

# Big Data Analytics

Isara Anantavasilp

Lecture 7: MapReduce Job

# Examine Word Count Code

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
...

public class WordCount {

    public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{

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

        public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            ...
        }

        public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
            private IntWritable result = new IntWritable();

            public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
                ...
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        ...
    }
}
```

# Structure of WordCount

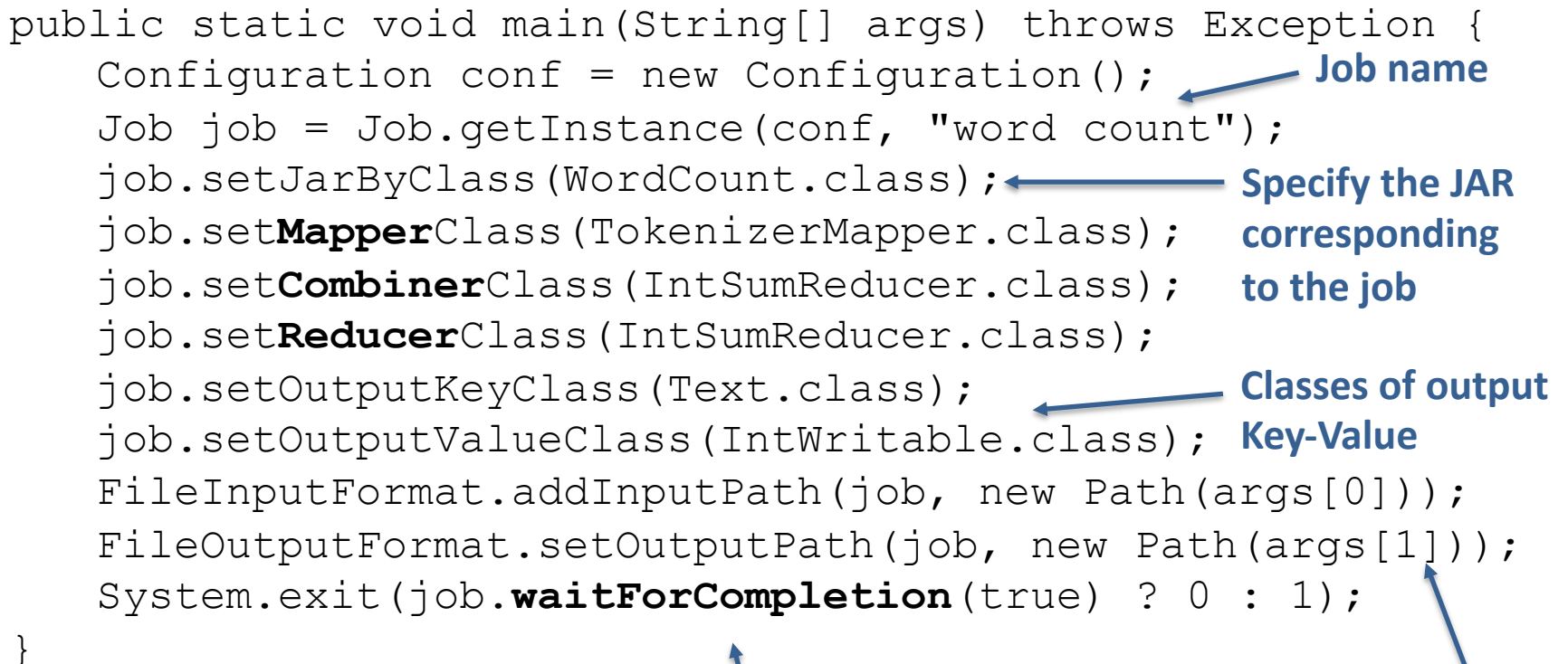
- Hadoop job consists of **Mapper** and **Reducer** (also Combiner, but we will talk about that later)
- When we initiate a Hadoop job, we have to specify what are mapper and reducer of that job
  - Scheduler will run the mapper and reducer on corresponding nodes
- In our WordCount class, we have two nested classes
  - TokenizerMapper extends Mapper
  - IntSumReducer extends Reducer
- There is the main method that initiate the task
  - Aka the **Driver**

# main Function (Driver)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

# main Function (Driver)

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    Job job = Job.getInstance(conf, "word count");  
    job.setJarByClass(WordCount.class);  
    job.setMapperClass(TokenizerMapper.class);  
    job.setCombinerClass(IntSumReducer.class);  
    job.setReducerClass(IntSumReducer.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
    System.exit(job.waitForCompletion(true) ? 0 : 1);  
}
```



Job submission

Input and output paths

# main Function

- Main method specifies job attributes
  - Mapper, Reducer and Combiner (optional)
  - Input and output classes
  - Input and output locations (in this case, passed from the program arguments)
- It also submits the job to the framework
  - `waitForCompletion`

# Tokenizer Mapper

```
public static class TokenizerMapper
extends Mapper<Object, Text, Text, IntWritable>{

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

    public void map(Object 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);
        }
    }
}
```

# Mapper Class

- Mapper class is a ***generic*** type with four parameters
  - Input key
  - Input value
  - Output key
  - Output value
- A given input pair may map to zero or many output pairs.

```
public class  
Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> extends  
Object
```

- Inside Mapper class, there is also a method `map` to execute map task



# Mapper Class

- In WordCount, we define the mapper as

```
public class TokenCounterMapper
    extends Mapper<Object, Text, Text,
IntWritable>
```

- This means the class `TokenCounterMapper` is a subclass of `Mapper`
- It takes `Object` as key, the value of the key is text, and the output key will be `Text` and the value is Integer object `IntWritable`
- **Map processes the input *line-by-line***

# Tokenizer Mapper

```
public static class TokenizerMapper
extends Mapper<Object, Text, Text, IntWritable>{

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

    public void map(Object 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);
        }
    }
}
```

**Output value (1)** (points to `one`)

**Input data (in our case, entire text)** (points to `value`)

**Output key** (points to `word`)

**Define output key** (points to `itr.nextToken()`)

**Key = word**  
**Value = one** (points to `context.write(word, one);`)

**Write one line of output to output buffer** (points to `context.write(word, one);`)

# Combiner

- Sometimes the map results are large.
- The map results could be **combined** before sending to the reducer
  - The combination can be done locally
  - It is optimization process, so it is not required
  - Hadoop does not guarantee if it will ever be executed
- Combiner has no interface. It must have the same interface as the reducer
- In our case, the combination is the same function as reduction

```
job.setCombinerClass(IntSumReducer.class);
```

# Integer Sum Reducer

```
public static class IntSumReducer extends Reducer
    <Text,IntWritable,Text,IntWritable> {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {

        int sum = 0;

        for (IntWritable val : values) {
            sum += val.get();
        }

        result.set(sum);
        context.write(key, result);
    }
}
```

# Reducer Class

- Reducer class is a ***generic*** type with four parameters
  - Input key
  - Input value
  - Output key
  - Output value
- Input to the reducer is a key and a corresponding list of values
- A given input pair may map to zero or many output pairs.

```
public class  
Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>  
extends Object
```

- Inside Reducer class, there is also a method `reduce` to execute reduce task

# Reducer Class

- In WordCount, we define the reducer as

```
public static class IntSumReducer  
    extends Reducer  
    <Text, IntWritable, Text, IntWritable>
```

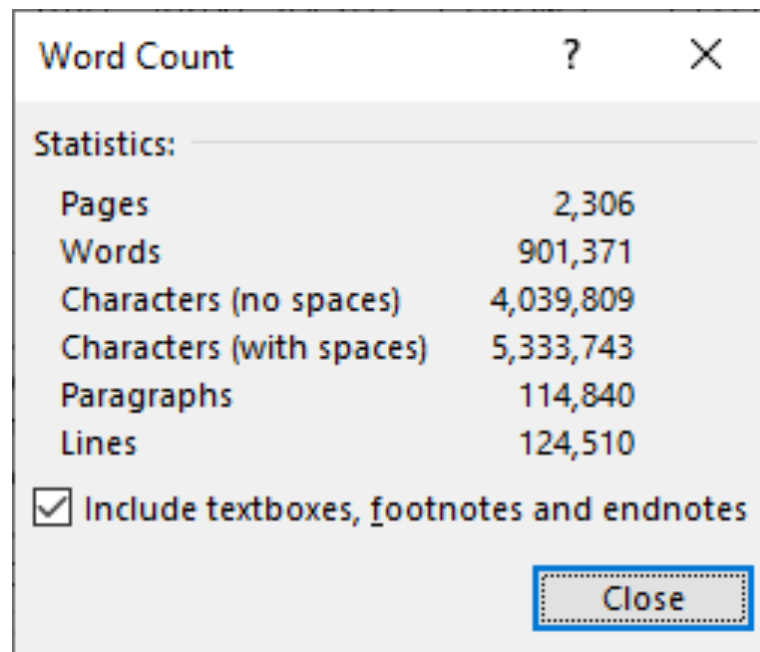
- This means the class `IntSumReducer` is a subclass of `Reducer`
- It takes `Text` as key, the value of the key is Integer, and the output key will be `Text` and the value is Integer object `IntWritable`

# Exercise

- Edit the mapper to count lines, instead of words
- Edit the mapper to count number of all words (not each word)
- Edit the mapper to count both lines and words

# Word Count of Shakespeare

```
part-r-00000 SUCCESS
[cloudera@quickstart out-s]$ cat part-r-00000
Line      124456
Word      901325
[cloudera@quickstart out-s]$
```





# Homework

- **Clean the word count results**
  - Currently, the results of word count are very redundant because the words may contain special characters. (e.g., “Caesars., Caesars’, “Caesars,”)
  - Clean those special characters
- **Handle the upper/lower cases**
  - Some words might be upper and lower cases or both
  - Count the words while ignoring the cases
- **Average words per line**