Lab Program-3: Wordcount Program

Prashanth BS

Assistant Professor,Department of ISE, Nitte Meenakshi Institute of Technology

June 16, 2022

Problem Statement

Map-Reduce

Use the Hadoop framework to write a custom MapReduce program to perform word count operation on a custom data set .

Hadoop Program Flow

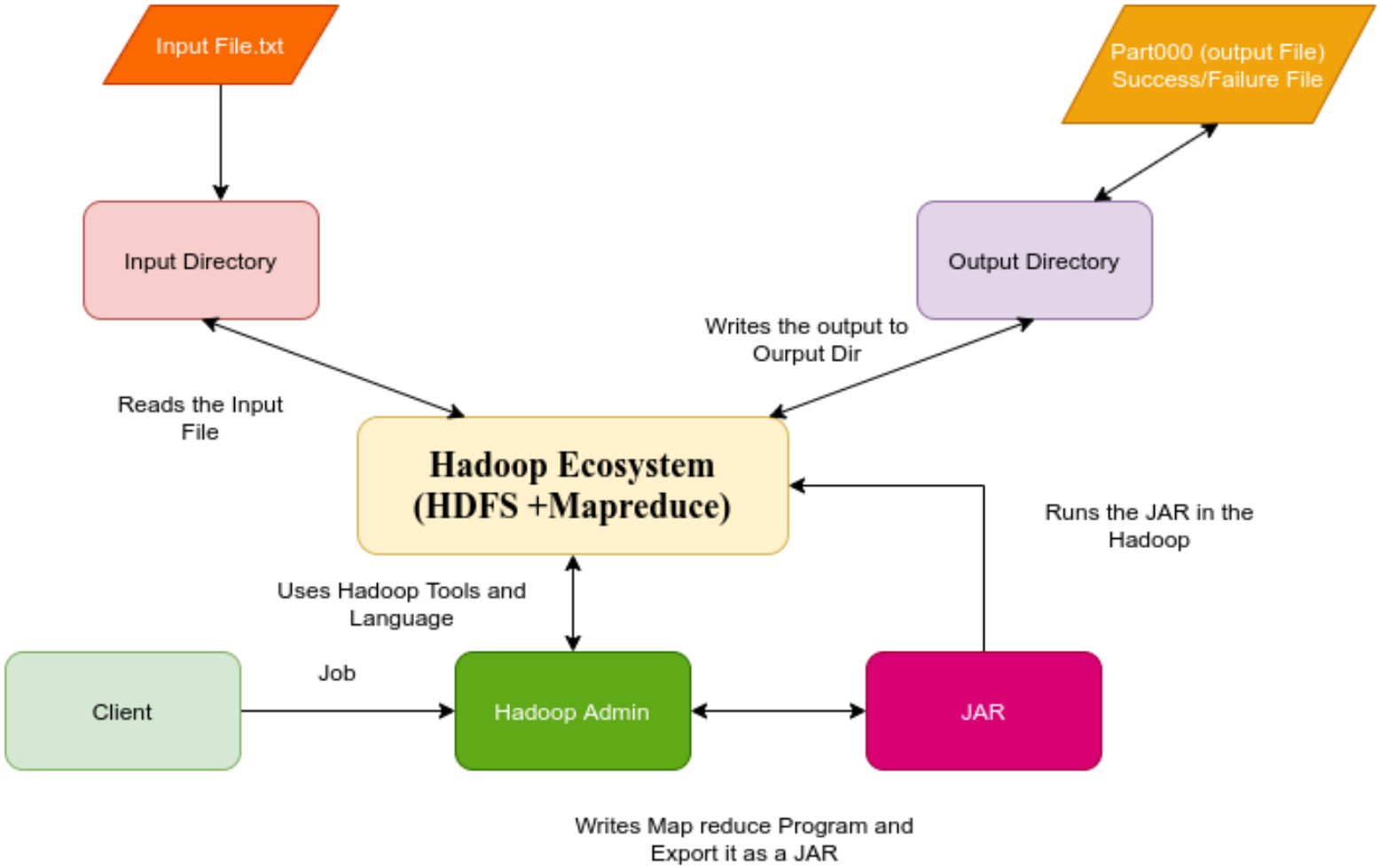


Figure: Mapreduce Workflow

Word-Count Program

- The following is an example of a word-count program

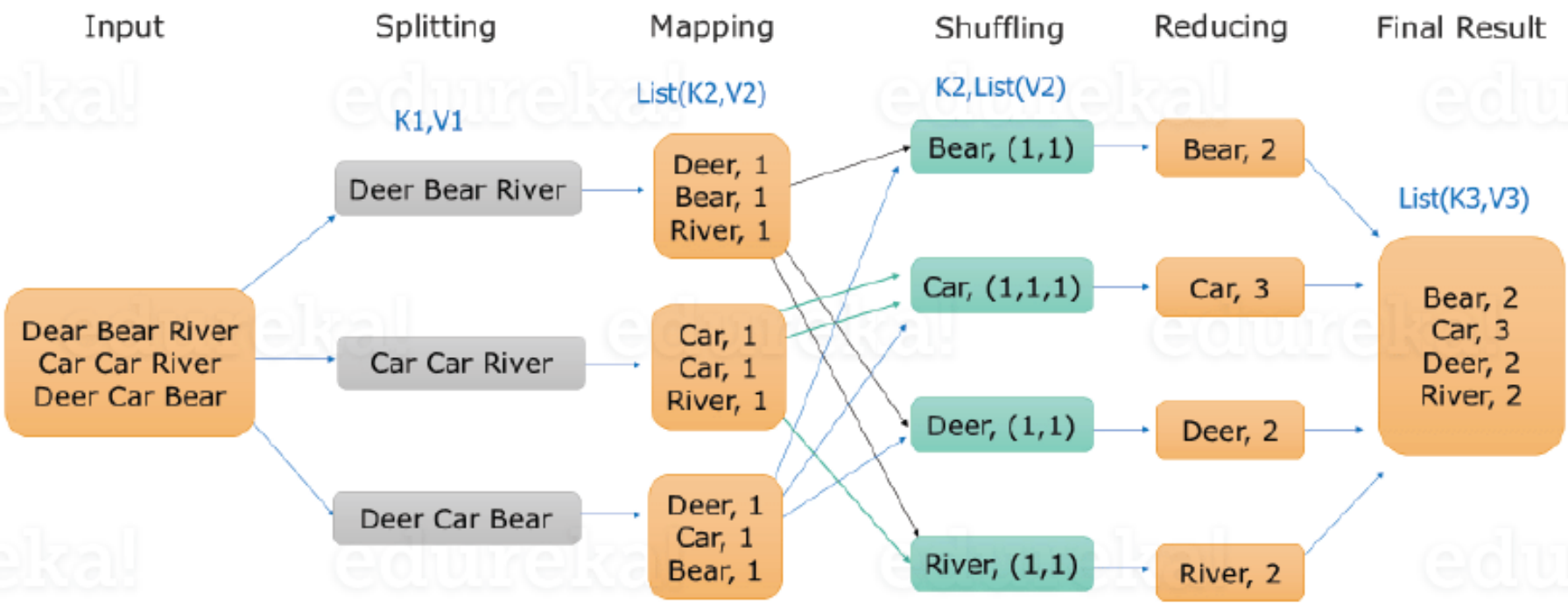
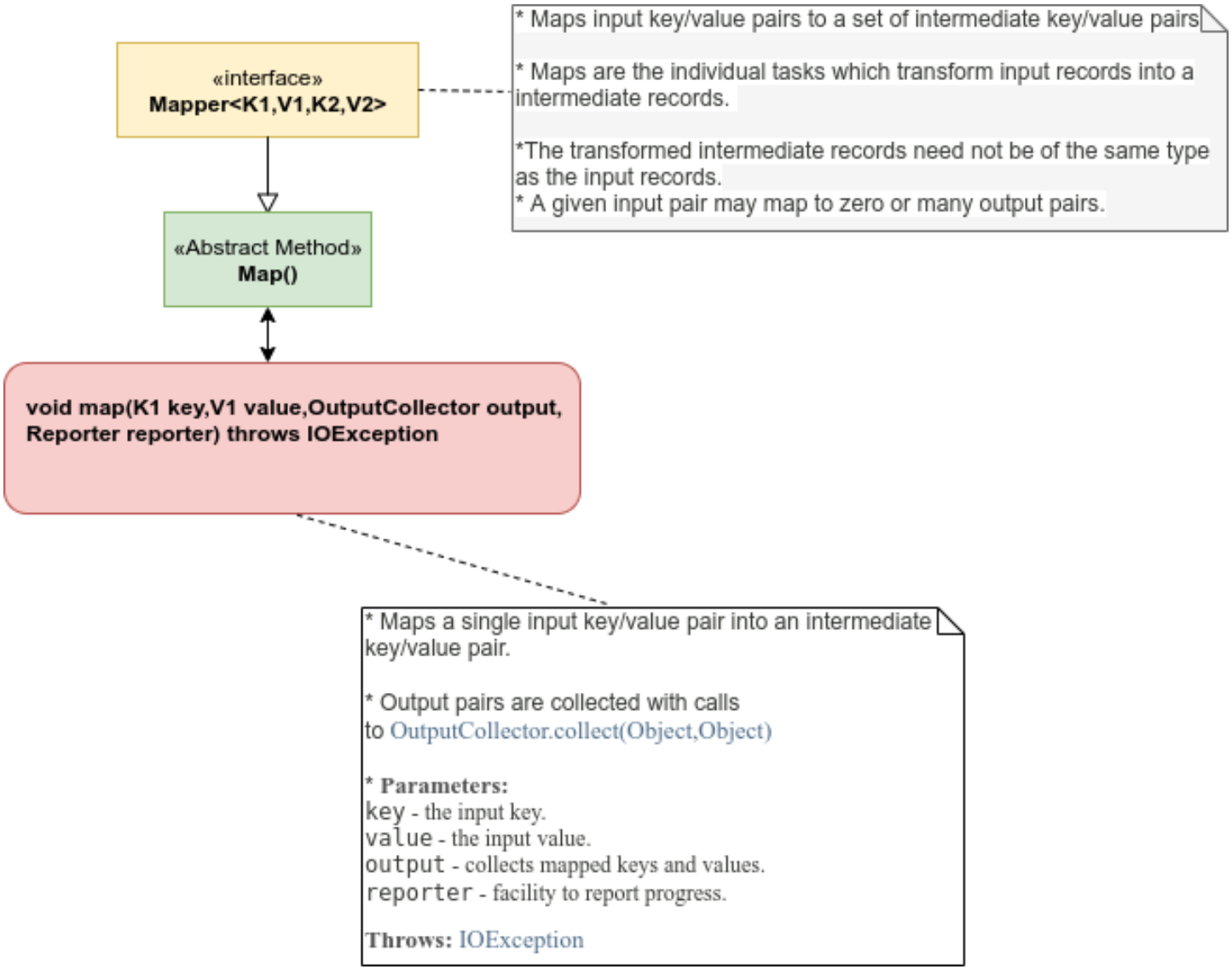


Figure: Mapreduce Workflow

Mapper Interface

The Mapper interface provided by the MapReduce base class is as shown below,



Reducer Interface

The Reducer interface provided by the MapReduce base class is as shown below,

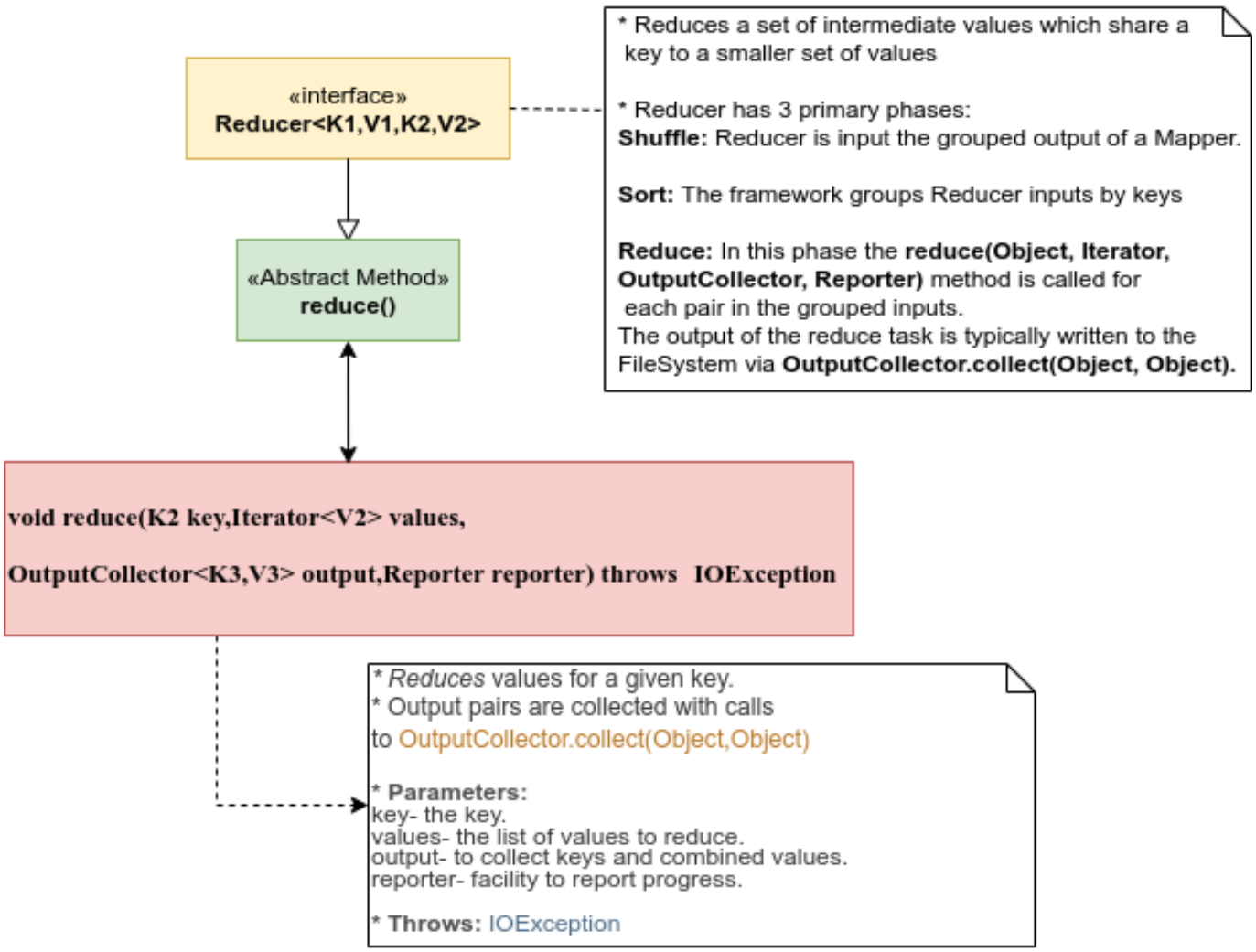


Figure: Reducer Interface

Map-Reduce Programming

The overall structure of the Map-Reduce Program is as shown below

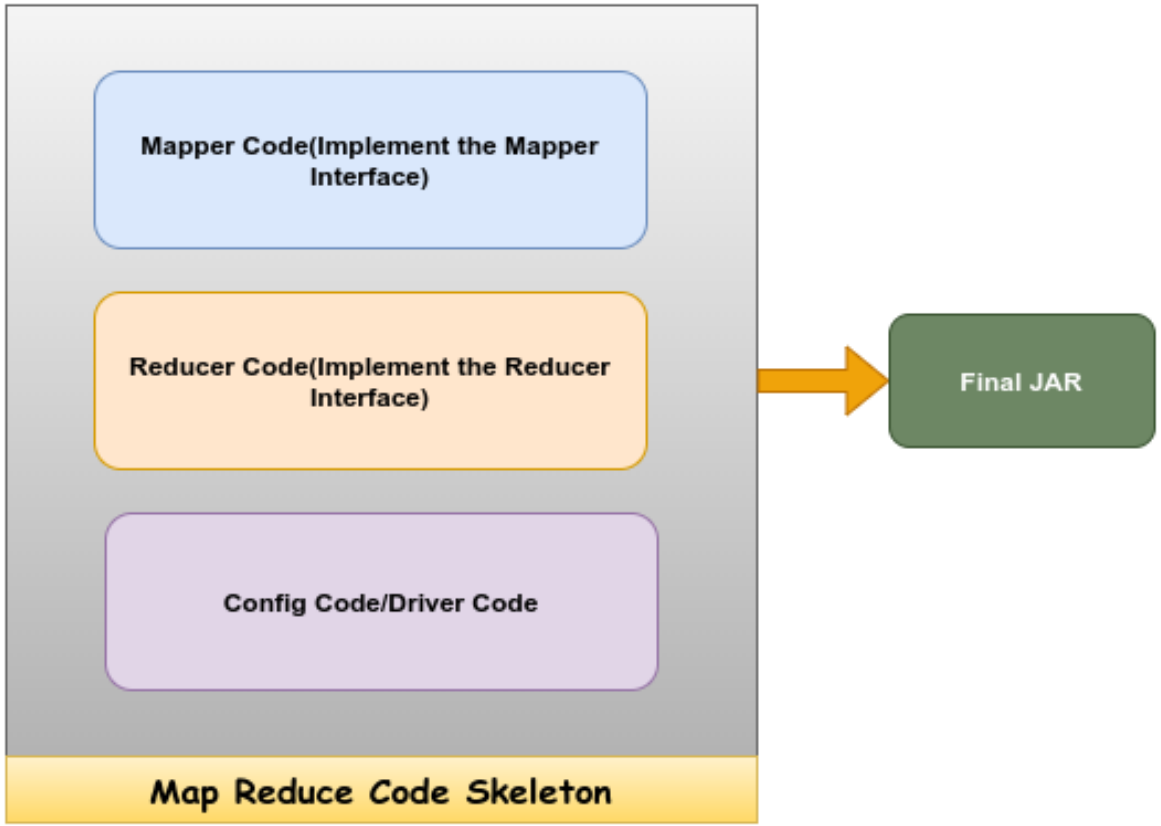


Figure: Mapreduce Workflow

Steps: Map-Reduce Programming

1. Create a class called as Wordcount which as 2 subclasses called as Map and Reduce
2. Make the Map and Reduce Subclass extend MapReduceBase class
3. Make the Map class implement Mapper Interface and Reduce class to implement Reducer Interface The code Snippet would look like below,

```
public class WordCount {
//Mapper code
public static class Map extends MapReduceBase implements
 Mapper<LongWritable, Text, Text, IntWritable> {
}
//Reducer code
public static class Reduce extends MapReduceBase implements
 Reducer<Text, IntWritable, Text, IntWritable> {
}
//Driver code
public static void main(String[] args) throws Exception {
}
}
```

Steps: Map-Reduce Programming

4. Design the Driver code which consists of code snippets that configure the jobs, files and paths

```
// Create a new Job & Set the Job Name
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");

// Set the type of value we get at the output <Text, IntWritable>
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class); // Set the Mapper class

// Set the Reducer and combiner class
conf.setCombinerClass(Reduce.class);
conf.setReducerClass(Reduce.class);
// Set the Input and Output Format class
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

// Configure the input path and output path
FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
JobClient.runJob(conf);    // Run the JOB
```

Steps: Map-Reduce Programming

5. Design the Mapper code which consists of code snippets of generating intermediate key-value pairs

```
public static class Map extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, IntWritable> {
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();

public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
        }
        }
}
```

Steps: Map-Reduce Programming

5. Design the Reducer code which consists of code snippets of taking in the output of Mapper and Producing aggregation results

```
public static class Reduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
        int sum = 0;
        while (values.hasNext()) {
                sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
        }
}
```

6. Prepare the INPUT directory, OUTPUT directory and INPUT file using HDFS commands

```
// Creating InputDir and OutputDir
$ hdfs dfs -mkdir -p ~/InputDir
$ hdfs dfs -mkdir -p ~/OuputDir

// Append contents of the file using -appendToFile
$ hdfs dfs -appendToFile - ~/InputDir/test.txt
// Add some lines + CTRL-D twice
```

Steps: Map-Reduce Programming

7. Add external JARS in the path **/usr/local/hadoop/share/hadoop** via build path such as
   - hadoop-common.x.jar
   - hadoop-mapred-clientcore.x.jar
8. Export the Wordcount Project as a .jar file
9. Run the Job by passing in the input and output directories created earlier as follows,

```
// General Syntax
$ hadoop jar full_path_to_jar input_directory output_directory
// Example Run
$ hadoop jar /home/hadoop/Desktop/input.jar ~/InputDir ~/OuputDir
```

10. The output directory consists of the JOBS output. Two files to be exact, a file that shows SUCCESS status and the partxx file which contains the Map-reduce Output. Cat the partxx file to get the output

```
$ hdfs dfs -cat ~/OuputDir/part*
```