

# Developing a Simple Map-Reduce Program for Hadoop

---

The University of Texas at Dallas

---

Big Data Course CS6350

Professor: Dr. Latifur Khan

TA: M Solaimani([mxs121731@utdallas.edu](mailto:mxs121731@utdallas.edu))

Release Date: Fall 2016

Content courtesy  
of

~~Gladebo~~ Ayode

## Introduction

The purpose of this document is to help those students who are not familiar with Hadoop to develop their first Map-Reduce program for Hadoop.

## So far from HW-0 :

So far from HW-0 we have a hadoop cluster in our machine and we know how to run a jar. But next questions comes in is -

- **How to write a map-reduce program ?**
- **How to get the jar of the map-reduce program ?**
- **We will demonstrate that and explain the WordCount example code.**

## The process

We assume that you already have Hadoop on your own machine, and now you are ready to develop your first Hadoop program. This document based on Ubuntu 14.04 and Hadoop 2.6.0.

In the following, we will discuss the steps in details.

### 1. Preparing the IDE

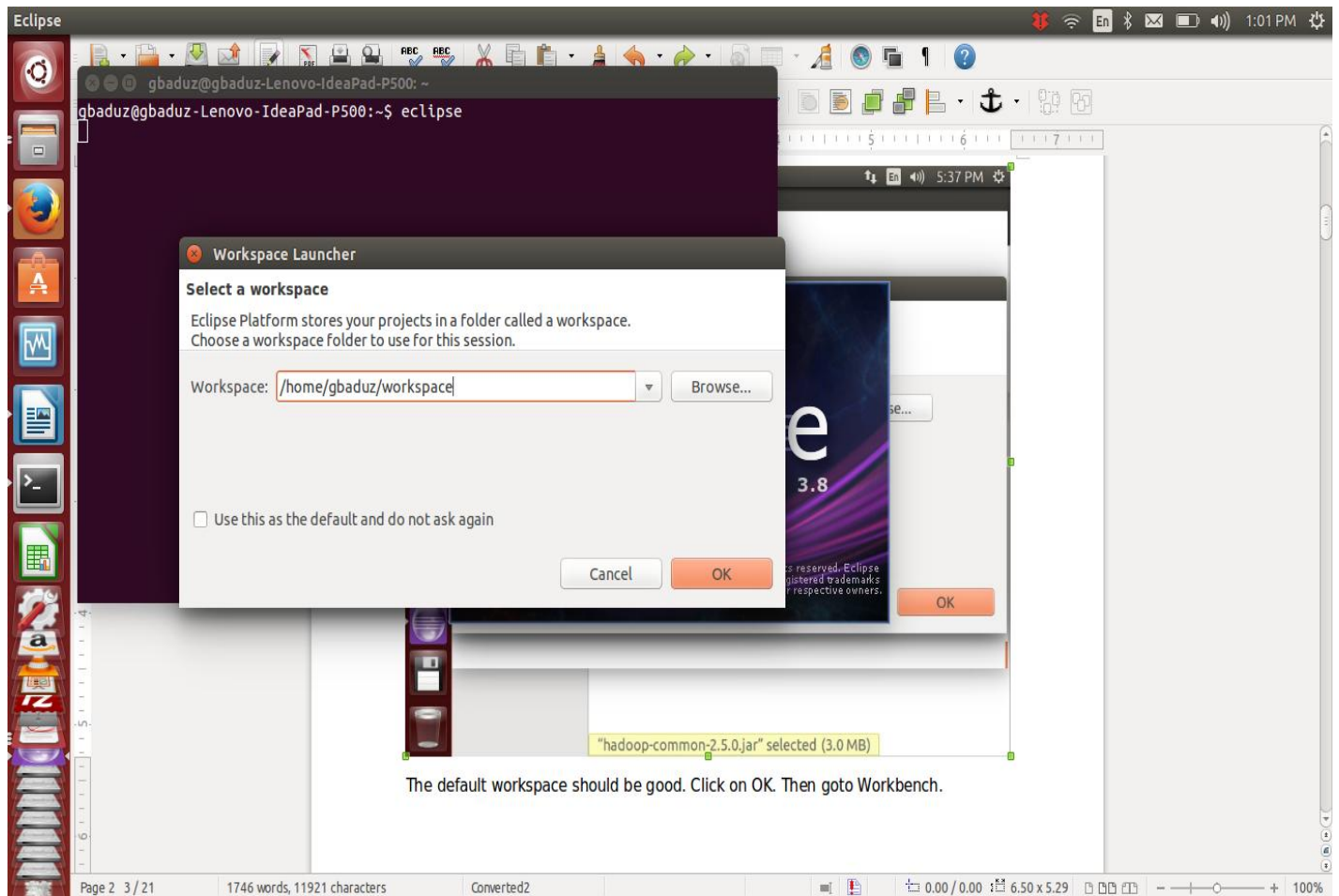
Hadoop programs are Java programs. You may use any Java IDE such as Eclipse, NetBeans, IntelliJ IDEA to develop your Map-Reduce program. We are going to use Eclipse in this document. If you have Eclipse on your own machine, you can skip this section.

To install Eclipse, you can run this command in the shell.

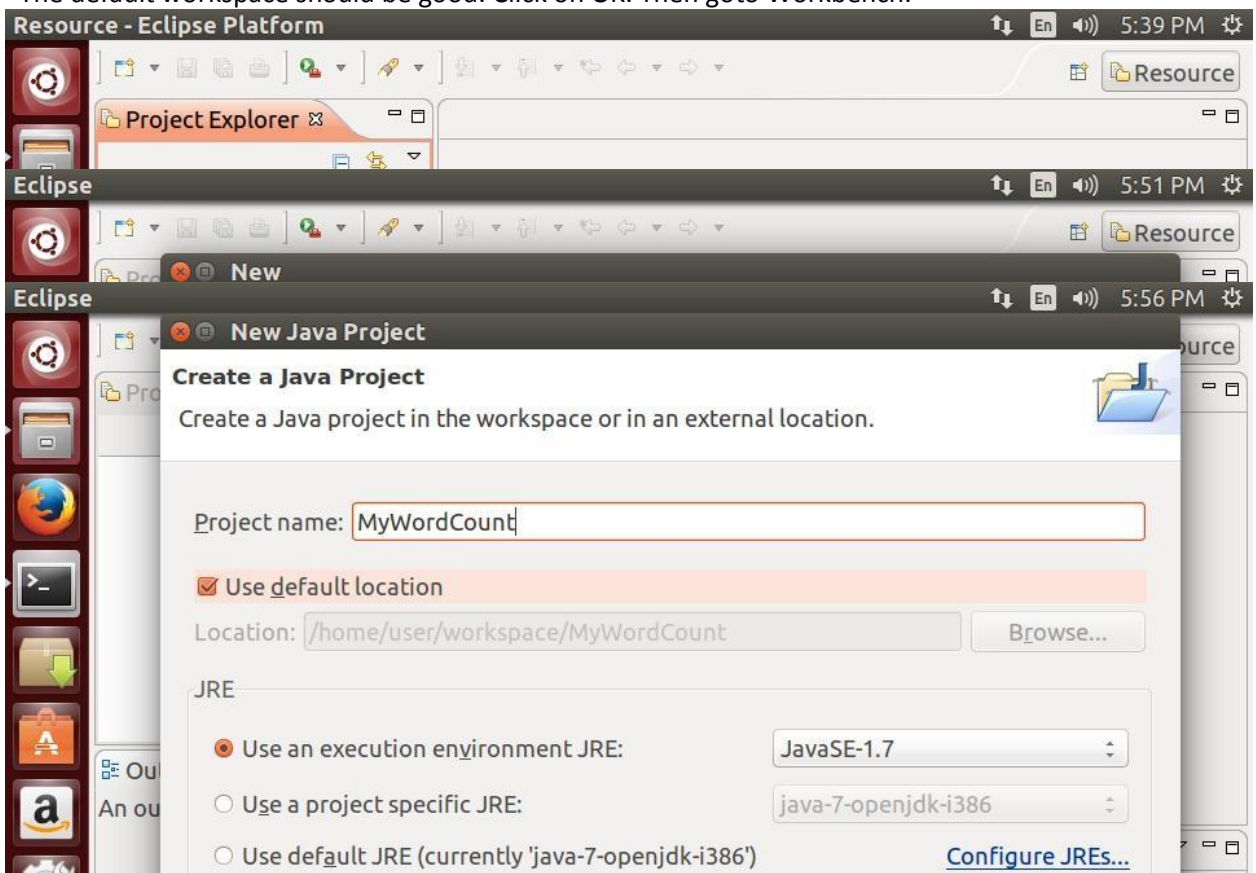
```
sudo apt-get install eclipse
```

Wait for it to be downloaded. Then use “eclipse” command to run the environment.

```
eclipse
```



The default workspace should be good. Click on OK. Then goto Workbench.



```

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.Mapper.Context;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

    public static class Map
        extends Mapper<LongWritable, Text, Text, IntWritable>{

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text(); // type of output key

        public void map(LongWritable key, Text value, Context context
            ) throws IOException, InterruptedException {
            String[] mydata = value.toString().split(" ");
            for (String data : mydata) {
                word.set(data); // set word as each input keyword
                context.write(word, one); // create a pair <keyword, 1>
            }
        }
    }

    public static class Reduce
        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; // initialize the sum for each keyword
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);

            context.write(key, result); // create a pair <keyword, number of occurrences>
        }
    }
}

```

```

    }
}

// Driver program
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    // get all args
    if (otherArgs.length != 2) {
        System.err.println("Usage: WordCount <in> <out>");
        System.exit(2);
    }

    // create a job with name "wordcount"
    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    // uncomment the following line to add the Combiner job.setCombinerClass(Reduce.class);

    // set output key type
    job.setOutputKeyClass(Text.class);
    // set output value type
    job.setOutputValueClass(IntWritable.class);
    //set the HDFS path of the input data
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    // set the HDFS path for the output
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

    //Wait till job completion
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

#### 4. Please download hadoop to your development machine

Please download hadoop to your development machine. This is required to get the dependent jar files for hadoop compilation.

<https://archive.apache.org/dist/hadoop/core/hadoop-2.4.1/hadoop-2.4.1.tar.gz>

Untar it

```
tar xvfz hadoop-2.4.1.tar.gz
```

## 5. Adding Hadoop reference (Very important)

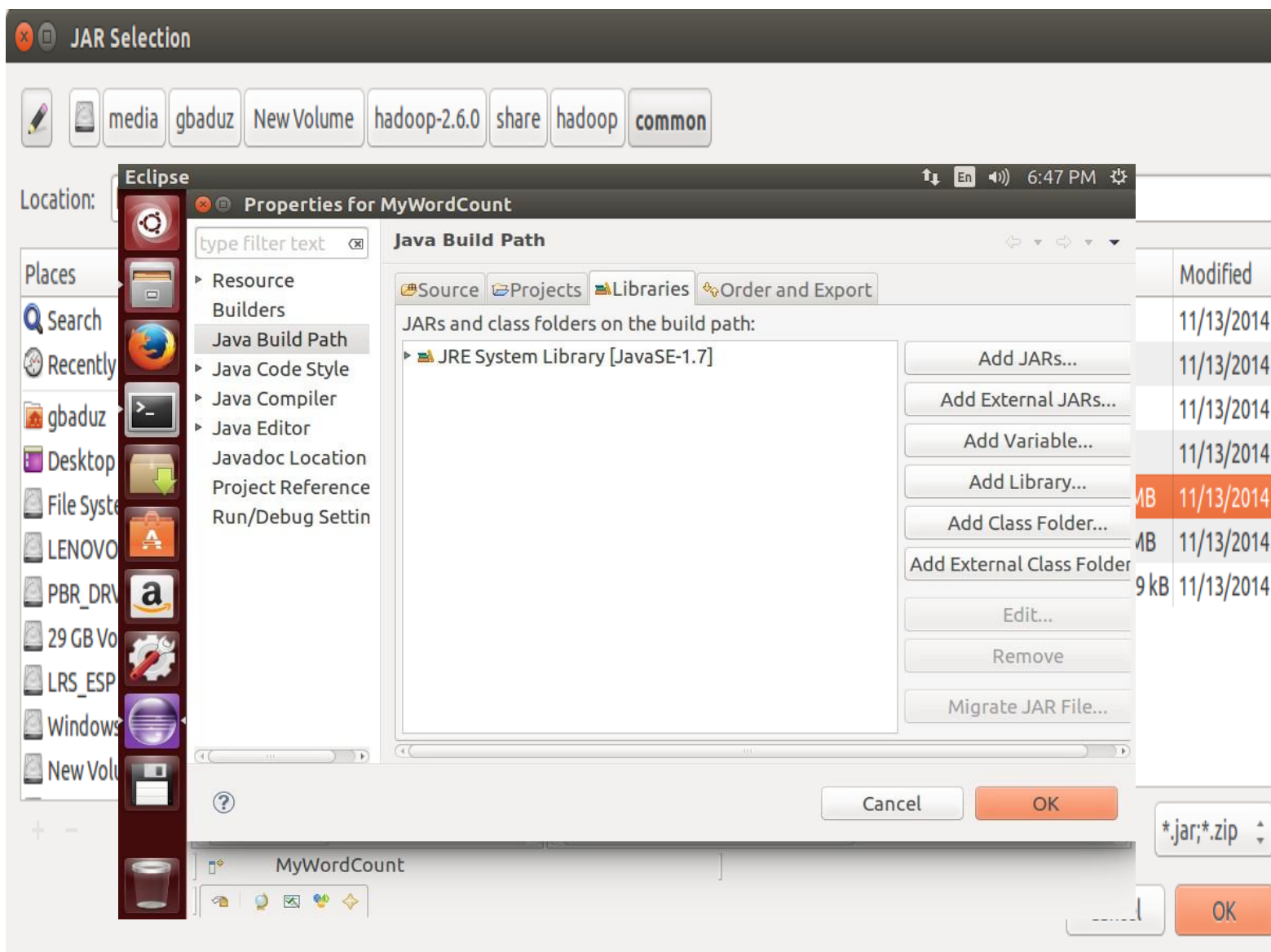
In order to compile Hadoop projects, you need to add Hadoop library as a reference to your projects. Right

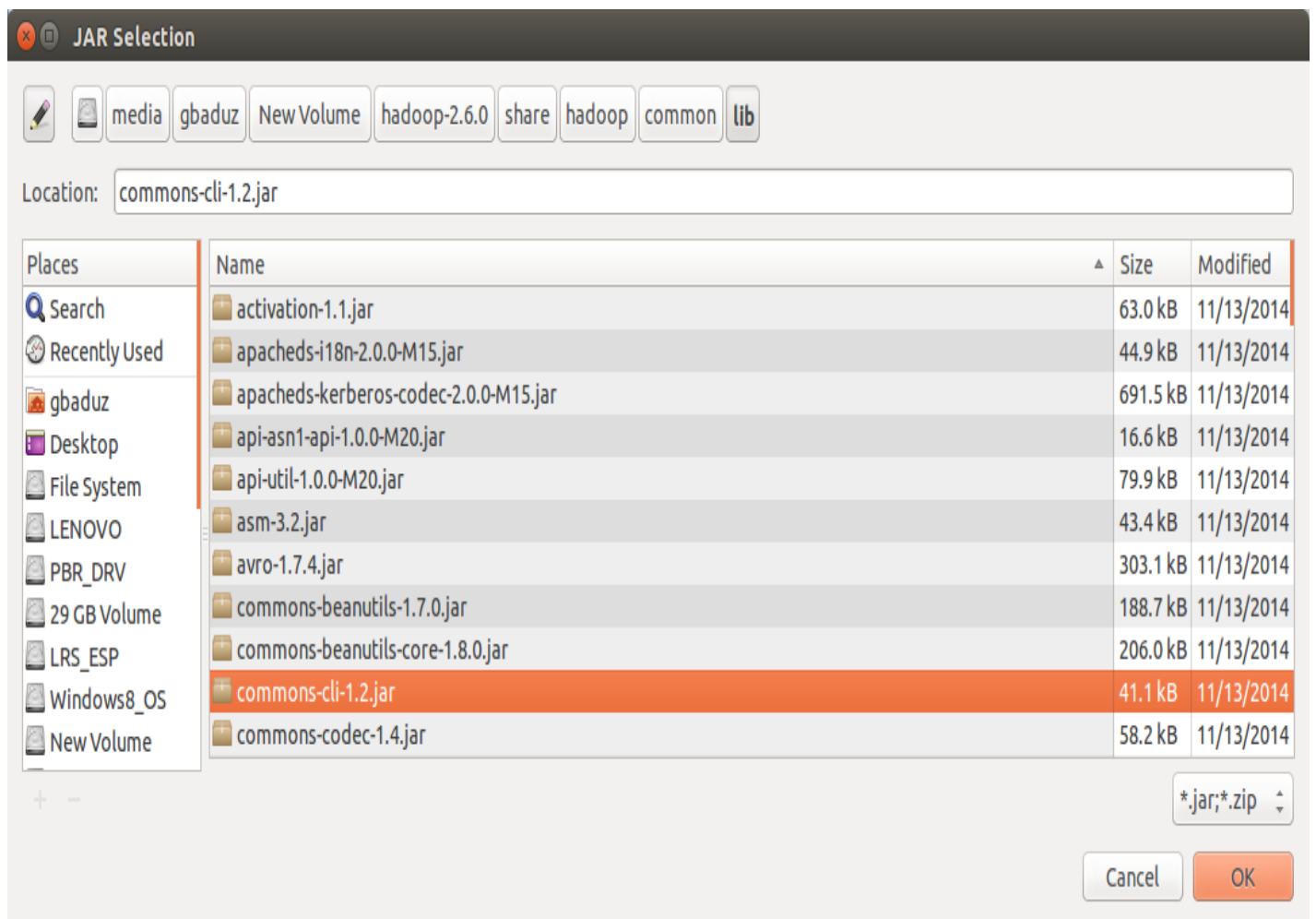
click on the project. Select "Build Path" -> "Configure Build Paths", select "Libraries" tab.

Click on "Add External JARs..." to continue. Find "hadoop-mapreduce-client-core-2.4.1.jar" in <Your hadoop folder>/share/hadoop/mapreduce folder, and add it.

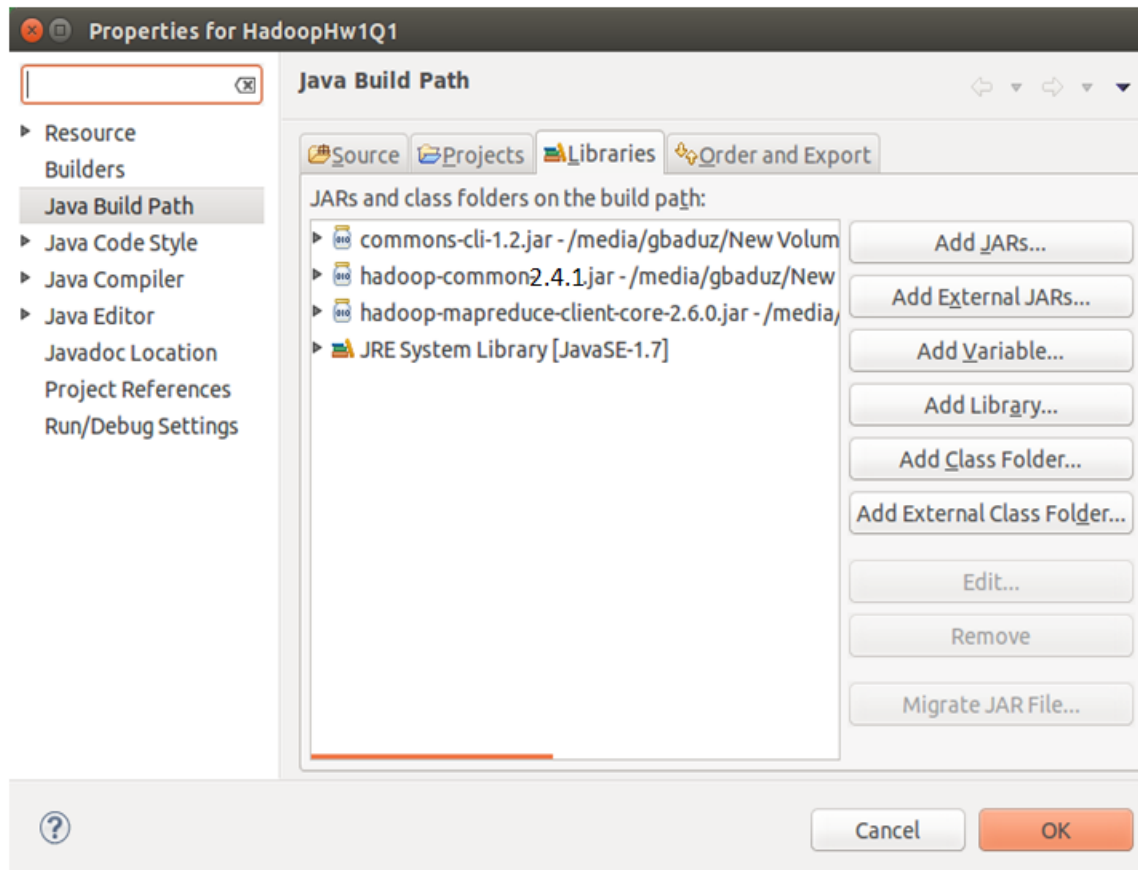
Click on "Add External JARs..." again. Find "hadoop-common-2.4.1.jar" in <Your hadoop folder>/share/hadoop/common folder, and add it.

You need also add "commons-cli-1.2.jar" in the folder <Your hadoop folder>/share/hadoop/common/lib folder.





Your build path configuration should be similar to this screen now:



## 6. Creating the JAR file for Hadoop

All you need to do now is to create the JAR file and run it in Hadoop. Right click on the project, and choose "export".

Then use "Browse..." button in front of the "JAR file:" label to specify the name of the export file.



For example, you may use `"/home/user/WordCountSample/wordcount.jar"` (you can use any other path)

Now, it should be two files inside WordCountSample folder:

## 7. Executing the example in Hadoop cluster.

You can login to **cs6360.utdallas.edu**, if you are on campus or you are using VPN.

If you are outside campus, first connect to **csgrads1.utdallas.edu**, then you can connect to **cs6360.utdallas.edu**.

Your username is your netid and use your school email password.

### 7.1 Copy the wordcount.jar to the cs6360 hadoop cluster.

**Change directory to where you exported the wordcount jar.**

Use the `scp` command to copy the jar file from your development machine to hadoop cluster.( Windows users can download **winscp** to load files to the hadoop cluster).

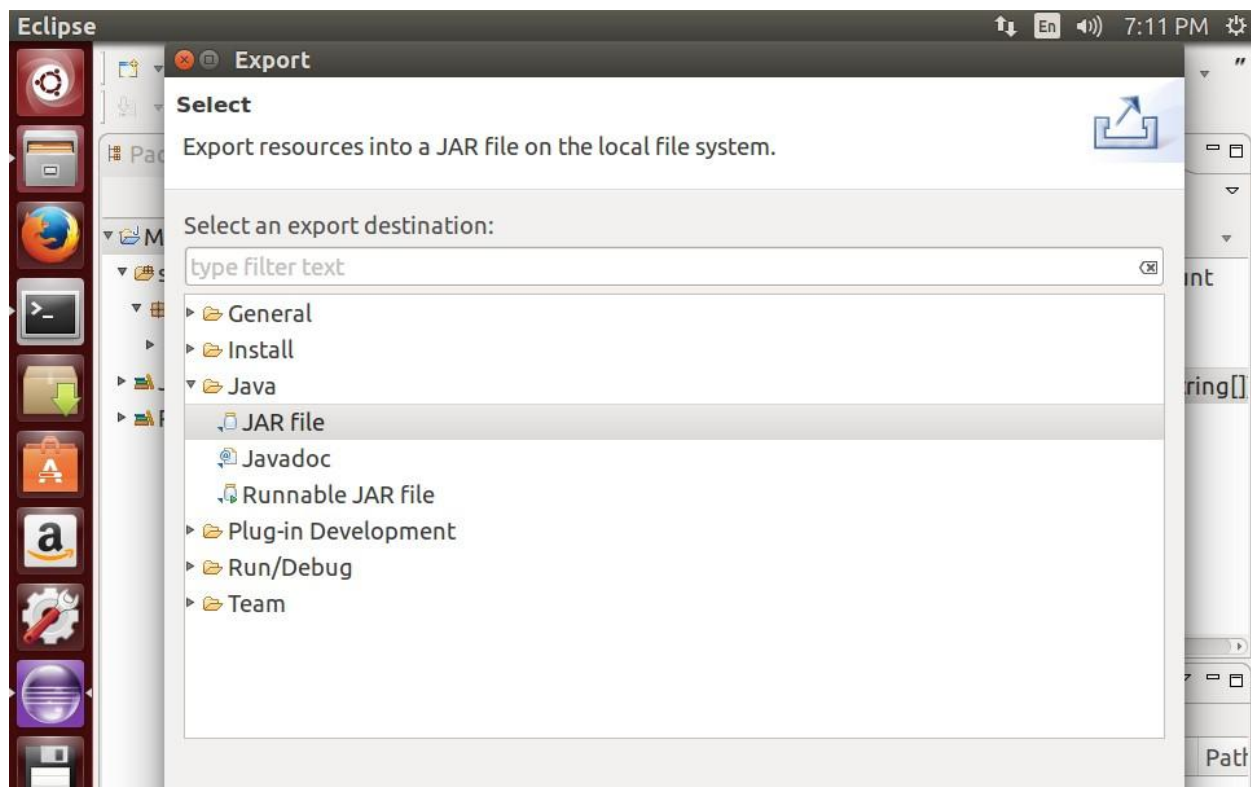
Linux users can use **filezilla**.

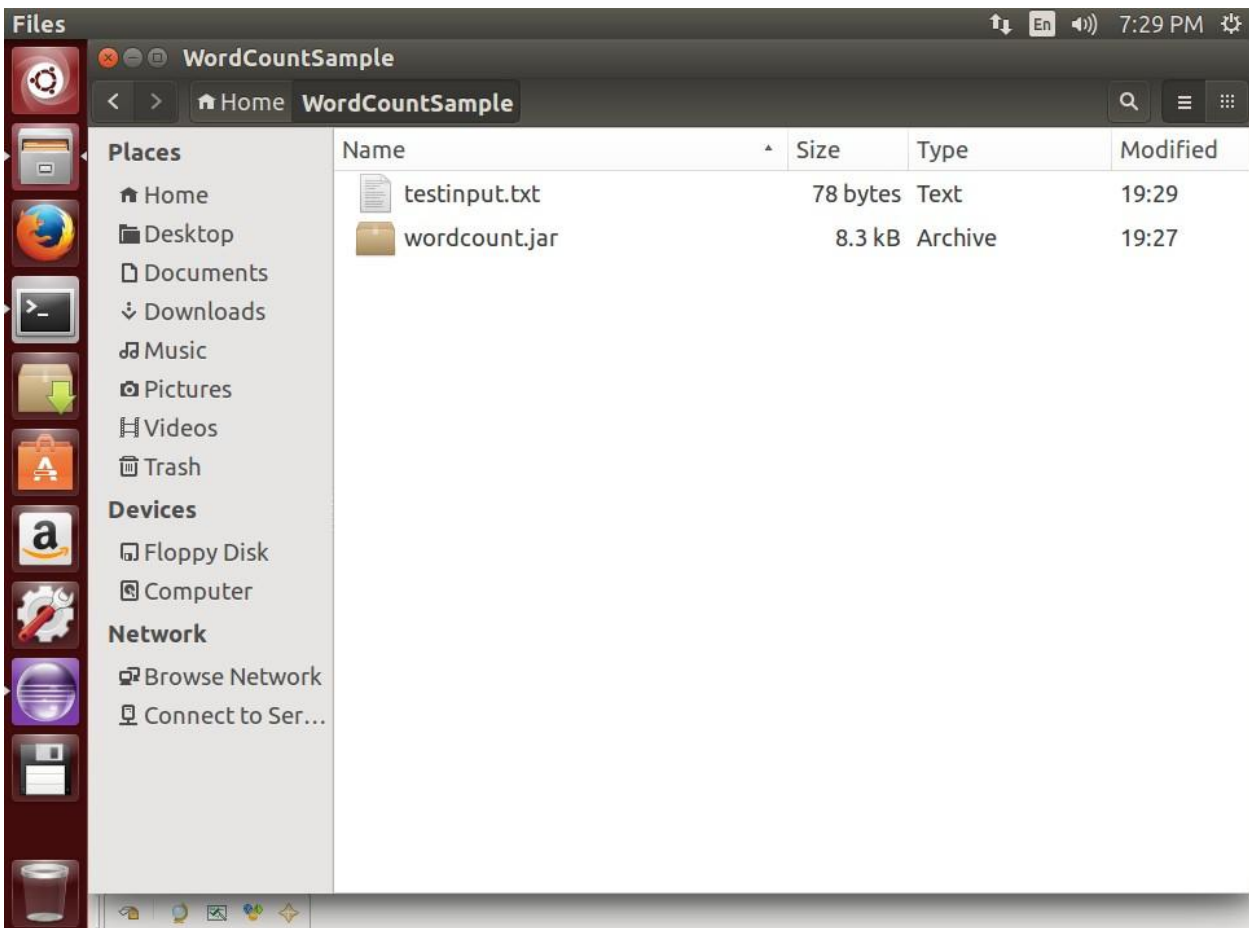
**`"scp wordcount.jar yournetid@csgrads1.utdallas.edu"`**

**enter your password.**

**Please ensure you are in cs6360.utdallas.edu, before running hadoop commands.**

**Also note that any file you copy to csgrads1.utdallas.edu is visible in the same location in cs6360.**





**Removing old folders:** (if you run the program again without deleting the previous output folder, You will get the error :: "Exception in thread "main" org.apache.hadoop.mapred.**FileAlreadyExistsException**: Output directory /user/hue/output already exists)"

So lets delete if there is any same output directory already. \*\*\* You do not need to delete input directory but it is shown here as well.

**hdfs dfs -rmr output**

**In CS6360, Upload your input file** to the input folder of Hadoop: (IF you have more than 1 input file

**hdfs dfs -put anytextfile /yournetid/input**

**In cs6360**

running Hadoop ::

**hadoop jar <Address of the jar> <ClassName> <location of input> <location of output>**

**hadoop jar WordCount.jar WordCount input o u t p u t**

If you get error **java.lang.ClassNotFoundException**: If needed use : packageName.WordCount **OR** you might have forgotten to mention the class name , **OR** if you create your jar from Netbeans **avoid the <ClassName>** as class name is internally mapped in manifest file.

**Viewing the results:**

After you run the program successfully - there will be part file generated inside that output directory that you mentioned in the above command. The part file name is part-r-00000 : You can cat this file and see the output.

**hdfs dfs -cat /yournetid/output/\***

**hdfs dfs -cat /yournetid/output/part-r-00000**

Output should be similar to this ::

bye, 1  
earth, 1  
hello, 2  
new, 1  
world, 3

## Walkthrough the code again ::

\*\*\*We will assume that, the input files goes to **two different mappers**. We will explain based on that assumption.

### MAPPER ::

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();    // type of output key

    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());    // set word as each input keyword
            context.write(word, one);    // create a pair <keyword, 1>
        }
    }
}
```

- **WHAT is actually going here ???**

- **How the above code segment generate this intermediate output ??**

- what is this `LongWritable, Text, Text, IntWritable` ?? NEW DATA TYPE ??

- Map class implements a public map method, that processes one line at a time from input data chunk and

- splits each line into tokens separated by whitespaces.

- It emits a key-value pair of `< <word>, 1>`, written to the Context.

1<sup>st</sup> map emits:

< hello, 1>  
< world, 1>  
< bye, 1>  
< world, 1>

2<sup>nd</sup> map emits:

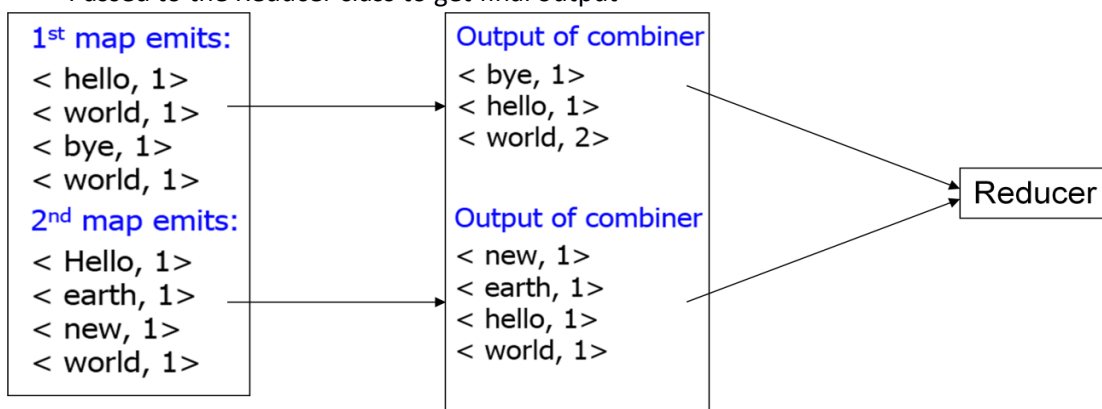
< Hello, 1>  
< earth, 1>  
< new, 1>  
< world, 1>

**Context object**: allows the Mapper to interact with the rest of the Hadoop system

Includes configuration data for the job as well as interfaces which allow it to emit output

**COMBINER ::** (not mandatory - depends on requirement)

- same as Reducer here
- As we used combiner here, So there would be a **local sum** :: Framework groups all intermediate values associated with a given output key
- Passed to the Reducer class to get final output



## REDUCER ::

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

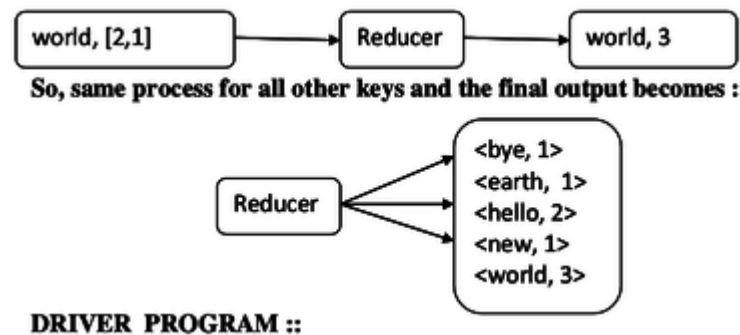
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {

        int sum = 0; // initialize the sum for each keyword
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result); // create a pair <keyword, number of occurrences>
    }
}
```

### Reducer.reduce( ) ::

- The intermediate key-value pair gets grouped by based on the key and each group comes at a time to a reducer
- Called once per key
- Passed in an **Iterable** which returns all values associated with that key and here we sum all those values.
- Emits output with Context.write()
-

lets think of the token "world" : 1st mapper emits <world,2> and 2nd mapper emits <world,1> : so they gets grouped by key "world" and their iterable values are : [2,1] , So look like this :



```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    // get all args
    if (otherArgs.length != 2) {
        System.err.println("Usage: WordCount <in> <out>");
        System.exit(2);
    }

    // create a job with name "wordcount"
    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    // OPTIONAL :: uncomment the following line to add the Combiner
    // job.setCombinerClass(Reduce.class);

    // set output key type
    job.setOutputKeyClass(Text.class);
    // set output value type
    job.setOutputValueClass(IntWritable.class);

    //set the HDFS path of the input data
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    // set the HDFS path for the output
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

    //Wait till job completion
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}

```

### \*\*\* Some other basic commands :

So far we have seen, **-cat, -ls, -rmr, -put** commands in our demonstration. Though we will use these commands a lot, but there are many other commands available. Please follow the link for other commands ::

<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html>

One thing NOT to get confused ::

When you type the command "**ls**" in unix terminal - you will find the list of files in your unix directory

BUT when you type "**hdfs dfs -ls <dir\_name>**" - you will see the list of files in the HDFS dir not in unix - please do not get confused - HDFS and Unix directory are different

## Summary

In this document, we described how develop and run a simple Map-Reduce program for Hadoop.