

MapReduce 2.0/YARN

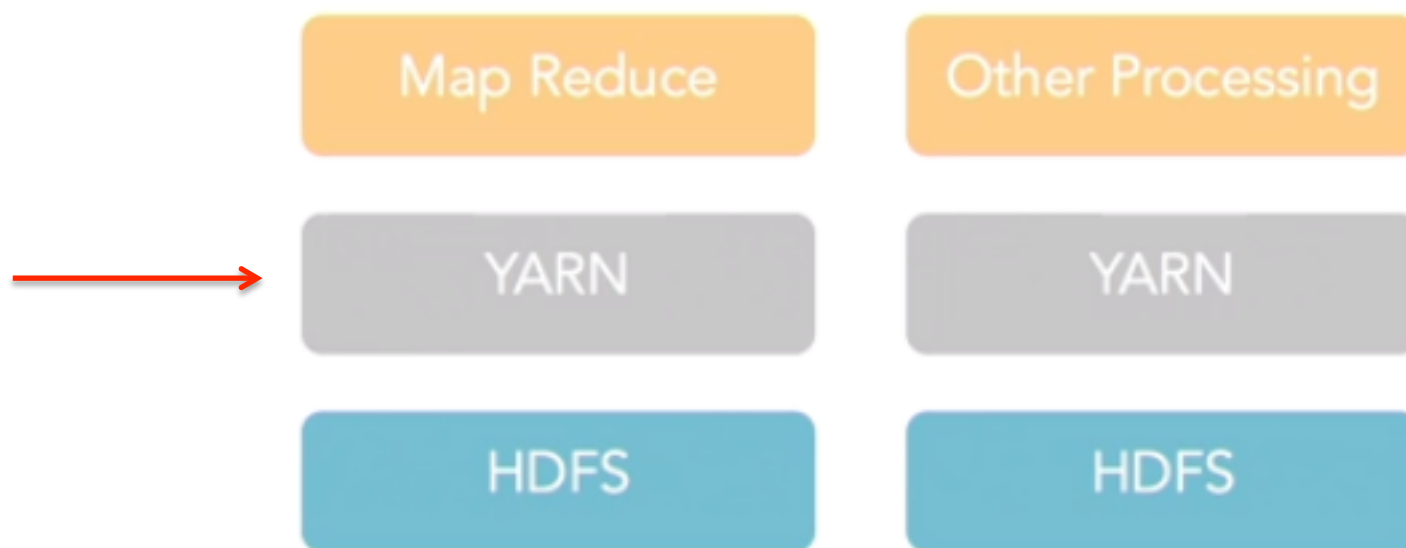
Jay Urbain, PhD

Limits of MapReduce 1.0

- ~10 years old
- Limits in the API
- Only batch processing, not interactive
- People want interactivity like RDBMS queries
- Coding MapReduce is complex, not enough developers
- Jobs don't fit all big data business scenarios
- Missing enterprise features, security, etc.

MapReduce 2.0 YARN

Yet Another Resource Negotiator



- YARN adds abstraction layer between MapReduce and HDFS
- Allows for other processing to occur on top of HDFS file system
- As volume of data increases, they want cheap HDFS storage, but better processing
- Spark, Impala

MapReduce 2.0

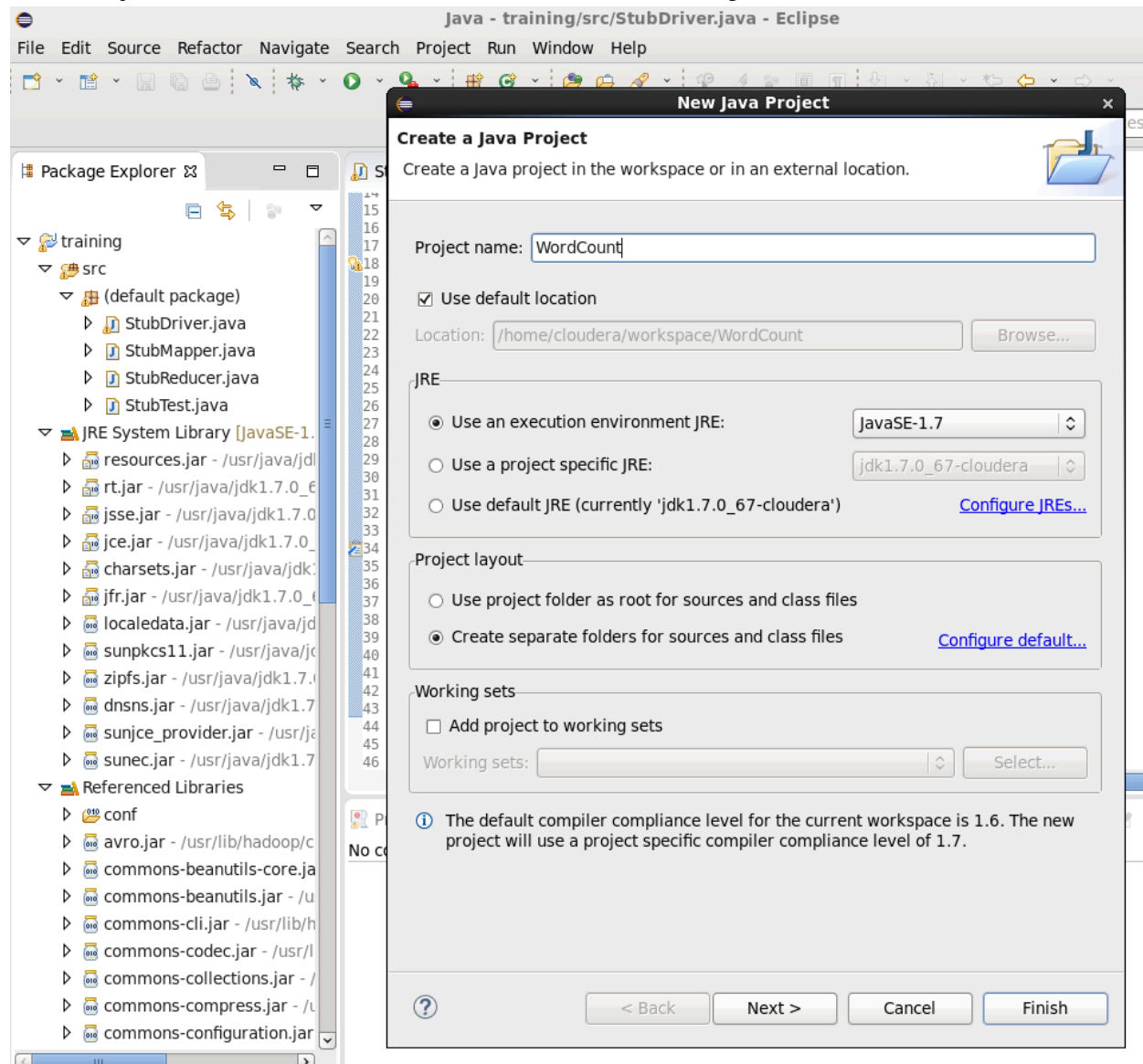
- Adds YARN
 - Supports multiple libraries
 - MapReduce not required
- Splits the existing JobTracker's roles
 - Resource management
 - Job life-cycle management
- Better scalability
- Batch or real-time + interactive processing
- Adds enterprise features, admin, security

MapReduce 2.0

- Execution flexibility/control
- Mapper/Reducer configure method takes params – can configure outside of program
- Tools/GenericOptionsParser takes options
- Reports can use app-specific info
- Distribute read only data for jobs

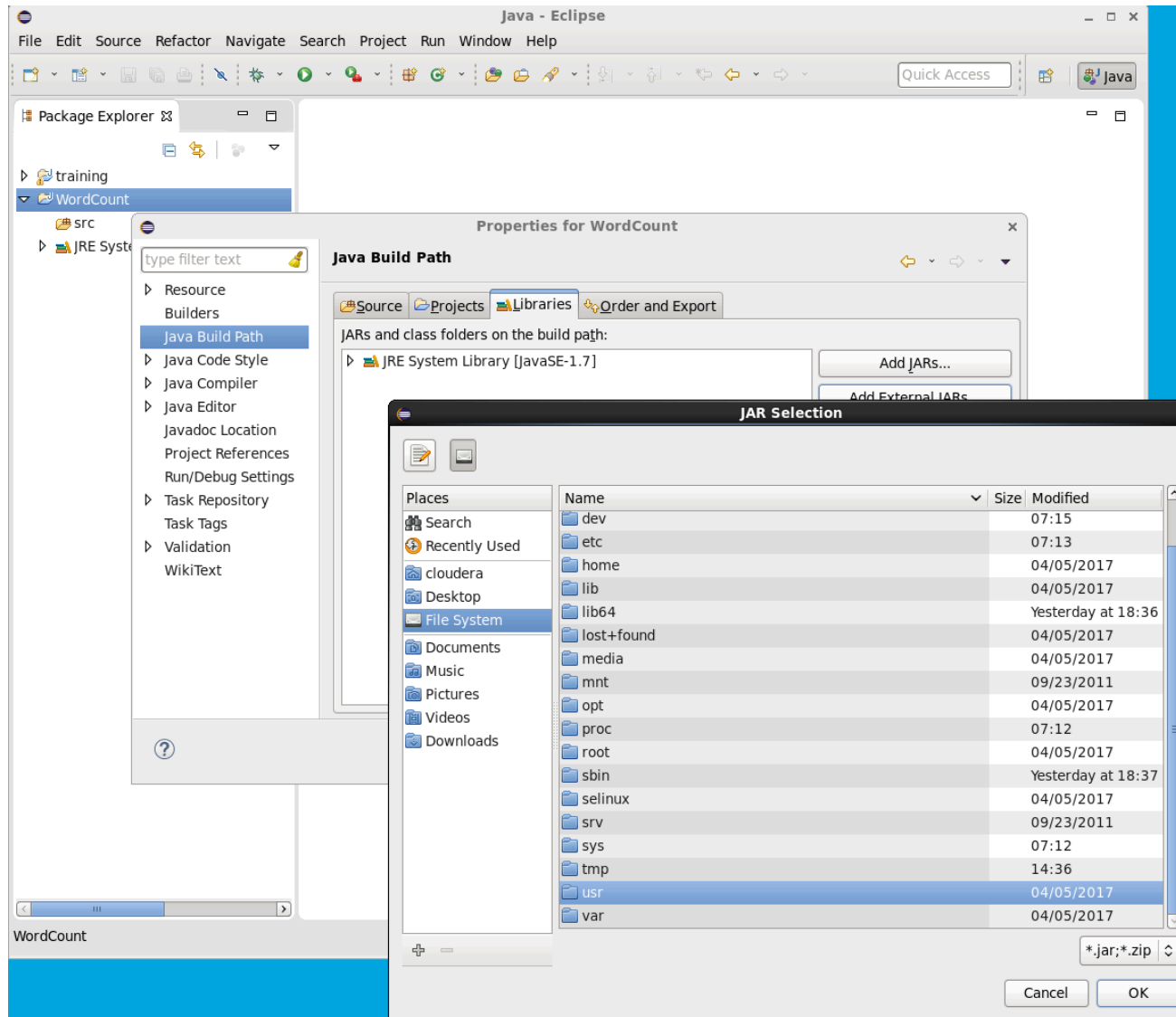
Coding MapReduce 2.0

In Eclipse: create a new JavaProject “WordCount”



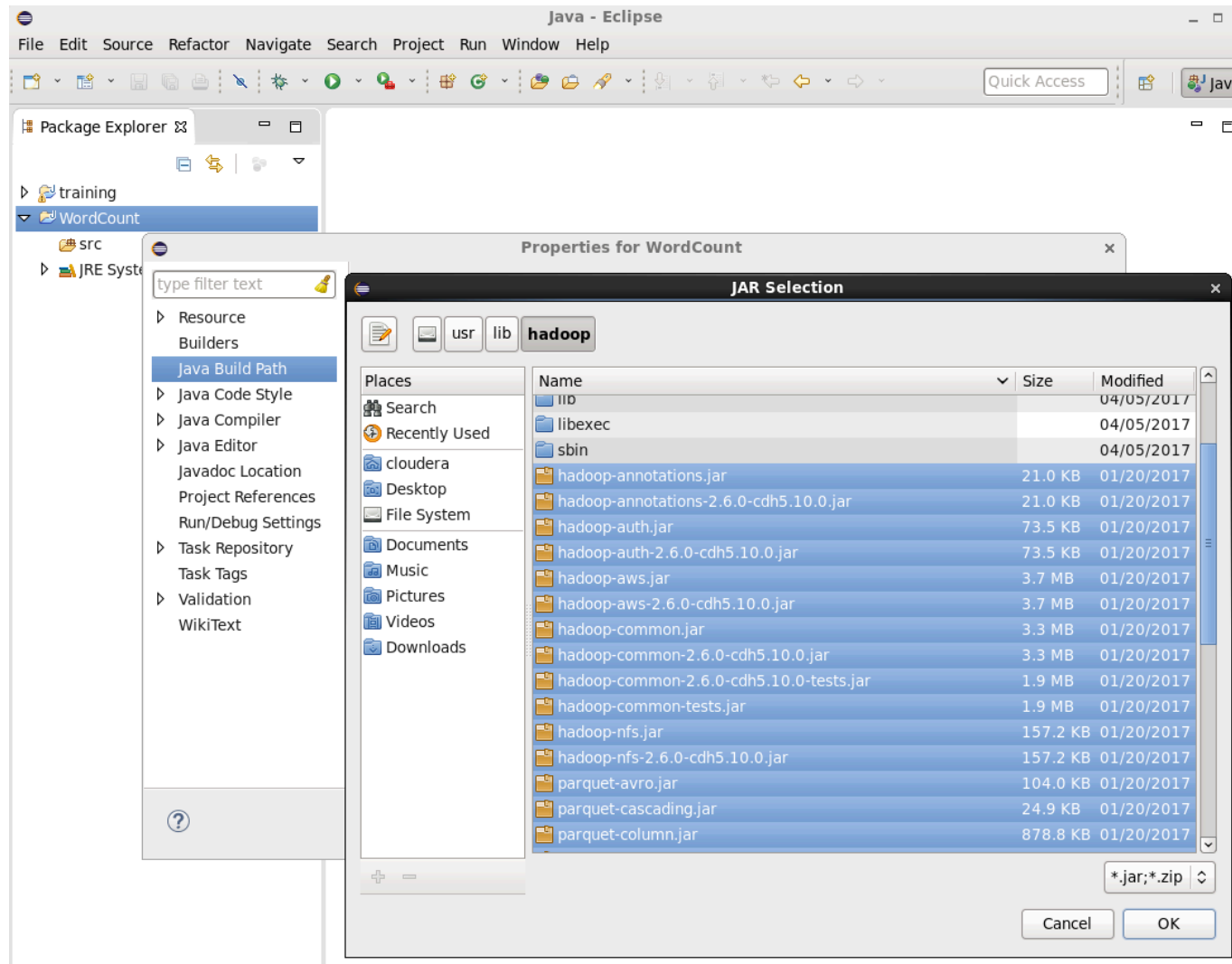
Add external libraries:

Right-mouse project -> Java Build Path -> Add External Jar -> File System -> usr

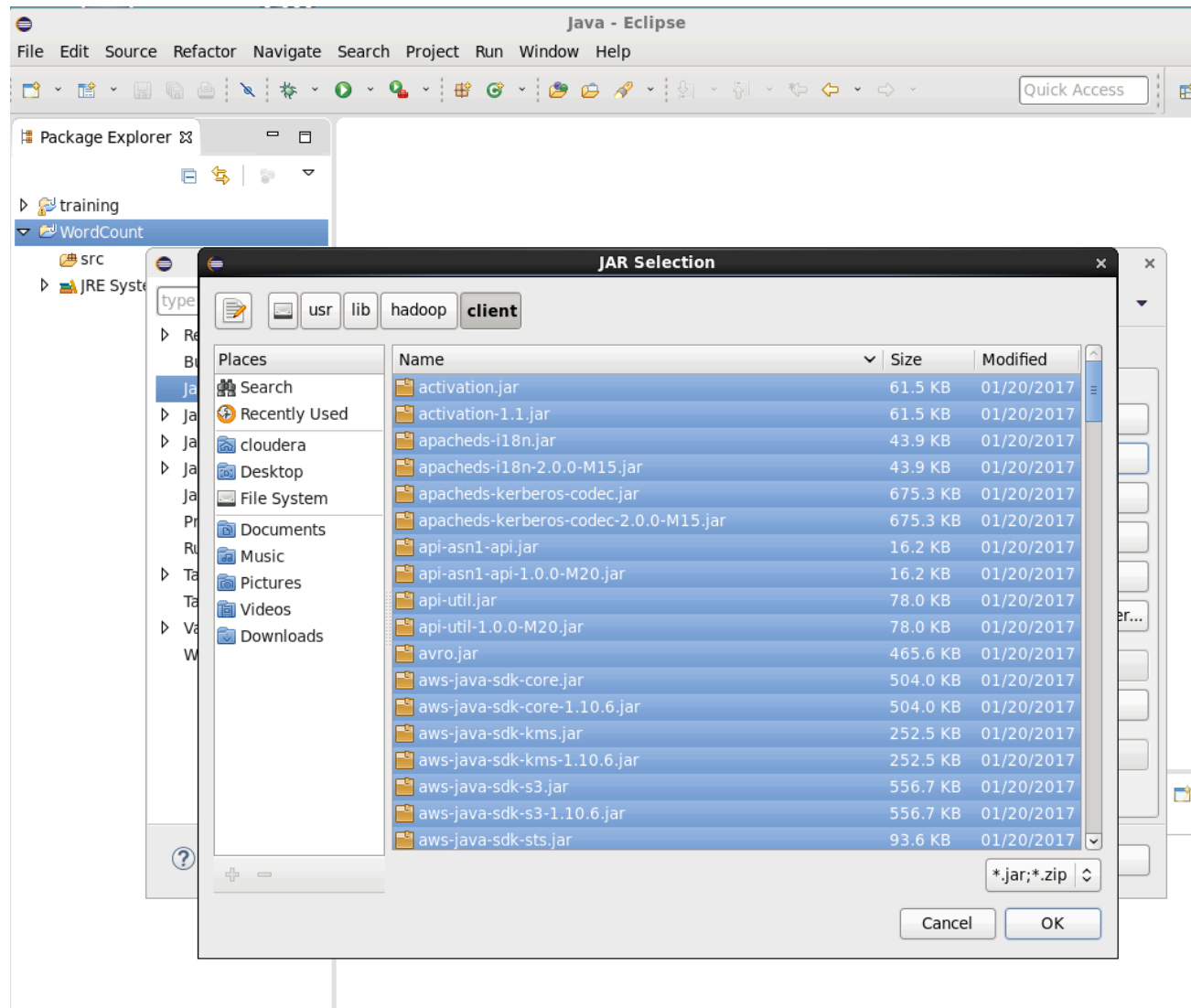


Add external libraries (continued)

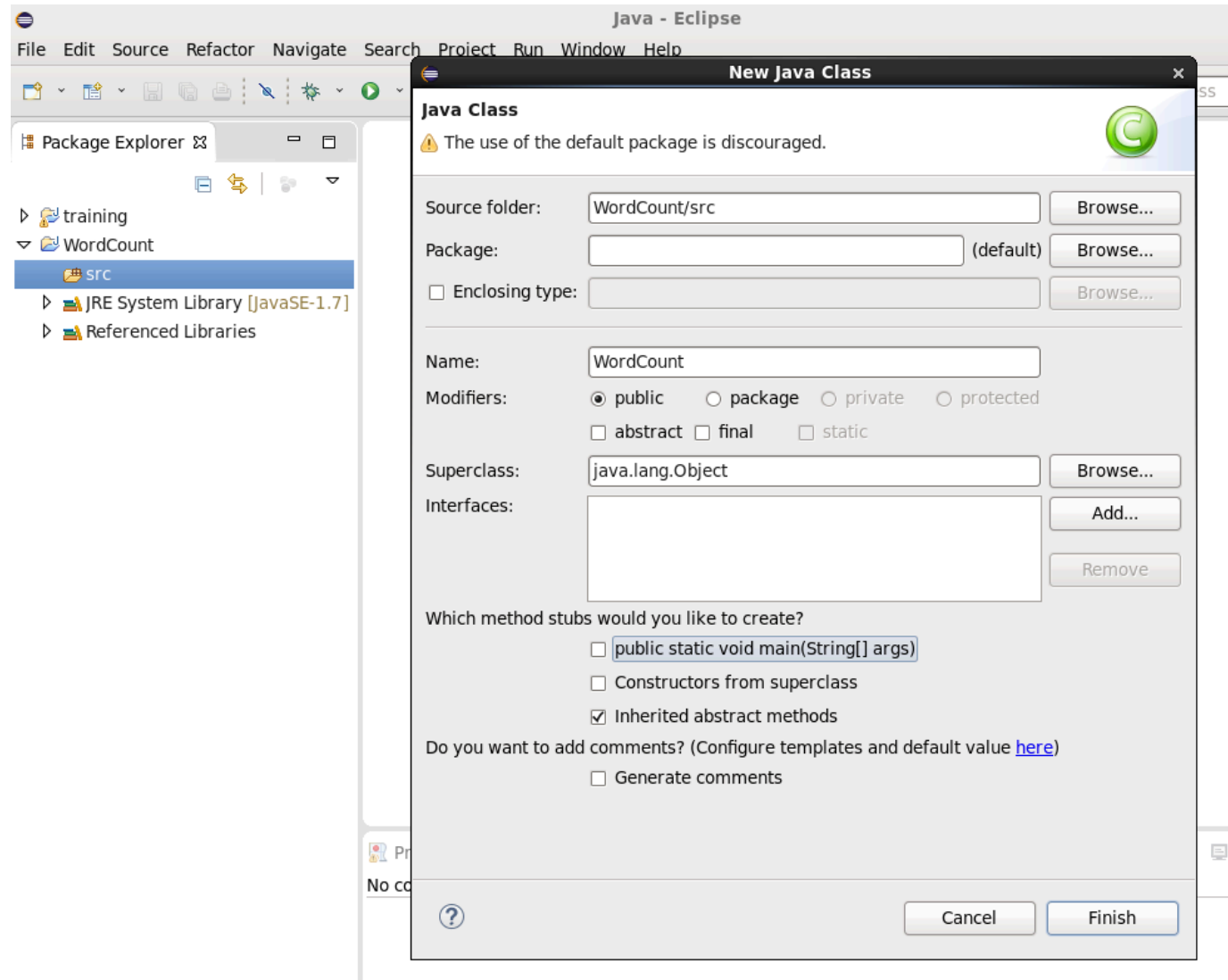
Right-mouse project -> Java Build Path -> Add External Jar -> File System ->
usr -> lib -> hadoop
Select all jar files



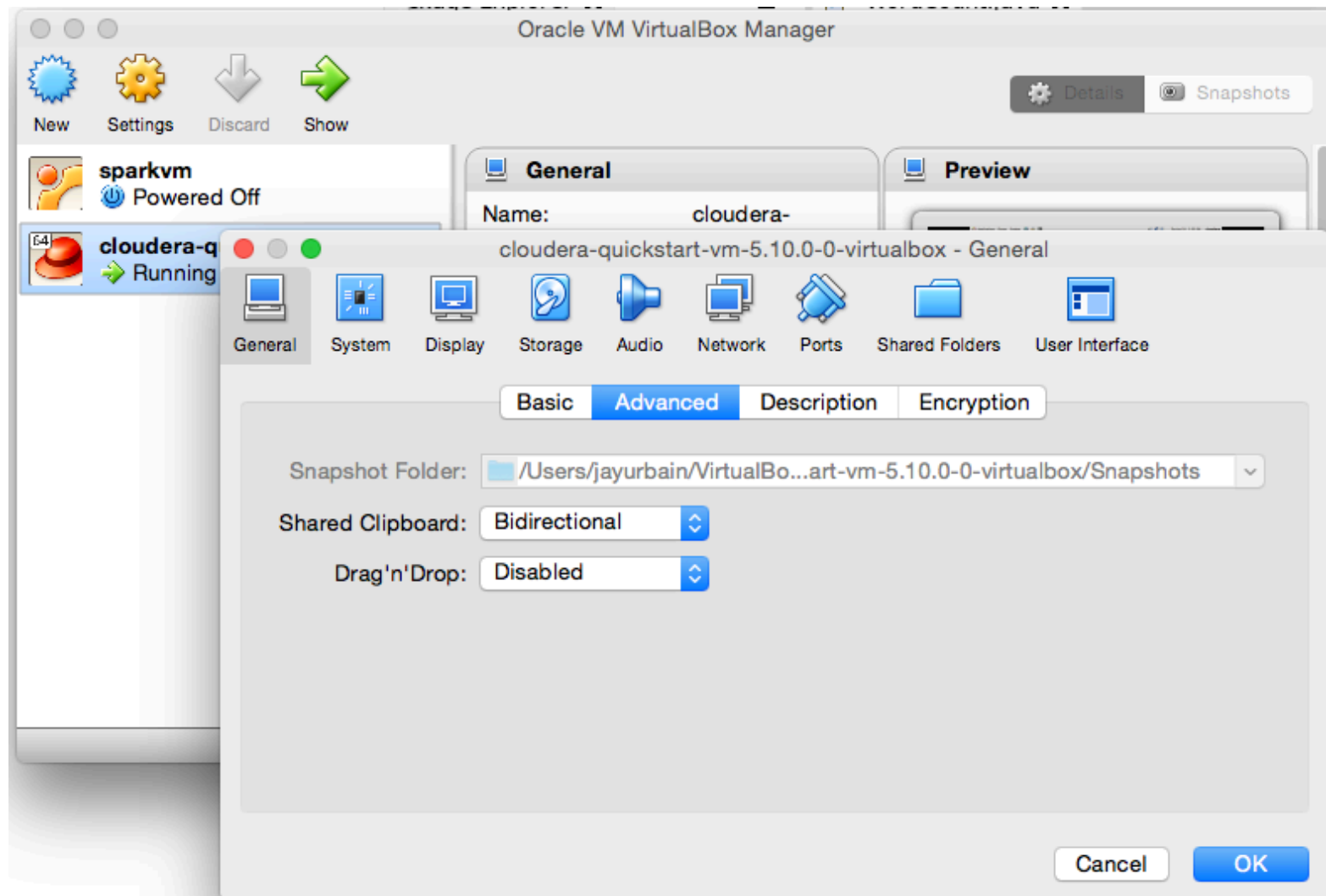
Repeat for client:
*Java Build Path -> Add External Jar -> File System -> usr -> lib -> hadoop
-> client -> Select all jar files*



- 1) Create a *org.myorg* package
- 2) Create a Java WordCount class within the *org.myorg* package



Oracle VM Settings -> Advanced Settings



Paste *WordCount.java* (course directory) into your project
WordCount.java,
Save your file, should compile cleanly

The image displays two side-by-side screenshots of the `WordCount.java` file. The left screenshot shows the raw code in a text editor, and the right screenshot shows the same code within the Eclipse IDE environment.

Left Screenshot (Text Editor): Shows the full source code of `WordCount.java`. The code includes imports for Hadoop and Java utilities, a package declaration, and the implementation of the `WordCount` class, which extends `Configured` and implements the `Tool` interface. It features a `main` method and a `run` method that processes input text and writes word counts to an output file.

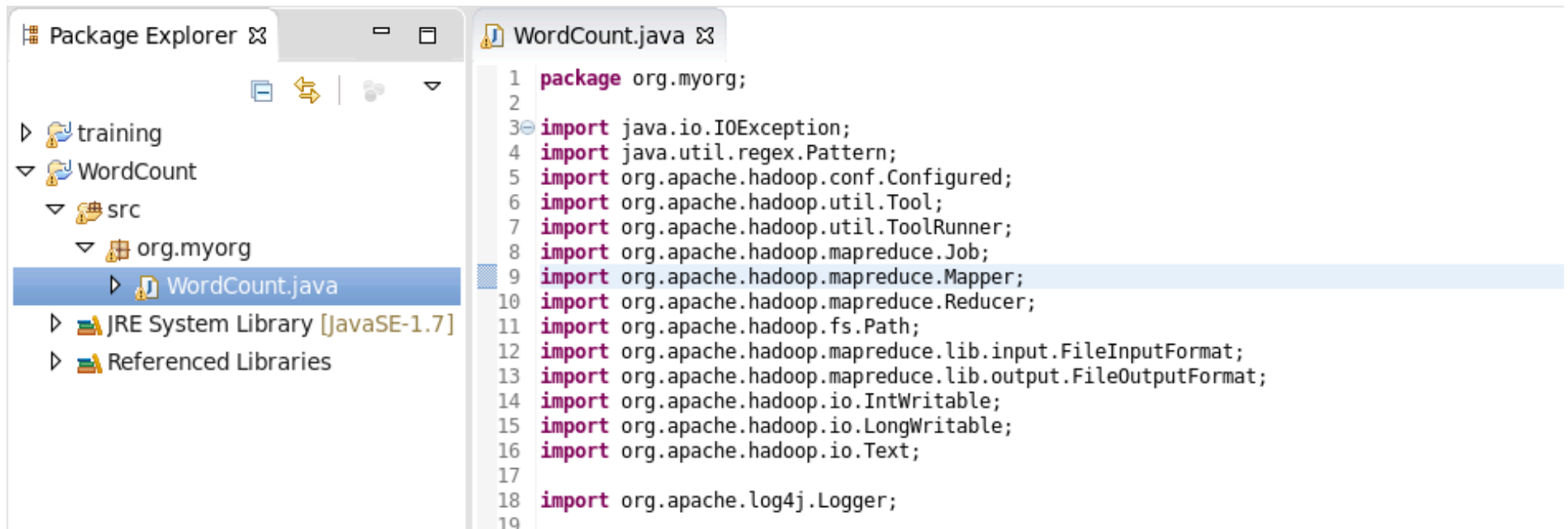
Right Screenshot (Eclipse IDE): Shows the same code in the Eclipse IDE. The Package Explorer on the left indicates the file is located in the `org.myorg` package. The IDE's status bar at the bottom shows "The value of the field WordCount.LOG is not used", "Writable", "Smart Insert", and "73 : 2".

```
1 package org.myorg;
2
3 import java.io.IOException;
4 import java.util.regex.Pattern;
5 import org.apache.hadoop.conf.Configured;
6 import org.apache.hadoop.util.Tool;
7 import org.apache.hadoop.util.ToolRunner;
8 import org.apache.hadoop.mapreduce.Job;
9 import org.apache.hadoop.mapreduce.Mapper;
10 import org.apache.hadoop.mapreduce.Reducer;
11 import org.apache.hadoop.fs.Path;
12 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.io.IntWritable;
15 import org.apache.hadoop.io.LongWritable;
16 import org.apache.hadoop.io.Text;
17
18 import org.apache.log4j.Logger;
19
20 public class WordCount extends Configured implements Tool {
21
22     private static final Logger LOG = Logger.getLogger(WordCount.class);
23
24     public static void main(String[] args) throws Exception {
25         int res = ToolRunner.run(new WordCount(), args);
26         System.exit(res);
27     }
28
29     public int run(String[] args) throws Exception {
30         Job job = Job.getInstance(getConf(), "wordcount");
31         job.setJarByClass(this.getClass());
32         // Use TextInputFormat, the default unless job.setInputFormatClass is used
33         FileInputFormat.addInputPath(job, new Path(args[0]));
34         FileOutputFormat.setOutputPath(job, new Path(args[1]));
35         job.setMapperClass(Map.class);
36         job.setReducerClass(Reduce.class);
37         job.setOutputKeyClass(Text.class);
38         job.setOutputValueClass(IntWritable.class);
39         return job.waitForCompletion(true) ? 0 : 1;
40     }
41
42     public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
43         private final static IntWritable one = new IntWritable(1);
44         private Text word = new Text();
45         private long numRecords = 0;
46         private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");
47
48         public void map(LongWritable offset, Text lineText, Context context)
49             throws IOException, InterruptedException {
50             String line = lineText.toString();
51             Text currentWord = new Text();
52             for (String word : WORD_BOUNDARY.split(line)) {
53                 if (word.isEmpty()) {
54                     continue;
55                 }
56                 currentWord.set(word);
57                 context.write(currentWord, one);
58             }
59         }
60     }
61 }
```

Code review – WordCount.java

Note: org.hadoop.mapreduce. -> MRv2*

- WordCount maps (extracts) words from an input source and reduces (aggregates) the results, returning a count of each word.
- The version of WordCount in this tutorial are implemented to take advantage of the features in the MRv2 API.



The screenshot shows an IDE interface. On the left, the Package Explorer displays a project structure with a 'training' folder containing a 'WordCount' folder. Inside 'WordCount', there is a 'src' folder containing a package 'org.myorg', which in turn contains the 'WordCount.java' file. The 'WordCount.java' file is selected and highlighted. On the right, the code editor displays the contents of 'WordCount.java'. The code starts with a package declaration 'package org.myorg;' followed by a series of import statements for various classes from the Java standard library and the Apache Hadoop MRv2 API. The imports include 'java.io.IOException', 'java.util.regex.Pattern', 'org.apache.hadoop.conf.Configured', 'org.apache.hadoop.util.Tool', 'org.apache.hadoop.util.ToolRunner', 'org.apache.hadoop.mapreduce.Job', 'org.apache.hadoop.mapreduce.Mapper', 'org.apache.hadoop.mapreduce.Reducer', 'org.apache.hadoop.fs.Path', 'org.apache.hadoop.mapreduce.lib.input.FileInputFormat', 'org.apache.hadoop.mapreduce.lib.output.FileOutputFormat', 'org.apache.hadoop.io.IntWritable', 'org.apache.hadoop.io.LongWritable', 'org.apache.hadoop.io.Text', and 'org.apache.log4j.Logger'.

```
1 package org.myorg;
2
3 import java.io.IOException;
4 import java.util.regex.Pattern;
5 import org.apache.hadoop.conf.Configured;
6 import org.apache.hadoop.util.Tool;
7 import org.apache.hadoop.util.ToolRunner;
8 import org.apache.hadoop.mapreduce.Job;
9 import org.apache.hadoop.mapreduce.Mapper;
10 import org.apache.hadoop.mapreduce.Reducer;
11 import org.apache.hadoop.fs.Path;
12 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.io.IntWritable;
15 import org.apache.hadoop.io.LongWritable;
16 import org.apache.hadoop.io.Text;
17
18 import org.apache.log4j.Logger;
19
```

Packages

- Standard Java classes: `IOException` and `regex.Pattern`.
- Use `regex.Pattern` to extract words from input files.

```
import java.io.IOException;  
import java.util.regex.Pattern;
```

- This application extends the class *Configured*, and implements the *Tool* utility class.
- You tell Hadoop what it needs to know to run your program in a configuration object.
- Then, you use *ToolRunner* to run your MapReduce application.

```
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;
```

- The `Logger` class sends debugging messages from inside the mapper and reducer classes.
- Messages you pass to `Logger` are displayed in the map or reduce logs for the job on your Hadoop server.

```
import org.apache.log4j.Logger;
```

Packages continued

- Use the Job class to create, configure, and run an instance of your MapReduce application.
- Extend the Mapper class with your own Mapper class and add your own processing instructions.
- Same is true for the Reducer: extend it to create and customize your own Reducer class.

```
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.Mapper;  
import org.apache.hadoop.mapreduce.Reducer;
```

- Use the Path class to access files in HDFS.
- Pass the required paths using the FileInputFormat and FileOutputFormat classes.

```
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

- Writable objects have methods for writing, reading, and comparing values during map and reduce processing.
- Think of the Text class as StringWritable.

```
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;
```

WordCount class, run method

- *WordCount* includes main and run methods, and the inner classes Map and Reduce. The class begins by initializing the logger.

```
public class WordCount extends Configured implements Tool {  
    private static final Logger LOG = Logger.getLogger(WordCount.class);
```

- The main method invokes *ToolRunner*, which creates and runs a new instance of *WordCount*, passing the command line arguments.
- When the application is finished, it returns an integer value for the status.

```
public static void main(String[] args) throws Exception {  
    int res = ToolRunner.run(new WordCount(), args);  
    System.exit(res);  
}
```

- The run method configures the job, starts the job, waits for the job to complete.

```
public int run(String[] args) throws Exception {
```

- Create a new instance of the Job object. This example uses the *Configured.getConf()* method to get the configuration object for this instance of *WordCount*, and names the job object wordcount. Default config.

```
Job job = Job.getInstance(getConf(), "wordcount");
```


Run method continued

- Set the JAR to use, based on the class in use.

```
job.setJarByClass(this.getClass());
```

- Set the input and output paths for your application. You store your input files in HDFS, and then pass the input and output paths as command-line arguments at runtime.

```
FileInputFormat.addInputPaths(job, new Path(args[0]));
```

```
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

- Set the map class and reduce class for the job. In this case, use the Map and Reduce inner classes defined in this class.

```
job.setMapperClass(Map.class);
```

```
job.setReducerClass(Reduce.class);
```

- Set the output key and the value.

```
job.setOutputKeyClass(Text.class);
```

```
job.setOutputValueClass(IntWritable.class);
```

- Launch the job and wait for it to finish. The method syntax is `waitForCompletion(boolean verbose)`. When true, the method reports its progress as the Map and Reduce classes run. When false, the method reports progress up to, but not including, the Map and Reduce processes.

```
return job.waitForCompletion(true) ? 0 : 1;
```

```
}
```

Mapper

- The Map class transforms key/value input into intermediate key/value pairs to be sent to the Reducer.

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {  
    private final static IntWritable one = new IntWritable(1);  
    private Text word = new Text();
```

- Regular expression pattern to parse each line of input text on word boundaries ("\\b").

```
private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");
```

- Hadoop invokes the map method once for every key/value pair from your input source.
- This does not necessarily correspond to the intermediate key/value pairs output to the reducer.
- In this case, the map method receives the offset of the first character in the current line of input as the key, and a Text object representing an entire line of text from the input file.

```
public void map(LongWritable offset, Text lineText, Context context)  
    throws IOException, InterruptedException {
```

Mapper

- Convert the Text object to a string. Create the current Word variable, which is used to capture individual words from each input string.

```
String line = lineText.toString();  
Text currentWord = new Text();
```

- Use the regular expression pattern to split the line into individual words.

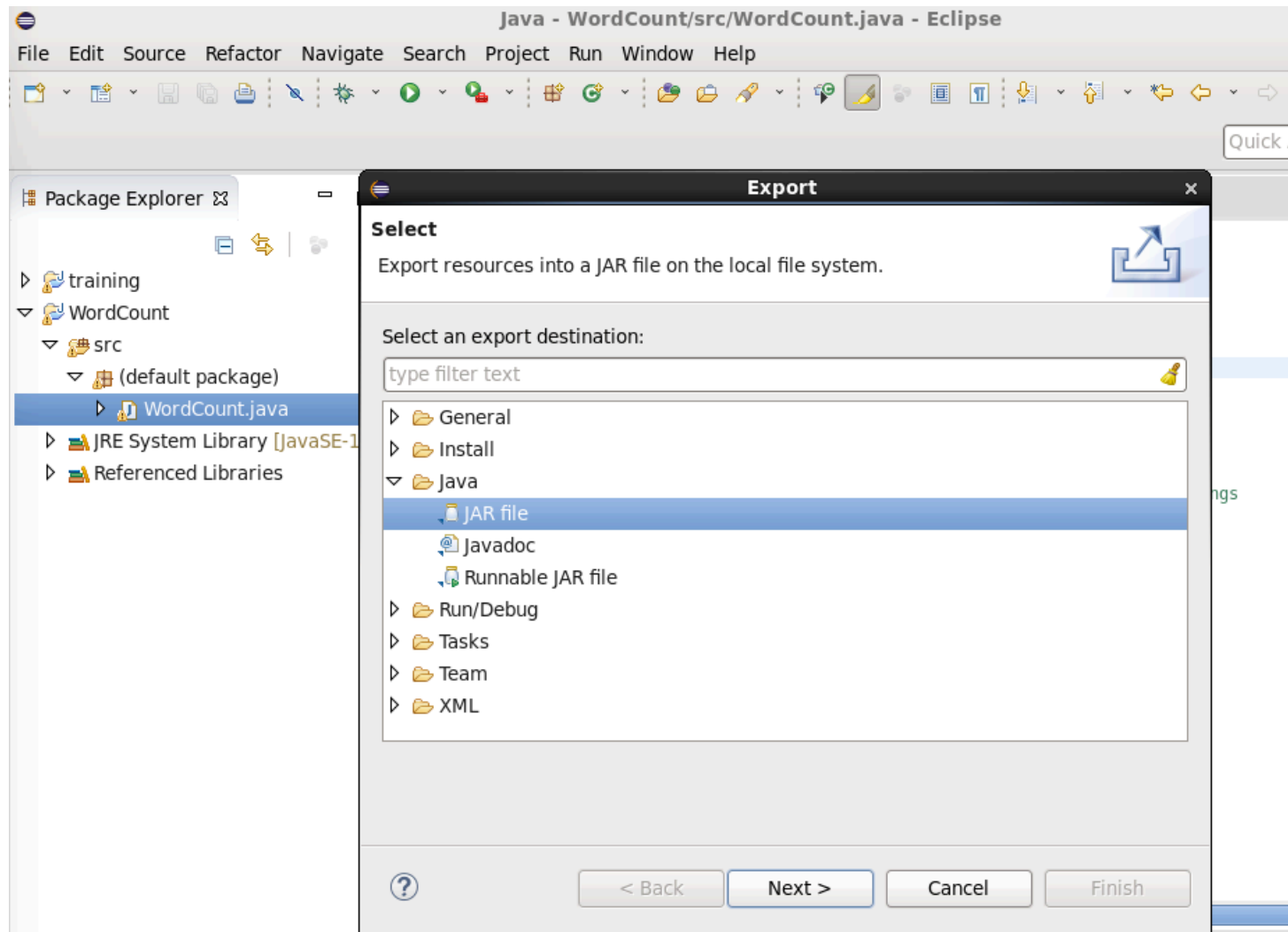
```
for ( String word : WORD_BOUNDARY.split(line)) {  
    if (word.isEmpty()) {  
        continue;  
    }  
    currentWord = new Text(word);  
    context.write(currentWord,one);  
}  
}
```

Reducer

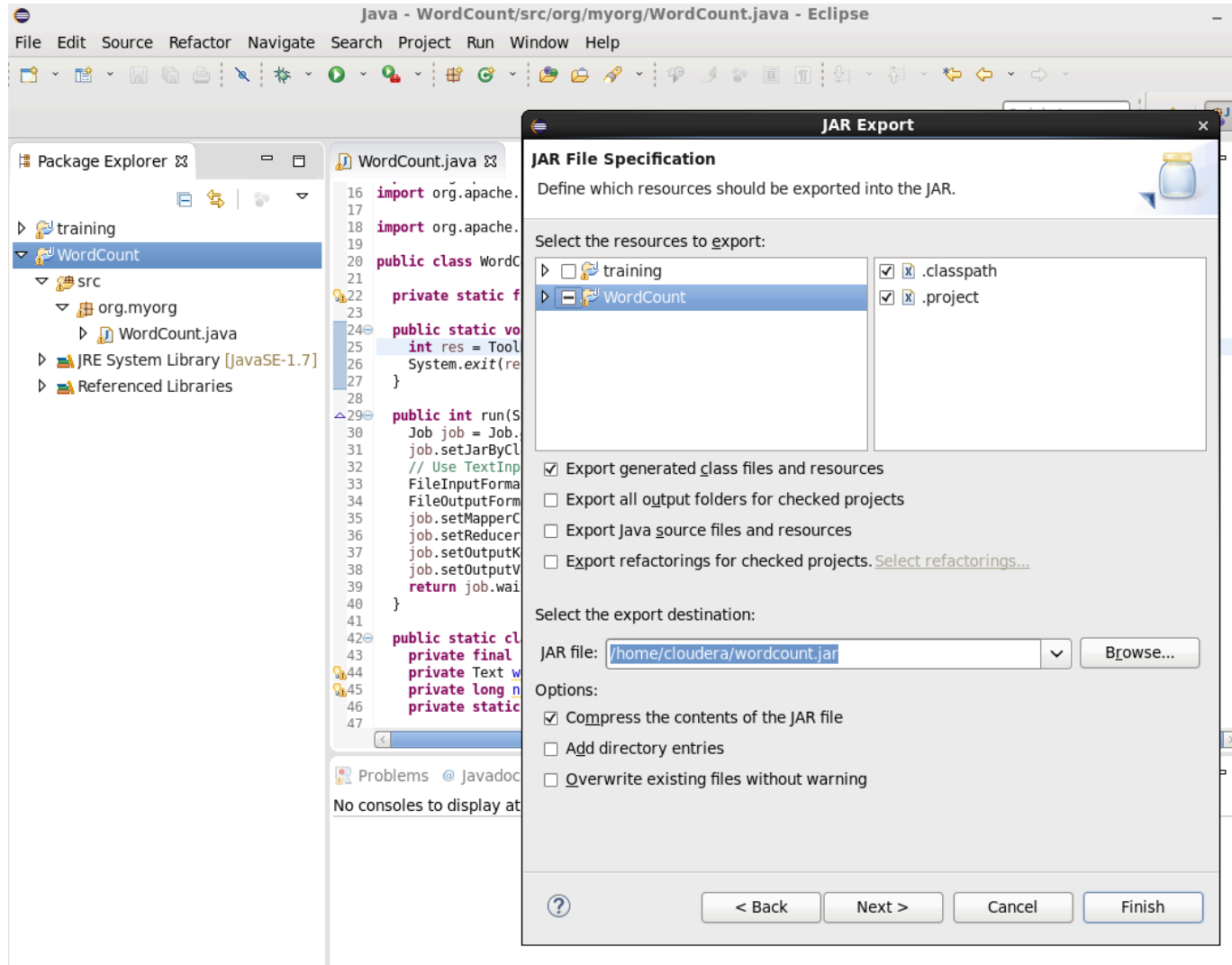
- The mapper creates a key/value pair for each word, composed of the word and the IntWritable value 1.
- The reducer processes each pair, adding one to the count for the current word.
- It then writes the result for that word to the reducer context object.
- When all of the intermediate key/value pairs are processed, the map/reduce task is complete. The application saves the results to the output location in HDFS.

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {  
    @Override public void reduce(Text word, Iterable<IntWritable> counts, Context context)  
        throws IOException, InterruptedException {  
        int sum = 0;  
        for (IntWritable count : counts) {  
            sum += count.get();  
        }  
        context.write(word, new IntWritable(sum));  
    }  
}
```

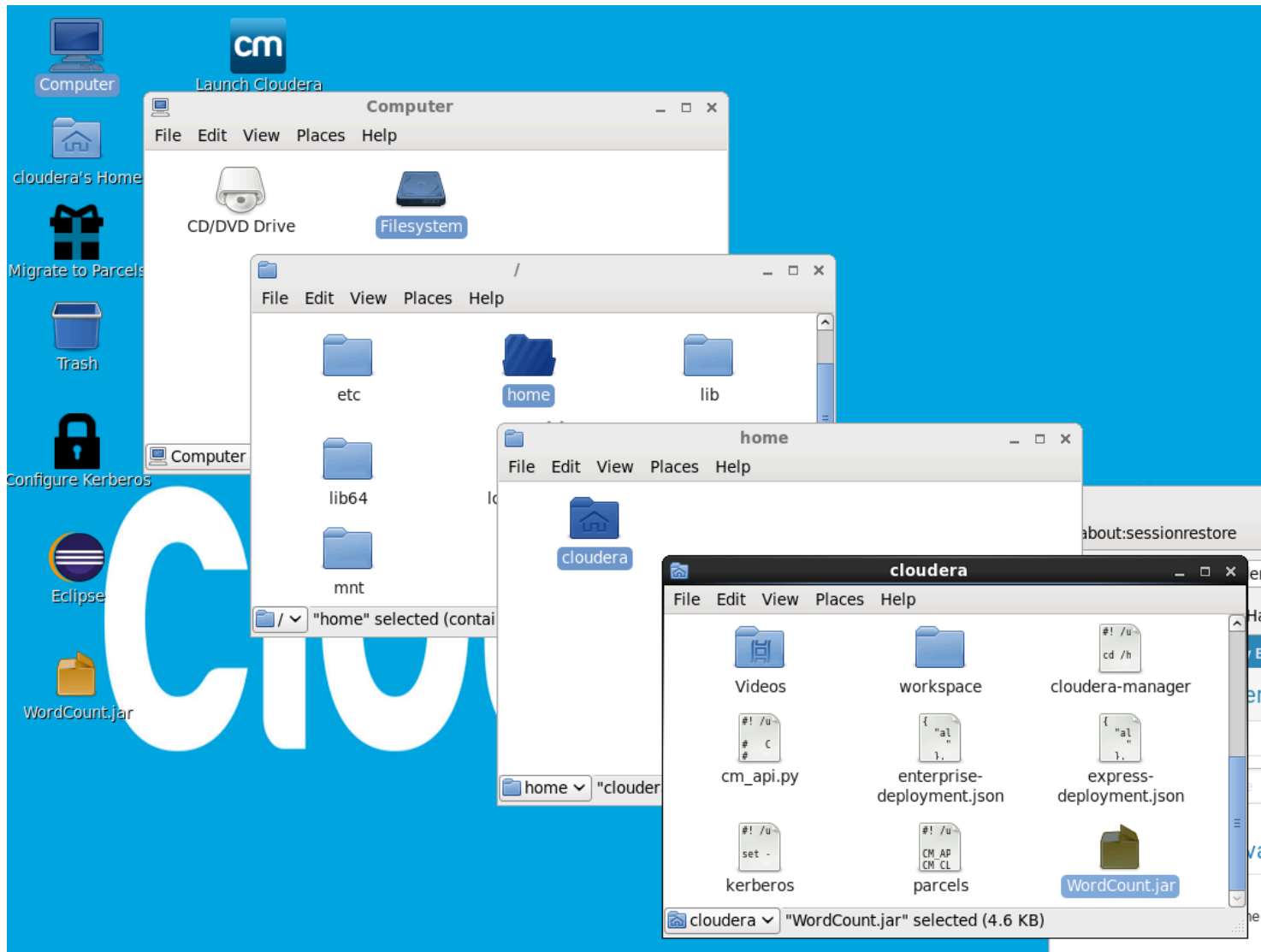
Export jar



Export jar



Export jar – Note: I renamed the jar file wordcount.jar



Create some data to process

From the terminal:

- create an input directory,
- create input files for processing, and
- move the input files into HDFS:

```
$ hadoop fs -mkdir /user/cloudera/wordcount /user/cloudera/wordcount/input
```

```
$ echo "Jay Urbain" > file0
```

```
$ echo "Kimberly Urbain" > file1
```

```
$ echo "Jenna Urbain Hoelz" > file2
```

```
$ echo "Jenna Urbain Hoelz" > file2
```

```
$ echo "Lauren Urbain Haehle" > file3
```

```
$ echo "Lucas Urbain" > file4
```

```
$ echo "Emmerson Hoelz" > file5
```

```
$ hadoop fs -put file* /user/cloudera/input
```


Execute your Hadoop job

```
$ hadoop jar wordcount.jar org.myorg.WordCount /user/cloudera/wordcount/input /  
user/cloudera/wordcount/output
```

Lots of output, watch for exceptions, some are benign:

```
17/06/21 06:40:40 INFO client.RMProxy: Connecting to ResourceManager at /  
0.0.0.0:8032  
17/06/21 06:40:41 INFO input.FileInputFormat: Total input paths to  
process : 7 ....
```

Note: you output directory /user/cloudera/wordcount/output can not exist. If it does, or you want to rerun your job, you can remove it and its contents as follows:

```
$ hadoop fs -rmr /user/cloudera/wordcount/output
```

Verify output

```
$ hadoop fs -cat /user/cloudera/wordcount/output/*
```

Carter 1

Emmerson 1

Haehle 2

Hoelz 2

Jay 1

Jenna 1

Kimberly 1

Lauren 1

Lucas 1

Urbain 5

Design Patterns

- On your own - SKIP

Adding Variables for Monitoring

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

    static enum Counters { INPUT_WORDS }

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private boolean caseSensitive = true;
    private Set<String> patternsToSkip = new HashSet<String>();

    private long numRecords = 0;
    private String inputFile;
```

Distributed Cache

```
public void configure(JobConf job) {  
    caseSensitive = job.getBoolean("wordcount.case.sensitive", true);  
    inputFile = job.get("map.input.file");  
    if (job.getBoolean("wordcount.skip.patterns", false)) {  
        Path[] patternsFile = new Path[0];  
        try {  
            patternsFiles = DistributedCache.getLocalCacheFiles(job);  
        } catch (IOException ioe) {  
            System.err.println("Caught exception while getting cached  
files: " + StringUtils.stringifyException(ioe));  
        }  
        for (Path patternsFile : patternsFiles) {  
            parseSkipFile(patternsFile);  
        }  
    }  
}
```

Optimization: Skipping Files

```
private void parseSkipFile(Path patternsFile) {  
    try {  
        BufferedReader fis = new BufferedReader(new  
        FileReader(patternsFile.toString()));  
        String pattern = null;  
        while ((pattern = fis.readLine()) != null) {  
            patternsToSkip.add(pattern);  
        }  
    } catch (IOException ioe) {  
        System.err.println("Caught exception while parsing the cached file '" +  
        patternsFile + "' : " + StringUtils.stringifyException(ioe));  
    }  
}
```

Mapper with Counter

```
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException {
    String line = (caseSensitive) ? value.toString() : value.toString.toLowerCase();
    for (String pattern : patternsToSkip) {
        line = line.replaceAll(pattern, "");
    }
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
        reporter.incrCounter(Counters.INPUT_WORDS, 1);
    }
    if ((++numRecords % 100) == 0) {
        reporter.setStatus("Finished processing " + numRecords + " records " + " from the
input file: " + inputFile);
    }
}
```

Reducer with Reporter

```
public static class Reduce extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```


Review

- Which of the following features of MapReduce 2.0 is very interesting and causing a lot of people to be excited about?
 - Splitting the existing JobTracker's role
 - **Batch or interactive processing**
 - Adding enterprise features
 - Supporting many frameworks
- How do you add references to Hadoop libraries in your project using Eclipse
 - Right-click on project -> Properties -> Add External jars -> bin -> lib
 - Right-click on project -> Properties -> Add External jars -> File System -> lib
 - Right-click on project -> Properties -> Add External jars -> Documents-> usr -> lib
 - **Right-click on project -> Properties -> Add External jars -> File System -> usr -> lib**
- Which of the following will allow you to add a variable for monitoring MapReduce 2.0?
 - Private Text
 - **static enum Counters {INPUT_WORDS}**
 - None of these
 - Private caseSensitive=true;