Big Data - MR Development

Dr. Qing "Matt" Zhang
ITU

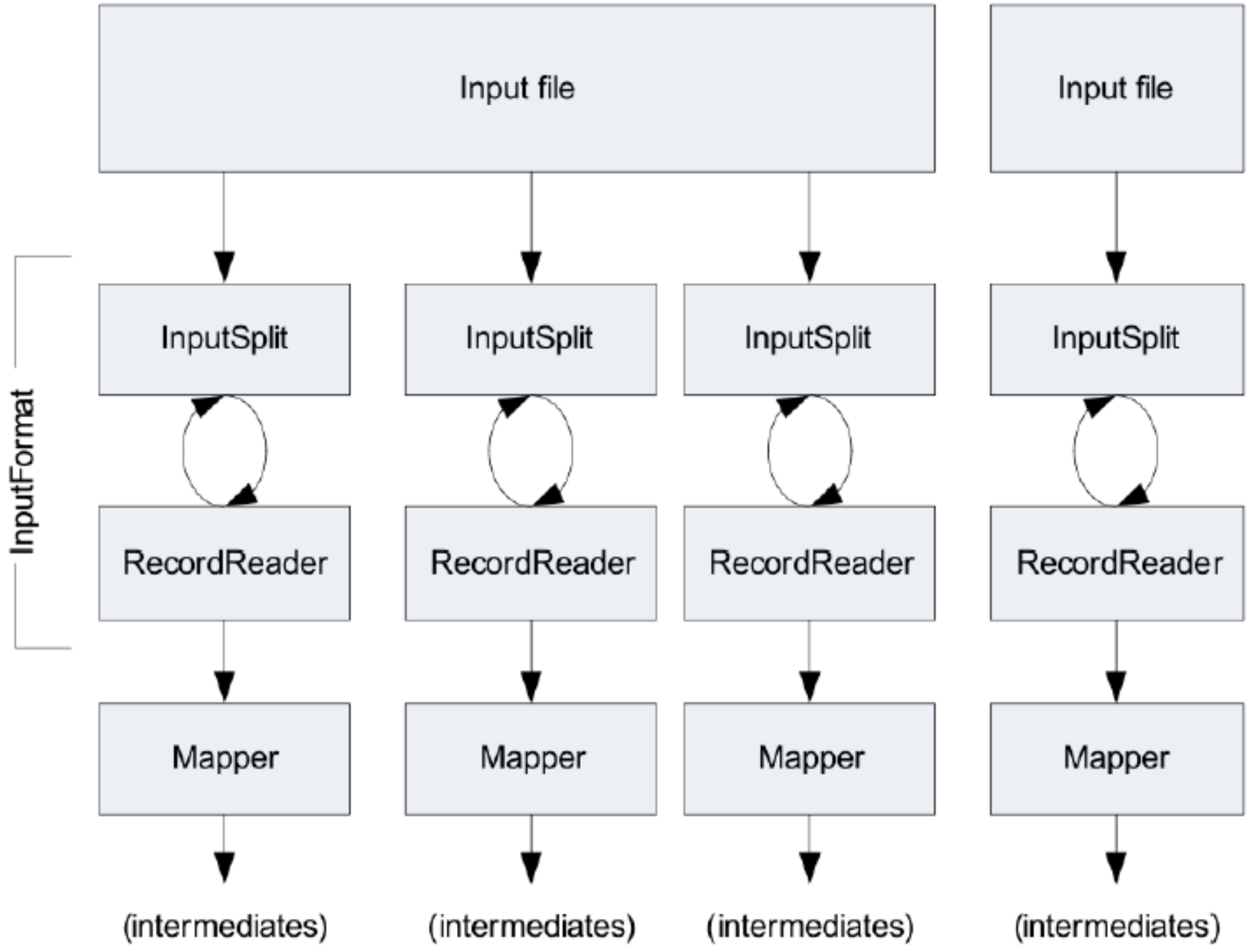
Outline

- Basic MapReduce Program
- Use Streaming API
- Implementing ToolRunner

MR Flow

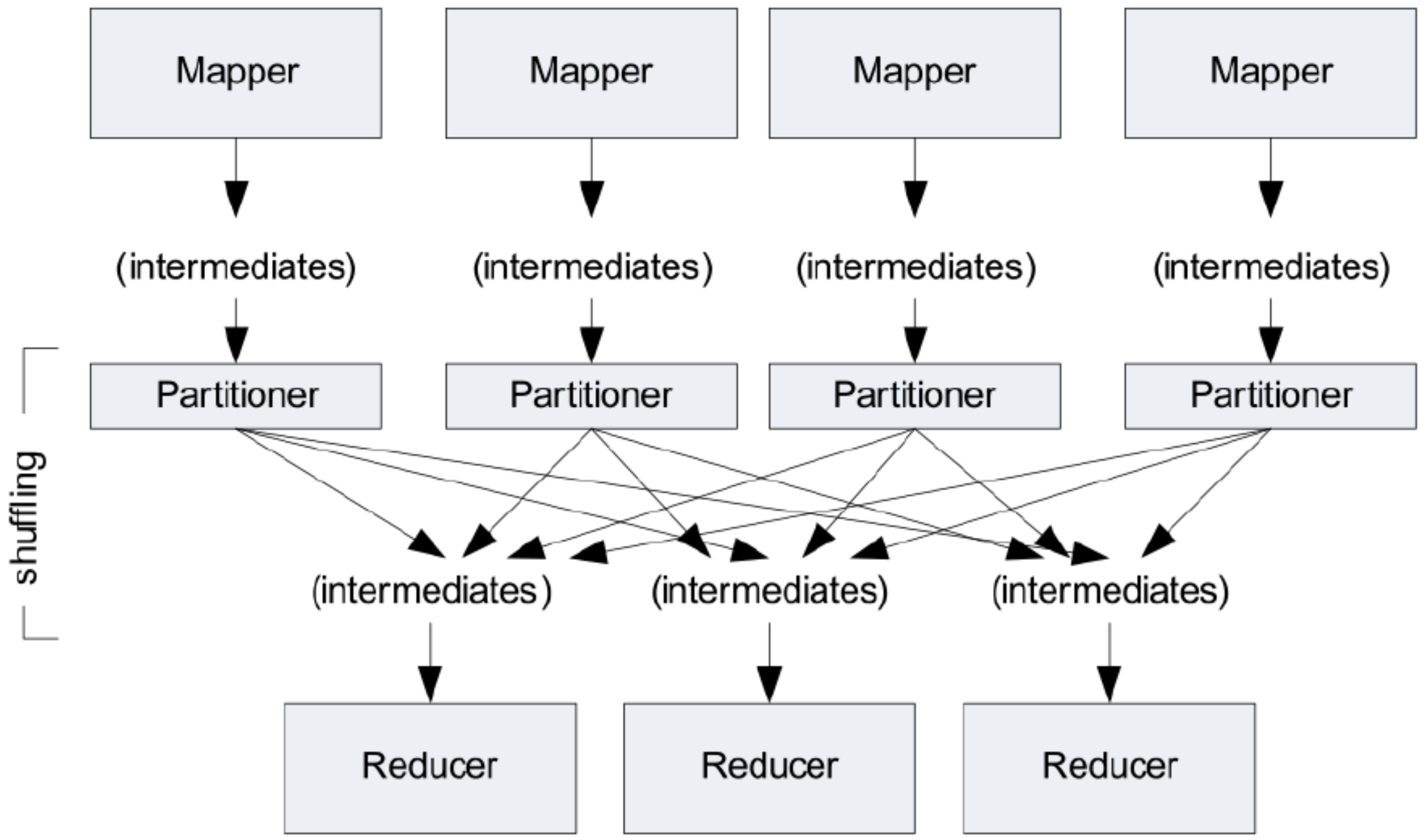
- Users need to create at least a Mapper, Reducer, and driver code
- Each of the portions can be created by the developer
  - Mapper, Reducer, RecordReader, Partitioner

Mapper

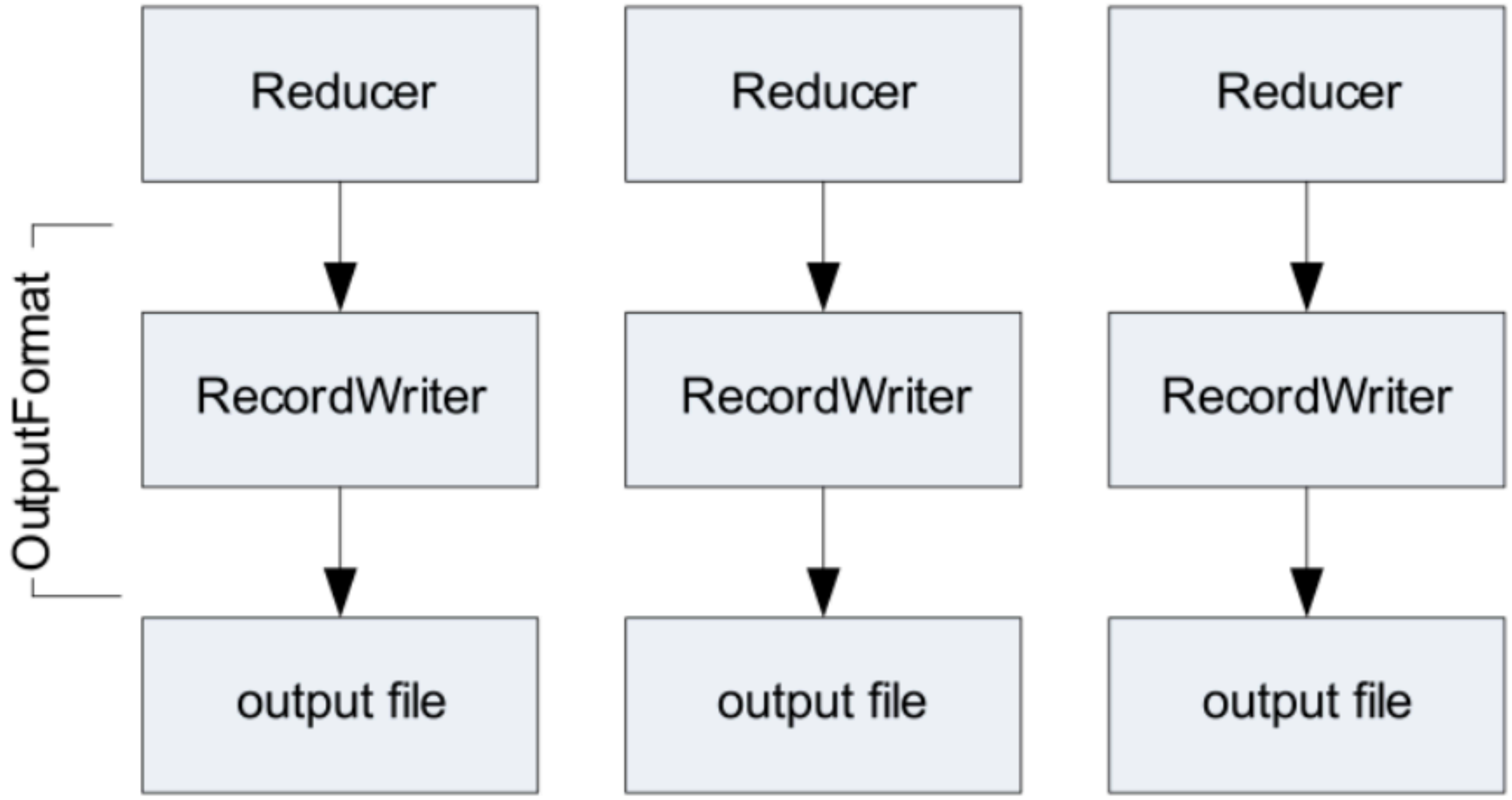


Input file
Input file
InputFormat
InputSplit
InputSplit
InputSplit
InputSplit
RecordReader
RecordReader
RecordReader
RecordReader
Mapper
Mapper
Mapper
Mapper
(intermediates)
(intermediates)
(intermediates)
(intermediates)

Shuffle and Sort



Reducer and Output



WordCount

- "HelloWorld" of Hadoop programming
- Three portions
  - Driver code
    - Code that runs on the client to configure and submit the job
  - Mapper
  - Reducer
- Need to study basic Hadoop APIs first

InputFormat

- The data passed to the Mapper is specified by an *InputFormat*
  - Specified in the driver code
  - Defines the location of the input data
    - A file or directory, for example
  - Determines how to split the input data into input splits
    - Each Mapper deals with a single input split
  - *InputFormat* is a factory for *RecordReader* objects to extract (key, value) records from the input source

Standard InputFormats

- FileInputFormat
  - The base class used for all file/based InputFormats
- TextInputFormat
  - The default
  - Treats each \n-terminated line of a file as a value
  - Key is the byte offset within the file of that line

Standard InputFormats

- KeyValueTextInputFormat
  - Maps \n-terminated lines as ‘key SEP value’
  - By default, separator is a tab
- SequenceFileInputFormat
  - Binary file of (key, value) pairs with some additional metadata
- SequenceFileAsTextInputFormat
  - Similar, but maps (*key.toString(), value.toString()*)

Keys and Values

- Keys and values in Hadoop are Objects
- Values are objects which implement *Writable*
- Keys are objects which implement *WritableComparable*

Writeable

- The Writable interface makes serialization quick and easy for Hadoop
- Hadoop defines its own ‘box classes’ for strings, integers and so on
  - *IntWritable* for ints
  - *LongWritable* for longs
  - *FloatWritable* for floats
  - *DoubleWritable* for doubles
  - *Text* for strings
  - Etc.
- Any value’s type must implement the Writable interface

WritableComparable

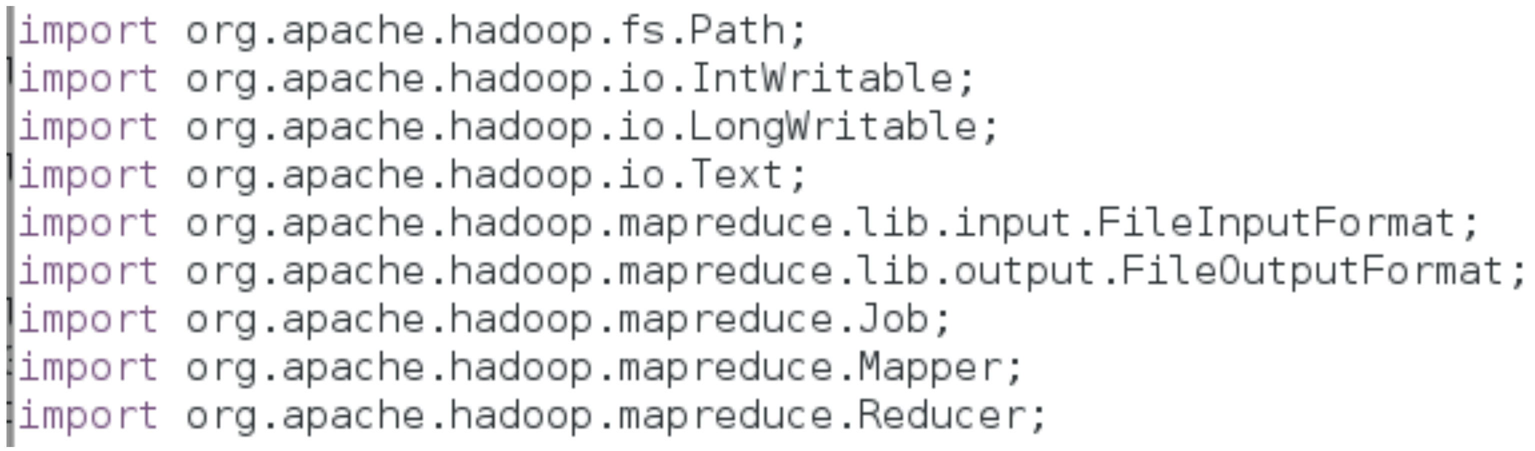
- A WritableComparable is a Writable which is also Comparable
  - Two WritableComparables can be compared against each other to determine their ‘order’
  - Keys must be WritableComparables because they are passed to the Reducer in sorted order
- Note that despite their names, all Hadoop box classes implement both Writable and WritableComparable
  - For example, IntWritable is actually a WritableComparable

Writing a MR code

- Driver code
  - Runs on client machine
  - It configures the job, then submits it to the cluster

Driver: Import

- Typical Hadoop pkgs often need to be imported



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

Driver: main

```java
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

Driver: main

```
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job
  job.set
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

We need two command line arguments.
First step is to check the input

Driver: main

```
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInpu
  FileOutp

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

Create a Job object, specify the class which will run the Hadoop job, and also give it a name.

Job

- The Job class allows you to set configuration options for your MapReduce job
  - The classes to be used for your Mapper and Reducer
  - The input and output directories
  - Many other options
- Any options not explicitly set in your driver code will be read from your Hadoop configuration files
  - Usually located in /etc/hadoop/conf
- Any options not specified in your configuration files will receive Hadoop’s default values
- You can also use the Job object to submit the job, control its execution, and query its state

Driver: main

```
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.
  job.

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

Setup the input and output directory, using the standard FileInputFormat and FileOutputFormat

FileInputFormat

- By default, *FileInputFormat.setInputPaths()* will read all files from a specified directory and send them to Mappers
  - Exceptions: items whose names begin with a period (.) or underscore (_)
  - Globs can be specified to restrict input
    - For example, /2010/*/01/*
- Alternatively, FileInputFormat.addInputPath() can be called multiple times, specifying a single file or directory each time

Specifying InputFormat

- No InputFormat is specified in this program
  - The default (TextInputFormat) will be used
- To use an InputFormat other than the default
  - *job.setInputFormatClass(KeyValueTextInputFormat.class)*

FileOutputFormat

- FileOutputFormat.setOutputPath() specifies the directory to which the Reducers will write their final output
- The driver can also specify the format of the output data
  - Default is a plain text file
  - Could be explicitly written as
    - *job.setOutputFormatClass(TextOutputFormat.class)*

Driver: main

```
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.se
  job.se

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

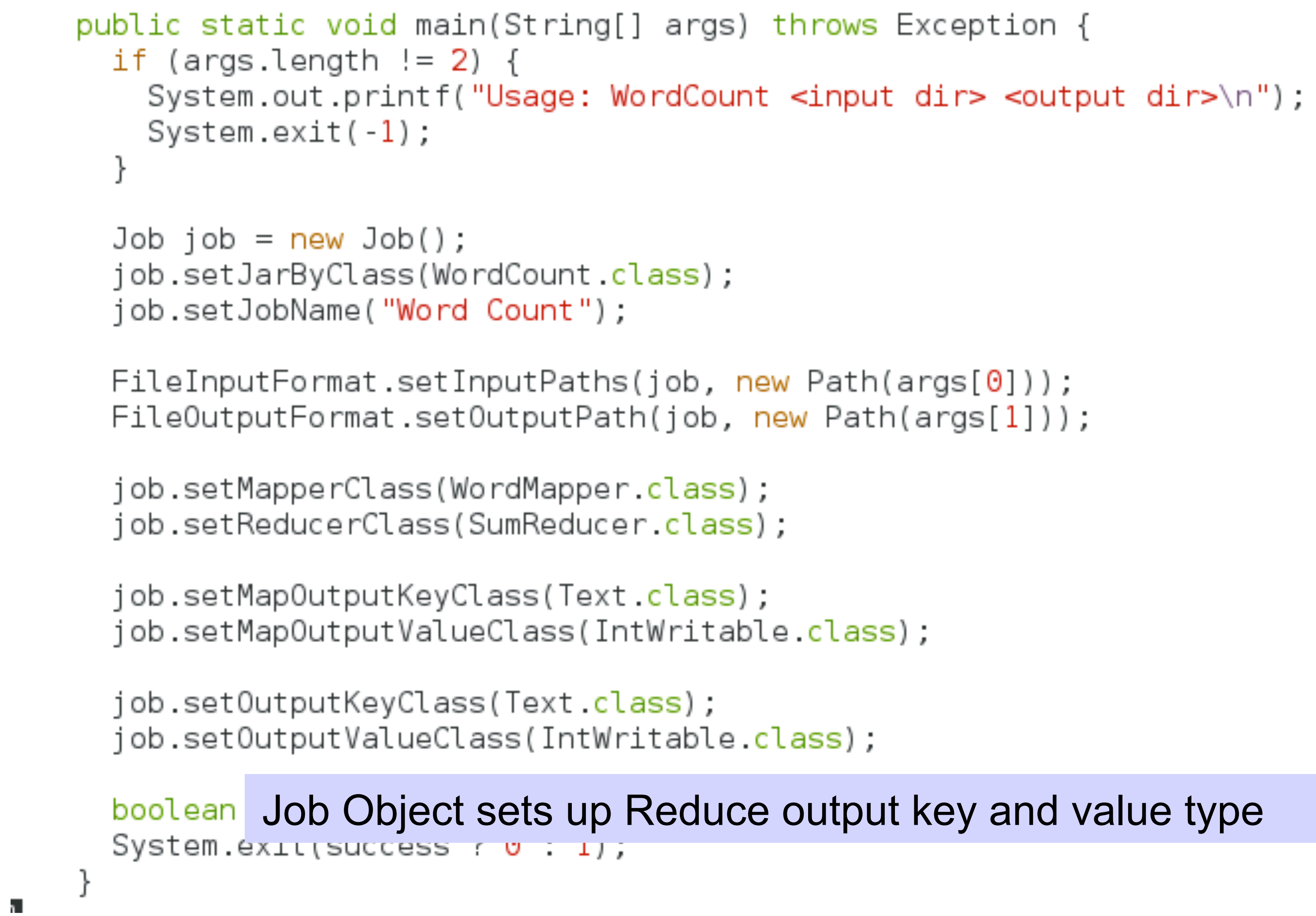
Job Object sets up the Mapper and Recuder class here

Driver: main

```java
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setO
  job.setO

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

Job Object sets up the Intermediate output key and value type, produced by the Mapper

Driver: main



```
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean
  System.exit(success ? 0 : 1);
}
```

Job Object sets up Reduce output key and value type

Driver: main

```
public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.out.printf("Usage: WordCount <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job();
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.setMapperClass(WordMapper.class);
  job.setReducerClass(SumReducer.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  boolean success = job.waitForCompletion(true);
  System.exit(success ? 0 : 1);
}
```

Start the job, wait for completion, then exit with return code

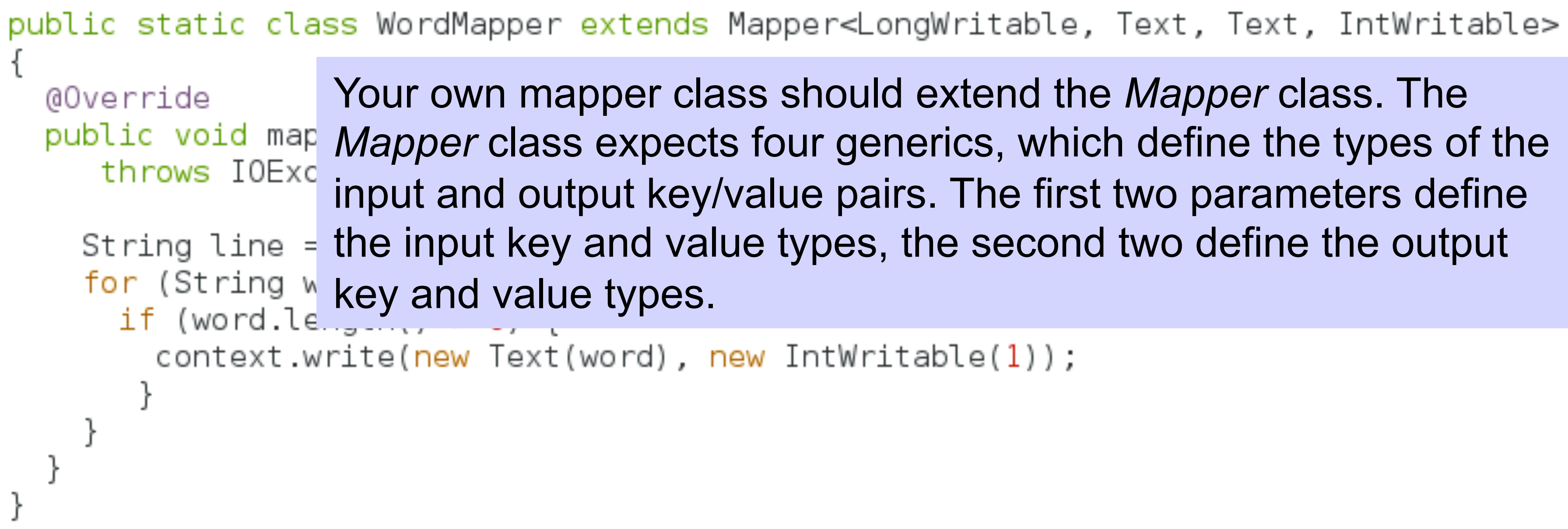
Run the Job

- Two ways to run your MapReduce job:
  - job.waitForCompletion()
    - Blocks (waits for the job to complete before continuing)
  - job.submit()
    - Does not block (driver code continues as the job is running)
- The job determines the proper division of input data into InputSplits, and then sends the job information to the JobTracker daemon on the cluster

Mapper

```
public static class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException {

    String line = value.toString();
    for (String word : line.split("\\W+")) {
      if (word.length() > 0) {
        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}
```

Mapper



```
public static class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map
     throws IOExc

    String line =
    for (String w
      if (word.le
        context.write(new Text(word), new IntWritable(1));
       }
    }
  }
}
```

Your own mapper class should extend the *Mapper* class. The *Mapper* class expects four generics, which define the types of the input and output key/value pairs. The first two parameters define the input key and value types, the second two define the output key and value types.

Mapper

```
public static class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    Stri
    for
      if
        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}
```

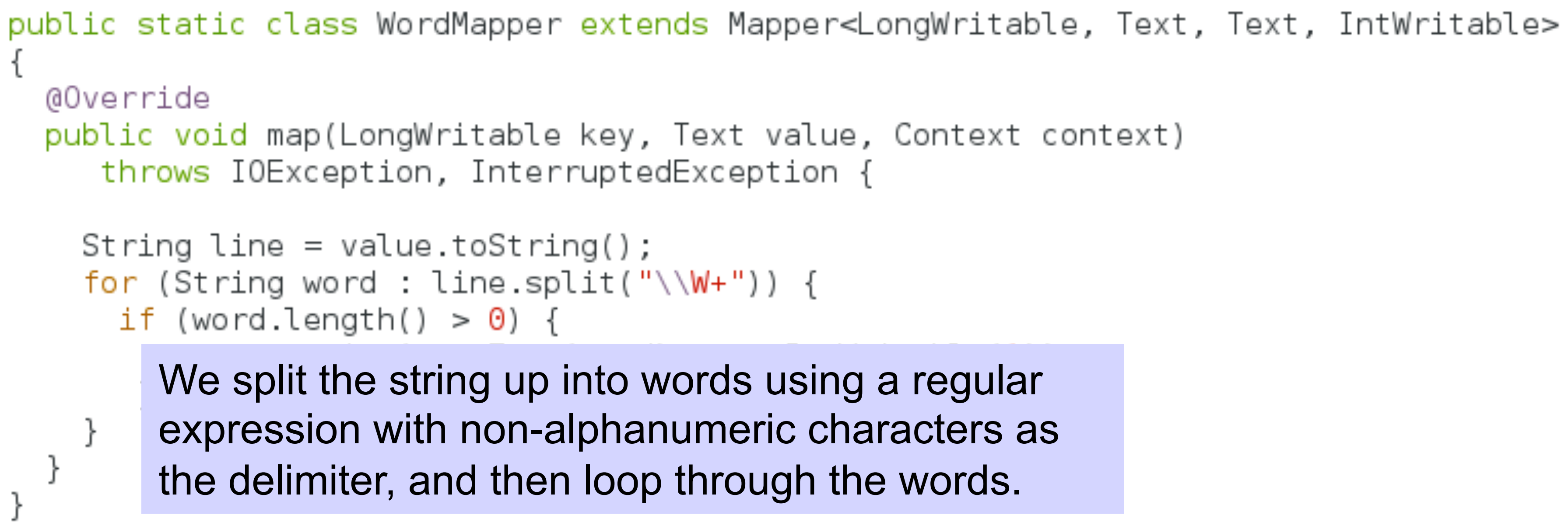
The map() signature. The Context is used to write the intermediate data. It also contains information about the job’s configuration.

Mapper

```
public static class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException {

    String line = value.toString();
    for
      i


    }
  }
}
```

value is the content of each line in the file, which is a Text object. The input *key* would be a long value assigned in default based on the position of Text in input file. If you use TextInputFormat, then it'll be the byte offset of the beginning of the record in the file (rather than the actual line number). We don't care about it here.

Mapper



```
public static class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException {

    String line = value.toString();
    for (String word : line.split("\\W+")) {
      if (word.length() > 0) {

    }
  }
}
```

We split the string up into words using a regular expression with non-alphanumeric characters as the delimiter, and then loop through the words.

Mapper

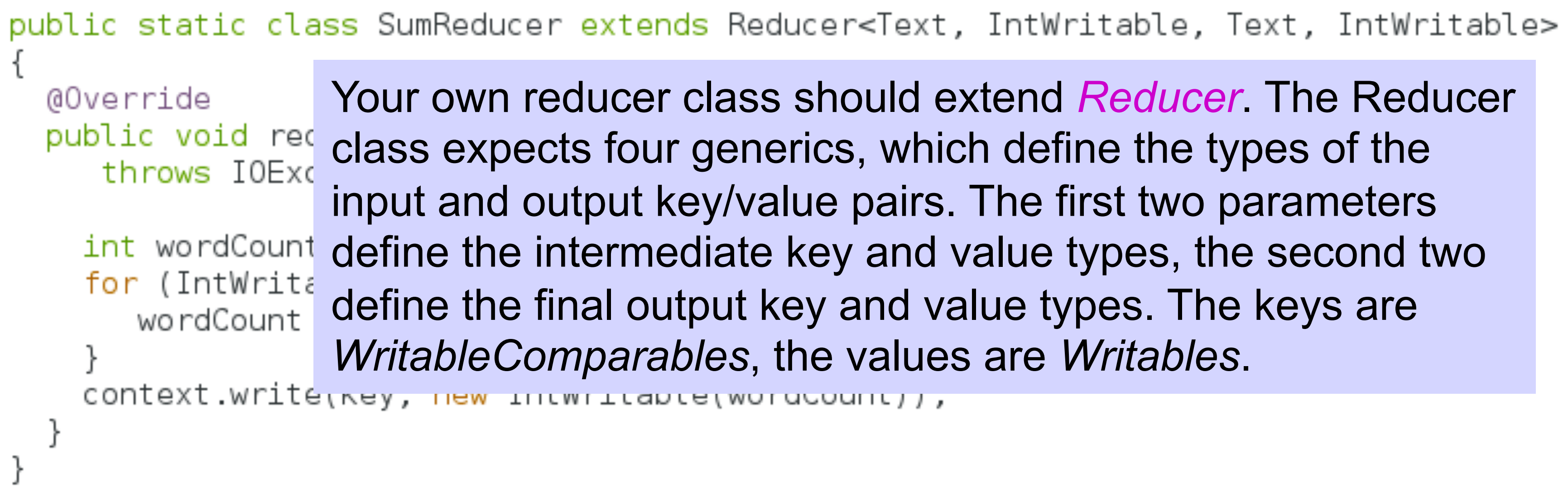
```
public static class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException {

    String line = value.toString();
    for (String word : line.split("\\W+")) {
      if (word.length() > 0) {
        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}
```

To emit a (key, value) pair, we call the *write* method of our Context object. The key will be the word itself, the value will be the number 1.
Recall that the output key must be a WritableComparable, and the value must be a Writable.

Reducer

```
public static class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {

    int wordCount = 0;
    for (IntWritable value : values) {
       wordCount += value.get();
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

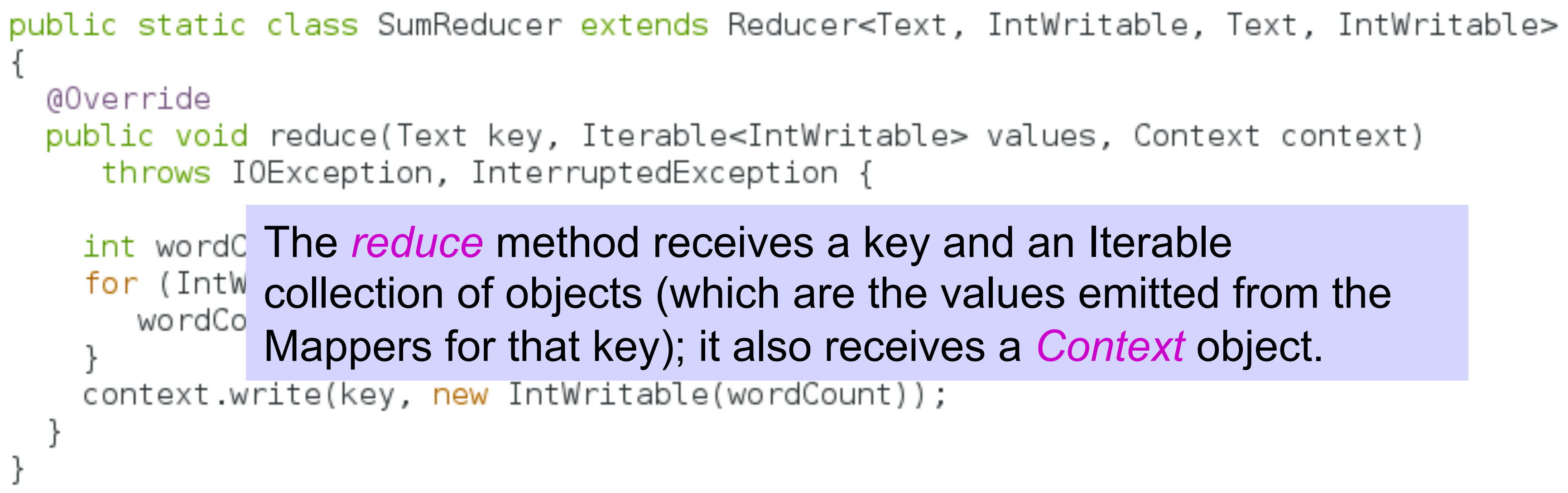
Reducer



```
public static class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  @Override
  public void rec
     throws IOExc

    int wordCount
    for (IntWrita
       wordCount
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

Your own reducer class should extend *Reducer*. The Reducer class expects four generics, which define the types of the input and output key/value pairs. The first two parameters define the intermediate key and value types, the second two define the final output key and value types. The keys are *WritableComparables*, the values are *Writables*.

Reducer



```
public static class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {

    int wordC
    for (IntW
       wordCo
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

The *reduce* method receives a key and an Iterable collection of objects (which are the values emitted from the Mappers for that key); it also receives a *Context* object.

Reducer

```
public static class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {

    int wordCount = 0;
    for (IntWritable value : values) {
       wordCount += value.get();
    }
    conte
  }
}
```

We use the Java for-each syntax to step through all the elements in the collection. In our example, we are merely adding all the values together. We use *value.get()* to retrieve the actual numeric values.

Reducer

```
public static class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {

    int wordCount = 0;
    for (IntWritable value : values) {
       wordCount += value.get();
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

Finally, we write the output key-value pair to HDFS using the *write* method of our *Context* object.
Here key is the word from the mapper, and value is the combined count for each word.

Outline

- Basic MapReduce Program
- **Use Streaming API**
- Implementing ToolRunner

Writing MR in other languages

- Many organizations have developers skilled in languages other than Java, such as
  - Ruby
  - Python
  - Perl
- Hadoop provides Streaming API
  - allows developers to use any language they wish to write Mappers and Reducers
  - As long as the language can read from standard input and write to standard output

Streaming API

- Advantages of the Streaming API:
  - No need for non-Java coders to learn Java
  - Fast development time
  - Ability to use existing code libraries
- Disadvantages of the Streaming API:
  - Performance
  - Primarily suited for handling data that can be represented as text
  - Streaming jobs can use excessive amounts of RAM or fork excessive numbers of processes
  - Although Mappers and Reducers can be written using the Streaming API, Partitioners, InputFormats etc. must still be written in Java

Implement Streaming

- Write separate Mapper and Reducer programs in the language of your choice
  - They will receive input via stdin
  - They should write their output to stdout
- If *TextInputFormat* (the default) is used, the streaming Mapper just receives each line from the file on stdin
  - No key is passed
- Streaming Mapper and streaming Reducer's output should be sent to stdout as key (tab) value (newline)
  - Separators other than tab can be specified

Example Mapper in Perl

```
#!/usr/bin/env perl
while (<>) {                   # Read lines from stdin
  chomp;                       # Get rid of the trailing newline
  (@words) = split /\s+/;      # Create an array of words
  foreach $w (@words) {        # Loop through the array
    print "$w\t1\n";           # Print out the key and value
  }
}
```

Launch a Streaming Job

- Use command line options to specify your mapper/reducer code
  - May use system commands as *grep*, *sed*, etc

```
hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/ \
   streaming/hadoop-streaming*.jar  \
   -input myInputDirs \
   -output myOutputDir \
   -mapper myMapScript.pl \
   -reducer myReduceScript.pl \
   -file myMapScript.pl \
   -file myReduceScript.pl
```

Outline

- Basic MapReduce Program
- Use Streaming API
- **Implementing ToolRunner**

ToolRunner

- You can use ToolRunner in MapReduce driver classes
  - This is not required, but is a best practice
- ToolRunner uses the GenericOptionsParser class internally
  - Allows you to specify configuration options on the command line
  - Also allows you to specify items for the Distributed Cache on the command line

ToolRunner Implementation

- Import the relevant classes in your driver

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

- Driver class should extend Configured and implements Tool

```
public class WordCountTr extends Configured implements Tool{
```

ToolRunner Implementation

- main class will call *ToolRunner.run()*

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

- Create a *run()* method
  - The Job object now created using the *getConf()* method

```
public int run(String[] args) throws Exception{
 if (args.length != 2) {
    System.out.printf("Usage: WordCountTr [generic options] <input dir> <output dir>\n");
    System.exit(-1);
  }

  Job job = new Job(getConf());
  job.setJarByClass(WordCountTr.class);
  job.setJobName("Word Count ToolRunner");
```

ToolRunner Command Line Options

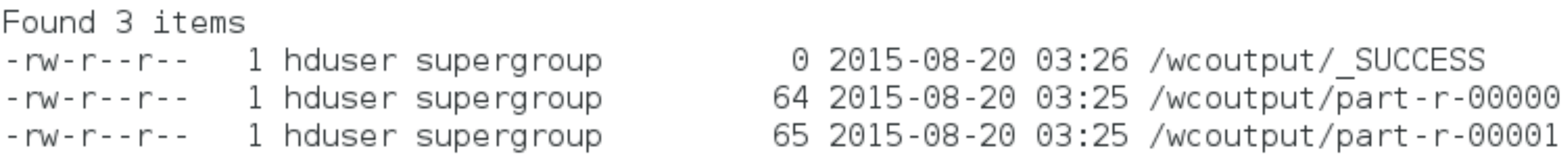
- ToolRunner allows the user to specify configuration options on the command line
- Commonly used to specify Hadoop properties using the *-D* flag
  - Will override any default or site properties in the configuration
  - But will not override those set in the driver code

```
hadoop jar wc.jar WordCountTr -D mapreduce.job.reduces=2 /wcinput /wcoutput
```

- Note that -D options must appear before any additional program arguments
- Can specify an XML configuration file with –conf
- Can specify the default filesystem with -fs uri
  - Shortcut for *–D fs.defaultFS=uri*

More observations

- Note that the output has two files
  - Each reducer write to one output file



```
Found 3 items
-rw-r--r--   1 hduser supergroup          0 2015-08-20 03:26 /wcoutput/_SUCCESS
-rw-r--r--   1 hduser supergroup         64 2015-08-20 03:25 /wcoutput/part-r-00000
-rw-r--r--   1 hduser supergroup         65 2015-08-20 03:25 /wcoutput/part-r-00001
```