Developing a Simple Map-Reduce Program for Hadoop

The University of Texas at Dallas

Big Data Course CS6350

Professor: Dr. Latifur Khan

TA: Gbadebo Ayoade(gga110020@utdallas.edu)

Release Date: Spring 2015

Content courtesy
of

Mohammad Ridwanur Rahman Mohammad Ali Ghaderi

Revised by Gbadebo Ayoade

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

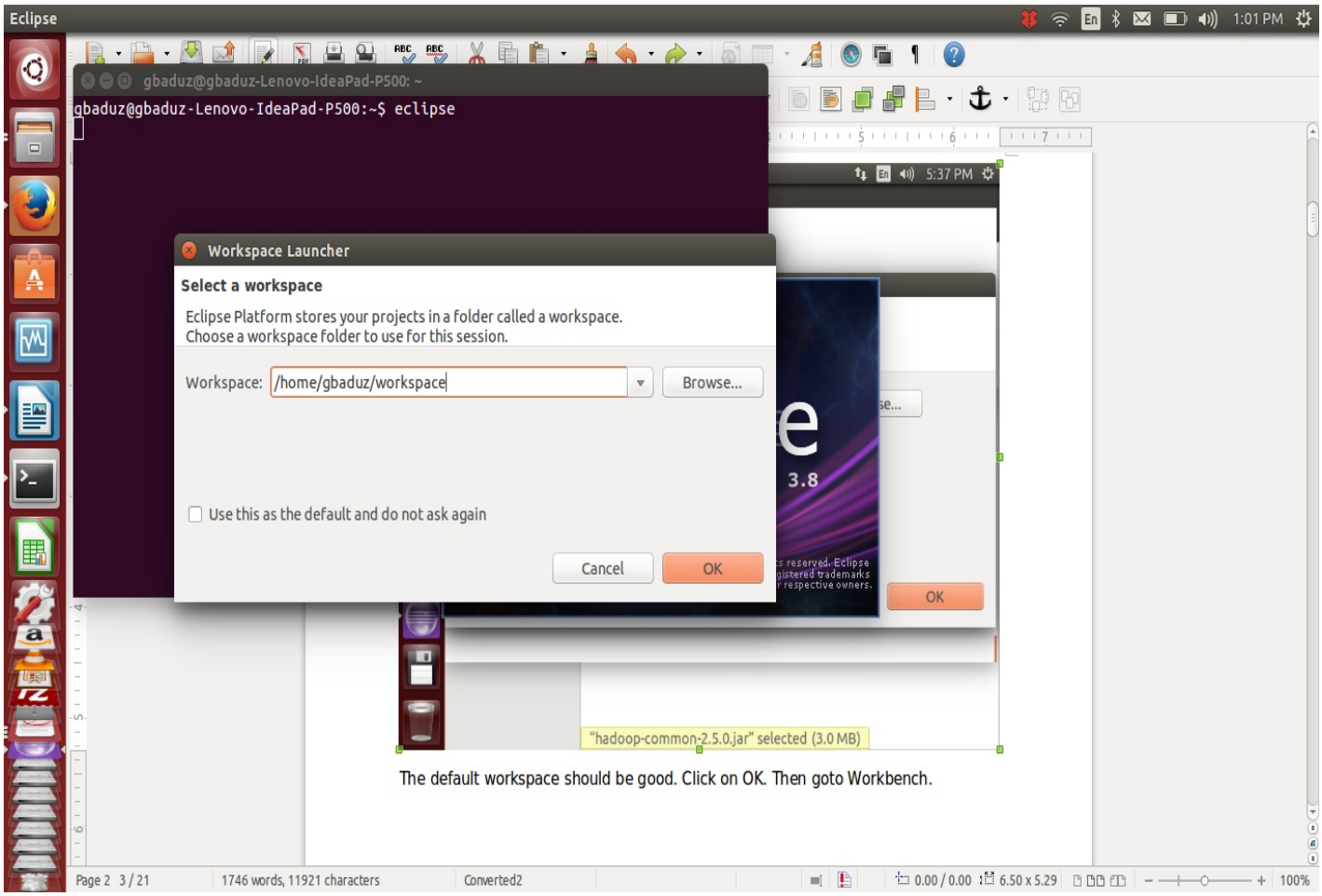
Hadoop programs are Java programs. You may use any Java IDE such as Eclipse, NetBeans, IntelliJ IDEA to develop your Map-Reduce program. We are going to use Eclipse in this document. If you have Eclipse on your own machine, you can skip this section.

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

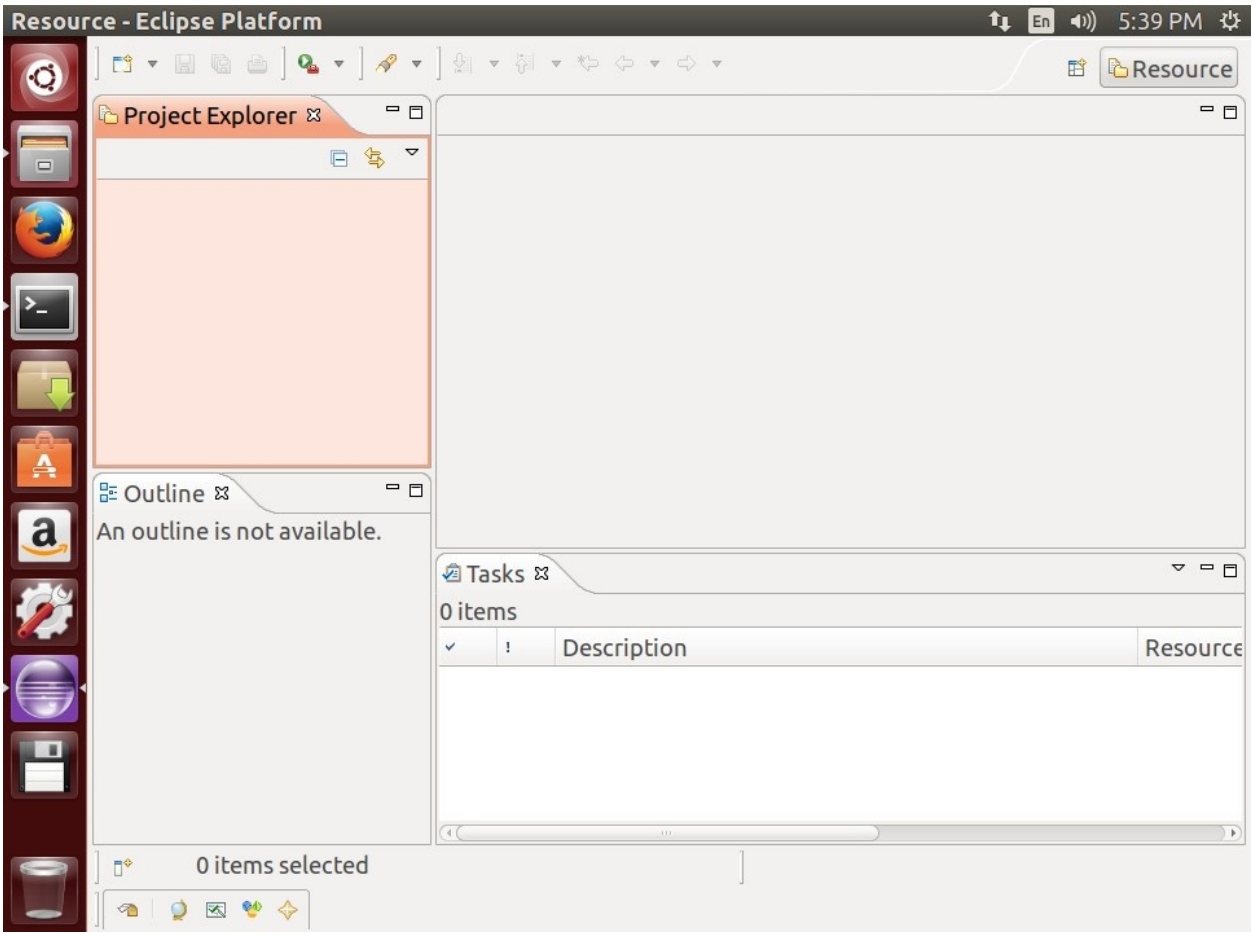
```
sudo apt-get install eclipse
```

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

eclipse

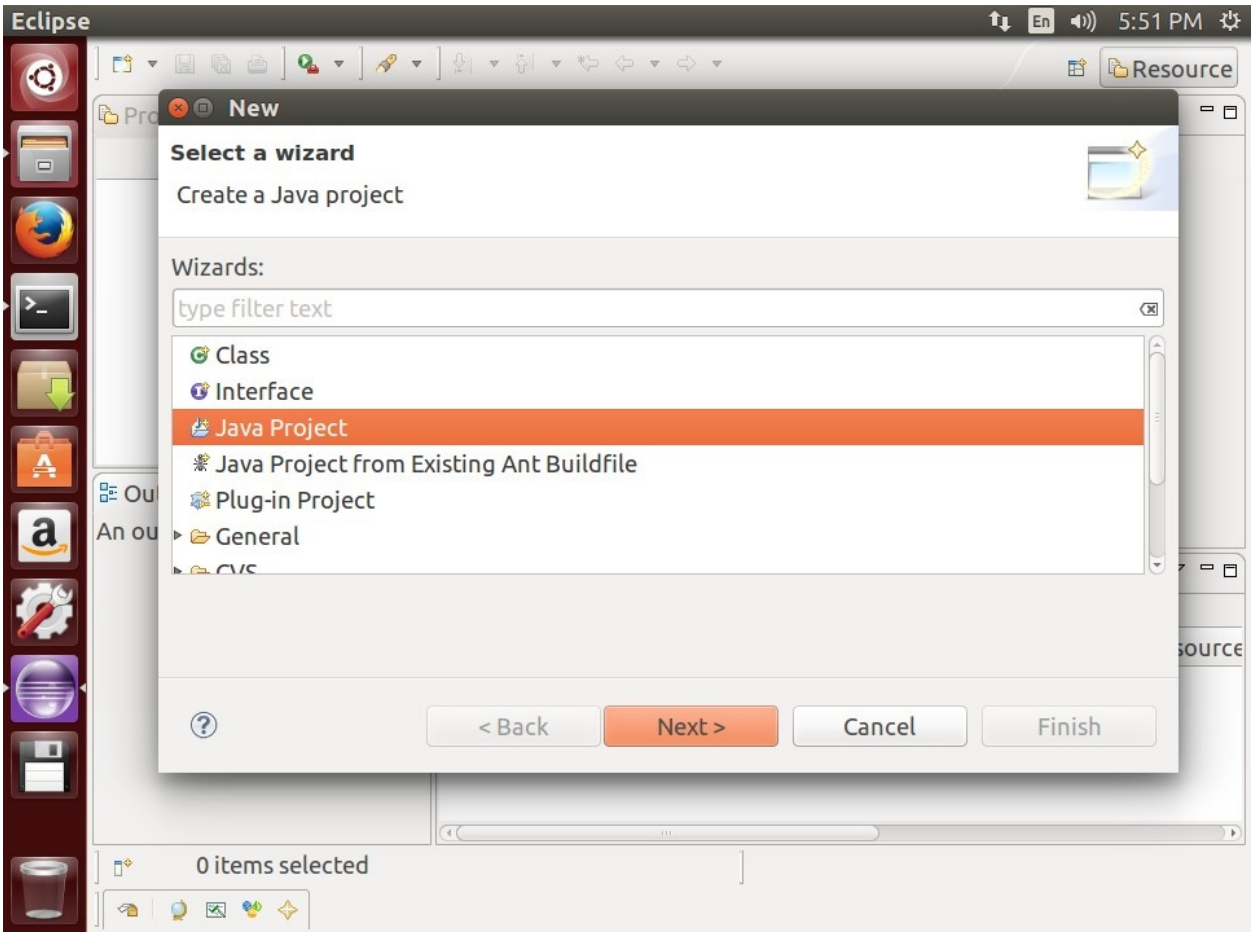


The default workspace should be good. Click on OK. Then goto Workbench.

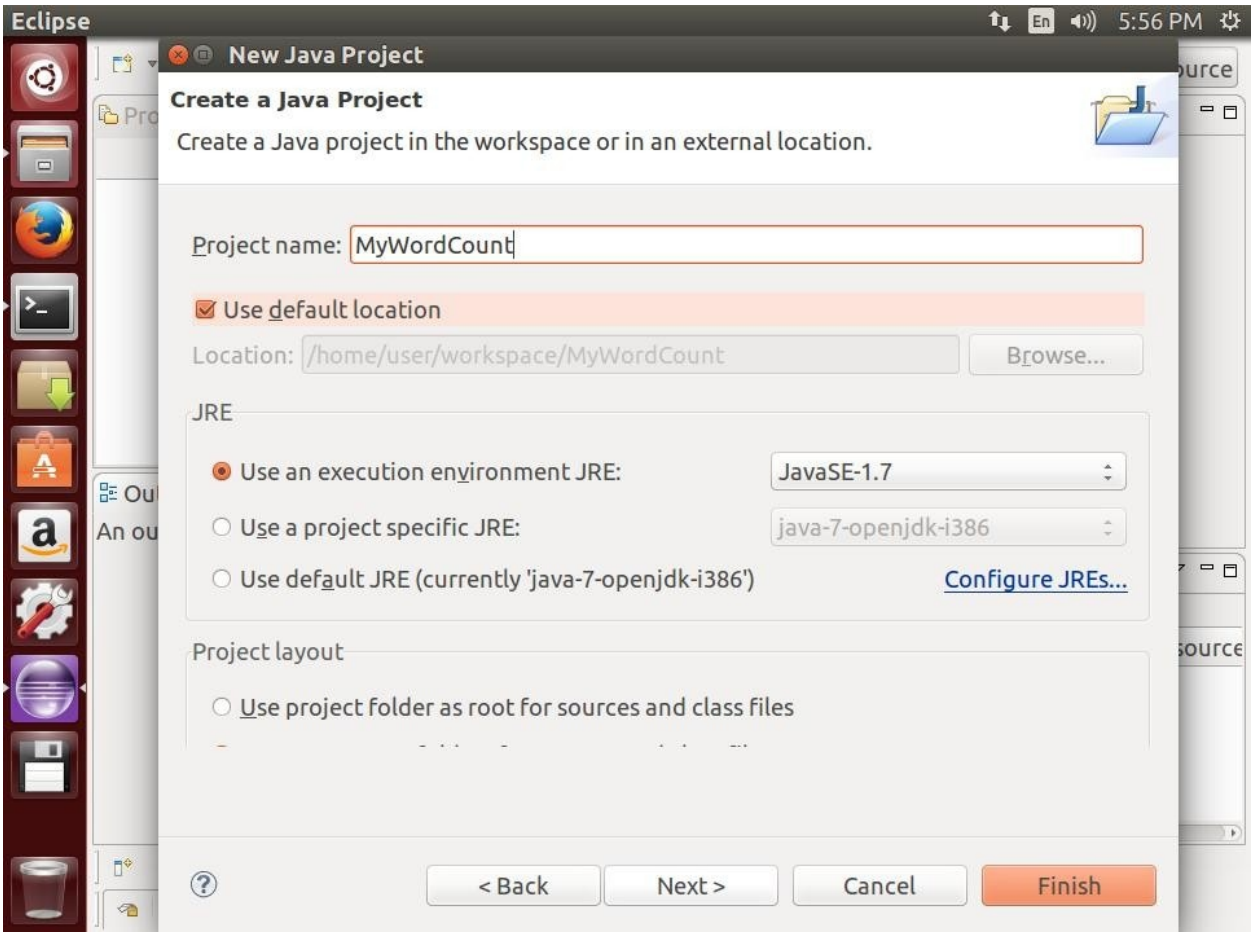


2. New Java Project

Hadoop projects are simple Java projects. Create a new Java project.



Write project name as “MyWordCount” and click on Finish to create the project.



3. Creating main file

Create a new file named “WordCount.java” and write the following lines there:

```
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.*;
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;
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 {
      StringTokenizer itr = new StringTokenizer(value.toString()); // line to
string token

      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());    // 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 occurences>
    }
  }

  // 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);
  }
}
```

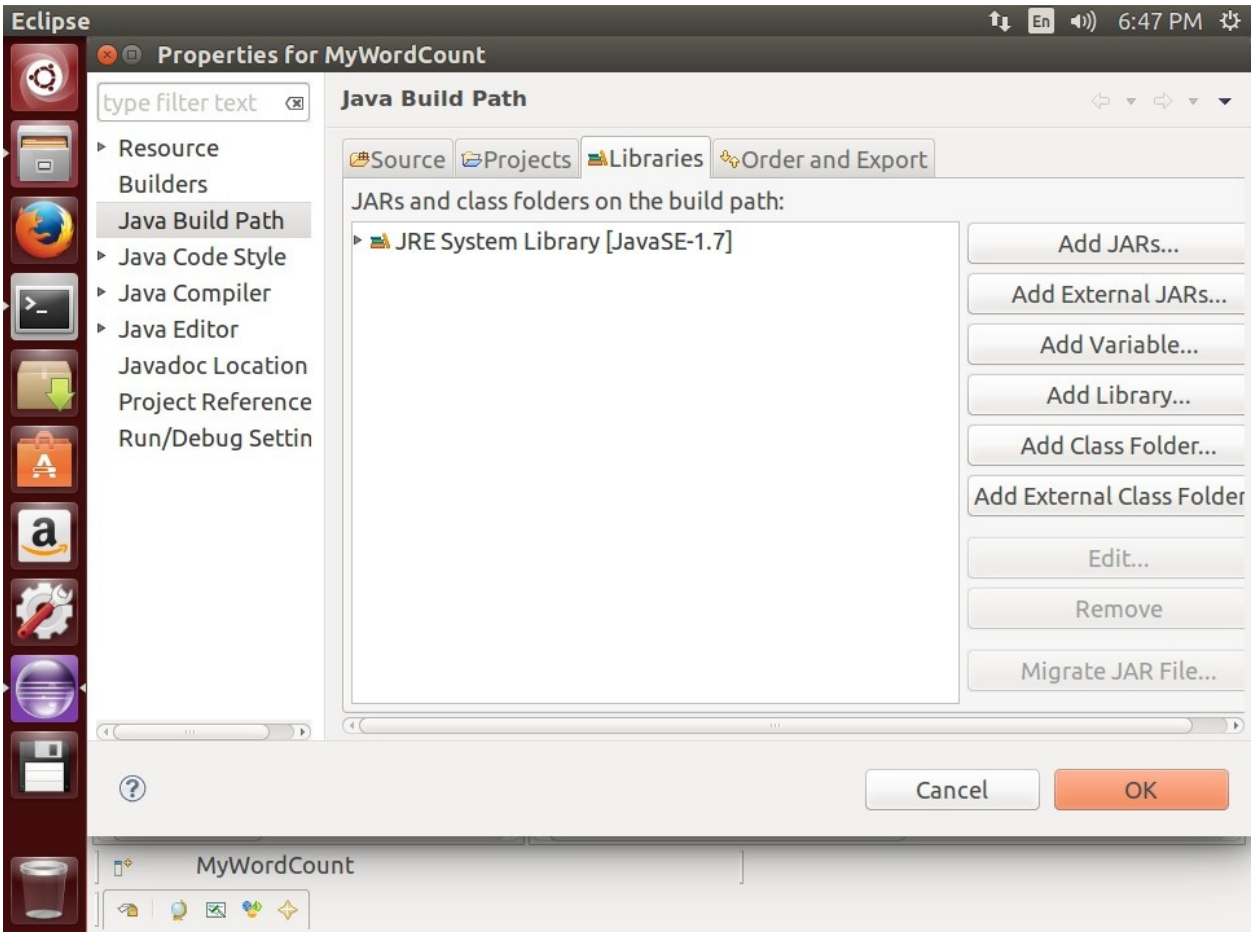
4. Please download hadoop to your development machine

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

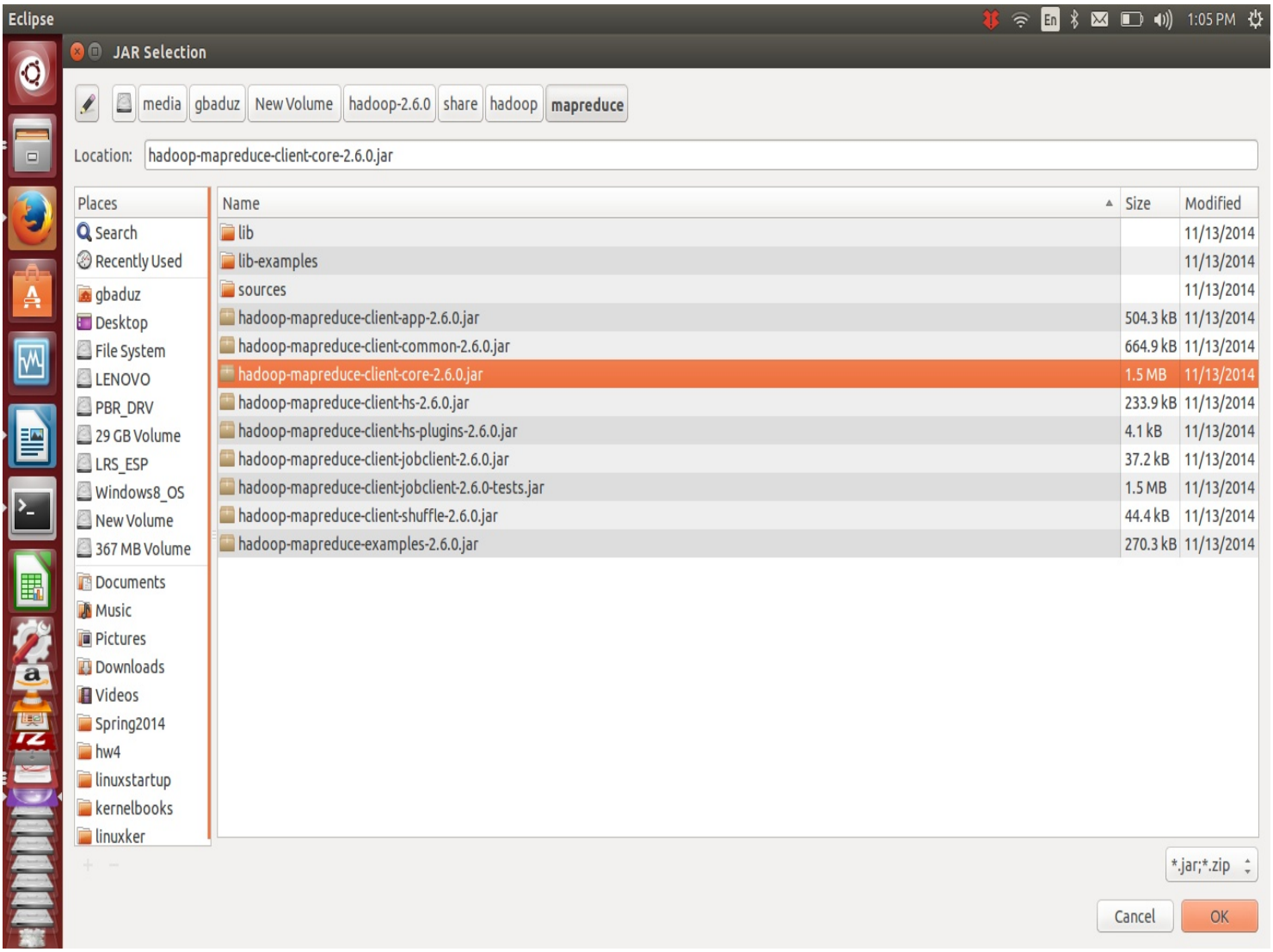
http://mirror.tcpdiag.net/apache/hadoop/common/hadoop-2.6.0/hadoop-2.6.0.tar.gz

5. Adding Hadoop reference (Very important)

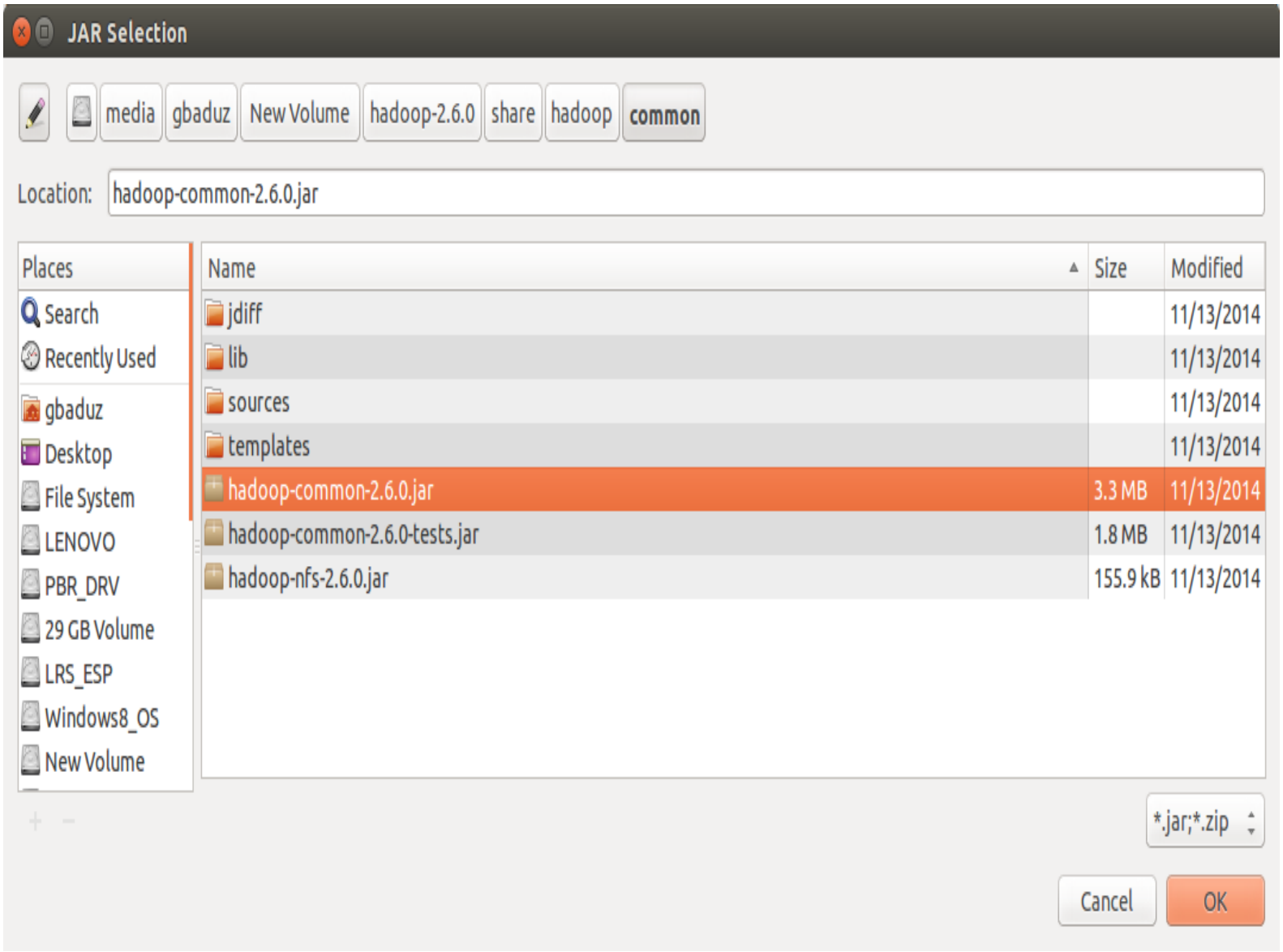
In order to compile Hadoop projects, you need to add Hadoop library as a reference to your projects. Right click on the project. Select “Build Path” -> “Configure Build Paths”, select “Libraries” tab.



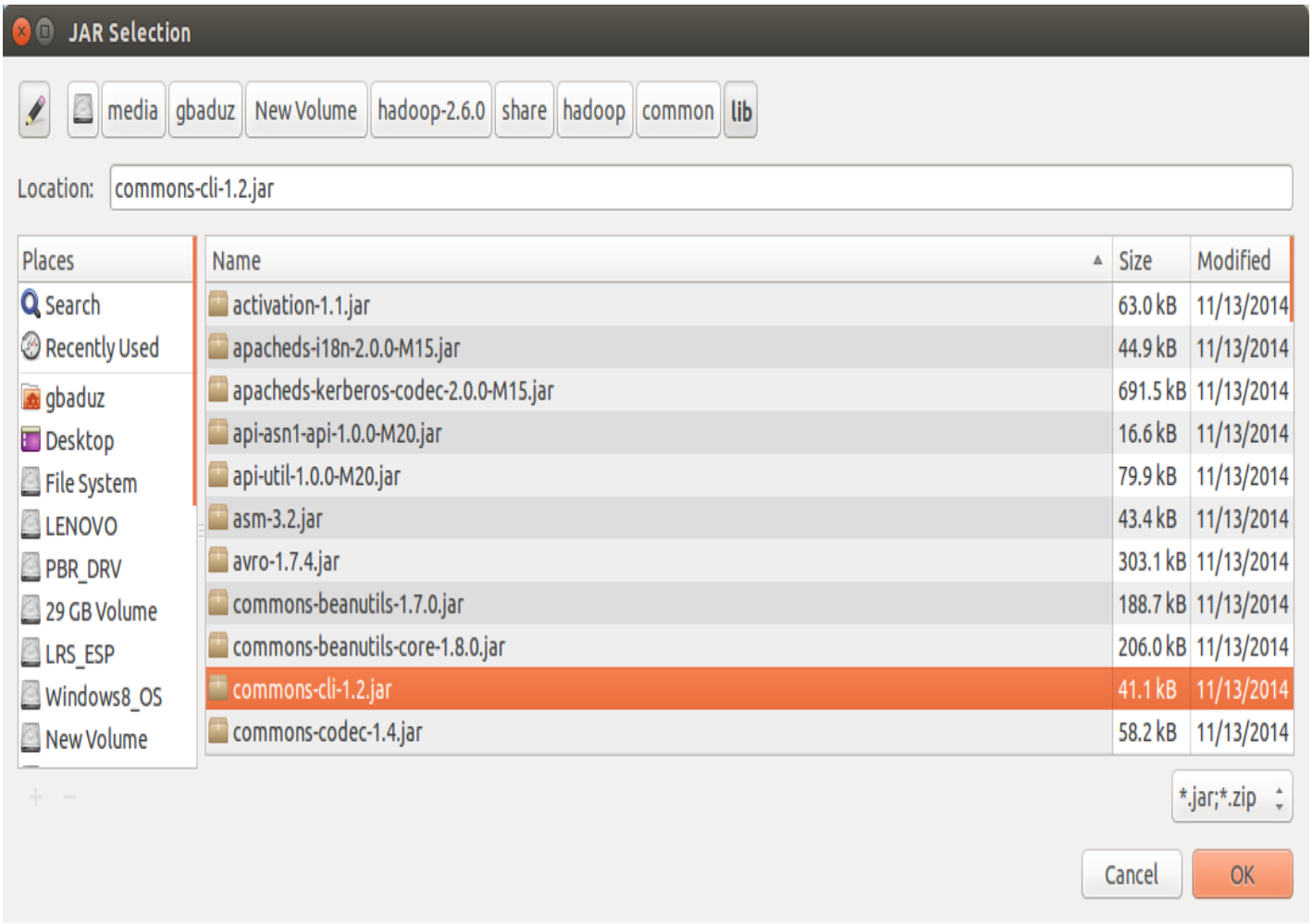
Click on “Add External JARs…” to continue. Find “hadoop-mapreduce-client-core-2.6.0.jar” in
<Your hadoop folder>/share/hadoop/mapreduce folder, and add it.



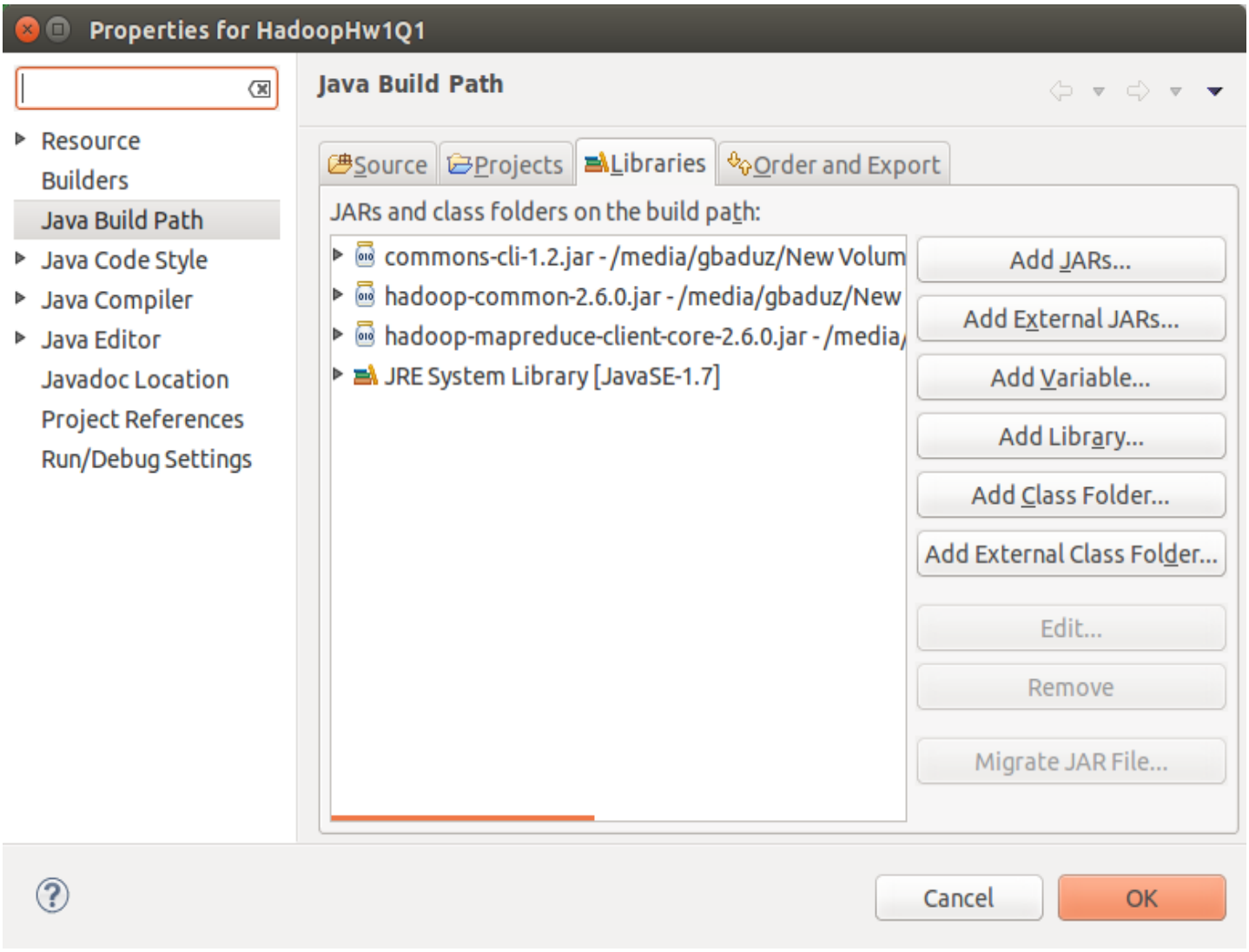
Click on “Add External JARs...”again. Find “hadoop-common-2.6.0.jar” in <Your hadoop folder>/share/hadoop/common folder, and add it.



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

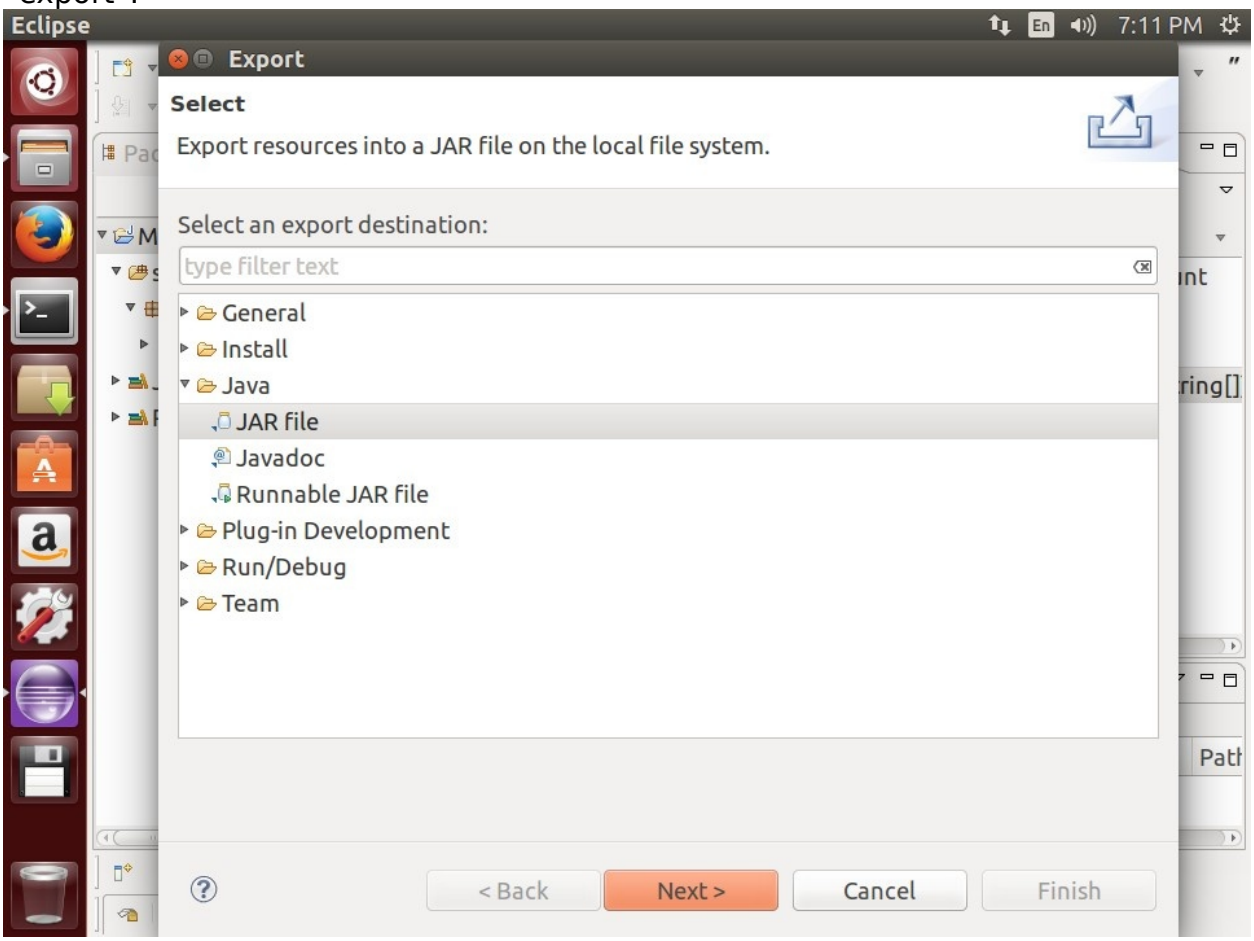


Your build path configuration should be similar to this screen now:


Properties for HadoopHw1Q1
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Run/Debug Settings
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
commons-cli-1.2.jar - /media/gbaduz/New Volum
hadoop-common-2.6.0.jar - /media/gbaduz/New
hadoop-mapreduce-client-core-2.6.0.jar - /media/
JRE System Library [JavaSE-1.7]
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
Cancel
OK

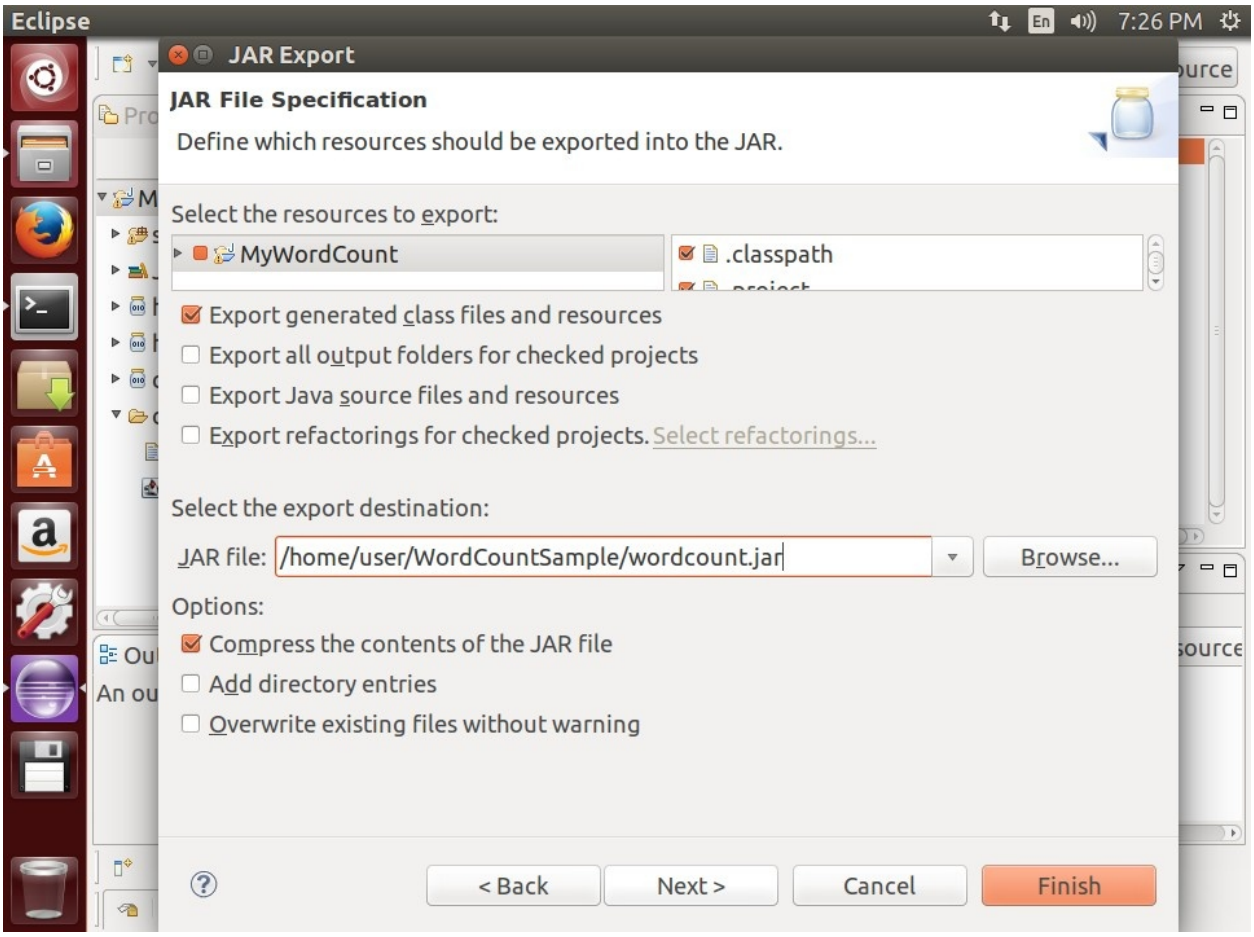
6. Creating the JAR file for Hadoop

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".

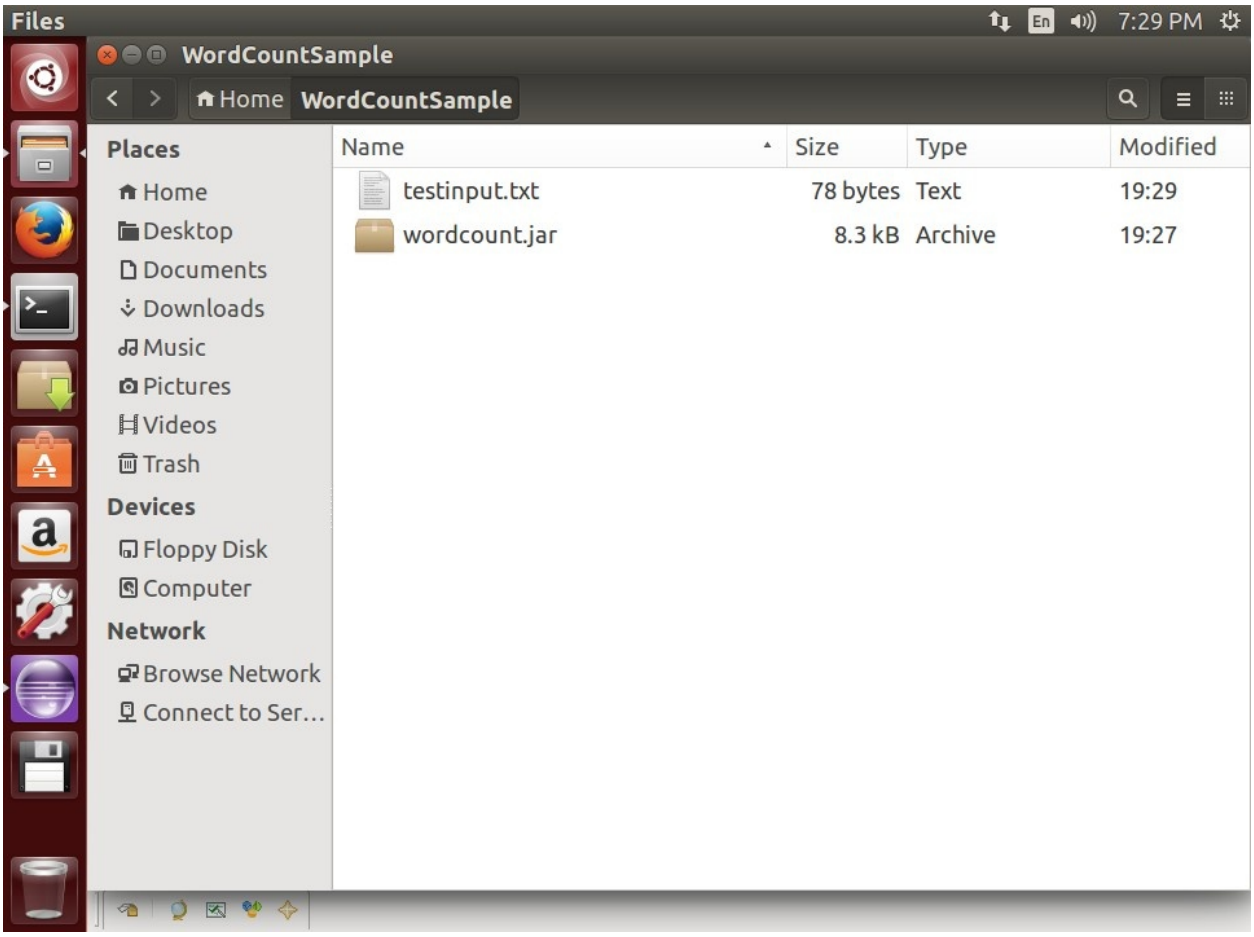


Then use "Browse…" button in front of the "JAR file:" label to specify the name of the export file.

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

Start the hortonworks VM as shown in the installation document. Ensure the VM is properly started. Get the IP of the VM following the steps in the installation document.

I am using the NAT configuration for the VM network so my IP is 127.0.0.1

7.1 Copy the wordcount.jar to the hortonworks hadoop sandbox VM.

Change directory to where you exported the wordcount jar.

Use the scp command to copy the jar file from your development machine to hortonworks vm.(Windows users can download winscp to load files to the hortonworks vm)

"scp -P 2222 wordcount.jar hue@127.0.0.1:"

NB: use user : hue, password: hadoop

7.2 Login to hortonworks VM directly or you can use ssh or putty. (User should be **hue**)

To ssh from your development machine use the command below

“ssh hue@127.0.0.1 -p 2222”

password : hadoop.

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 Hortonworks VM, Upload your input file to the input folder of Hadoop:

(IF you have more than 1 input file you have to upload all input files from Unix to HDFS)

hdfs dfs -put anytextfile input

In hortonworks VM

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

hadoop jar /home/lib/hue/WordCount.jar WordCount input output

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 output/*

hdfs dfs -cat 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** ???

1st map emits:
< hello, 1>
< world, 1>
< bye, 1>
< world, 1>

2nd map emits:
< Hello, 1>
< earth, 1>
< new, 1>
< world, 1>

- **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 chuck and

- splits each line into tokens separated by whitespaces.

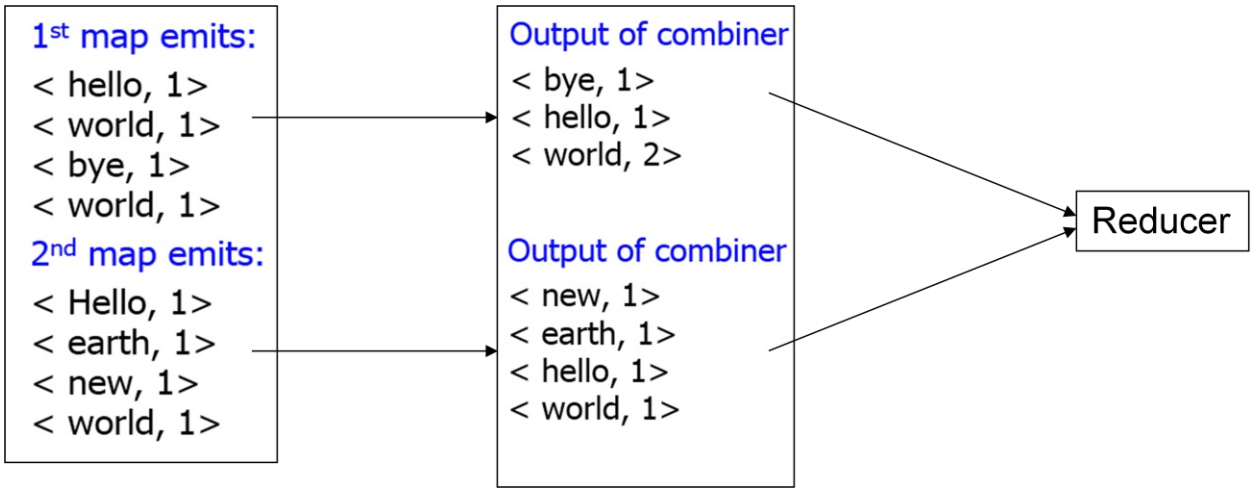
- It emits a key-value pair of < <word>, 1>, written to the Context.

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



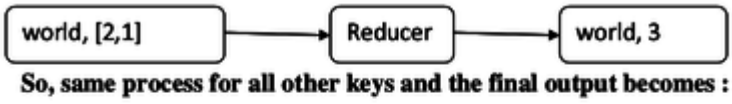
REDUCER ::

```
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 occurences>
    }
  }
```

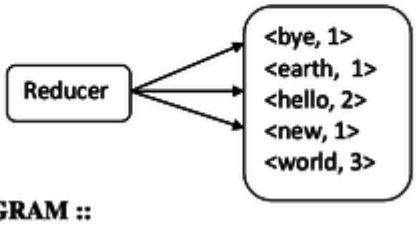
Reducer.reduce( ) ::

- **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 :



So, same process for all other keys and the final output becomes :



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);
    // 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);
  }
}
```

Name of the class which hadoop will run

*** 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 willl 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.