Task 1

Write a simple program to show inheritance in scala.

Inheritance.scala

```
package Assignment9

abstract class Bike{
  def run()
  def display(){
    println("Display function in Bike Class")
  }
}

class Hero extends Bike{
  def run(){
    println("Hero Bike is running!")
  }
}

object Inheritance {
  def main(args:Array[String]){
    var heroObj = new Hero()
    heroObj.run()
    heroObj.display()
  }
}
```

Problems Tasks Console

```
<terminated> Inheritance$ [Scala Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (Sep 18, 2018, 3:00:42 PM)
Hero Bike is running!
Display function in Bike Class
```

Task 2

Write a simple program to show multiple inheritance in scala

Multiple_Inheritance.scala

```scala
package Assignment9

abstract class Animal{
  val legs:Int
  val noise:String
  def makeNoise() = println(noise)
}
trait Quadriped{
  self:Animal =>
    val legs = 4
}

class Dog extends Animal with Quadriped{
  val noise = "Bow Bow"
  override def makeNoise() = println(noise + " " + noise)
}

object Multiple_Inheritance {
  def main(args:Array[String]){
    var dogObj = new Dog()
    dogObj.makeNoise()
  }
}
```

Problems Tasks Console

```
<terminated> Multiple_Inheritance$ [Scala Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (Sep 18, 2018, 3:13:33 PM)
Bow Bow Bow Bow
```

Task 3

Write a partial function to add three numbers in which one number is constant and two numbers can be passed as inputs and define another method which can take the partial function as input and squares the result.

PartialFunction.scala

```scala
package Assignment9

class PartialClass{
  def squareFun(x: Int) = println("Square = "+ x*x)
  def addNum(x:Int, y:Int, z:Int) = x+y+z
  val sum=addNum(5,_:Int,_:Int)
  def partialFun(x:Int, y:Int): Unit = {
    println("Addition of two numbers is = "+sum(x,y))
    squareFun(sum(x,y))
  }
}

object PartialFunction {
  def main(args:Array[String]): Unit = {
    print("Enter the value of x : ")
    var x:Int = readInt()
    print("Enter the value of y : ")
    var y:Int = readInt()
    var obj = new PartialClass()
    obj.partialFun(x,y)

  }
}
```

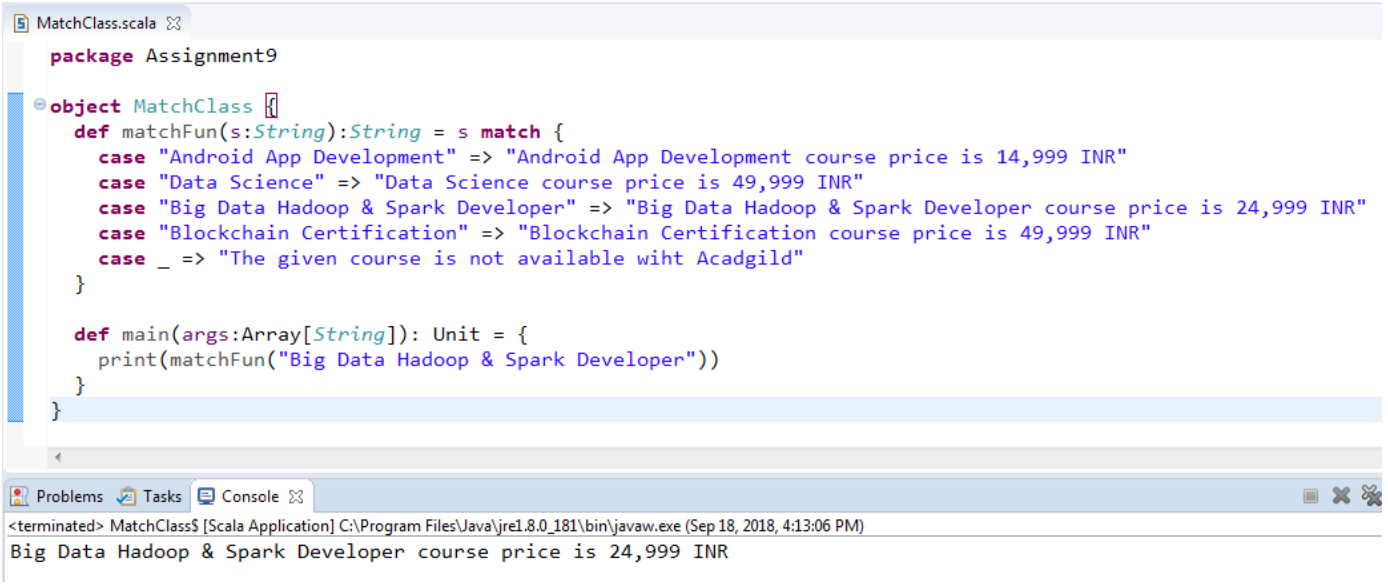
Problems Tasks Console

```
<terminated> PartialFunction$ [Scala Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (Sep 18, 2018, 4:02:41 PM)
Enter the value of x : 10
Enter the value of y : 20
Addition of two numbers is = 35
Square = 1225
```

Task 4

Write a program to print the prices of 4 courses of Acadgild:
Android App Development -14,999 INR
Data Science - 49,999 INR
Big Data Hadoop & Spark Developer – 24,999 INR
Blockchain Certification – 49,999 INR

Using match and add a default condition if the user enters any other course



Task 5

Create a calculator to work with rational numbers.
Requirements:

➢It should provide capability to add, subtract, divide and multiply rational Numbers

➢Create a method to compute GCD (this will come in handy during operations on rational)

Add option to work with whole numbers which are also rational numbers i.e. (n/1)

➢achieve the above using auxiliary constructors

Enable method overloading to enable each function to work with numbers and rational

CalObj.scala

```scala
package Assignment8

class Calc(n:Int, d:Int){
  val g = gcd(n.abs,d.abs)
  val num = n/g
  val den = d/g

  def gcd(a:Int, b:Int) : Int = {
    if (a == 0 || b == 0)
       return 0
    if (a == b)
        return a
    if (a > b)
        return gcd(a-b, b)
    return gcd(a, b-a)
  }

  def this(n:Int) = this(n,1)

  def add(r:Calc): Calc = new Calc(num * r.den + r.num * den, den * r.den)
  def add(i:Int): Calc = new Calc(num + i * den, den)

  def sub(r:Calc): Calc = new Calc(num * r.den - r.num * den, den * r.den)
  def sub(i:Int): Calc = new Calc(num - i * den, den)

  def mul(r:Calc): Calc = new Calc(num * r.num, den * r.den)
  def mul(i:Int): Calc = new Calc(num * i, den)

  def div(r:Calc): Calc = new Calc(num * r.den , den * r.num)
  def div(i:Int): Calc = new Calc(num, den * i)

  def display() : Unit = println(num + "/" + den)
}
```

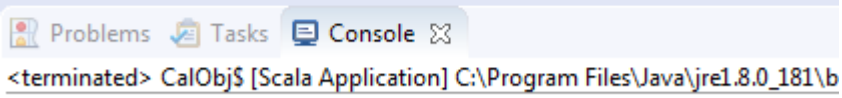
```scala
object CalObj {
  def main(args: Array[String]){
    val n = new Calc(10,20)

    val addVar = n add 5
    addVar.display()

    val subVar = n sub new Calc(5,30)
    subVar.display()

    val mulVar = n mul new Calc(5,4)
    mulVar.display()

    val divVar = n div 50
    divVar.display()
  }
}
```




```
11/2
1/3
5/8
1/100
```

Task 6

Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- find the sum of all numbers
- find the total elements in the list
- calculate the average of the numbers in the list
- find the sum of all the even numbers in the list
- find the total number of elements in the list divisible by both 5 and 3

ListExample.scala

```
package Assignment9

object ListExample {
  def main(args:Array[String]): Unit = {
    var intList:List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11,12,13,14,15)
    //find the sum of all numbers
    println("Sum of the numbers in the list using built in function 'sum' is: "+intList.sum)
    var sum:Int = 0
    for(n<-intList){
      sum += n
    }
    println("Sum of the numbers in the list using traversing the list is: "+sum)

    //find the total elements in the list
    println("Length of the list using built in function 'length' is: "+intList.length)
    var count:Int = 0
    for(n <- intList)
      count += 1
    println("Length of the list using traversing the list is: "+count)

    //calculate the average of the numbers in the list
    println("Average of the numbers present in the list using builtin functions 'sum' and 'length' is: "+intList.sum / intList.length)
    var avg:Int = 0
    var len:Int = 0
    sum = 0
    for(n<-intList){
      sum+=n
      len+=1
    }
    avg = sum/len
    println("Average of the numbers present in the list using traversing is: "+avg)

    //find the sum of all the even numbers in the list
    sum=0
    for(n<-intList){
      if(n % 2 == 0)
        sum+=n
    }
    println("Sum of all the even numbers present in the list is "+sum)

    //find the total number of elements in the list divisible by both 5 and 3
    count = 0
    for(n<-intList){
      if((n % 5 == 0) && (n % 3 == 0)){
        println(n+" is divisible by both 5 and 3")
        count += 1
      }
    }
    println("Number of digits that are divisible by both 3 and 5 is/are: "+count)
  }
}
```

Problems Tasks Console

```
<terminated> ListExample$ [Scala Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (Sep 18, 2018, 5:01:17 PM)
Sum of the numbers in the list using built in function is: 120
Sum of the numbers in the list using traversing the list is: 120
Length of the list using built in function is: 15
Length of the list using traversing the list is: 15
Average of the numbers present in the list using builtin functions is: 8
Average of the numbers present in the list using traversing is: 8
Sum of all the even numbers present in the list is 56
15 is divisible by both 5 and 3
Number of digits that are divisible by both 3 and 5 is/are: 1
```

Task 7

1) Pen down the limitations of MapReduce.

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

- It's based on disk computing
- Suitable for single pass computations - not iterative computations
- Needs a sequence of MR jobs to run iterative tasks,
- Needs integration with several other frameworks/tools to solve BigData use cases, like Hive, Hbase, Flume, Sqoop, PIG
- Hadoop Map Reduce supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum.
- Slow Processing Speed,
- No Real-time Data Processing
- Lengthy Line of Code and

- MapReduce only ensures that data job is complete, but it's unable to guarantee when the job will be complete.

2) What is RDD? Explain few features of RDD?

RDD stands for **Resilient Distributed Datasets** are Apache Spark's data abstraction; RDD is a logical reference of a dataset which is partitioned across many server machines in the cluster. RDDs are **Immutable** and are self-recovered in case of failure. Dataset could be the data loaded externally by the user. RDDs can only be created by reading data from a stable storage such as HDFS or by transformations on existing RDDs.

Why RDD?

When it comes to iterative distributed computing, i.e. processing data over multiple jobs in computations such as Logistic Regression, K-means clustering, and Page rank algorithms, it is fairly common to reuse or share the data among multiple jobs or you may want to do multiple ad-hoc queries over a shared data set.

Few features of RDD:

- **In-memory computation**

The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes.

- **Lazy Evaluation**

The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do.

- **Fault Tolerance**

Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data

- **Partitioning**

RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

3) List down few Spark RDD operations and explain each of them.

Apache Spark RDD supports two types of Operations:

- ✓ **Transformations**
- ✓ **Actions**

RDD Transformation

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature. Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as **RDD operator graph** or **RDD dependency graph**. It is a logical execution plan i.e., it is **Directed Acyclic Graph (DAG)** of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a **map(), filter().**
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. **flatMap(), union(), Cartesian()**) or the same size (e.g. map).

There are two types of transformations:

- ***Narrow transformation*** – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of **map(), filter().**
- ***Wide transformation*** – In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. Wide transformations are the result of **groupbyKey() and reducebyKey().**

RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give **non-RDD** values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

count(), collect(), take(n), top(), countByValue(), reduce(),fold(), aggregate() and foreach().

Thank you,