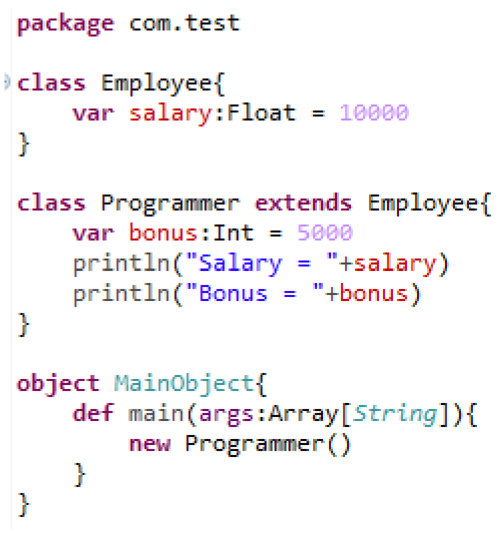
Big Data – Hadoop – Scala and Spark-Assignment 6

Task 1

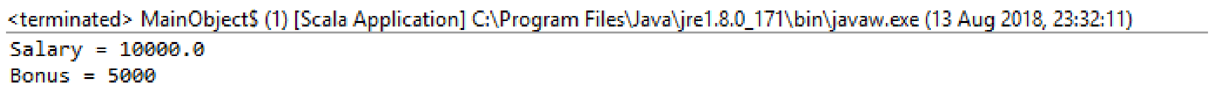
Write a simple program to show inheritance in scala.

Inheritance is an object-oriented concept which is used to reusability of code. To achieve inheritance a class must extend to other class using extends keyword. A class which is extended called super or parent class. A class which extends class is called derived or base class.



```
package com.test

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 :

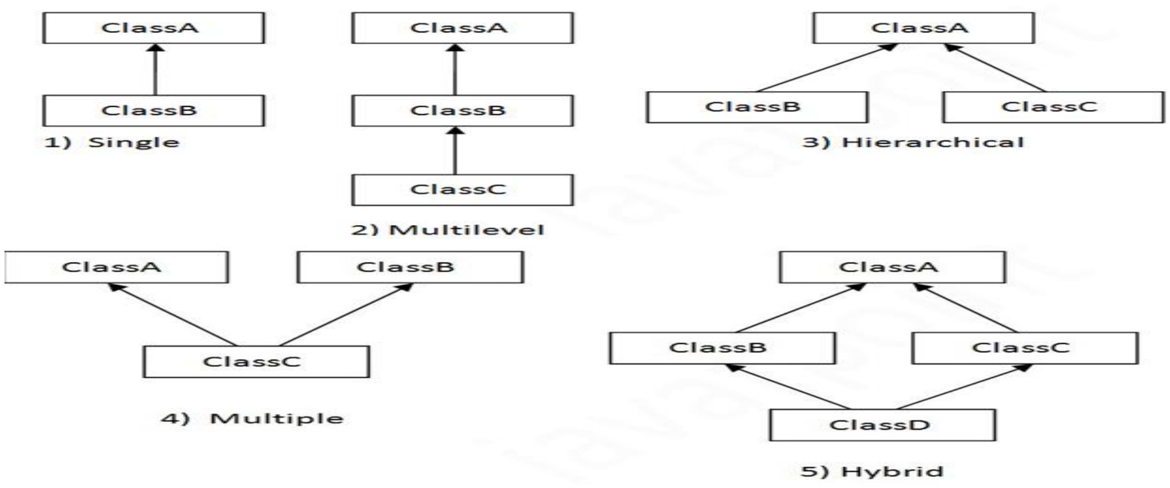


```
<terminated> MainObject$ (1) [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (13 Aug 2018, 23:32:11)
Salary = 10000.0
Bonus = 5000
```

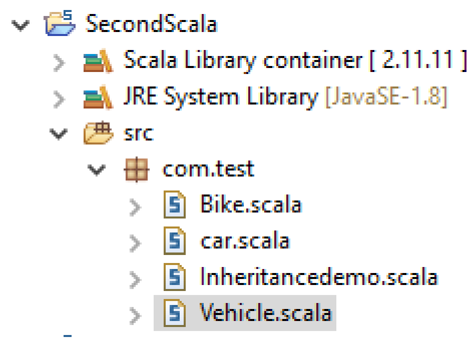
Big Data – Hadoop – Scala and Spark-Assignment 6

Task 2

Write a simple program to show multiple inheritance in scala

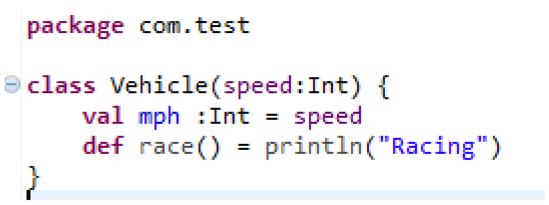


Step 1: Creating a project to show the inheritance in scala



```
SecondScala
  Scala Library container [ 2.11.11 ]
  JRE System Library [JavaSE-1.8]
  src
    com.test
      Bike.scala
      car.scala
      Inheritancedemo.scala
      Vehicle.scala
```

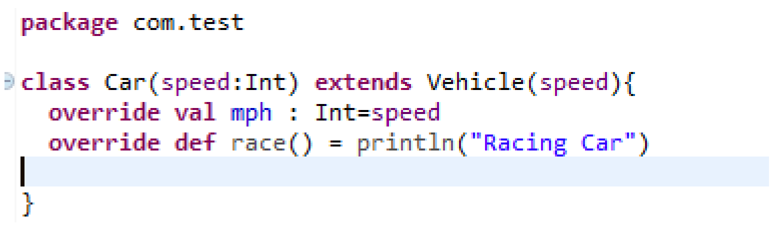
Step 2: Creating a Parent class called Vehicle.



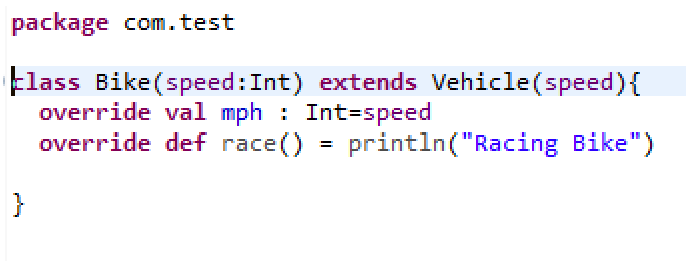
```
package com.test

class Vehicle(speed:Int) {
     val mph :Int = speed
     def race() = println("Racing")
}
```

Step 3 : Creating Child classes Car and Bike that inherit the Vehicle class.

We can override the mph and race() methods accordingly.

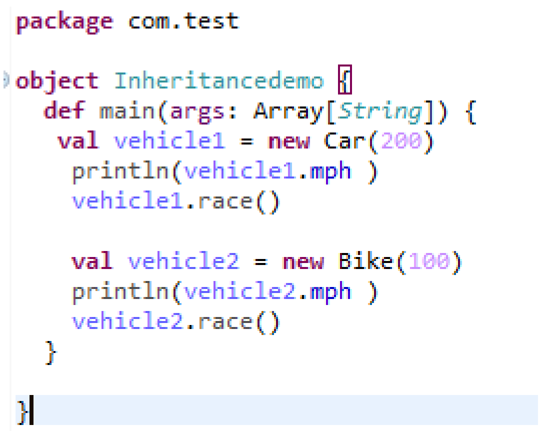


```
package com.test

class Car(speed:Int) extends Vehicle(speed){
  override val mph : Int=speed
  override def race() = println("Racing Car")

}
```



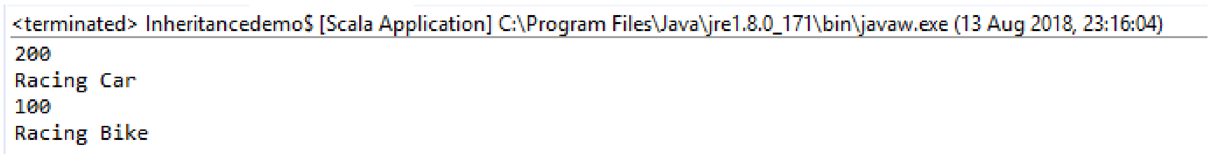
```
package com.test

class Bike(speed:Int) extends Vehicle(speed){
  override val mph : Int=speed
  override def race() = println("Racing Bike")

}
```

Step 4 : Inside the main method, creating a Car object and a Bike object and then access their property mph and method race.



```
package com.test

object Inheritancedemo {
  def main(args: Array[String]) {
   val vehicle1 = new Car(200)
    println(vehicle1.mph )
    vehicle1.race()

    val vehicle2 = new Bike(100)
    println(vehicle2.mph )
    vehicle2.race()
  }

}
```

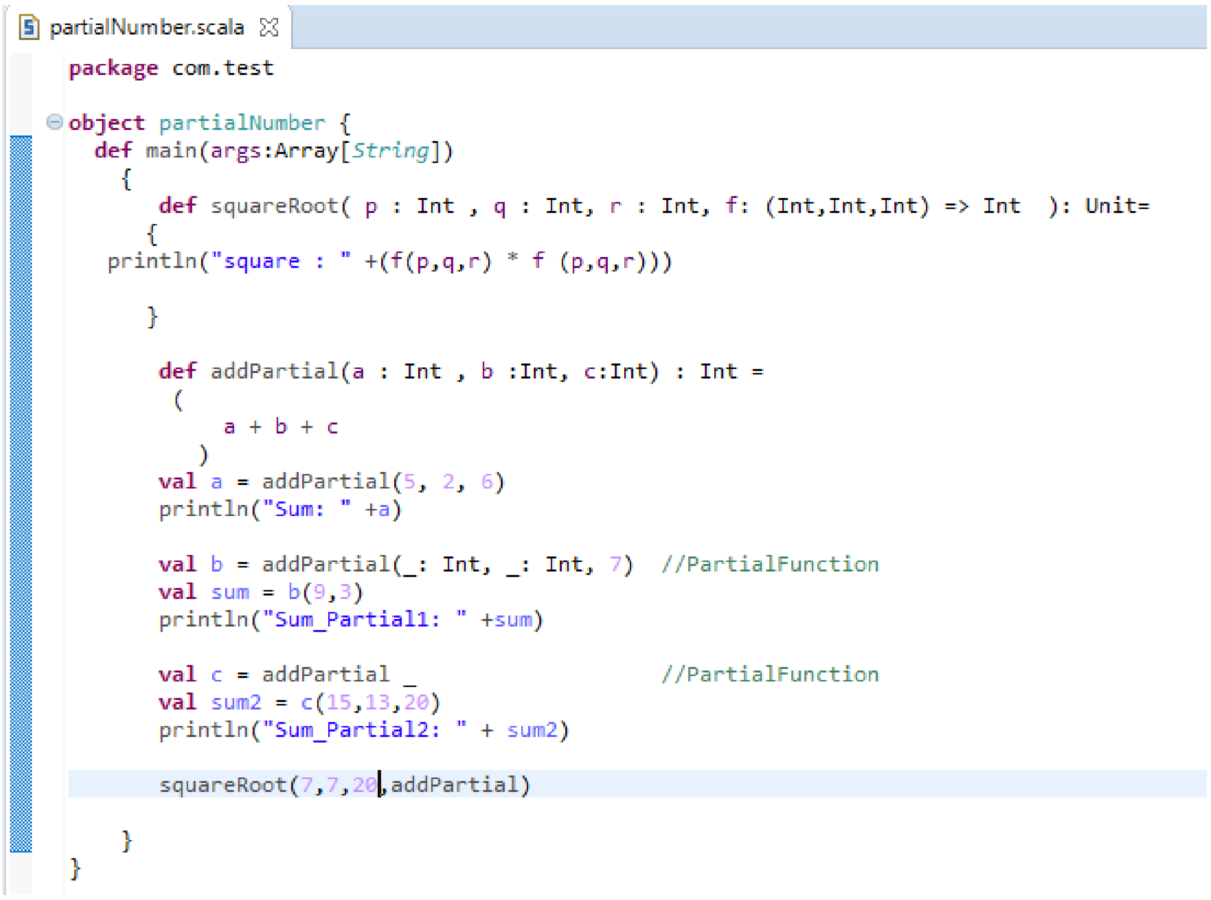
<u>OUTPUT:</u>



```
<terminated> Inheritancedemo$ [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (13 Aug 2018, 23:16:04)
200
Racing Car
100
Racing Bike
```

Task 3

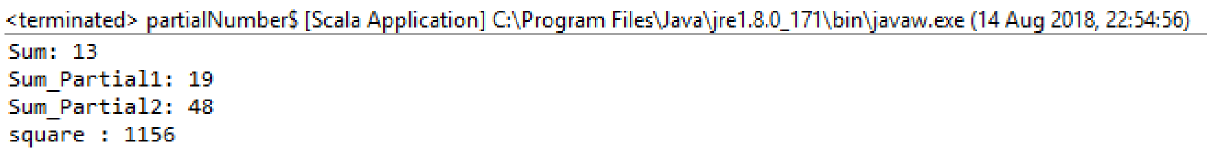
Write a partial function to add three numbers in which one number is constant and two numbers can be passed as inputs and define another method which can take the partial function as input and squares the result.



partialNumber.scala

```
package com.test

object partialNumber {
  def main(args:Array[String])
   {
      def squareRoot( p : Int , q : Int, r : Int, f: (Int,Int,Int) => Int  ): Unit=
     {
   println("square : " +(f(p,q,r) * f (p,q,r)))

     }

      def addPartial(a : Int , b :Int, c:Int) : Int =
       (
           a + b + c
         )
      val a = addPartial(5, 2, 6)
      println("Sum: " +a)

      val b = addPartial(_: Int, _: Int, 7)  //PartialFunction
      val sum = b(9,3)
      println("Sum_Partial1: " +sum)

      val c = addPartial _                    //PartialFunction
      val sum2 = c(15,13,20)
      println("Sum_Partial2: " + sum2)

      squareRoot(7,7,20,addPartial)

   }
}
```

OUTPUT :



```
<terminated> partialNumber$ [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (14 Aug 2018, 22:54:56)
Sum: 13
Sum_Partial1: 19
Sum_Partial2: 48
square : 1156
```

Big Data – Hadoop – Scala and Spark-Assignment 6

Task 4

Write a program to print the prices of 4 courses of Acadgild:

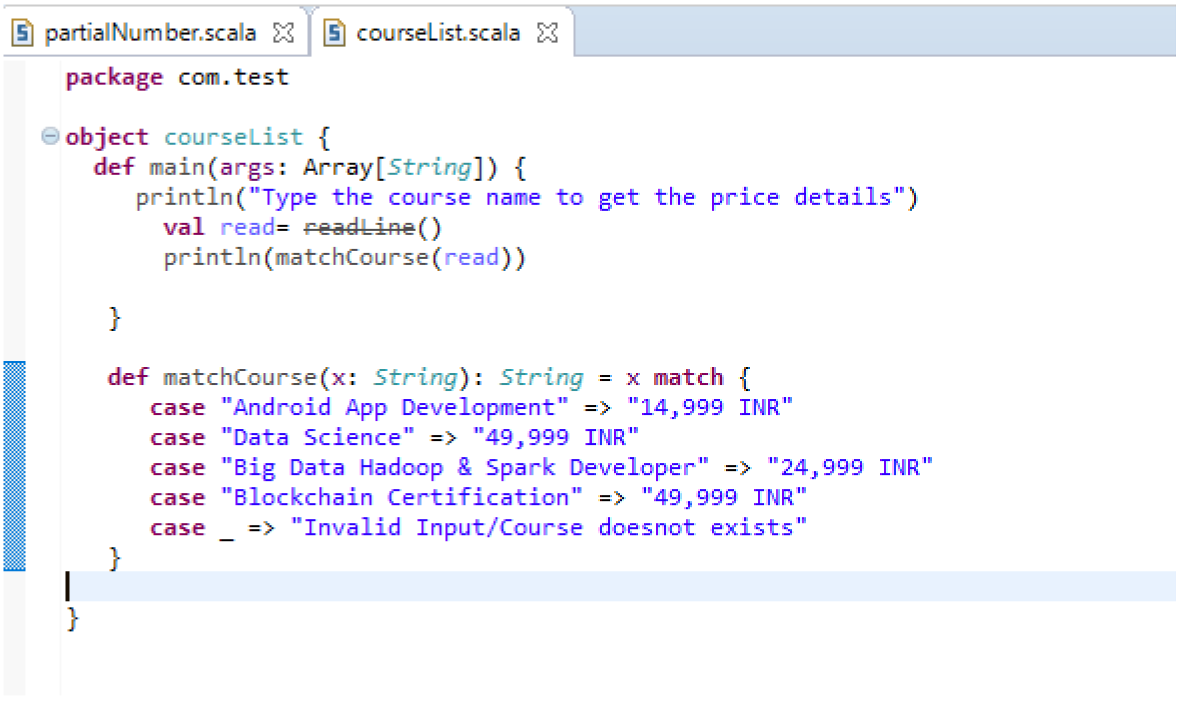
Android App Development -14,999 INR

Data Science - 49,999 INR

Big Data Hadoop & Spark Developer – 24,999 INR

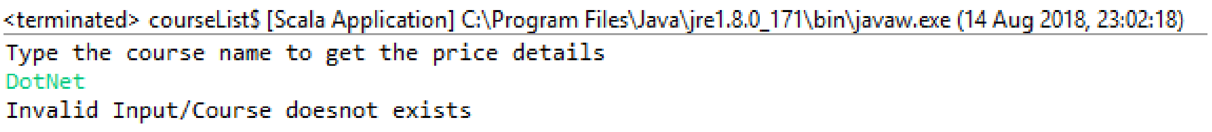
Blockchain Certification – 49,999 INR

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

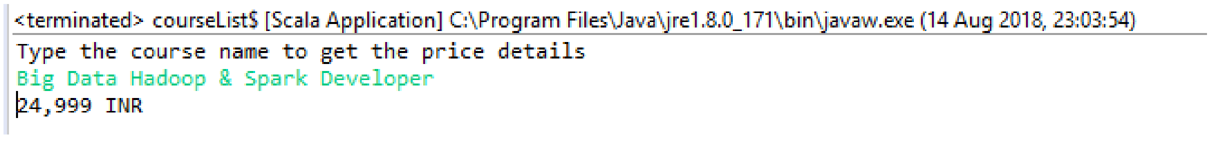

partialNumber.scala courseList.scala

```
package com.test

object courseList {
  def main(args: Array[String]) {
     println("Type the course name to get the price details")
       val read= ~~readLine~~()
       println(matchCourse(read))

    }

    def matchCourse(x: String): String = x match {
       case "Android App Development" => "14,999 INR"
       case "Data Science" => "49,999 INR"
       case "Big Data Hadoop & Spark Developer" => "24,999 INR"
       case "Blockchain Certification" => "49,999 INR"
       case _ => "Invalid Input/Course doesnot exists"
    }

}
```

OUTPUT – Selection of invalid course


```
<terminated> courseList$ [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (14 Aug 2018, 23:02:18)
Type the course name to get the price details
DotNet
Invalid Input/Course doesnot exists
```

OUTPUT – Selection of available course


```
<terminated> courseList$ [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (14 Aug 2018, 23:03:54)
Type the course name to get the price details
Big Data Hadoop & Spark Developer
24,999 INR
```

Big Data – Hadoop – Scala and Spark-Assignment 6

Task 5

Create a calculator to work with rational numbers.

Requirements:

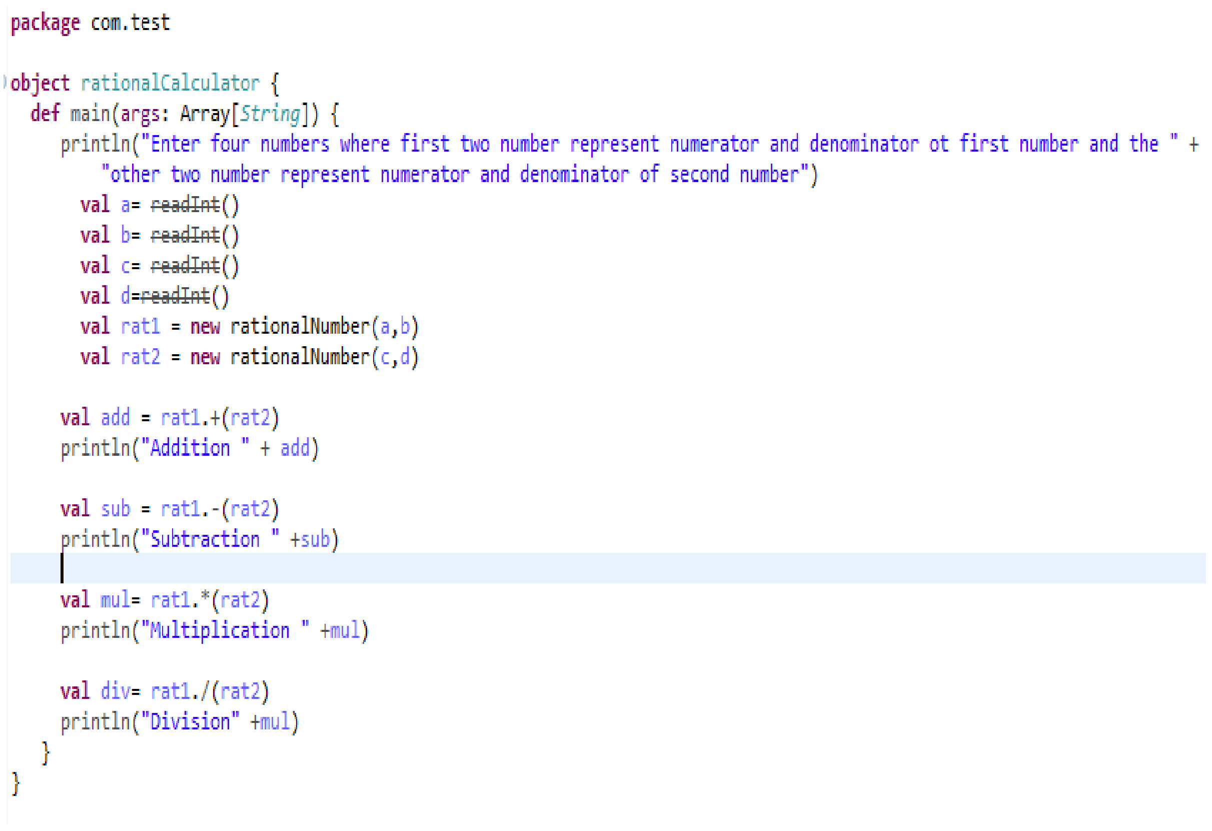
➢ **It should provide capability to add, subtract, divide and multiply rational Numbers**

➢ **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**

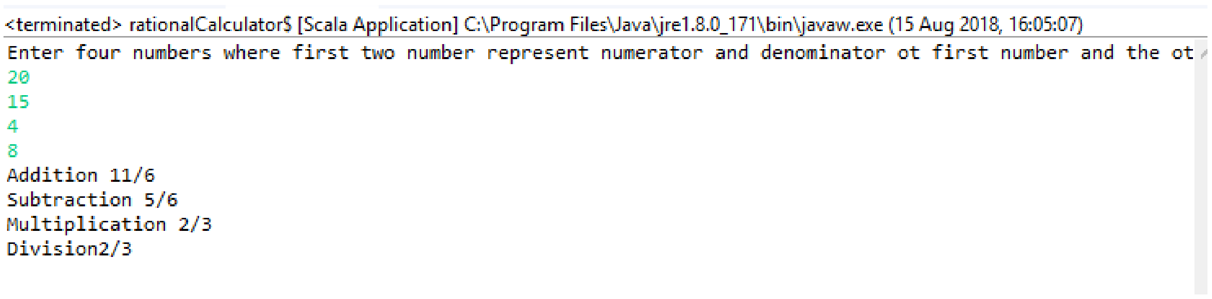
enable method overloading to enable each function to work with numbers and rational



```
package com.test

object rationalCalculator {
  def main(args: Array[String]) {
     println("Enter four numbers where first two number represent numerator and denominator ot first number and the " +
         "other two number represent numerator and denominator of second number")
       val a= ~~readInt~~()
       val b= ~~readInt~~()
       val c= ~~readInt~~()
       val d=~~readInt~~()
       val rat1 = new rationalNumber(a,b)
       val rat2 = new rationalNumber(c,d)

     val add = rat1.+(rat2)
     println("Addition " + add)

     val sub = rat1.-(rat2)
     println("Subtraction " +sub)

     val mul= rat1.*(rat2)
     println("Multiplication " +mul)

     val div= rat1./(rat2)
     println("Division" +mul)
   }
}
```

```scala
val numer = n / g
val denom = d / g

def + (that: rationalNumber): rationalNumber =
  new rationalNumber(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )

 def + (i: Int): rationalNumber = {
new rationalNumber(numer + i * denom, denom)
}

  def - (that: rationalNumber): rationalNumber =
  new rationalNumber(
    numer * that.denom - that.numer * denom,
    denom * that.denom
  )

def - (i: Int):rationalNumber =
  new rationalNumber(numer - i * denom, denom)

def * (that: rationalNumber): rationalNumber =
  new rationalNumber(numer * that.numer, denom * that.denom)

def * (i: Int):rationalNumber=
  new rationalNumber(numer * i, denom)

def / (that: rationalNumber): rationalNumber =
  new rationalNumber(numer * that.denom, denom * that.numer)

def / (i: Int): rationalNumber =
  new rationalNumber(numer, denom * i)

override def toString = numer +"/"+ denom

 def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)
```

OUTPUT :

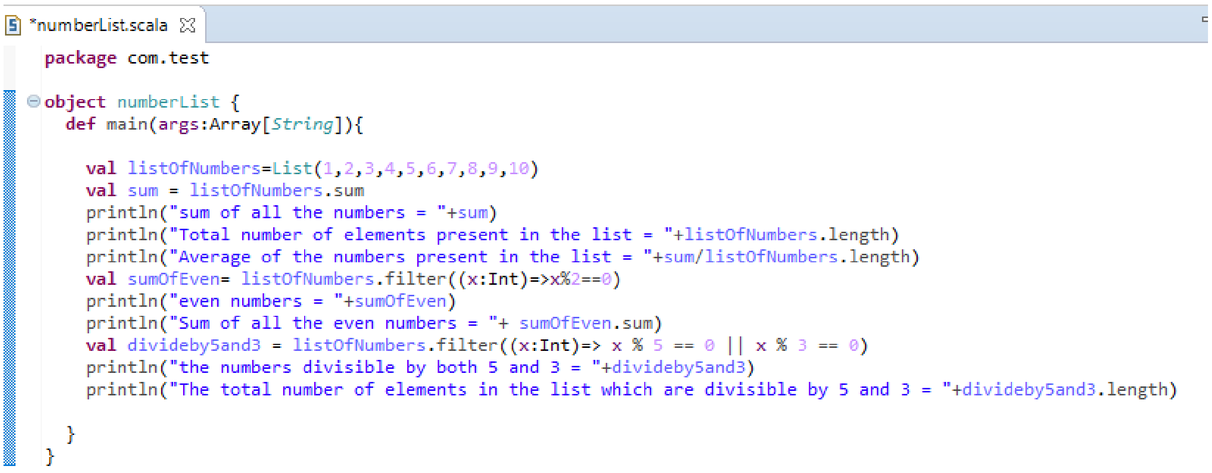


```
<terminated> rationalCalculator$ [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (15 Aug 2018, 16:05:07)
Enter four numbers where first two number represent numerator and denominator ot first number and the ot
20
15
4
8
Addition 11/6
Subtraction 5/6
Multiplication 2/3
Division2/3
```

Big Data – Hadoop – Scala and Spark-Assignment 6

Task 6

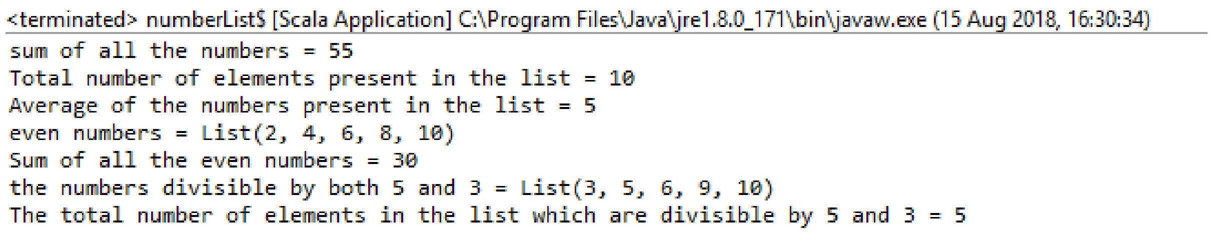
Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
find the sum of all numbers
find the total elements in the list
calculate the average of the numbers in the list
find the sum of all the even numbers in the list
find the total number of elements in the list divisible by both 5 and 3



*numberList.scala

```
package com.test

object numberList {
  def main(args:Array[String]){

    val listOfNumbers=List(1,2,3,4,5,6,7,8,9,10)
    val sum = listOfNumbers.sum
    println("sum of all the numbers = "+sum)
    println("Total number of elements present in the list = "+listOfNumbers.length)
    println("Average of the numbers present in the list = "+sum/listOfNumbers.length)
    val sumOfEven= listOfNumbers.filter((x:Int)=>x%2==0)
    println("even numbers = "+sumOfEven)
    println("Sum of all the even numbers = "+ sumOfEven.sum)
    val divideby5and3 = listOfNumbers.filter((x:Int)=> x % 5 == 0 || x % 3 == 0)
    println("the numbers divisible by both 5 and 3 = "+divideby5and3)
    println("The total number of elements in the list which are divisible by 5 and 3 = "+divideby5and3.length)

  }
}
```

OUTPUT :



```
<terminated> numberList$ [Scala Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (15 Aug 2018, 16:30:34)
sum of all the numbers = 55
Total number of elements present in the list = 10
Average of the numbers present in the list = 5
even numbers = List(2, 4, 6, 8, 10)
Sum of all the even numbers = 30
the numbers divisible by both 5 and 3 = List(3, 5, 6, 9, 10)
The total number of elements in the list which are divisible by 5 and 3 = 5
```

Big Data – Hadoop – Scala and Spark-Assignment 6

Task 7

1) Pen down the limitations of MapReduce.

MapReduce cannot handle:

1. Interactive Processing
2. Real-time (stream) Processing
3. Iterative (delta) Processing
4. In-memory Processing
5. Graph Processing

1. Issue with Small Files

Hadoop is not suited for small data. Hadoop distributed file system lacks the ability to efficiently support the random reading of small files because of its high capacity design.

2. Slow Processing Speed

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: Map and Reduce and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

3. Support for Batch Processing only

Hadoop 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.

4. No Real-time Data Processing

Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

5. No Delta Iteration

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow (i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

6. Latency

In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In MapReduce, Map takes a set of data and converts it into another set of data, where individual element is broken down into key value pair and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

7. Not Easy to Use

In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as hive and pig makes working with MapReduce a little easier for adopters.

8. No Caching

Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

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

RDD (Resilient Distributed Dataset) is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

Decomposing the name RDD:

- Resilient, i.e. fault-tolerant with the help of RDD lineage graph (DAG) and so able to recompute missing or damaged partitions due to node failures.
- Distributed, since Data resides on multiple nodes.
- Dataset represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Hence, each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. It possess self-recovery in the case of failure.

There are several advantages of using RDD. Some of them are-

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.

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.

Immutability

RDDS are immutable in nature meaning once we create an RDD we cannot manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

Persistence

We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function.

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.

Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAG Scheduler** places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation.

Coarse-grained Operation

We apply coarse-grained transformations to RDD**.** Coarse-grained meaning the operation applies to the whole dataset not on an individual element in the data set of RDD.

Typed

We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

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

Apache Spark RDD supports two types of Operations-

- Transformations
- Actions

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. They get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().

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().

Map

The map function iterates over every line in RDD and split into new RDD.
Using **map()** transformation we take in any function, and that function is applied to every element of RDD.

In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.

For example, in RDD {1, 2, 3, 4, 5} if we apply “rdd.map(x=>x+2)” we will get the result as (3, 4, 5, 6, 7).

flatMap()

With the help of flatMap() function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.

Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

```
val data = spark.read.textFile("spark_test.txt").rdd
val flatmapFile = data.flatMap(lines => lines.split(" "))
flatmapFile.foreach(println)
```

filter()

Spark RDD **filter()** function returns a new RDD, containing only the elements that meet a predicate. It is a *narrow operation* because it does not shuffle data from one partition to many partitions.
For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

sortByKey()

When we apply the sortByKey() function on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.

sortByKey() example:

Big Data – Hadoop – Scala and Spark-Assignment 6

```
val data = spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82))
val sorted = data.sortByKey()
sorted.foreach(println)
```

union(dataset)

With the union() function, we get the elements of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

For example, the elements of RDD1 are (Spark, Spark, Hadoop, Flink) and that ofRDD2 are (Big data, Spark, Flink) so the resultant rdd1.union(rdd2) will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

RRD 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()

Action **count()** returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.count()" will give the result 8.

Count() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())
```

take(n)

The action **take(n)** returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.
For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "take (4)" will give result { 2, 2, 3, 4}
Take() example:

```
val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
val twoRec = result.take(2)
twoRec.foreach(println)
```

reduce()

The **reduce()** function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.

Reduce() example:

```
val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
val sum = rdd1.reduce(_+_)
println(sum)
```

aggregate ()

It gives us the flexibility to get data type different from the input type. The **aggregate ()** takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the *driver*. In this case, **foreach()** function is useful. For example, inserting a record into the database.

Foreach() example:

```
val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)
```