creating listmap with
elements
      listMap.foreach{
         case(key,value)=>println(key+"->"+value)
      }
      println(listMap("Gram"))
      var newListMap = listMap+("Pulses"->"550")
      newListMap.foreach {
         case (key, value) => println (key + " -> " + value)
      }
   }
}
```

Output:

```
Rice->100
Wheat->50
Gram->500
500
Rice -> 100
Wheat -> 50
Gram -> 500
Pulses -> 550
```

Scala Tuples

A tuple is a collection of elements in ordered form. If there is no element present, it is called empty tuple. You can use tuple to store any type of data.

You can store similar type or mix type data also. You can use it to return multiple values from a function

Scala Tuple Example

In the following example, we have created tuple of different types of elements.

```
object MainObject{
   def main(args:Array[String]){
       var tuple = (1,5,8,6,4)                    // Tuple of integer values
       var tuple2 = ("Apple","Banana","Gavava")        // Tuple of string values
       var tuple3 = (2.5,8.4,10.50)               // Tuple of float values
       var tuple4 = (1,2.5,"India")               // Tuple of mix type values
       println(tuple)
       println(tuple2)
       println(tuple3)
       println(tuple4)
   }
}
```

Output:

```
(1,5,8,6,4)
(Apple,Banana,Gavava)
(2.5,8.4,10.5)
(1,2.5,India)
```

Scala Tuple Example: Accessing Tuple Elements

In this example, we are accessing tuple elements by using index. Here, we are using productIterator for iterating tuple elements.

```
object MainObject{
   def main(args:Array[String]){
       var tupleValues = (1,2.5,"India")
       println("iterating values: ")
       tupleValues.productIterator.foreach(println)    // Iterating tuple values using productIterator
       println("Accessing values: ")
       println(tupleValues._1) // Fetching first value
       println(tupleValues._2) // Fetching second value
   }
}
```

Output:

```
iterating values:
1
2.5
India
Accessing values:
1
2.5
```

Scala Tuple Example: Function Return Multiple Values

You can return multiple values by using tuple. Function does not return multiple values but you can do this with the help of tuple. In the following example,

we are describing this process.

```
object MainObject{
   def main(args:Array[String]){
      var tupleValues = tupleFunction()
      println("Iterating values: ")
      tupleValues.productIterator.foreach(println)    // Iterating tuple values using productIterator
   }
   def tupleFunction()={
      var tuple = (1,2.5,"India")
      tuple
   }
}
```

Output:

```
Iterating values:
1
2.5
India
```

Scala File handling

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

In this chapter, we will discuss all these file operations in detail.

Scala Creating a File Example

Scala doesn't provide file writing methods. So, you have to use the Java PrintWriter or FileWriter methods.

```
import java.io._

val fileObject = new File("ScalaFile.txt" )     // Creating a file

val printWriter = new PrintWriter(fileObject)       // Passing reference of file to the printwriter

printWriter.write("Hello, This is scala file")  // Writing to the file

printWriter.close()            // Closing printwriter
```

The above code will create a text file ScalaFile.txt. After creating file printwriter is used to write content to this file.

Scala Reading File Example: Reading Each Charecter

```
import scala.io.Source


object MainObject{

  def main(args:Array[String]){

    val filename = "ScalaFile.txt"

    val fileSource = Source.fromFile(filename)

    while(fileSource.hasNext){

      println(fileSource.next)

    }

    fileSource.close()

  }

}
```

Scala Reading a File Example: Reading Each Line

```
import scala.io.Source
object MainObject{
  def main(args:Array[String]){
    val filename = "ScalaFile.txt"
    val fileSource = Source.fromFile(filename)
    for(line<-fileSource.getLines){
      println(line)
    }
    fileSource.close()
  }
}
```

Output:

```
Hello, This is scala file
```

Scala Multithreading

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

You 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 you are familiar with multithreading concept of Java, you will come to know that it

is similar except the syntax of Scala language itself.

You 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:

1.New

2.Runnable

3.Running

4.Non-Runnable (Blocked)

5.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

Scala Thread Example by Extending Thread Class

The following example extends Thread class and overrides run method. The start() method is used to start thread.

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

Output:

```
Thread is running...
```

Scala Thread Example by Extending Runnable Interface

The following example implements Runnable interface and overrides run method. The start() method is used to start thread.

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

Output:

```
Thread is running...
```

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


}


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

____

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.

Scala Pattern Matching Example

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

Output:

```
One
```

____

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)
```

Scala Case Class Example

```
case class CaseClass(a:Int, b:Int)

object ss{
   def main(args:Array[String]){
      var c =  CaseClass(10,10)      // Creating object of case class
      println("a = "+c.a)            // Accessing elements of case class
      println("b = "+c.b)
   }
}
```

Output:

```
a = 10
b = 10
```

____