Task 1

Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

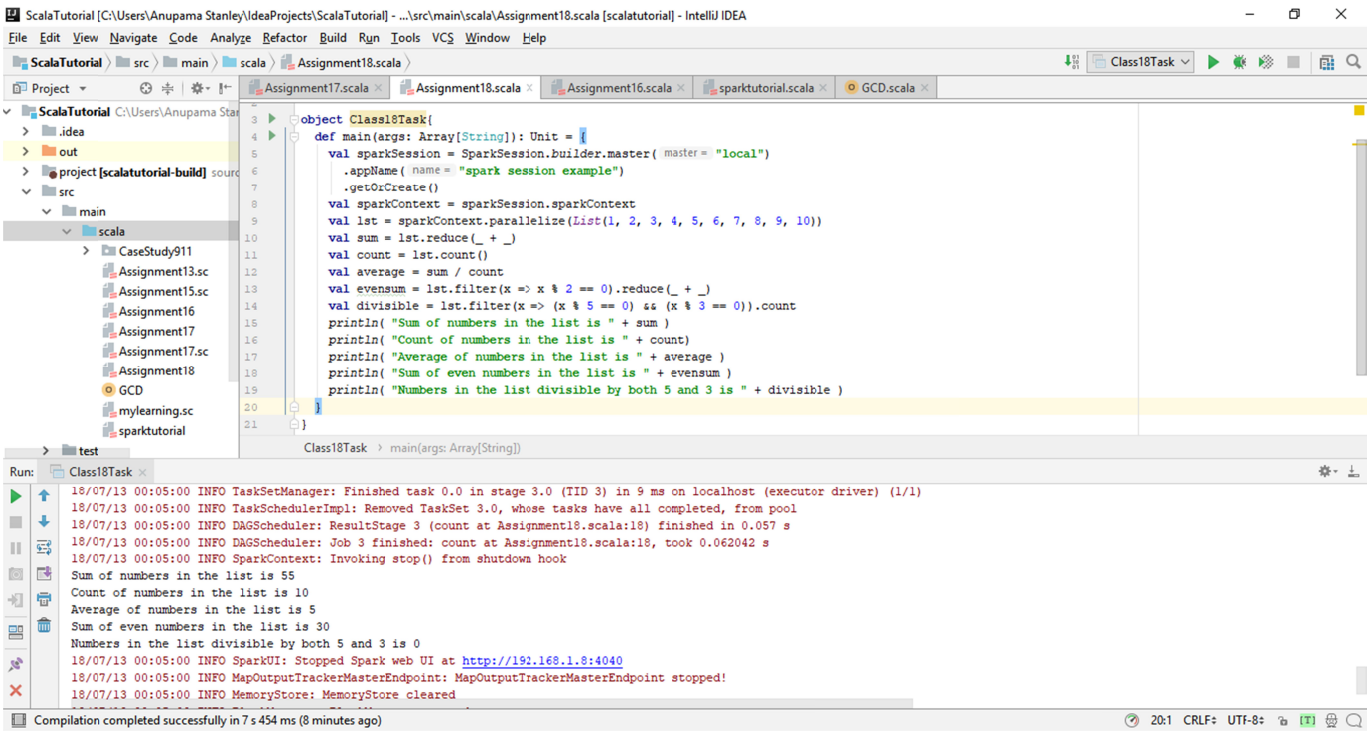
- find the sum of all numbers

- 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



Task 2

1. *Pen down the limitations of MapReduce.*

MapReduce cannot handle below:

- Interactive Processing
- Real-time (stream) Processing
- Iterative (delta) Processing
- In-memory Processing
- Graph Processing

Issue with Small Files: Hadoop is not suited for small data. (HDFS) Hadoop distributed file system lacks the ability to efficiently support the random reading of small files because of its high capacity design.

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.

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.

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.

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).

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 are 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.

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.

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 stands for “Resilient Distributed Dataset”. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node 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 posses self-recovery in the case of failure.

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

Sparkling Features of Spark RDD

1. **In-memory computation**: 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.
2. **Lazy Evaluation:** 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. They are not executed immediately. Two most basic type of transformations is a map(), filter().
   Eg of 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).

3. **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.
4. **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.
5. **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.
6. **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.
7. **Parallel:** Rdd, process the data in parallel over the cluster.
8. **Location-Stickiness**: RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The DAGScheduler places the partitions in such a way that task is close to data as much as possible. Thus speed up computation.

9. **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.

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

11. **No limitation:** We can have any number of RDD. There is no limit to its number. The limit depends on the size of disk and memory.