CSCI4180 Tutorial-3

# Assignment 1 Review (Part 1)

REN, Yanjing
yjren22@cse.cuhk.edu.hk

Sep. 28, 2022

# Outline

- Assignment 1 Overview
  - Specifications
- Hadoop configuration
  - Pseudo distributed mode & Fully distributed mode
- MapReduce Programming
  - WordCount

# Assignment 1 Overview

- Four parts
  - Part 1: Getting Started on Hadoop (25%)
  - Part 2: Word Length Count (25%)
  - Part 3: *N*-gram Initial Count (25%)
  - Part 4: Counting *N*-gram Initials Relative Frequencies (25%)

# Assignment 1 Overview

- Part 1: Getting Started on Hadoop (25%)
    - Goals
        - Configure Hadoop in ***pseudo distributed mode*** on single VM
        - Bouns: configure Hadoop in ***fully distributed mode*** with 1 NameNode and 2 DataNodes on pre-assigned 3 VMs
        - Run WordCount program
    - Demo
        - Show your VM and Hadoop configurations
        - Run WordCount with 2 datasets: **KJV Bible** and **the complete works of Shakespeare**

# Assignment 1 Overview

- Part 2: Word Length Count (25%)
  - Goals
    - Extend ***WordCount*** program to ***WordLengthCount***
    - Implement **in-mapper combining** to optimize your program
    - Program output: **(length, count)**, separated by an **empty space**.
  - **Important Notes**
    - Double check your file name (***WordLengthCount.java***)
    - Do **NOT** specify the package (e.g., “package <pkg_name>”) in your program

# Assignment 1 Overview

- Part 3: *N*-gram Initial Count (25%)
    - Goals
        - Extend ***WordLengthCount*** program to ***NgramInitialCount***
    - Input
        - Additional Parameters
            - N: number of words in N-gram
    - Data
        - You may assume that the datasets contain only printable ASCII characters

# Assignment 1 Overview

- Part 3: *N*-gram Initial Count (25%)
  - Definitions
    - **Word**
      - A sequence of English alphabets characters (Case-sensitive)
    - **Ngram**
      - A sequence of **N consecutive words**
      - The word in the end of a line and the word in the beginning of the next line also form an N-gram
    - **Ngram Initial**
      - The first letters of the N words of the N-gram

# Assignment 1 Overview

- Part 3: *N*-gram Initial Count (25%)
    - **Delimiter**
        - Non-alphabet characters
            - {ASCII } \ {alphabetic characters}
        - Multiple consequent non-alphabet characters are treated as a single delimiter

# Assignment 1 Overview

- Part 4: Counting *N*-gram Initials Relative Frequencies (25%)
  - Extend ***NgramInitialCount*** program to ***NgramInitialRF***
    - Relative frequency = count("X Y Z")/count("X *")
      - "*" stands for the remaining N - 1 characters of **any** N-gram Initials.
      - In this example, *N* = 3
      - Both "X Y Z" and "X *" are 3-gram words
    - Output
      - **ONLY** output sequences with frequency ≥ θ

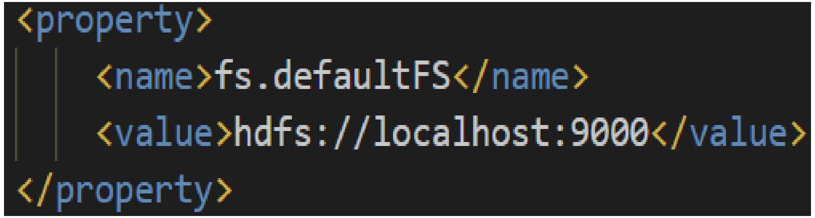
# Outline

- Assignment 1 Overview
  - Specifications
- Hadoop configuration
  - Pseudo distributed mode & Fully distributed mode
- MapReduce Programming
  - WordCount

# Hadoop Configuration

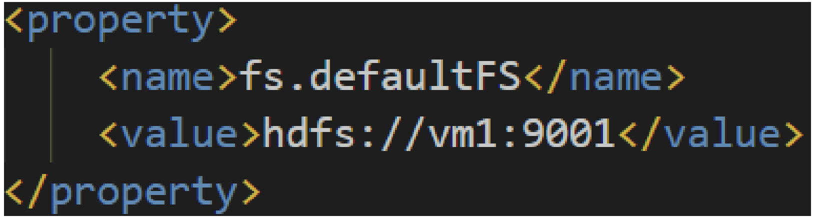
- core-site.xml

***pseudo distributed mode***



```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
</property>
```

***fully distributed mode***



```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://vm1:9001</value>
</property>
```

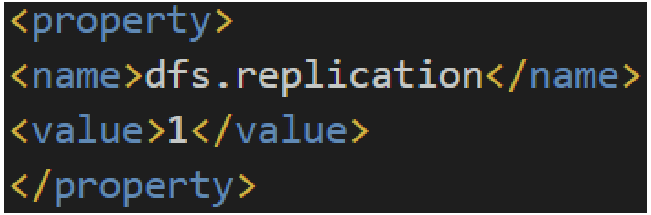
Localhost -> vm1

**Follow the link to set up pseudo distributed mode**
**We also provide a set of configuration files on Blackboard: Link**

# Hadoop Configuration

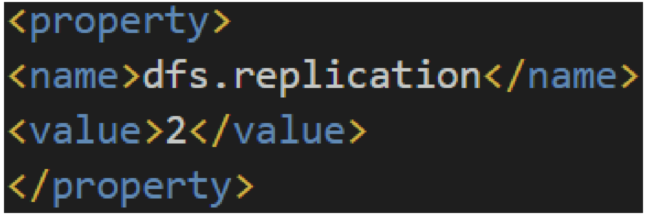
- hdfs-site.xml

***pseudo distributed mode***



```
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
```

***fully distributed mode***



```
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
```

Replication 1 -> 2

- slaves

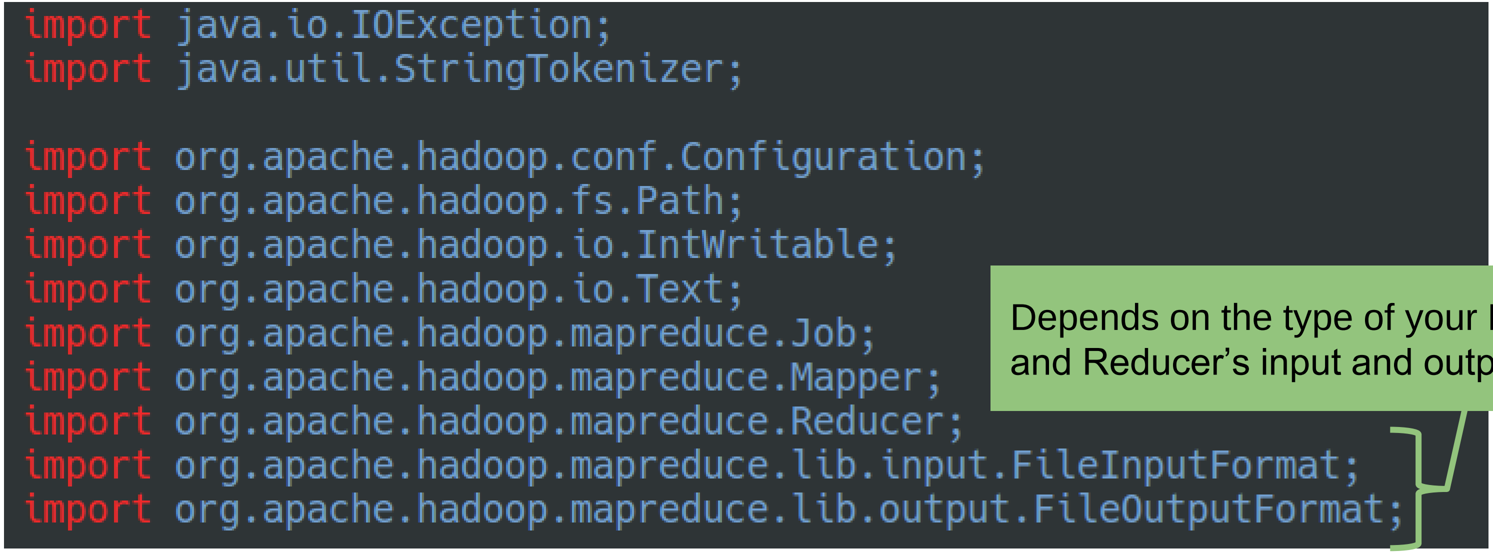
localhost-> vm2 vm3

# Outline

- Assignment 1 Overview
  - Specifications
- Hadoop configuration
  - Pseudo distributed mode & Fully distributed mode
- MapReduce Programming
  - WordCount

# MapReduce Programming

- Basic Structure



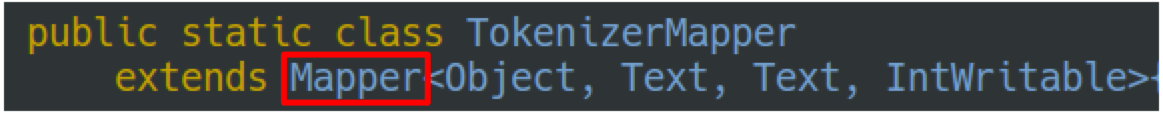
```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

Depends on the type of your Mapper and Reducer's input and output

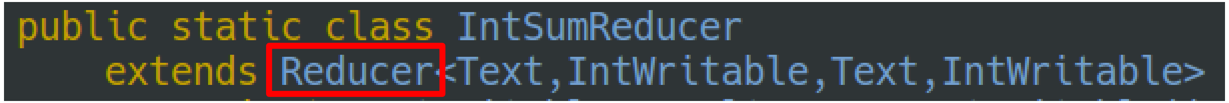
# MapReduce Programming

- Basic Structure
  - public class **WordCount**



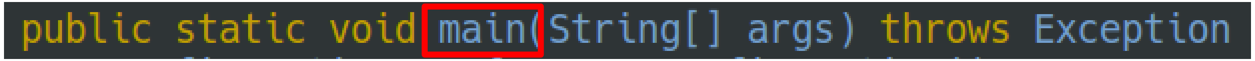
```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{
```

The **map** function is put inside this class



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

The **reduce** function is put inside this class

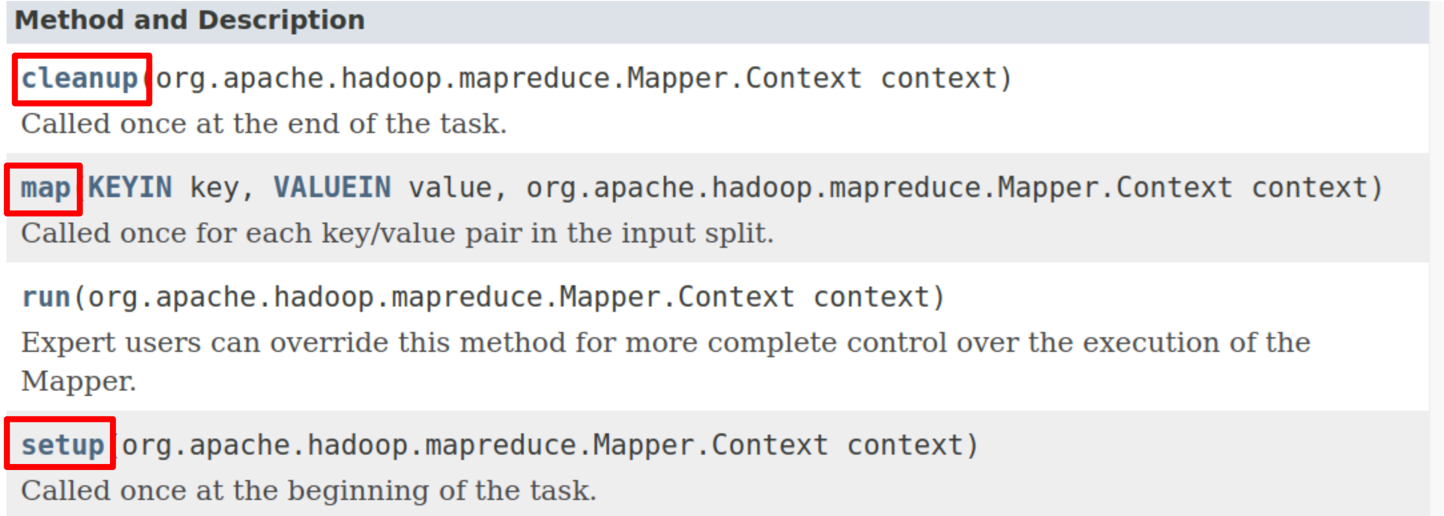


```
public static void main(String[] args) throws Exception
```

The **starting** point of the job, which includes job configuration

# MapReduce Programming

- Mapper
  - Refer to Mapper API



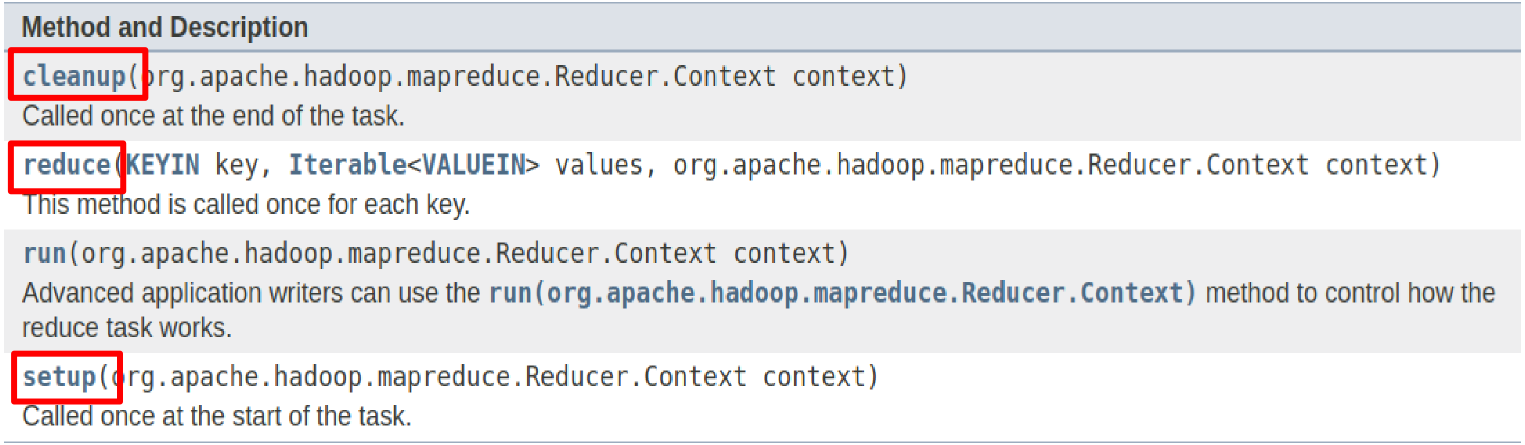
| Method and Description |
|---|
| `cleanup(org.apache.hadoop.mapreduce.Mapper.Context context)`<br>Called once at the end of the task. |
| `map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)`<br>Called once for each key/value pair in the input split. |
| `run(org.apache.hadoop.mapreduce.Mapper.Context context)`<br>Expert users can override this method for more complete control over the execution of the Mapper. |
| `setup(org.apache.hadoop.mapreduce.Mapper.Context context)`<br>Called once at the beginning of the task. |

# MapReduce Programming

- Mapper
  - Class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
    - We use **Generics** to specify the data type of the input key-value pair (KEYIN, VALUEIN) and output key-value pair (KEYOUT, VALUEOUT)
    - We can use the following types here
    - By default, the KEYIN and VALUEIN of the mapper class is **LongWritable** and **Text** respectively, which represents *the line number* and *the whole line* in the input

# MapReduce Programming

- Reducer
  - Refer to Reducer API



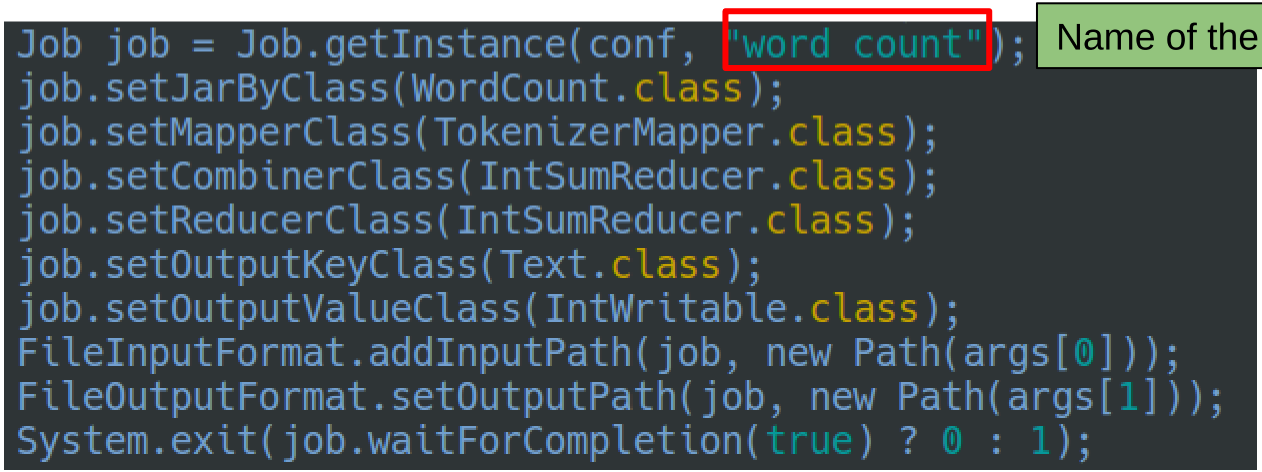
| Method and Description |
|---|
| `cleanup(org.apache.hadoop.mapreduce.Reducer.Context context)`<br>Called once at the end of the task. |
| `reduce(KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context)`<br>This method is called once for each key. |
| `run(org.apache.hadoop.mapreduce.Reducer.Context context)`<br>Advanced application writers can use the `run(org.apache.hadoop.mapreduce.Reducer.Context)` method to control how the reduce task works. |
| `setup(org.apache.hadoop.mapreduce.Reducer.Context context)`<br>Called once at the start of the task. |

# MapReduce Programming

- Reducer
    - Class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
        - Like mapper, we need to specify the data type of the input key-value pair (KEYIN, VALUEIN) and output key-value pair (KEYOUT, VALUEOUT)
        - The mapper's (KEYOUT, VALUEOUT) should be the same as the reducer's (KEYIN, VALUEIN)
            - This is because the input of reducers comes from the output of mappers

# MapReduce Programming

- Configure Job

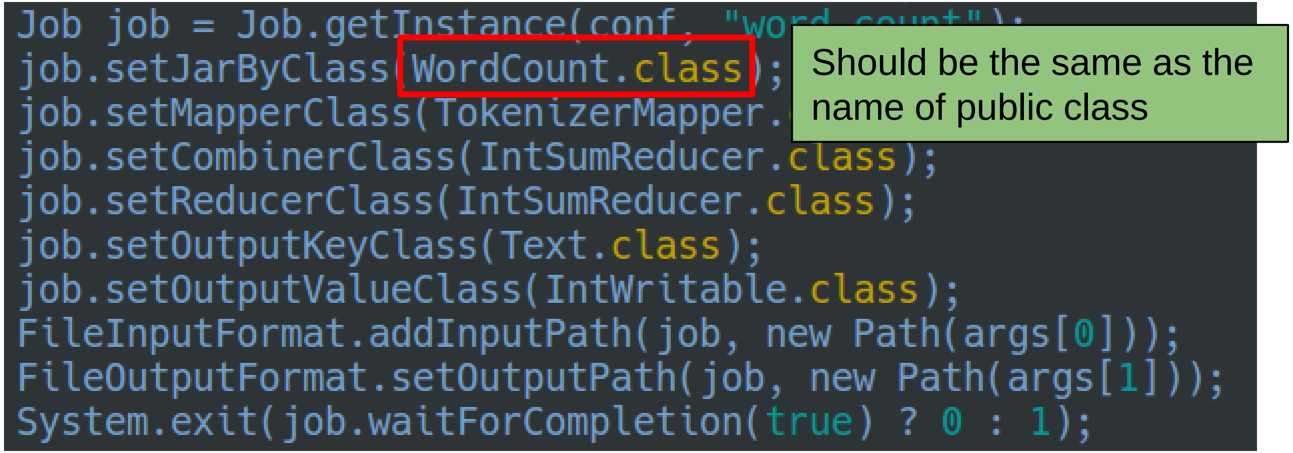


```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job,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

Name of the Job

# MapReduce Programming

- Configure Job

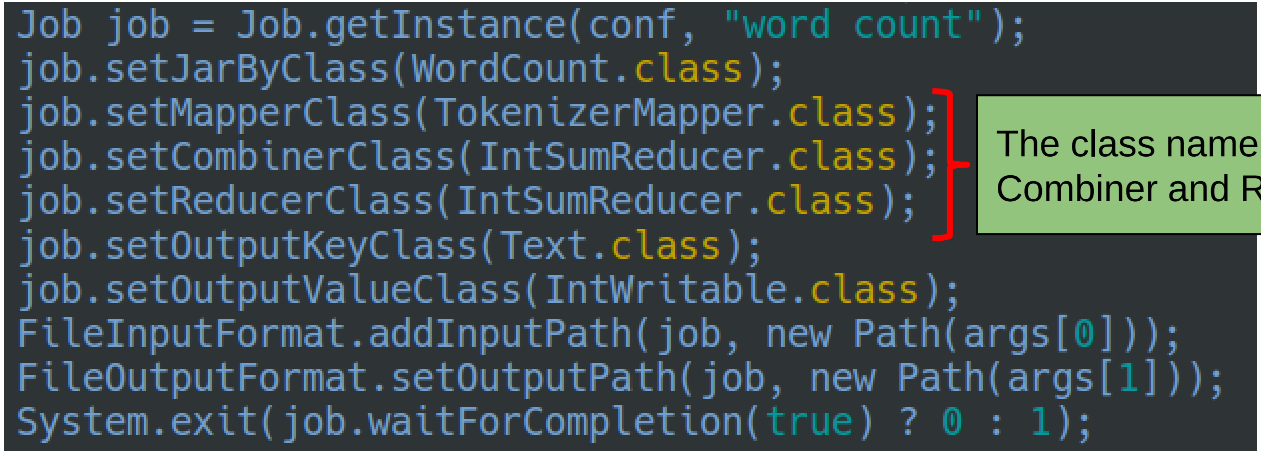


```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job,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

Should be the same as the name of public class

# MapReduce Programming

- Configure Job

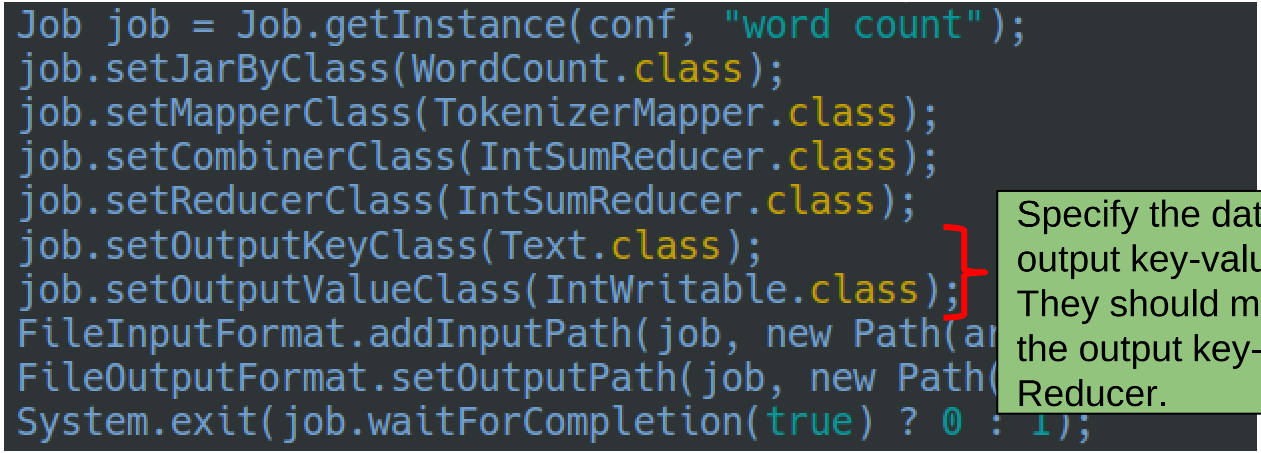


```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job,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

The class name of Mapper, Combiner and Reducer

# MapReduce Programming

- Configure Job

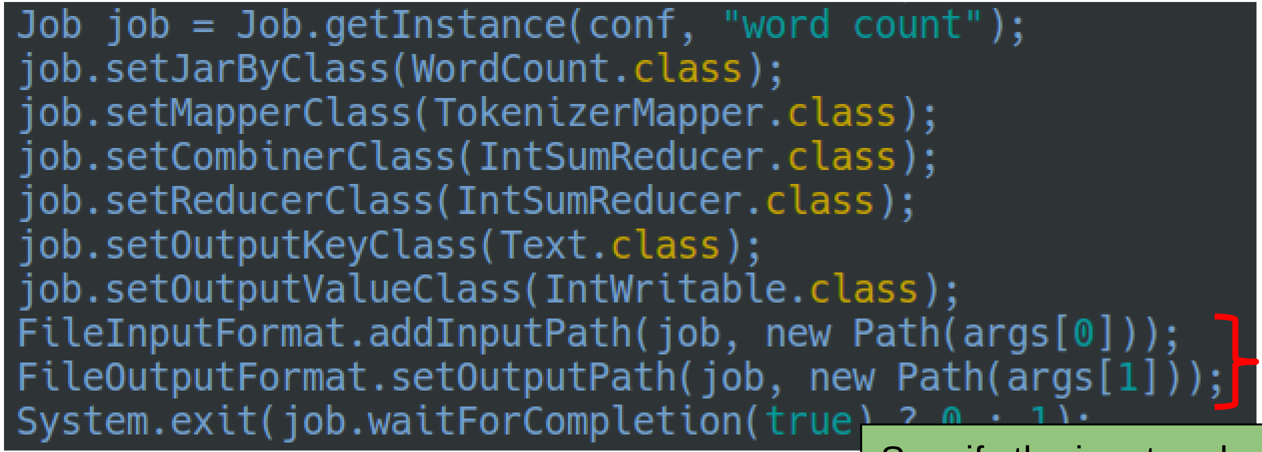


```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(ar
FileOutputFormat.setOutputPath(job, new Path(
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Specify the data types of the output key-value pair.
They should match with that of the output key-value pair of Reducer.

# MapReduce Programming

- Configure Job

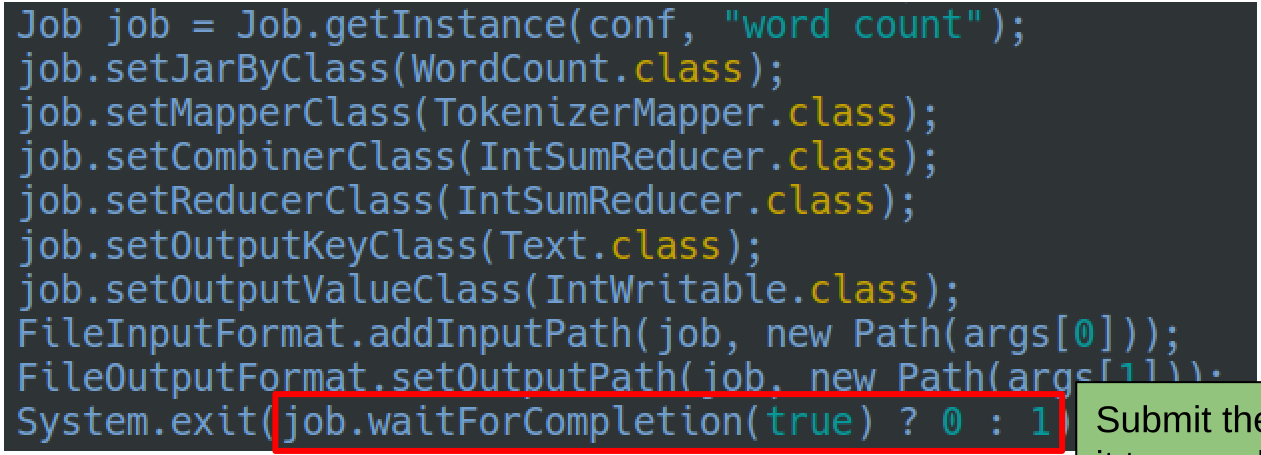


```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job,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

Specify the input and output directory. Here, we use command-line arguments to define the path.

# MapReduce Programming

- Configure Job



```
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)
```

Submit the job and wait for it to complete

# MapReduce Programming

- **Notes**
  - If you write codes in VMs, please **backup** your code
  - Check the assignment specifications very carefully, especially the **file names, input / output formats**
    - Make sure we can compile and run your codes, and double check the format of your program output

# Q & A