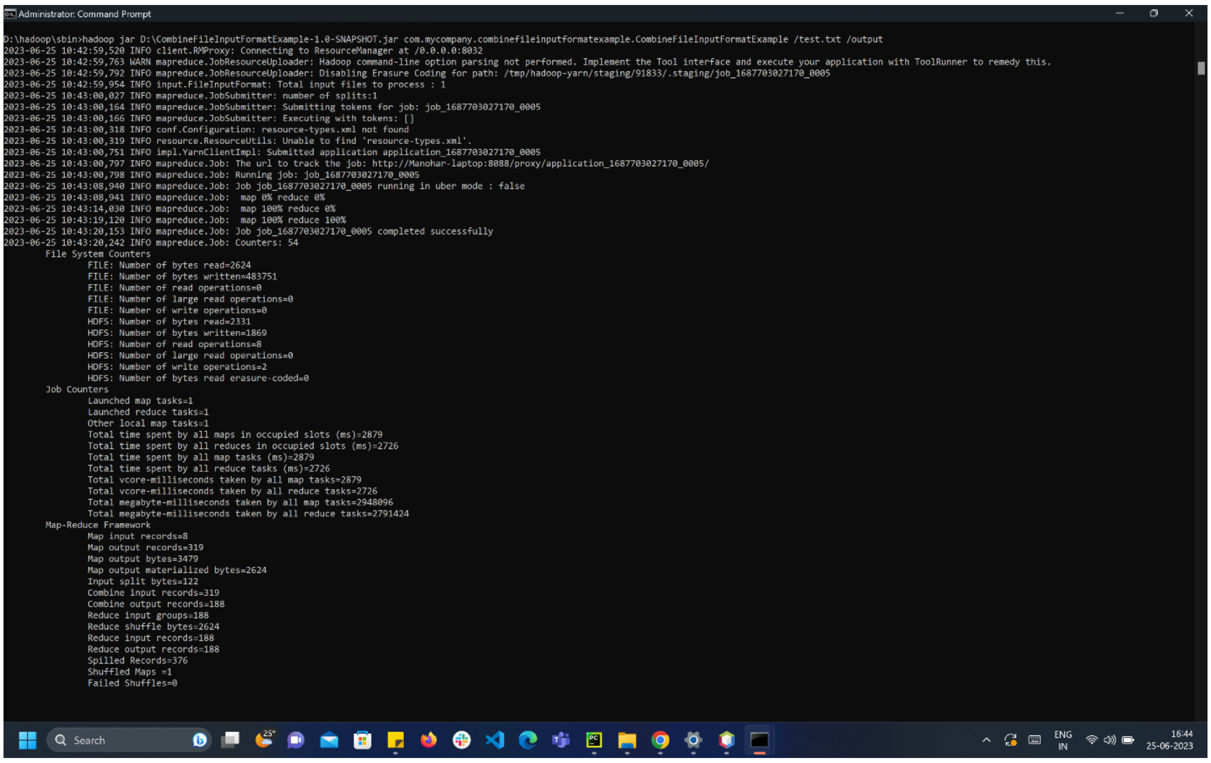
PART – 4

Write one MapReduce program using each of the classes that extend FileInputFormat<k,v> (CombineFileInputFormat, FixedLengthInputFormat, KeyValueTextInputFormat, NLineInputFormat, SequenceFileInputFormat, TextInputFormat)



```
D:\hadoop\sbin>hadoop jar D:\CombineFileInputFormatExample-1.0-SNAPSHOT.jar com.mycompany.combinefileinputformatexample.CombineFileInputFormatExample /test.txt /output
2023-06-25 10:42:59,520 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
2023-06-25 10:42:59,763 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2023-06-25 10:42:59,792 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/91833/.staging/job_1687703027170_0005
2023-06-25 10:42:59,954 INFO input.FileInputFormat: Total input files to process : 1
2023-06-25 10:43:00,027 INFO mapreduce.JobSubmitter: number of splits:1
2023-06-25 10:43:00,164 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1687703027170_0005
2023-06-25 10:43:00,166 INFO mapreduce.JobSubmitter: Executing with tokens: []
2023-06-25 10:43:00,318 INFO conf.Configuration: resource-types.xml not found
2023-06-25 10:43:00,319 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2023-06-25 10:43:00,751 INFO impl.YarnClientImpl: Submitted application application_1687703027170_0005
2023-06-25 10:43:00,797 INFO mapreduce.Job: The url to track the job: http://Manohar-laptop:8088/proxy/application_1687703027170_0005/
2023-06-25 10:43:00,798 INFO mapreduce.Job: Running job: job_1687703027170_0005
2023-06-25 10:43:08,940 INFO mapreduce.Job: Job job_1687703027170_0005 running in uber mode : false
2023-06-25 10:43:08,941 INFO mapreduce.Job:  map 0% reduce 0%
2023-06-25 10:43:14,030 INFO mapreduce.Job:  map 100% reduce 0%
2023-06-25 10:43:19,120 INFO mapreduce.Job:  map 100% reduce 100%
2023-06-25 10:43:20,153 INFO mapreduce.Job: Job job_1687703027170_0005 completed successfully
2023-06-25 10:43:20,242 INFO mapreduce.Job: Counters: 54
        File System Counters
                FILE: Number of bytes read=2624
                FILE: Number of bytes written=483751
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=2331
                HDFS: Number of bytes written=1869
                HDFS: Number of read operations=8
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Other local map tasks=1
                Total time spent by all maps in occupied slots (ms)=2879
                Total time spent by all reduces in occupied slots (ms)=2726
                Total time spent by all map tasks (ms)=2879
                Total time spent by all reduce tasks (ms)=2726
                Total vcore-milliseconds taken by all map tasks=2879
                Total vcore-milliseconds taken by all reduce tasks=2726
                Total megabyte-milliseconds taken by all map tasks=2948096
                Total megabyte-milliseconds taken by all reduce tasks=2791424
        Map-Reduce Framework
                Map input records=8
                Map output records=319
                Map output bytes=3479
                Map output materialized bytes=2624
                Input split bytes=122
                Combine input records=319
                Combine output records=188
                Reduce input groups=188
                Reduce shuffle bytes=2624
                Reduce input records=188
                Reduce output records=188
                Spilled Records=376
                Shuffled Maps =1
                Failed Shuffles=0
```

CombineFileInputFormat:

This code example uses the CombineFileInputFormat class, which allows combining multiple small input files into larger splits to improve efficiency.

The WordCountMapper class reads each line from the input file and tokenizes it into words.

The WordCountReducer class receives the words as keys and counts their occurrences.

The program counts the occurrence of each word in the input files and writes the result to the output file.

The significance of using CombineFileInputFormat is to optimize the processing of small files by combining them into larger input splits, reducing overhead.

Code: (Full code is submitted in the part-4 file with reducer & main method)

public class CombineFileInputFormatExample {

```
public static class WordCountMapper extends 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, Context context) throws IOException,
InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString());

      while (itr.hasMoreTokens()) {

        word.set(itr.nextToken());

        context.write(word, one);

      }

    }

  }
```

FixedLengthInputFormat:

This code example uses the FixedLengthInputFormat class, which reads fixed-length records from the input file.

The WordCountMapper class reads each character from the input file and emits it as a separate key-value pair.

The WordCountReducer class receives the characters as keys and counts their occurrences.

The program counts the occurrence of each character in the input file and writes the result to the output file.

The significance of using FixedLengthInputFormat is to process input files with fixed-length records, such as files with fixed-width columns.

Code: (Full code is submitted in the part-4 file with reducer & main method)

```
public class FixedLengthInputFormatExample {

public static class WordCountMapper extends 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, Context context) throws IOException,
InterruptedException {
            String line = value.toString();
            word.set(line);
            context.write(word, one);
        }
    }
```

KeyValueTextInputFormat:

This code example uses the KeyValueTextInputFormat class, which reads input files in key-value pair format, where each line contains a key and a value separated by a delimiter (default is tab).

The WordCountMapper class reads each value from the input file and tokenizes it into words.

The WordCountReducer class receives the words as keys and counts their occurrences.

The program counts the occurrence of each word in the input file and writes the result to the output file.

The significance of using KeyValueTextInputFormat is to process input files with key-value pairs, such as log files or configuration files.

Code: (Full code is submitted in the part-4 file with reducer & main method)

```
public class KeyValueTextInputFormatExample {
    public static class WordCountMapper extends Mapper<Text, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
public void map(Text key, Text value, Context context) throws IOException,
InterruptedException {
            String line = value.toString();
            String[] words = line.split("\\s+"); \\ Regex one or more whitespace characters (spaces,
tabs, or line breaks).
            for (String word : words) {
                this.word.set(word);
                context.write(this.word, one);
        }
      }
  }
```

NLineInputFormat:

This code example uses the NLineInputFormat class, which reads N lines of input as a split.

The WordCountMapper class reads each line from the input file and tokenizes it into words.

The WordCountReducer class receives the words as keys and counts their occurrences.

The program counts the occurrence of each word in the input file and writes the result to the output file.

The significance of using NLineInputFormat is to control the number of lines per input split, which can be useful when the input data has a specific structure or when you want to process a specific number of lines at once.

Code: (Full code is submitted in the part-4 file with reducer & main method)

```
public class NLineInputFormatExample {

  public static class WordCountMapper extends 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, Context context) throws IOException,
InterruptedException {

      String line = value.toString();

      String[] words = line.split("\\s+"); \\ Regex one or more whitespace characters (spaces,
tabs, or line breaks).

      for (String word : words) {

        this.word.set(word);

        context.write(this.word, one);

      }

    }

  }
```

SequenceFileInputFormat:

This code example uses the SequenceFileInputFormat class, which reads input files stored in the SequenceFile format, which is a binary file format that stores key-value pairs.

The WordCountMapper class reads each key from the input file and tokenizes it into words.

The WordCountReducer class receives the words as keys and counts their occurrences.

The program counts the occurrence of each word in the input file and writes the result to the output file.

The significance of using SequenceFileInputFormat is to process input files stored in the SequenceFile format, which is commonly used in Hadoop for efficient data storage and retrieval.

Code: (Full code is submitted in the part-4 file with reducer & main method)

```
public class SequenceFileInputFormatExample {

  public static class WordCountMapper extends Mapper<Text, IntWritable, Text,
IntWritable> {

    private final static IntWritable one = new IntWritable(1);

    public void map(Text key, IntWritable value, Context context) throws IOException,
InterruptedException {

      context.write(key, one);

    }

  }
```

TextInputFormat:

This code example uses the TextInputFormat class, which reads input files as plain text files, where each line is considered a record.

The WordCountMapper class reads each line from the input file and tokenizes it into words.

The WordCountReducer class receives the words as keys and counts their occurrences.

The program counts the occurrence of each word in the input file and writes the result to the output file.

The significance of using TextInputFormat is to process plain text files where each line represents a record, which is a common format for textual data processing.

Code: (Full code is submitted in the part-4 file with reducer & main method)

```
public class TextInputFormatExample {
    public static class WordCountMapper extends 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, Context context) throws IOException,
InterruptedException {

            StringTokenizer itr = new StringTokenizer(value.toString());

            while (itr.hasMoreTokens()) {

                word.set(itr.nextToken());

                context.write(word, one);

            }

        }

    }
```