Experiment No 11

Amitesh Sawarkar
60009210105

Aim: - Implement programs using Abstract class and Traits in Scala

Theory: -

Abstract class in Scala: -

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.

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()
   }
}
```

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()
    }
}
```

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
```

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

Traits in Scala: -

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.

Example 1–

```
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()
   }
}
```

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 2

```
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()
   }
```

```
}
```

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()
   }
}
```

Important differences between Traits and Abstract Classes in Scala.

Sr. No.	Key	Trait	Abstract Class
1	Definition	Traits are similar to interfaces in Java and are created using trait keyword.	Abstract Class is similar to abstract classes in Java and are created using abstract keyword.
2	Multiple inheritance	Trait supports multiple inheritance.	Abstract Class supports single inheritance only.
3	Constructor parameters	Trait cannot have parameters in its constructors.	Abstract class can have parameterised constructor.
4	Interoperability	Traits are interoperable with java if they don't have any implementation.	Abstract classes are interoperable with java without any restriction.

Lab exercise to be performed in this session

1. We have to calculate the area of a rectangle, a square and a circle. Create an abstract class 'Shape' with three abstract methods namely 'RectangleArea' taking two parameters, 'SquareArea' and 'CircleArea' taking one parameter each. The parameters of 'RectangleArea' are its length and breadth, that of 'SquareArea' is its side and that of 'CircleArea' is its radius. Now create another class 'Area' containing all the three methods 'RectangleArea', 'SquareArea' and 'CircleArea' for printing the area of rectangle, square and circle respectively. Create an object of class 'Area' and call all the three methods.

```
trait Rectangle {
  def RectangleArea(length: Double, breadth: Double): Double = length * breadth
}

trait Square {
  def SquareArea(side: Double): Double = side * side
}

trait Circle {
  def CircleArea(radius: Double): Double = Math.PI * radius * radius
}

class Area extends Rectangle with Square with Circle

val areaObj = new Area

val rectangleArea = areaObj.RectangleArea(5.0, 8.0)
val squareArea = areaObj.SquareArea(4.0)
val circleArea = areaObj.CircleArea(3.0)

println(s"Area of Rectangle: $rectangleArea")
println(s"Area of Square: $squareArea")
println(s"Area of Circle: $circleArea")
```

```
Area of Rectangle: 40.0
Area of Square: 16.0
Area of Circle: 28.274333882308138
```

2. We have to calculate the percentage of marks obtained in three subjects (each out of 100) by student A and in four subjects (each out of 100) by student B. Create an abstract class 'Marks' with an abstract method 'getPercentage'. It is inherited by two other classes 'A' and 'B' each having a method with the same name which returns the percentage of the students. The constructor of student A takes the marks in three subjects as its parameters and the marks in four subjects as its parameters for student B. Create an object of the two classes and print the percentage of marks for both the students.

```
trait Marks {
  def getPercentage: Double
}

class A(subject1: Double, subject2: Double, subject3: Double) extends Marks {
  def getPercentage: Double = (subject1 + subject2 + subject3) / 3.0
}

class B(subject1: Double, subject2: Double, subject3: Double, subject4: Double) extends Marks {
  def getPercentage: Double = (subject1 + subject2 + subject3 + subject4) / 4.0
}

val studentA = new A(80.0, 90.0, 75.0)
val studentB = new B(85.0, 92.0, 78.0, 88.0)

println(s"Percentage of marks for Student A: ${studentA.getPercentage}%")
println(s"Percentage of marks for Student B: ${studentB.getPercentage}%")

```

```
Percentage of marks for Student A: 81.66666666666667%
Percentage of marks for Student B: 85.75%
```

3. Create an abstract class 'Bank' with an abstract method 'getBalance'. $100, $150 and $200 are deposited in banks A, B and C respectively. 'BankA', 'BankB' and 'BankC' are subclasses of class 'Bank', each having a method named 'getBalance'. Call this method by creating an object of each of the three classes.

```
trait Bank {
  def getBalance: Double
}

class BankA extends Bank {
  private val balance: Double = 100.0
  def getBalance: Double = balance
}

class BankB extends Bank {
  private val balance: Double = 150.0
  def getBalance: Double = balance
}

class BankC extends Bank {
  private val balance: Double = 200.0
  def getBalance: Double = balance
}

val bankA = new BankA
val bankB = new BankB
val bankC = new BankC

println(s"Balance in Bank A: ${bankA.getBalance}")
println(s"Balance in Bank B: ${bankB.getBalance}")
println(s"Balance in Bank C: ${bankC.getBalance}")
```

```
Balance in Bank A: 100.0
Balance in Bank B: 150.0
Balance in Bank C: 200.0
```

4. Create an abstract class 'Animals' with two abstract methods 'cats' and 'dogs'. Now create a class 'Cats' with a method 'cats' which prints "Cats meow" and a class 'Dogs' with a method 'dogs' which prints "Dogs bark", both inheriting the class 'Animals'. Now create an object for each of the subclasses and call their respective methods.

```
trait Animals {
  def cats: Unit
  def dogs: Unit
}

class Cats extends Animals {
  def cats: Unit = println("Cats meow")
  def dogs: Unit = ()
}

class Dogs extends Animals {
  def cats: Unit = ()
  def dogs: Unit = println("Dogs bark")
}

val catsObj = new Cats
val dogsObj = new Dogs

catsObj.cats
catsObj.dogs

dogsObj.cats
dogsObj.dogs
```

```
Cats meow
Dogs bark
```

5. We have to calculate the area of a rectangle, a square and a circle. Create an abstract class 'Shape' with three abstract methods namely 'RectangleArea' taking two parameters, 'SquareArea' and 'CircleArea' taking one parameter each. The parameters of 'RectangleArea' are its length and breadth, that of 'SquareArea' is its side and that of 'CircleArea' is its radius. Now create another class 'Area' containing all the three methods 'RectangleArea', 'SquareArea' and 'CircleArea' for printing the area of rectangle, square and circle respectively. Create an object of class 'Area' and call all the three methods.

```
trait Rectangle {
  def RectangleArea(length: Double, breadth: Double): Double = length * breadth
}

trait Square {
  def SquareArea(side: Double): Double = side * side
}

trait Circle {
  def CircleArea(radius: Double): Double = Math.PI * radius * radius
}

class Area extends Rectangle with Square with Circle

val areaObj = new Area

val rectangleArea = areaObj.RectangleArea(5.0, 8.0)
val squareArea = areaObj.SquareArea(4.0)
val circleArea = areaObj.CircleArea(3.0)

println(s"Area of Rectangle: $rectangleArea")
println(s"Area of Square: $squareArea")
println(s"Area of Circle: $circleArea")
```

```
Area of Rectangle: 40.0
Area of Square: 16.0
Area of Circle: 28.274333882308138
```