Writing a MapReduce Program in Java

Chapter 3.1

Writing a MapReduce Program in Java

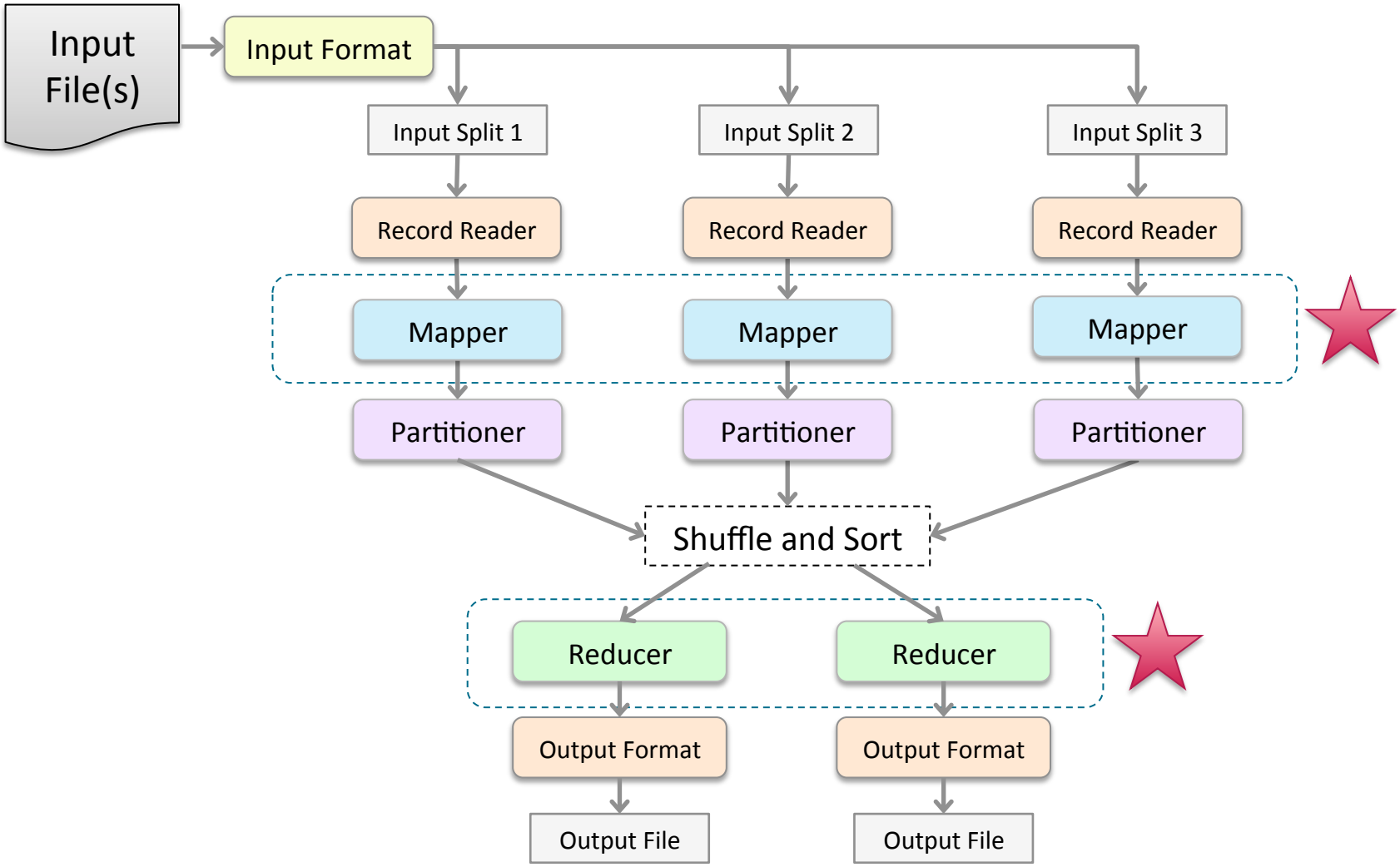
- **Basic MapReduce API concepts**
- **How to write MapReduce drivers, Mappers, and Reducers in Java**
- **The differences between the old and new MapReduce APIs**

Chapter Topics

Writing a MapReduce Program in Java

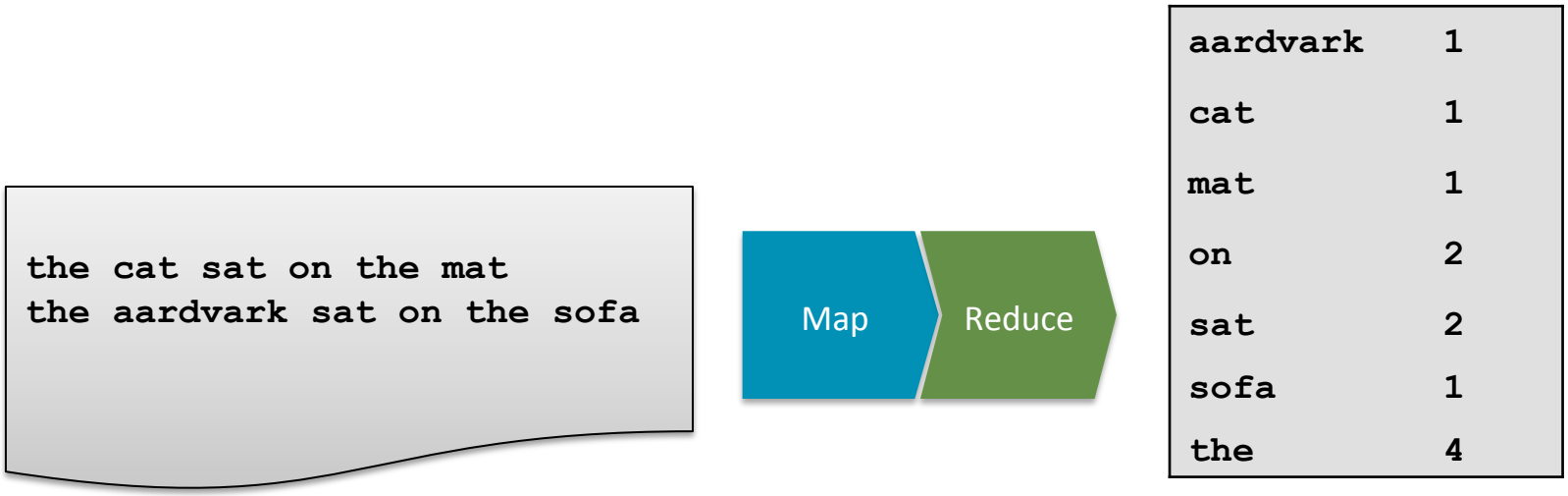
- **Basic MapReduce API Concepts**
- Writing MapReduce Applications in Java
  - The Driver
  - The Mapper
  - The Reducer
- Speeding up Hadoop Development by Using Eclipse
- Differences Between the Old and New MapReduce APIs

Review: The MapReduce Flow



A Sample MapReduce Program: WordCount

- **In an earlier chapter, you ran a sample MapReduce program**
  - WordCount, which counted the number of occurrences of each unique word in a set of files
- **In this chapter, we will examine the code for WordCount**
  - This will demonstrate the Hadoop API



Our MapReduce Program: WordCount

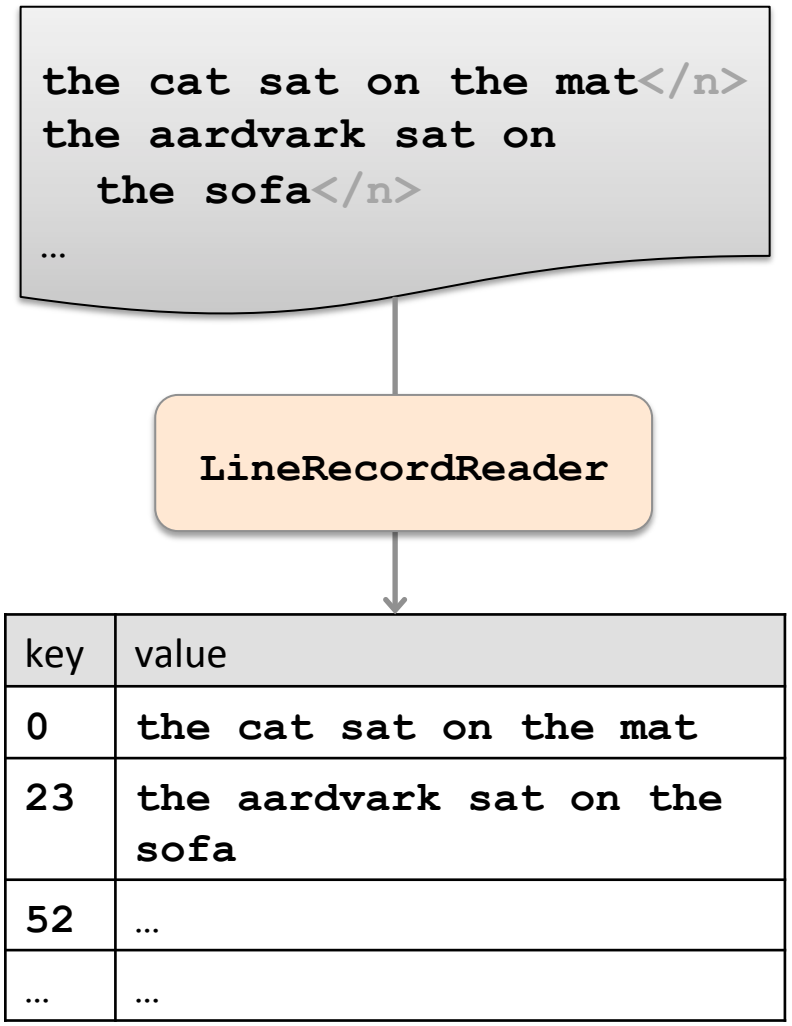
- **To investigate the API, we will dissect the WordCount program we covered in the previous chapter**
- **This consists of three portions**
  - The driver code
    - Code that runs on the client to configure and submit the job
  - The Mapper
  - The Reducer
- **Before we look at the code, we need to cover some basic Hadoop API concepts**

Getting Data to the Mapper

- **The data passed to the Mapper is specified by an *InputFormat***
  - Specified in the driver code
  - Defines the location of the input data
    - Typically a file or directory
  - Determines how to split the input data into *input splits*
    - Each Mapper deals with a single input split
  - Creates a RecordReader object
    - RecordReader parses the input data into key/value pairs to pass to the Mapper

Example: `TextInputFormat`

- **`TextInputFormat`**
  - The default
  - Creates `LineRecordReader` objects
  - Treats each `\n`-terminated line of a file as a value
  - Key is the byte offset of that line within the file



key	value
0	the cat sat on the mat
23	the aardvark sat on the sofa
52	…
…	…

Other Standard InputFormats

- **`FileInputFormat`**
  - Abstract base class used for all file-based InputFormats
- **`KeyValueTextInputFormat`**
  - Maps `\n`-terminated lines as 'key [*separator*] 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 are Objects

- **Keys and values in Hadoop are Java `Objects`**
  - Not primitives
- **Values are objects which implement `Writable`**
- **Keys are objects which implement `WritableComparable`**

What is `Writable`?

- **The `Writable` interface makes serialization quick and easy for Hadoop**
- **Any value's type must implement the `Writable` interface**
- **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.

What is `WritableComparable`?

- **A `WritableComparable` is a `Writable` which is also `Comparable`**
  - Two `WritableComparable`s can be compared against each other to determine their 'order'
  - Keys must be `WritableComparable`s because they are passed to the Reducer in sorted order
  - We will talk more about `WritableComparable`s later
- **Note that despite their names, all Hadoop box classes implement both `Writable` and `WritableComparable`**
  - For example, `IntWritable` is actually a `WritableComparable`

Chapter Topics

Writing a MapReduce Program in Java

- Basic MapReduce API Concepts
- **Writing MapReduce Applications in Java**
  - **The Driver**
  - The Mapper
  - The Reducer
- Speeding up Hadoop Development by Using Eclipse
- Differences Between the Old and New MapReduce APIs

The Driver Code: Introduction

- **The driver code runs on the client machine**
- **It configures the job, then submits it to the cluster**

The Driver: Complete Code (1)

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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;

public class WordCount {

  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);
```

The Driver: Complete Code (2)

```
    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);
  }
}
```

The Driver: Import Statements

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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;

public
  publ
    if
    }

    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);
```

You will typically import these classes into every MapReduce job you write. We will omit the `import` statements in future slides for brevity.

The Driver: Main Code

```
public class WordCount {
  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);
  }
}
```

The Driver Class: `main` Method

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.println("usage: WordCount <input dir> <output dir>");
      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);
  }
}
```

The `main` method accepts two command-line arguments: the input and output directories.

Sanity Checking The Job's Invocation

```
public class WordCount {
  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.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);
  }
}
```

The first step is to ensure we have been given two command-line arguments. If not, print a help message and exit.

Configuring The Job With the `Job` Object

```
public class WordCount {
  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");

    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);
  }
}
```

To configure the job, create a new `Job` object. Identify the Jar which contains the Mapper and Reducer by specifying a class in that Jar.

Creating a New `Job` Object

- **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 use Hadoop's default values**
- **You can also use the `Job` object to submit the job, control its execution, and query its state**

Configuring the Job: Setting the Name

```
public class WordCount {
  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");

    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);
  }
}
```

Give the job a meaningful name.

Configuring the Job: Specifying Input and Output Directories

```
public class WordCount {
  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.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

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

Next, specify the input directory from which data will be read, and the output directory to which final output will be written.

Configuring the Job: Specifying the InputFormat

- **The default InputFormat (`TextInputFormat`) will be used unless you specify otherwise**
- **To use an InputFormat other than the default, use e.g.**

```
job.setInputFormatClass(KeyValueTextInputFormat.class)
```

Configuring the Job: Determining Which Files To Read

- **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**
- **More advanced filtering can be performed by implementing a `PathFilter`**
  - Interface with a method named `accept`
    - Takes a path to a file, returns `true` or `false` depending on whether or not the file should be processed

Configuring the Job: Specifying Final Output With OutputFormat

- **`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)`
- **We will discuss OutputFormats in more depth in a later chapter**

Configuring the Job: Specifying the Mapper and Reducer Classes

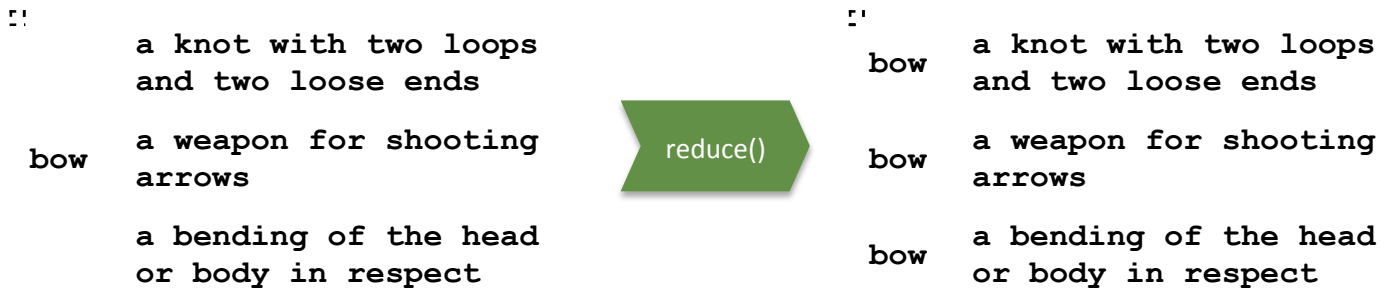
```
public class WordCount {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.printf("Usage: WordCount <input dir> <output dir>\n");
      System.exit(-1);
    }
    Job job = new Job();
    job.setJarByClass(WordCount.class);
    job.setJobName("Word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

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

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

    job
    job

    job
    job

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

Give the `Job` object information about which classes are to be instantiated as the Mapper and Reducer.

Default Mapper and Reducer Classes

- **Setting the Mapper and Reducer classes is optional**
- **If not set in your driver code, Hadoop uses its defaults**
  - **IdentityMapper**

  - **IdentityReducer**



Configuring the Job: Specifying the Intermediate Data Types

```
public class WordCount {
  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);

    jo
    jo

    bo
    Sy
  }
}
```

Specify the types for the intermediate output keys and values produced by the Mapper.

Configuring the Job: Specifying the Final Output Data Types

```
public class WordCount {
  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);
    j
    j
    j

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

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

Specify the types for the Reducer's output keys and values.

Running The Job (1)

```
public class WordCount {
  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]));

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

Start the job and wait for it to complete. Parameter is a Boolean, specifying verbosity: if true, display progress to the user. Finally, exit with a return code.

Running The Job (2)

- **There are 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 client determines the proper division of input data into InputSplits, and then sends the job information to the JobTracker daemon on the cluster**

Reprise: Driver Code

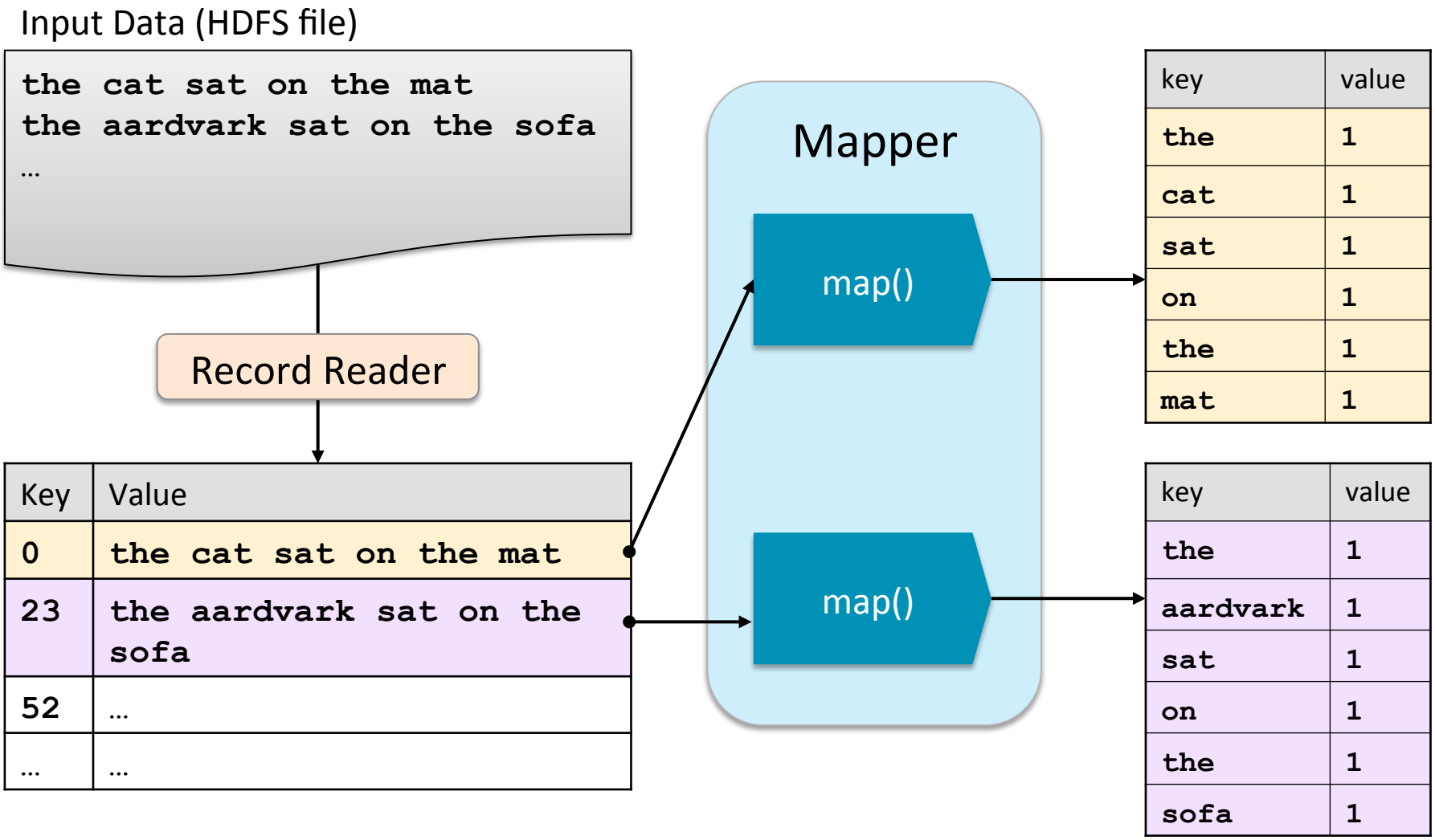
```
public class WordCount {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.printf("Usage: WordCount <input dir> <output dir>\n");
      System.exit(-1);
    }
    Job job = new Job();
    job.setJarByClass(WordCount.class);
    job.setJobName("Word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

Chapter Topics

Writing a MapReduce Program in Java

- Basic MapReduce API Concepts
- **Writing MapReduce Applications in Java**
  - The Driver
  - **The Mapper**
  - The Reducer
- Speeding up Hadoop Development by Using Eclipse
- Differences Between the Old and New MapReduce APIs

WordCount Mapper Review



The Mapper: Complete Code

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public 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));
      }
    }
  }
}
```

The Mapper: `import` Statements

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

publ
IntW
  @O
  pu
    for (String word : line.split("\\W+")) {
      if (word.length() > 0) {

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

You will typically import `java.io.IOException`, and the `org.apache.hadoop` classes shown, in every Mapper you write. We will omit the `import` statements in future slides for brevity.

The Mapper: Main Code

```
public 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));
      }
    }
  }
}
```

The Mapper: Class Declaration (1)

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Overr
  public
      th

    String line = value.toString();

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

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

Mapper classes extend the `Mapper` base class.

The Mapper: Class Declaration (2)

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

    Str

    for
      i

    }
  }
}
```

Input key and value types

Intermediate key and value types

Specify generic types that declare four type parameters: the input key and value types, and the output (intermediate) key and value types. Keys must be `WritableComparable`, and values must be `Writable`.

The `map` Method

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

    String

    for (St
      if (w

        con
      }
    }
  }
}
```

The `map` method is passed a key, a value, and a `Context` object. The `Context` is used to write the intermediate data. It also contains information about the job's configuration.

The `map` Method: Processing The Line (1)

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

`value` is a `Text` object, so retrieve the string it contains.

The `map` Method: Processing The Line (2)

```
public 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) {

        cont
      }
    }
  }
}
```

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

The `map` Method: Outputting Intermediate Data

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Ov
  pub
      context.write(new Text(word), new IntWritable(1));
    }
  }
}
```

To emit a (key, value) pair, call the `write` method of the `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`.

Reprise: The `map` Method

```
public 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));
      }
    }
  }
}
```

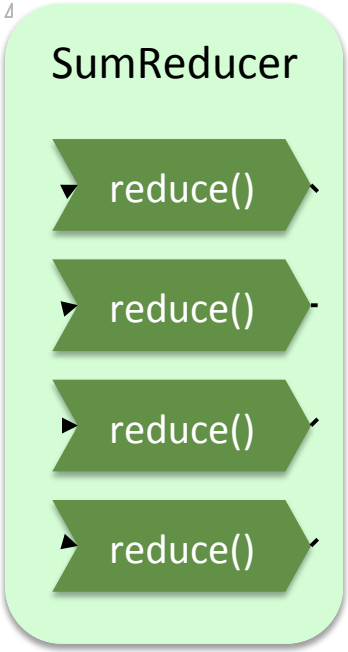
Chapter Topics

Writing a MapReduce Program in Java

- Basic MapReduce API Concepts
- **Writing MapReduce Applications in Java**
  - The Driver
  - The Mapper
  - **The Reducer**
- Speeding up Hadoop Development by Using Eclipse
- Differences Between the Old and New MapReduce APIs

WordCount Review: SumReducer

key	value list
on	1,1
sat	1,1
sofa	1
the	1,1,1,1,1,1



on	2
sat	2
sofa	1
the	6

The Reducer: Complete Code

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public 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));
  }
}
```

The Reducer: Import Statements

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

As with the Mapper, you will typically import `java.io.IOException`, and the `org.apache.hadoop` classes shown, in every Reducer you write. We will omit the import statements in future slides for brevity.

The Reducer: Main Code

```
public 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));
  }
}
```

The Reducer: Class Declaration

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{

  @Override
  public void reduce(Text key, Iterable<
      throws IOException, InterruptedException {

    int

    for
      wo
    }

    cont
  }
}
```

Intermediate key and value types

Output key and value types

Reducer classes extend the `Reducer` base class. The four generic type parameters are: input (intermediate) key and value types, and final output key and value types.

The `reduce` Method

```
public 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
      wordCou
    }

    context.w
  }
}
```

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.

The `reduce` Method: Processing The Values

```
public 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();
    }

    cont
  }
}
```

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 value each time.

The `reduce` Method: Writing The Final Output

```
public class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int

    for
      wo
    }

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

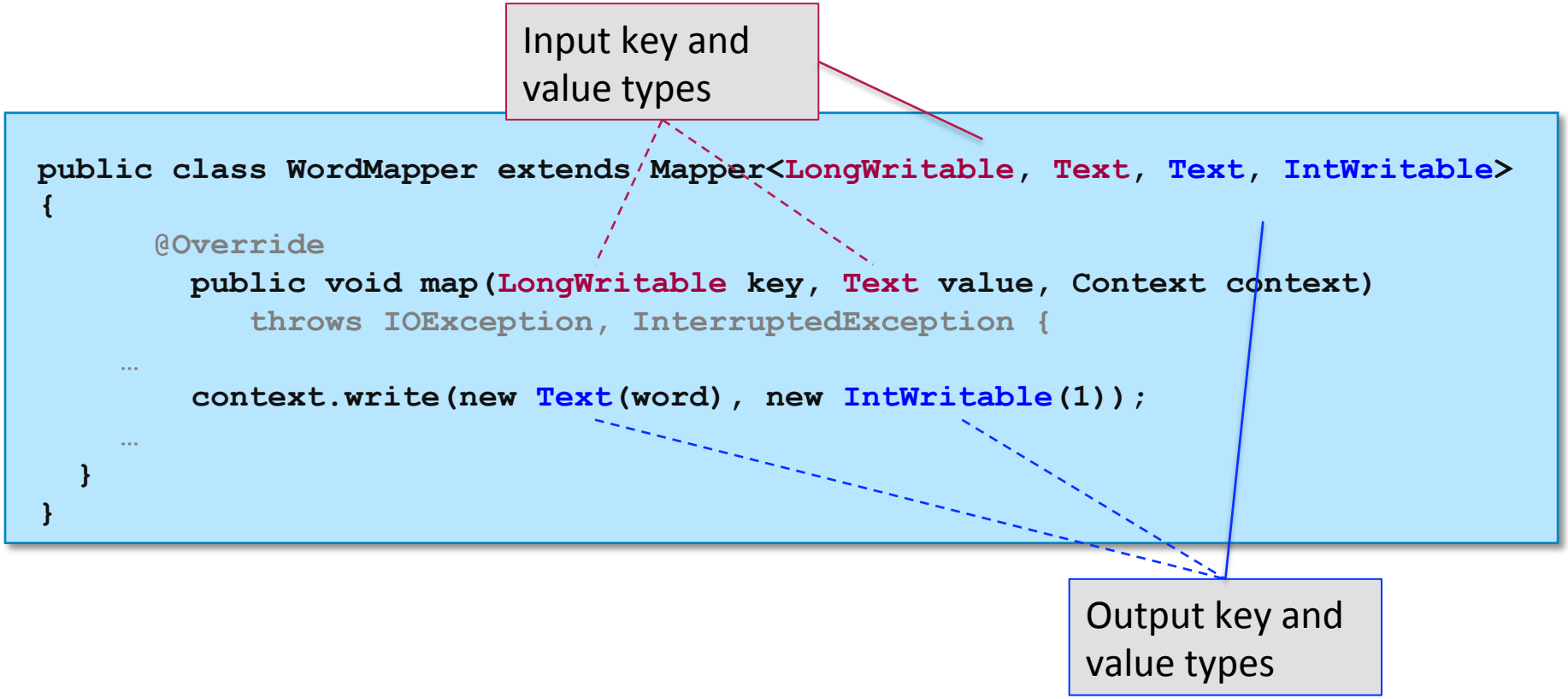
Finally, we write the output key-value pair to HDFS using the `write` method of our `Context` object.

Reprise: The Reducer Main Code

```
public 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));
  }
}
```

Ensure Types Match (1)

- **Mappers and Reducers declare input and output type parameters**
- **These must match the types used in the class**



Ensure Types Match (2)

- **Output types must also match those set in the driver**

```
public class WordMapper extends Mapper<LongWritable,
Text, Text, IntWritable> {
    …
}
```

Mapper

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    …
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);

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

driver code

```
public class SumReducer extends Reducer<Text,
IntWritable, Text, IntWritable> {
…
}
```

Reducer

Chapter Topics

Writing a MapReduce Program in Java

- Basic MapReduce API Concepts
- Writing MapReduce Applications in Java
  - The Driver
  - The Mapper
  - The Reducer
- **Speeding up Hadoop Development by Using Eclipse**
- Differences Between the Old and New MapReduce APIs

Integrated Development Environments

- **There are many Integrated Development Environments (IDEs) available**
- **Eclipse is one such IDE**
  - Open source
  - Very popular among Java developers
  - Has plug-ins to speed development in several different languages

Chapter Topics

Writing a MapReduce Program in Java

- Basic MapReduce API Concepts
- Writing MapReduce Applications in Java
  - The Driver
  - The Mapper
  - The Reducer
- Speeding up Hadoop Development by Using Eclipse
- **Differences Between the Old and New MapReduce APIs**

What Is The Old API?

- **When Hadoop 0.20 was released, a 'New API' was introduced**
  - Designed to make the API easier to evolve in the future
  - Favors abstract classes over interfaces
- **Some developers still use the Old API**
  - Until CDH4, the New API was not absolutely feature-complete
- **All the code examples in this course use the New API**

New API vs. Old API: Some Key Differences (1)

New API

```
import org.apache.hadoop.mapreduce.*
```

Driver code:

```
Configuration conf = new Configuration();
Job job = new Job(conf);
job.setJarByClass(Driver.class);
job.setSomeProperty(...);
...
job.waitForCompletion(true);
```

Mapper:

```
public class MyMapper extends Mapper {

  public void map(Keytype k, Valuetype v,
                                  Context c) {
    ...
    c.write(key, val);
  }
}
```

Old API

```
import org.apache.hadoop.mapred.*
```

Driver code:

```
JobConf conf = new JobConf(Driver.class);
conf.setSomeProperty(...);
...
JobClient.runJob(conf);
```

Mapper:

```
public class MyMapper extends MapReduceBase
                            implements Mapper {

  public void map(Keytype k, Valuetype v,
            OutputCollector o, Reporter r) {
    ...
    o.collect(key, val);
  }
}
```

New API vs. Old API: Some Key Differences (2)

New API	Old API

Reducer:
 	
`public class MyReducer extends Reducer {`
 	
`  public void reduce(Keytype k,`
`  Iterable<Valuetype> v, Context c) {`
`  for(Valuetype eachval : v) {`
`  // process eachval`
`  c.write(key, val);`
`  }`
`  }`
`}`
	Reducer:
	
	`public class MyReducer extends MapReduceBase`
	`  implements Reducer {`
	
	`  public void reduce(Keytype k,`
	`  Iterator<Valuetype> v,`
	`  OutputCollector o, Reporter r) {`
	`  while(v.hasnext()) {`
	`  // process v.next()`
	`  o.collect(key, val);`
	`  }`
	`  }`
	`}`

`setup(Context c)` **(See later)**	`configure(JobConf job)`
`cleanup(Context c)` **(See later)**	`close()`

MRv1 vs MRv2, Old API vs New API

- There is a lot of confusion about the New and Old APIs, and MapReduce version 1 and MapReduce version 2
- The chart below should clarify what is available with each version of MapReduce

	Old API	New API
MapReduce v1	✔	✔
MapReduce v2	✔	✔

- Summary: Code using either the Old API or the New API will run under MRv1 and MRv2

Key Points

- **InputFormat**
  - Parses input files into key/value pairs
- **WritableComparable, Writable classes**
  - "Box" or "wrapper" classes to pass keys and values
- **Driver**
  - Sets InputFormat and input and output types
  - Specifies classes for the Mapper and Reducer
- **Mapper**
  - `map()` method takes a key/value pair
  - Call `Context.write()` to output intermediate key/value pairs
- **Reducer**
  - `reduce()` method takes a key and iterable list of values
  - Call `Context.write()` to output final key/value pairs

Bibliography

The following offer more information on topics discussed in this chapter

- **References for InputFormats**
  - TDG 3e pages 237 and 245-251
  - TDG 3e page 247
- **Configuring jobs: See TDG 3e pages 65-66 for more explanation and examples.**
- **Running jobs: For more information, see page 190 of TDG 3e.**
- **Old API vs. new API: See TDG 3e pages 27-30.**

Writing a MapReduce Program Using Streaming

Chapter 3.2

Chapter Topics

Writing a MapReduce Program Using Streaming

- **Writing Mappers and Reducers with the Streaming API**

The Streaming API: Motivation

- **Many organizations have developers skilled in languages other than Java, such as**
  - Ruby
  - Python
  - Perl
- **The 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

The Streaming API: Advantages and Disadvantages

- **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

How Streaming Works

- **To implement streaming, write separate Mapper and Reducer programs in the language(s) 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
- **Mapper and Reducer output should be sent to `stdout` as**
  - `key` *`[tab]`* `value` *`[newline]`*
- **Separators other than tab can be specified**

Streaming: Example Mapper

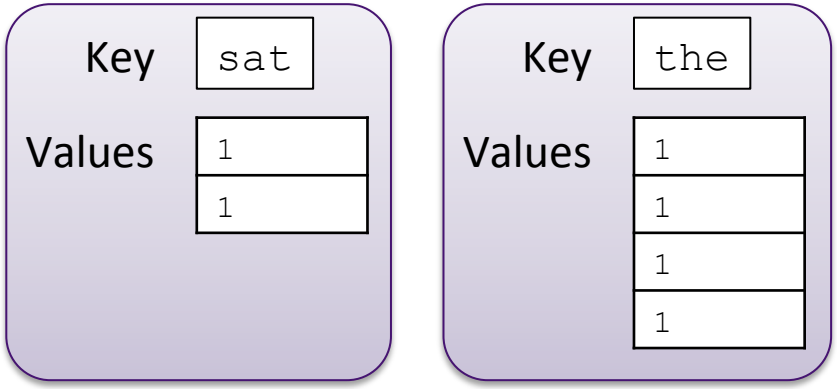
- **Example streaming wordcount Mapper:**

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

Streaming Reducers: Caution

- **Recall that in Java, all the values associated with a key are passed to the Reducer as an `Iterable`**
- **Using Hadoop Streaming, the Reducer receives its input as one key/value pair per line**
- **Your code will have to keep track of the key so that it can detect when values from a new key start**

Java: Iterable Objects



Streaming: stdin

```
sat     1
sat     1
the     1
the     1
the     1
the     1
```

Streaming: Example Reducer

- **Example streaming wordcount Reducer:**

```
#!/usr/bin/env perl
$sum = 0;
$last = "";
while (<>) {                          # read lines from stdin
  ($key, $value) = split /\t/;        # obtain the key and value
  $last = $key if $last eq "";        # first time through
  if ($last ne $key) {                # has key has changed?
    print "$last\t$sum\n";            # if so output last key/value
    $last = $key;                     # start with the new key
    $sum = 0;                         # reset sum for the new key
  }
  $sum += $value;                     # add value to tally sum for key
}
print "$key\t$sum\n";                 # print the final pair
```

Launching a Streaming Job

- **To launch a Streaming job, use e.g.:**

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

- **Many other command-line options are available**
  - See the documentation for full details
- **Note that system commands can be used as a Streaming Mapper or Reducer**
  - For example: `awk`, `grep`, `sed`, or `wc`

Key Points

- **The Hadoop Streaming API allows you to write Mappers and Reducers in any language**
  - Data is read from `stdin` and written to `stdout`
- **Other Hadoop components (InputFormats, Partitioners, etc.) still require Java**

Bibliography

The following offer more information on topics discussed in this chapter

- **Streaming:**
  - TDG 3e, pages 36-40
  - http://hadoop.apache.org/common/docs/r0.20.203.0/streaming.html
  - http://code.google.com/p/hadoop-stream-mapreduce/wiki/Performanc
  - http://stackoverflow.com/questions/1482282/java-vs-python-on-hadoop
  - http://hadoop.apache.org/common/docs/r0.20.203.0/streaming.html#Hadoop+Comparator+Class

Unit Testing MapReduce Programs

Chapter 3.3

Unit Testing MapReduce Programs

- **What is unit testing, why write unit tests**
- **What is the JUnit testing framework, and how does MRUnit build on the JUnit framework**
- **How to write unit tests with MRUnit**
- **How to run unit tests**

Chapter Topics

Unit Testing MapReduce Programs

- **Unit Testing**
- The JUnit and MRUnit Testing Frameworks
- Writing Unit Tests with MRUnit
- Running Unit Tests

An Introduction to Unit Testing

- **A 'unit' is a small piece of your code**
  - A small piece of functionality
- **A unit test verifies the correctness of that unit of code**
  - A purist might say that in a well-written unit test, only a single 'thing' should be able to fail
  - Generally accepted rule-of-thumb: a unit test should take less than a second to complete

Why Write Unit Tests?

- **Unit testing provides verification that your code is functioning correctly**
- **Much faster than testing your entire program each time you modify the code**
  - Fastest MapReduce job on a cluster will take many seconds
    - Even in pseudo-distributed mode
  - Even running in LocalJobRunner mode will take several seconds
    - LocalJobRunner mode is discussed later in the course
  - Unit tests help you iterate faster in your code development

Chapter Topics

Unit Testing MapReduce Programs

- Unit Testing
- **The JUnit and MRUnit testing frameworks**
- Writing Unit Tests with MRUnit
- Running Unit Tests

Why MRUnit?

- **JUnit is a popular Java unit testing framework**
- **Problem: JUnit cannot be used directly to test Mappers or Reducers**
  - Unit tests require mocking up classes in the MapReduce framework
    - A lot of work
- **MRUnit is built on top of JUnit**
  - Works with the mockito framework to provide required mock objects
- **Allows you to test your code from within an IDE**
  - Much easier to debug

JUnit Basics (1)

- **`@Test`**
  - Java annotation
  - Indicates that this method is a test which JUnit should execute
- **`@Before`**
  - Java annotation
  - Tells JUnit to call this method before every `@Test` method
    - Two `@Test` methods would result in the `@Before` method being called twice

JUnit Basics (2)

- **JUnit test methods:**
  - `assertEquals()`, `assertNotNull()` etc
    - Fail if the conditions of the statement are not met
  - `fail(`*`msg`*`)`
    - Explicitly fails the test with the given error message
- **With a JUnit test open in Eclipse, run all tests in the class by going to Run ➔ Run**
- **Eclipse also provides functionality to run all JUnit tests in your project**
- **Other IDEs have similar functionality**

JUnit: Example Code

```
import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

public class JUnitHelloWorld {
  protected String s;
  @Before
  public void setup() {
    s = "HELLO WORLD";
  }
  @Test
  public void testHelloWorldSuccess() {
    s = s.toLowerCase();
    assertEquals("hello world", s);
  }
  // following will fail even if testHelloWorldSuccess is run first
  @Test
  public void testHelloWorldFail() {
    assertEquals("hello world", s);
  }
}
```

Chapter Topics

Unit Testing MapReduce Programs

- Unit Testing
- The JUnit and MRUnit Testing Frameworks
- **Writing unit tests with MRUnit**
- Running Unit Tests

Using MRUnit to Test MapReduce Code

- **MRUnit builds on top of JUnit**
- **Provides a mock `InputSplit` and other classes**
- **Can test just the Mapper, just the Reducer, or the full MapReduce flow**

MRUnit: Example Code – Mapper Unit Test (1)

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class TestWordCount {
  MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

  @Before
  public void setUp() {
    WordMapper mapper = new WordMapper();
    mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
    mapDriver.setMapper(mapper);
  }

  @Test
  public void testMapper() {
    mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
    mapDriver.withOutput(new Text("cat"), new IntWritable(1));
    mapDriver.withOutput(new Text("dog"), new IntWritable(1));
    mapDriver.runTest();
  }
}
```

MRUnit: Example Code – Mapper Unit Test (2)

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.Before;
import org.junit.Test;

public
  MapD

  @Bef
  publ
    Wo
    mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
    mapDriver.setMapper(mapper);
  }

  @Test
  public void testMapper() {
    mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
    mapDriver.withOutput(new Text("cat"), new IntWritable(1));
    mapDriver.withOutput(new Text("dog"), new IntWritable(1));
    mapDriver.runTest();
  }
}
```

Import the relevant JUnit classes and the MRUnit `MapDriver` class as we will be writing a unit test for our Mapper.

MRUnit: Example Code – Mapper Unit Test (3)

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class TestWordCount {
  MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

  @Bef
  publ
    Wo
    ma
    mapDriver.setMapper(mapper);
  }

  @Test
  public void testMapper() {
    mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
    mapDriver.withOutput(new Text("cat"), new IntWritable(1));
    mapDriver.withOutput(new Text("dog"), new IntWritable(1));
    mapDriver.runTest();
  }
}
```

`MapDriver` is an MRUnit class (not a user-defined driver).

MRUnit: Example Code – Mapper Unit Test (4)

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class TestWordCount {
  MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

  @Before
  public void setUp() {
    WordMapper mapper = new WordMapper();
    mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
    mapDriver.setMapper(mapper);
  }

  @Tes
  publ
    ma
    ma
    ma
    mapDriver.runTest();
  }
}
```

Set up the test. This method will be called before every test, just as with JUnit.

MRUnit: Example Code – Mapper Unit Test (5)

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class TestWordCount {
  MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

  @Before
  public
    WordM
    mapDr
    mapDr
  }

  @Test
  public void testMapper() {
    mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
    mapDriver.withOutput(new Text("cat"), new IntWritable(1));
    mapDriver.withOutput(new Text("dog"), new IntWritable(1));
    mapDriver.runTest();
  }
}
```

The test itself. Note that the order in which the output is specified is important – it must match the order in which the output will be created by the Mapper.

MRUnit Drivers (1)

- **MRUnit has a `MapDriver`, a `ReduceDriver`, and a `MapReduceDriver`**
- **Methods to specify test input and output:**
  - `withInput`
    - Specifies input to the Mapper/Reducer
    - Builder method that can be chained
  - `withOutput`
    - Specifies expected output from the Mapper/Reducer
    - Builder method that can be chained
  - `addOutput`
    - Similar to `withOutput` but returns `void`

MRUnit Drivers (2)

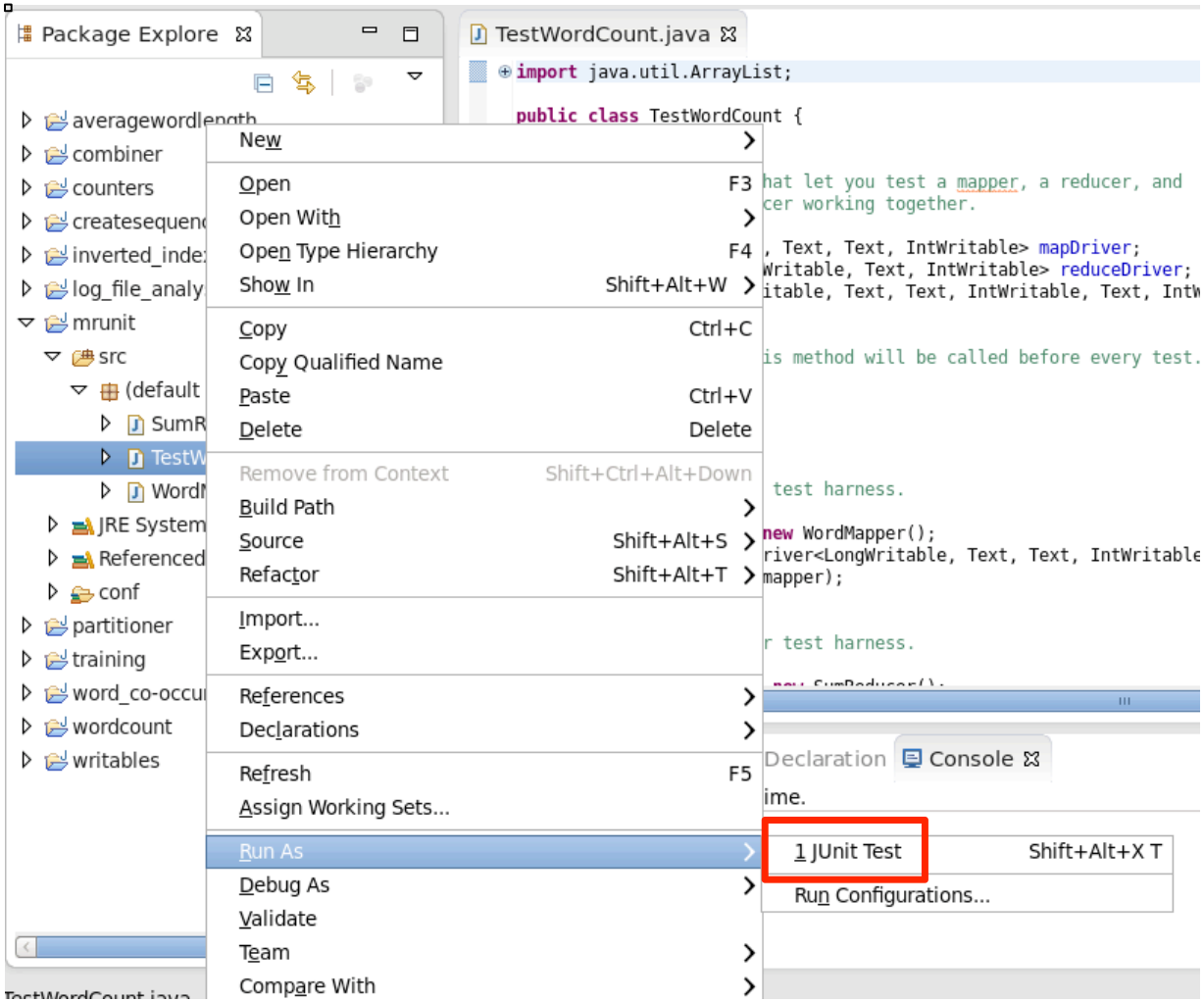
- **Methods to run tests:**
  - `runTest`
    - Runs the test and verifies the output
  - `run`
    - Runs the test and returns the result set
    - Ignores previous `withOutput` and `addOutput` calls
- **Drivers take a single (key, value) pair as input**
- **Can take multiple (key, value) pairs as expected output**
- **If you are calling `driver.runTest()` or `driver.run()` multiple times, call `driver.resetOutput()` between each call**
  - MRUnit will fail if you do not do this

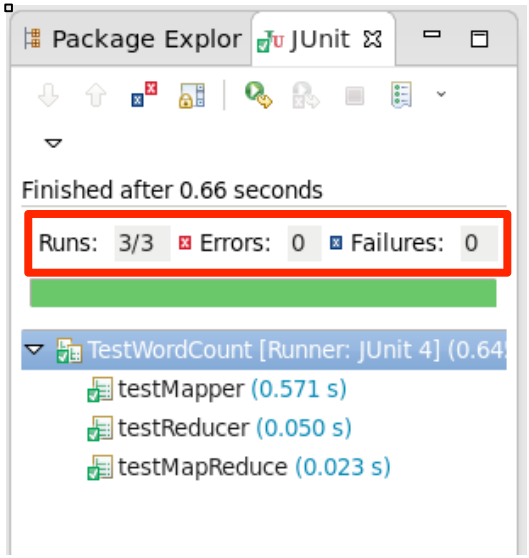
Chapter Topics

Unit Testing MapReduce Programs

- Unit Testing
- The JUnit and MRUnit Testing Frameworks
- Writing Unit Tests with MRUnit
- **Running Unit Tests**
- Hands-On Exercise: Writing Unit Tests with the MRUnit Framework

Running Unit Tests From Eclipse





Compiling and Running Unit Tests From the Command Line

```
$ javac -classpath `hadoop classpath`:\
/home/training/lib/mrunit-0.9.0-incubating-hadoop2.jar:. *.java

$ java -cp `hadoop classpath`:/home/training/lib/\
mrunit-0.9.0-incubating-hadoop2.jar:. \
org.junit.runner.JUnitCore TestWordCount

JUnit version 4.8.2
...
Time: 0.51

OK (3 tests)
```

Key Points

- **Unit testing is important**
- **MRUnit is a framework for MapReduce programs**
  - Built on JUnit
- **You can write tests for Mappers and Reducers individually, and for both together**
- **Run tests from the command line, Eclipse, or other IDE**
- **Best practice: always write unit tests!**

Bibliography

The following offer more information on topics discussed in this chapter

- **Junit reference: Reference**
  - http://pub.admc.com/howtos/junit4x/
- **MRUnit was developed by Cloudera and donated to the Apache project. You can find the MRUnit project at the following URL**
  - http://mrunit.apache.org