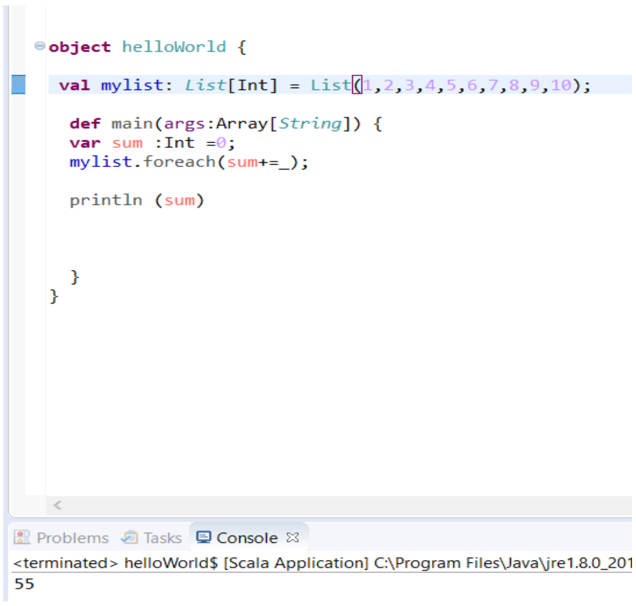
Task 1

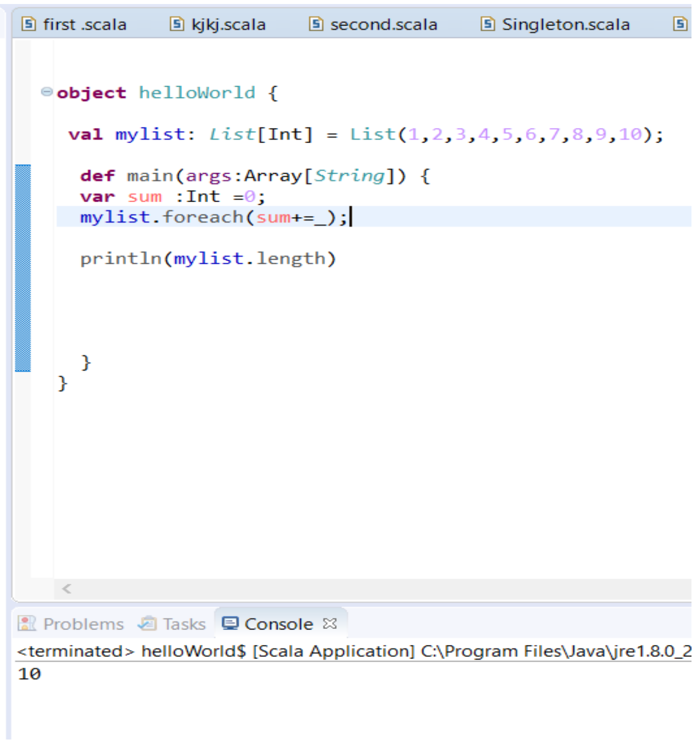
a



```
object helloWorld {

 val mylist: List[Int] = List(1,2,3,4,5,6,7,8,9,10);

  def main(args:Array[String]) {
  var sum :Int =0;
  mylist.foreach(sum+=_);

  println (sum)


  }
}
```

```
<terminated> helloWorld$ [Scala Application] C:\Program Files\Java\jre1.8.0_201
55
```

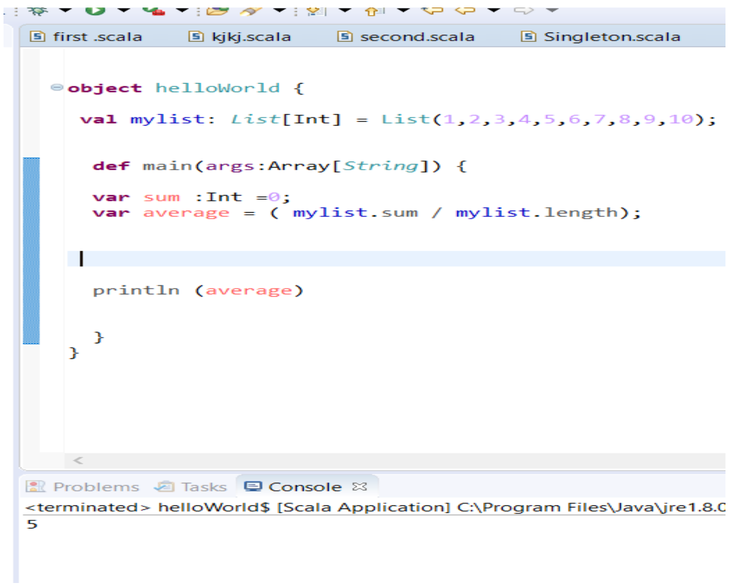
b.



```
object helloWorld {

 val mylist: List[Int] = List(1,2,3,4,5,6,7,8,9,10);

  def main(args:Array[String]) {
  var sum :Int =0;
  mylist.foreach(sum+=_);

  println(mylist.length)



  }
}
```

```
<terminated> helloWorld$ [Scala Application] C:\Program Files\Java\jre1.8.0_2
10
```

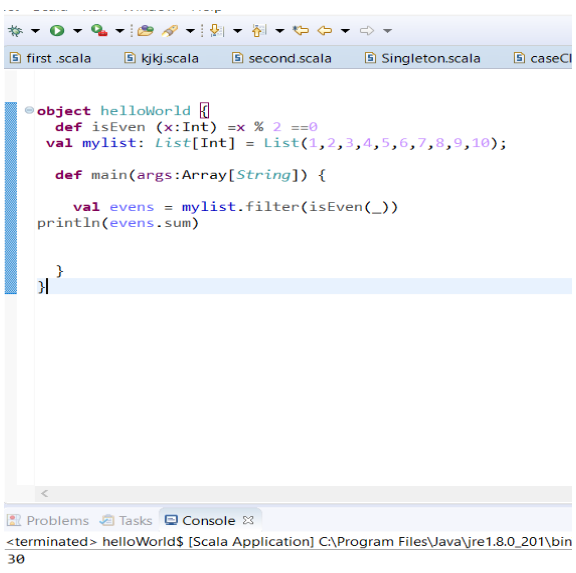
c.



```
object helloWorld {
  val mylist: List[Int] = List(1,2,3,4,5,6,7,8,9,10);

   def main(args:Array[String]) {

   var sum :Int =0;
   var average = ( mylist.sum / mylist.length);



   println (average)

   }
}
```

```
<terminated> helloWorld$ [Scala Application] C:\Program Files\Java\jre1.8.0
5
```

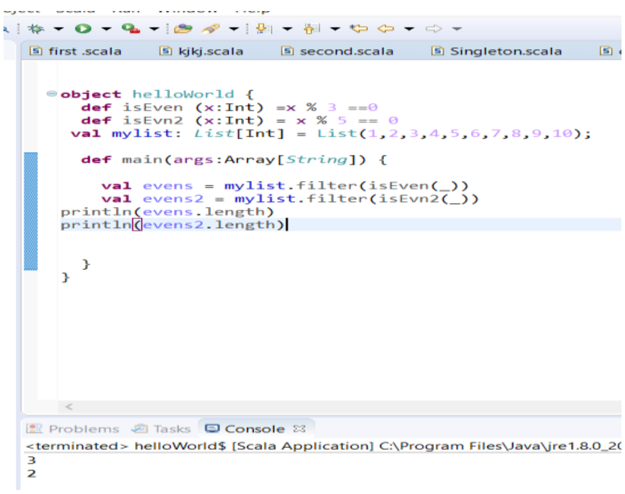
d.



```
object helloWorld {
   def isEven (x:Int) =x % 2 ==0
  val mylist: List[Int] = List(1,2,3,4,5,6,7,8,9,10);

   def main(args:Array[String]) {

     val evens = mylist.filter(isEven(_))
println(evens.sum)


   }
}
```

```
<terminated> helloWorld$ [Scala Application] C:\Program Files\Java\jre1.8.0_201\bin
30
```

e.



```
object helloWorld {
   def isEven (x:Int) =x % 3 ==0
   def isEvn2 (x:Int) = x % 5 == 0
  val mylist: List[Int] = List(1,2,3,4,5,6,7,8,9,10);

   def main(args:Array[String]) {

     val evens = mylist.filter(isEven(_))
     val evens2 = mylist.filter(isEvn2(_))
println(evens.length)
println(evens2.length)


   }
}
```

```
<terminated> helloWorld$ [Scala Application] C:\Program Files\Java\jre1.8.0_2
3
2
```

Task 2

a. **Limitations of Hadoop**
    - **Issue with small files**- Hadoop is not suited for small files. Small files are the major problems in HDFS. A small file is significantly smaller than the HDFS Blocksize (default 128MB). If you are storing these large number of small files, HDFS can't handle these lots of files. As HDFS works with a small number of large files for storing data sets rather than larger number of small files. If one use the huge number of small files, then this will overload the namenode. Since namenode stores the namespace of HDFS.HAR files, Sequence files, and Hbase overcome small files issues.
    - **Processing Speed**- With parallel and distributed algorithm, MapReduce process large data sets. MapReduce performs the task: Map and Reduce. MapReduce requires a lot of time to perform these tasks thereby increasing latency. As data is distributed and processed over the cluster in MapReduce. So, it will increase the time and reduces processing speed.
    - **Support only Batch Processing**- Hadoop supports only batch processing. It does not process streamed data and hence, overall performance is slower. MapReduce framework does not leverage the memory of the cluster to the maximum.
    - **Iterative Processing-** Hadoop is not efficient for iterative processing. As hadoop does not support cyclic data flow. That is the chain of stages in which the input to the next stage is the output from the previous stage

b. **What is RDD and few features of RDD**

(Resilient Distributed Datasets) are a basic abstraction of spark which is immutable. These are logically partitioned that we can also apply parallel operations on them.

Features

- In-memory – means operation of information is done in the main random access memory
- Lazy evaluations- means to trigger an execution an action is a must since execution process doesn't start instantly
- Immutable and Read-only- they are unchanged overtime
- Partitioned – each dataset is logically partitioned and distributed across nodes over the cluster

c. **A few RDD operations**
- Map()- is a function that iterates over every line in RDD and splits into new RDD
- FlatMap()- splits each input string into words
- Filter()- returns a new RDD, containing only elemets that meet a predicate
- Union() – gets the elements of both the RDD in new RDD
- Distinct()- returns a new dataset that contains the distinct elements of the source dataset