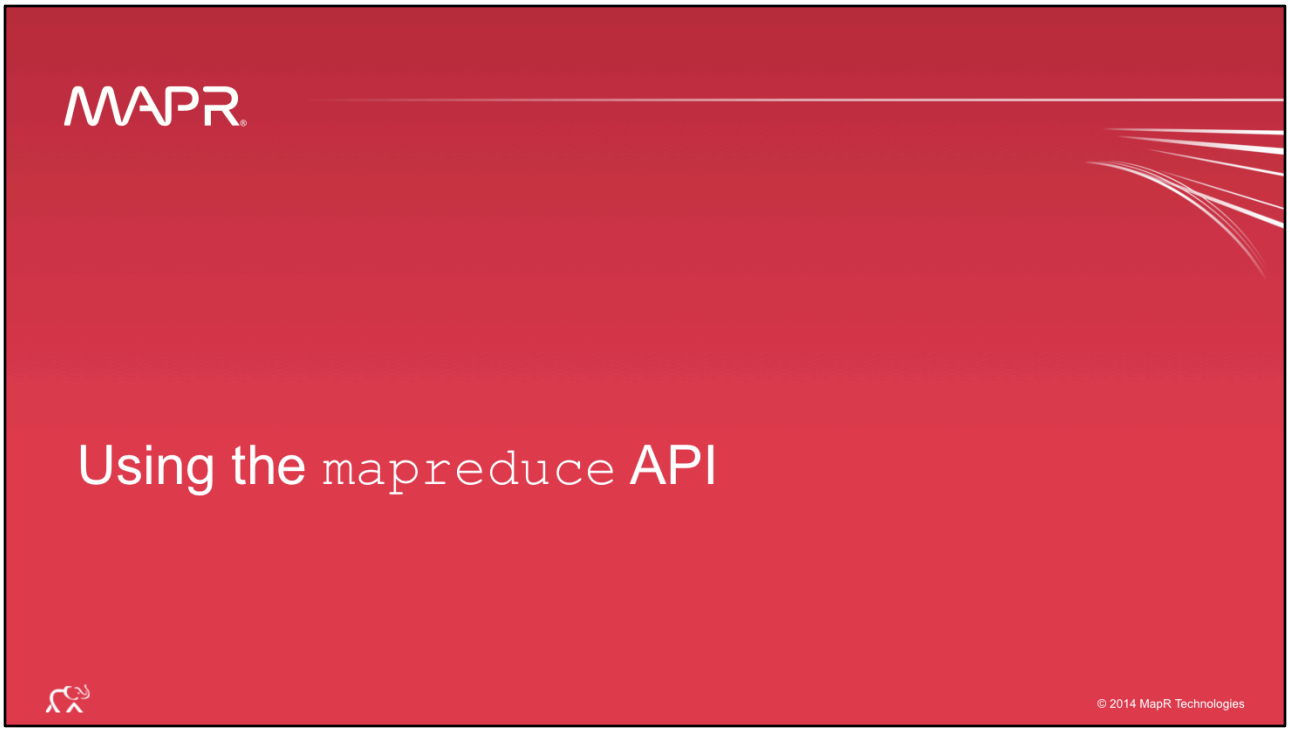

Using the mapreduce API
© 2014 MapR Technologies

A Few Quotes to Start Us Off

Le bon Dieu est dans le detail (more recently translated to "the devil is in the details").

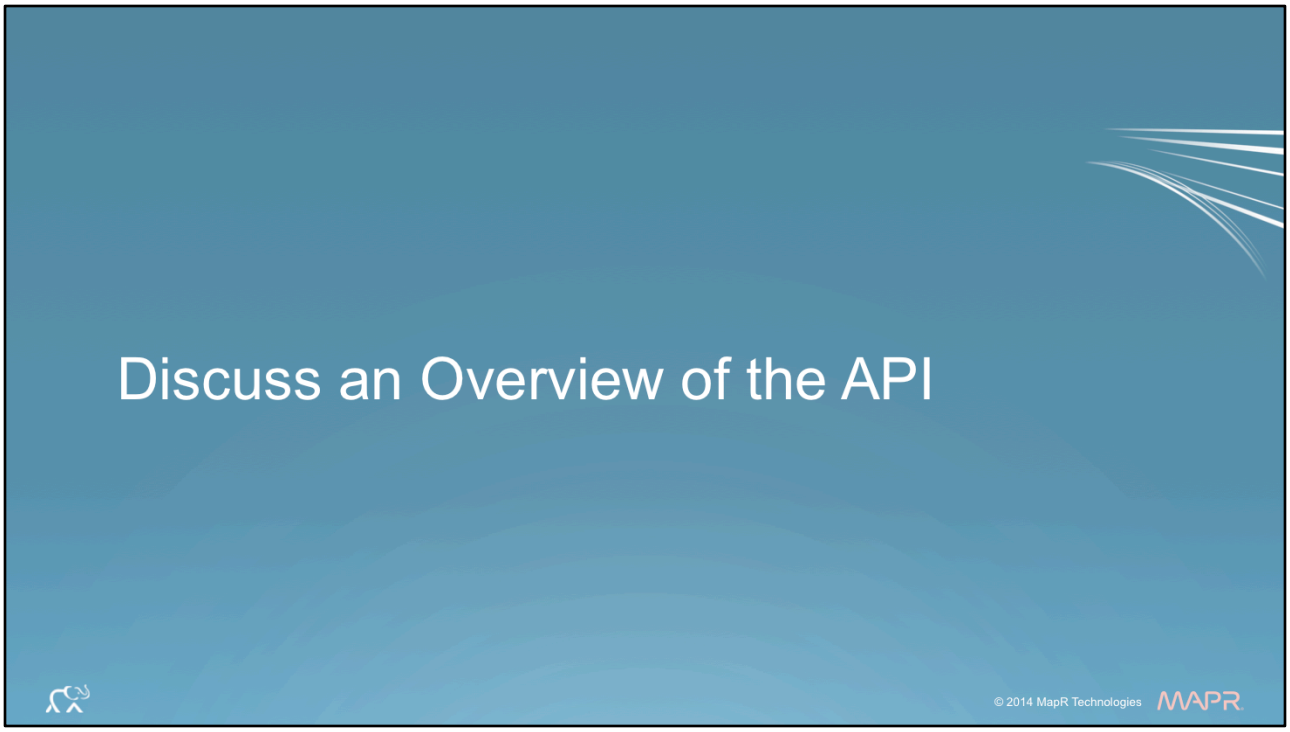
(Gustave Flaubert, 1821-1880)

The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.

(Donald Knuth, 1996)

Learning Objectives

- Discuss an overview of the API
- Describe how Mapper input is processed
- Describe how Reducer output is processed
- Describe the Mapper class
- Describe the Reducer class
- Describe the Job class



In this section, we discuss an overview of the MapReduce API.

Cite Hadoop Version Support on MapR

- MapR supports and ships with a specific version of Hadoop
- `HADOOP_HOME=/opt/mapr/hadoop/hadoop-0.20.2`
- Accessing 0.20.2 javadocs:
  - Download Hadoop-0.20.2.tar.gz
    http://archive.apache.org/dist/hadoop/core/hadoop-0.20.2
  - Extract the file
    gzip -dc hadoop-0.20.2.tar.gz | tar xvf –
  - Access the javadocs in your browser
    file://path-to-/hadoop-0.20.2/docs/api/index.html

Each MapR software release supports and ships with a specific version of Hadoop. For example, MapR 3.0.1 shipped with Hadoop 0.20.2.

The slide above contains some pointers on how to access the correct API for MapReduce. Note that Apache does not keep older versions of the API available online, so you'll have to download and access the documentation as described above.

Differentiate `mapred` and `mapreduce`

Aspect	mapred	mapreduce
Supported on MapR	yes	yes
Deprecated	no	no
YARN-compatible	yes	yes
Types	interfaces	abstract classes

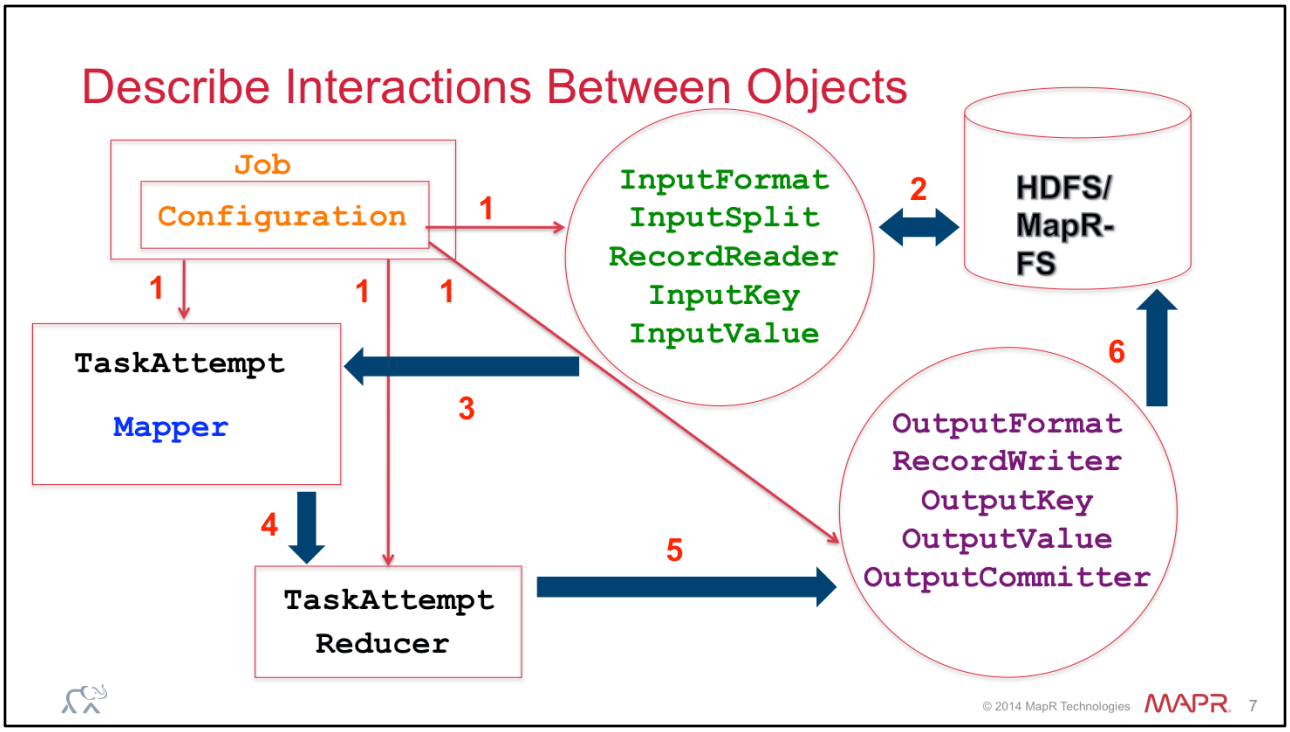
Objects	`OutputCollector`
	`Reporter`
	`JobConf`
		`Context`

Methods	`map() reduce()`	`map() reduce()`
		`cleanup() setup()`
		`run()`

Output files	`part-nnnnn`	`part-r-nnnnn` `part-m-nnnnn`
Reducer input values	`java.lang.Iterable`	`java.lang.iterator`

 MAPR 6

There are 2 API packages to choose when developing MapReduce applications: org.apache.hadoop.mapred and org.apache.hadoop.mapreduce. The mapred package was deprecated in the Apache Hadoop project when the mapreduce pacakge was introduced. Shortly thereafter, it was un-deprecated. Use the API which is most flexible for your use. Note that this course will use a mixture of each API.



The graphic above depicts the relationships between the objects in a running Hadoop job.

The driver class instantiates a Job object and it's configuration, and then uses set() methods to define each element of the Configuration object.

The Mapper class is the one that reads data from HDFS to do its processing.

The reducer class is the one that writes data to HDFS after it is done processing.

Define Writable Types

- All keys/values serialized as `Writables` before being written
- `Writable` is an interface
  - `void write(DataOutput out) throws IOException`
  - `void readFields(DataInput in) throws IOException`
- Each primitive type has a corresponding `Writable` :
  - `boolean` → `booleanWritable`

...

  - `double` → `doubleWritable`
- `Text` = `Writable` string encoded in UTF-8 format
- BytesWritable = Writable binary

Writable is an interface from Hadoop which is used to serialize keys and values before they are written to a stream (network or storage). Each java primitive has a corresponding Writable class (e.g. int → IntWritable). The integral java primitives (int and long) have a variable length Writable type that you can use to make your memory footprint more compact where possible.

The Text Writable type stores text using standard UTF8 encoding. It provides methods for reading and manipulating substrings in the Text object.

The bytesWritable Writable type is a byte sequence that is usable as a key or value. It is resizable and distinguishes between the size of the sequence and the current capacity.

Define WritableComparable Types

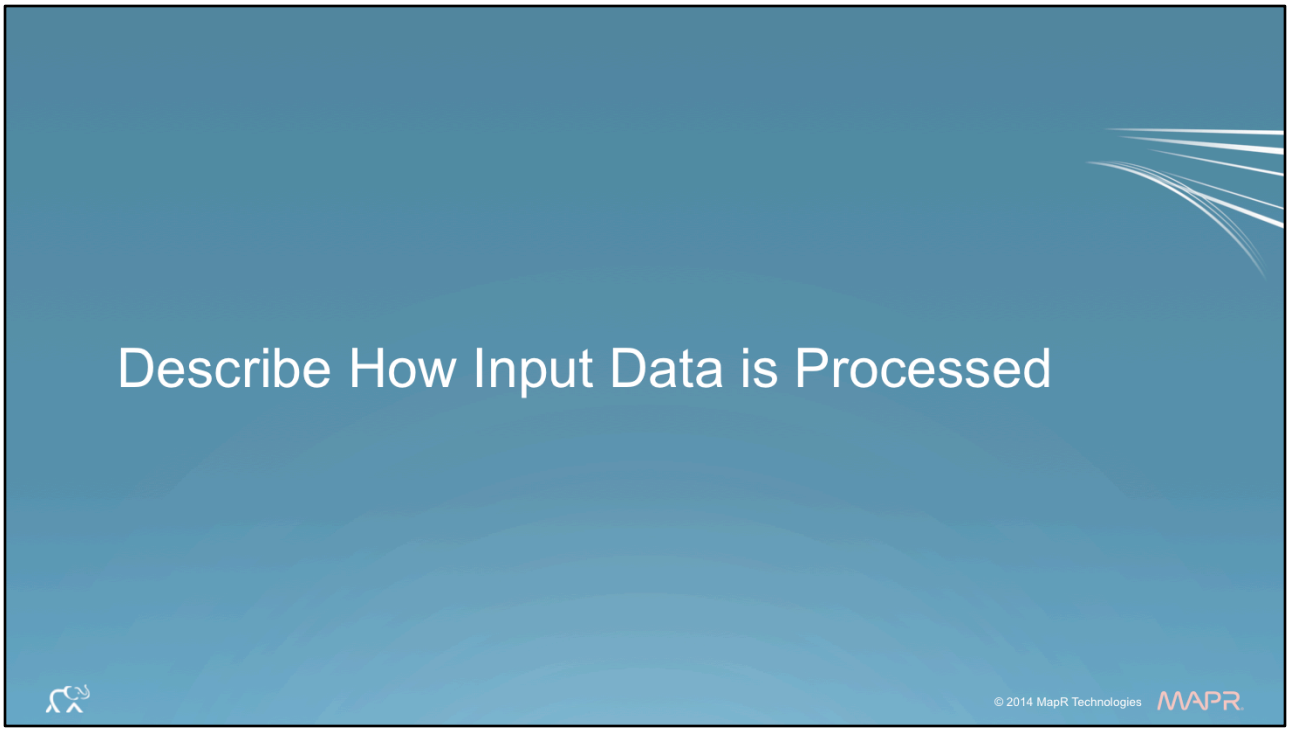
- **All keys must implement `WritableComparable` interface**

```
public interface WritableComparable extends Writable, Comparable
{
    void readFields(DataInput in);
    void write(DataOutput out);
    int compareTo(WritableComparable o)
}
```

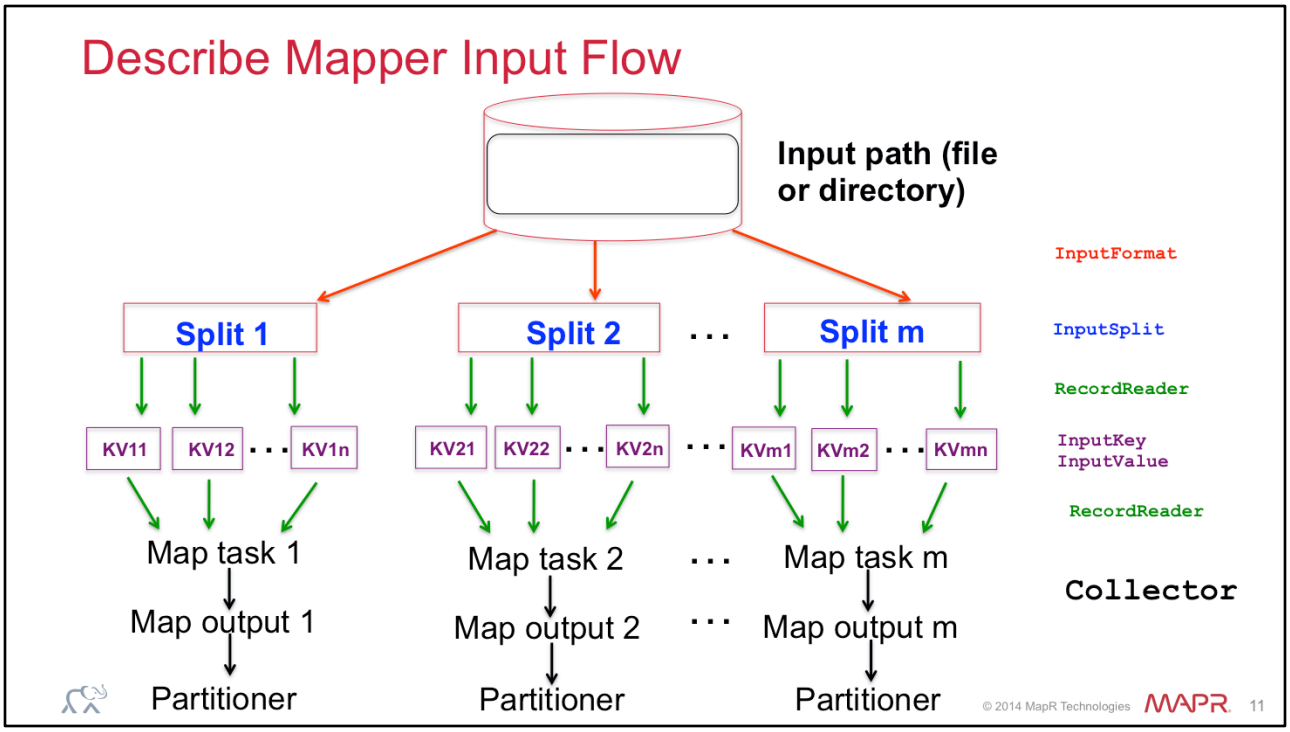
NOTE: the `compareTo()` method provides a total ordering of keys for the partitioner

All keys and values must implement the Writable interface, and all keys must further implement the Comparable interface. Since key types and value types are often interchanged, there is an interface called WritableComparable which extends both interfaces.

The compareTo() method returns -1, 0, or 1 depending on the lexicographic comparison between an object and the input to the method.



In this section, we describe how input data is processed in a MapReduce job.



Input file(s) are split using an implementation of the InputFormat class. Key-value pairs from the input splits are generated for each split using the RecordReader class. The InputKey and InputValue objects are subclasses of the Writable class which we discuss later in this lesson. All the key-value pairs from a given input split are sent to the same Mapper. The map() method is called once for each key-value pair, and all the results for each mapper are collected and sent to the partitioner which breaks out results based on hashed key values. These results are stored in the file system local to the Mapper task where they await pickup by the Hadoop framework running on the Reducer nodes to perform the shuffle and sort phase.

Define `InputFormat` Class

```
public abstract class InputFormat<K, V> {

 // returns array of InputSplits for job
 public abstract List<InputSplit> getSplits(JobContext) throws
IOException, InterruptedException;

 // create new record reader
 public abstract RecordReader<K, V> createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException, InterruptedException;

}
```

Implementations include:

- `TextFileInputFormat` (for single-line records in text files)
- `SequenceFileInputFormat` (for binary files)

The InputFormat class performs the following:

1. Validate the input file(s)/dir(s) exist for the job (throws IOException on invalid input).
2. Split the input file(s) into InputSplits
3. Instantiate the RecordReader to be used for parsing out records from each input split

File-based input formats split the input into logical splits based on the total size of the input files.

TextInputFormat is a subclass of FileInputFormat you can use for plain text files. TextInputFormat breaks files into lines (terminated by new-line character). The key is the offset into the file and the value is the line of text up to the new-line character. Note that the TextInputFormat class is not parameterized -- the key (LongWritable) and value (Text) are

Define `InputSplit` Class

```
public abstract class InputSplit {

   // returns number of bytes in the split
   public abstract long getLength() throws IOException;

   // returns array of nodes
   public abstract String{[] getLocations() throws IOException;

}
```

Implementations include:

- `FileSplit` (breaks each file into splits)

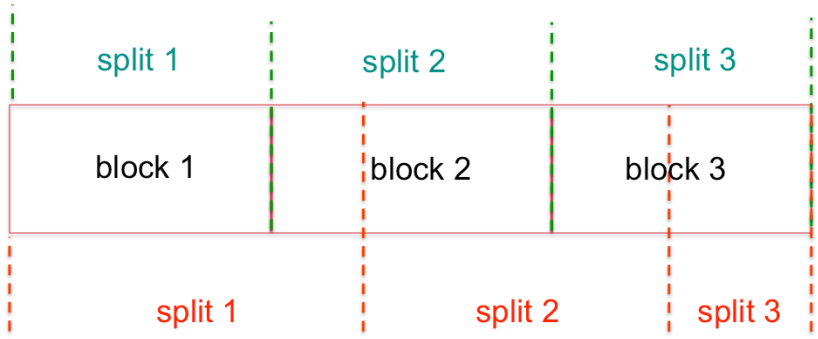
 13

InputSplit is a logical representation of a subset of the data to be processed by an individual Mapper.

The size of a split is given by the following formula:
Max(minimumSize, min(maximumSize, blockSize))
So for example,
Minimum split size = 10mb
Maximum split size = 32mb
Block size = 64mb
Split size = max (10, min(32,64) = max(10, 32) = 32mb

Note that the maximum split size is configurable (mapred.max.split.size) up to a hard bound of the block size. The minimum split size is also configurable (via mapred.min.split.size). Note that InputSplit is a logical representation of a part of a file, and that split may not (most

Calculating Input Splits



|input split| = max(minimumSize, min(maximumSize, blockSize))

The graphic above depicts a common situation where the input splits may not line up exactly with block sizes. The actual size of an input split is given in the formula above. It is a function of the minimum and maximum sizes of an input split which is configurable by the end user. Note that an input split may be smaller, larger, or the same size as the block size, depending on the configuration the user provides to the mapreduce job.

Define `RecordReader` Interface

```
public interface RecordReader<K, V> {
   boolean next(K key, V value) throws IOException
   K createKey();
   V createValue();
   long getPos() throws IOException;
   public void close() throws IOException;
   float getProgress() throws IOException;
}
```

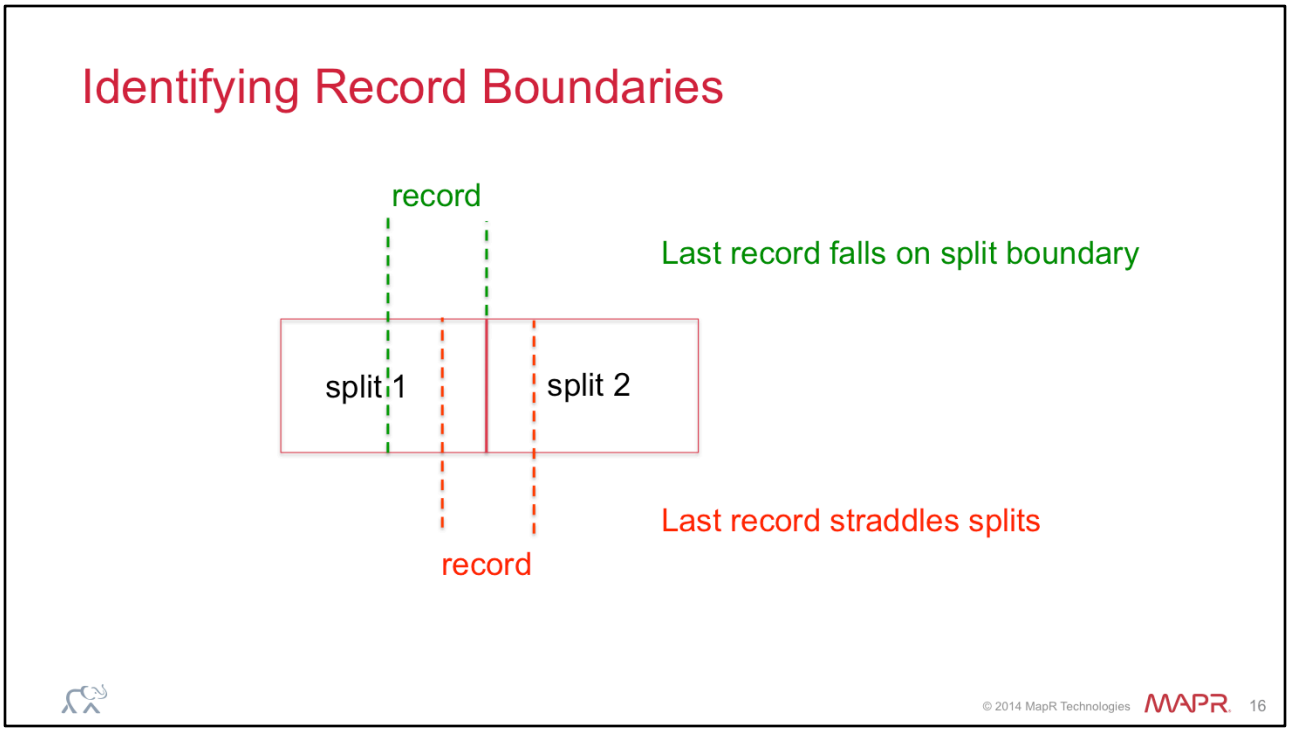
Implementations include:

- `LineRecordReader` (key=offset, value=line)
- `SequenceFileRecordReader` (for binary input files)

The record reader breaks up the data in an input split into key-value pairs and presents them to the Mapper assigned to that split.

Note that records won't necessarily start and end on whole split boundaries (or block boundaries for that matter). It is the job of the RecordReader to ensure that it discards incomplete records at the start of a split and "reads ahead" into the next split to get an entire record that is broken across split boundaries. Here is the code snippet from the LineRecordReader class that deals with partial records:

```
if (codec != null) {
  in = new LineReader(codec.createInputStream(fileIn), job);
  end = Long.MAX_VALUE;
} else {
```



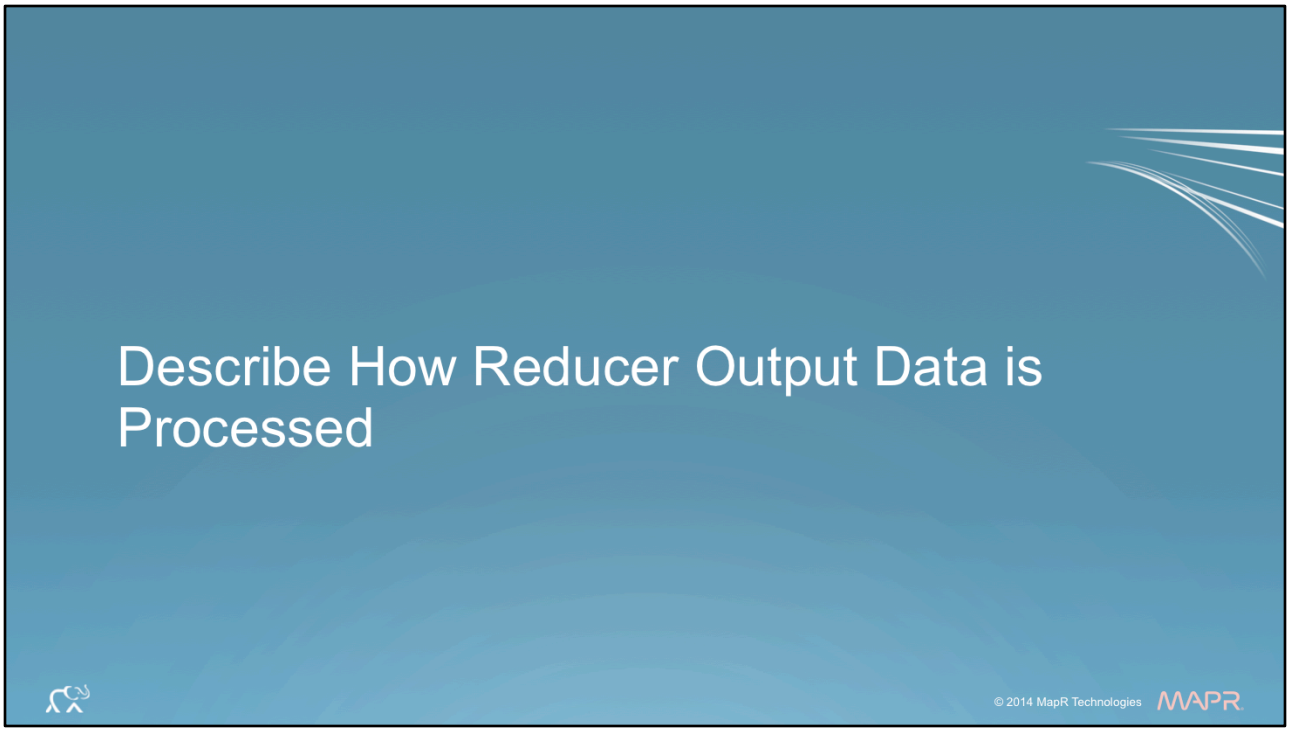
It is the most common case that a record does not end on a split boundary. Usually, a record starts in one split and is terminated in the next split. The record reader logic accounts for this, as described in the next slide. Users do not need to configure anything with default record readers, but custom record readers that a programmer implements must comprehend this phenomenon.

Examining `RecordReader` Code

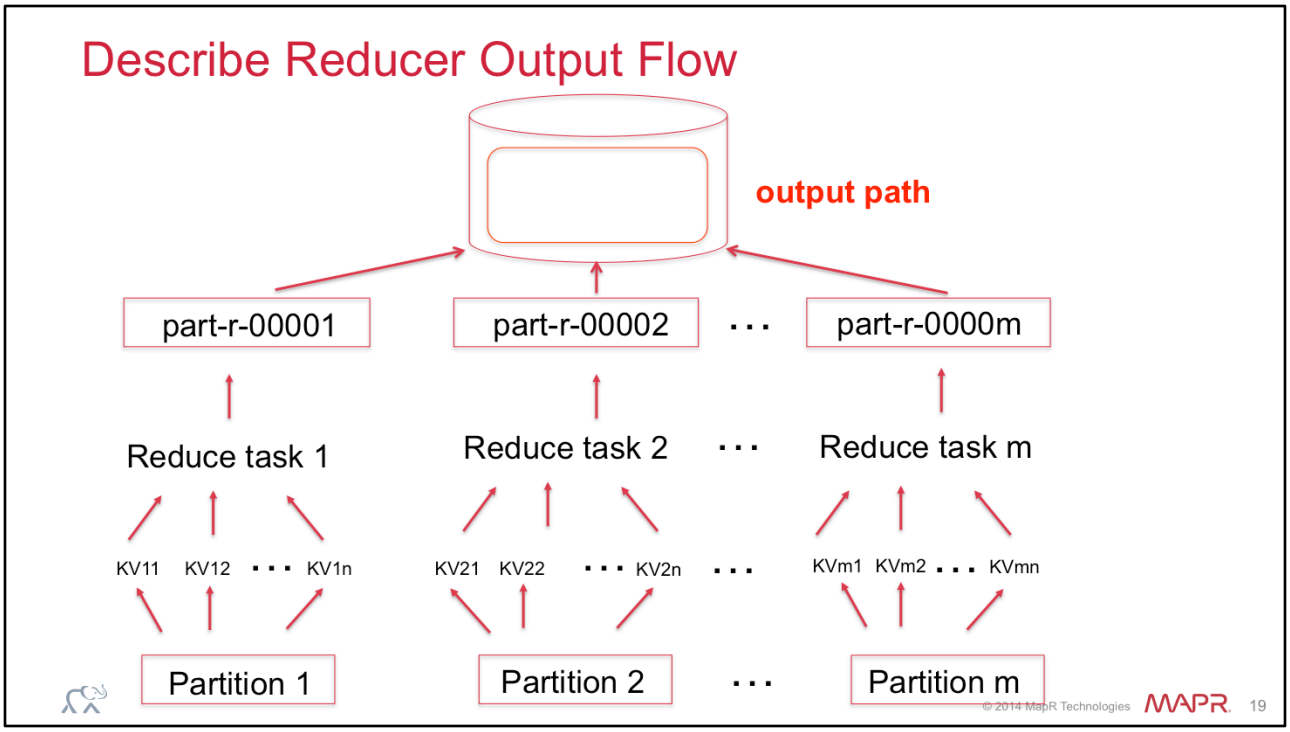
```
if (codec != null) {
   in = new LineReader(codec.createInputStream(fileIn), job);
   end = Long.MAX_VALUE;
} else {
   if (start != 0) {
     skipFirstLine = true;
     --start;
     fileIn.seek(start);
   }
   in = new LineReader(fileIn, job);
}
if (skipFirstLine) {  // skip first line and re-establish "start".
  start += in.readLine(new Text(), 0, (int)Math.min((long)Integer.MAX_VALUE, end - start));
}
this.pos = start;
```

Since the splits are calculated in the client, the mappers don't need to run in sequence. Every mapper already knows if it needs to discard the first line or not.

Assume you have 2 lines of each 100Mb in the same file and the split size is 64Mb. Then when the input splits are calculated, we will have the following scenario:
Split 1 containing the path and the hosts to this block. Initialized at start 200-200=0Mb, length 64Mb.
Split 2 initialized at start 200-200+64=64Mb, length 64Mb.
Split 3 initialized at start 200-200+128=128Mb, length 64Mb.
Split 4 initialized at start 200-200+192=192Mb, length 8Mb.
Mapper A will process split 1, start is 0 so don't skip first line, and read a full line which goes beyond the 64Mb limit so needs remote read.
Mapper B will process split 2, start is != 0 so skip the first line



In this section, we describe how reducer output data is processed in a MapReduce job.



Note the data flows from the bottom of the graphic to the top.

The Hadoop framework is responsible for taking the partitions from each of the Mappers, shuffles and sorts the results, and presenting a sorted set of key-value pairs as single partition to each Reducer. The key (or a subset of the key) is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job.

Each reducer takes its partition as input, processes one iterable list of key-value pairs at a time, and produces an output file called 'part-r-0000x' (where x denotes the number of the reducer). The output directory for this file is specified in the Job configuration and must reside on the HDFS/Mapr-FS file system. The directory cannot both exist and already contain reduce task output files, or the OutputFormat class will generate an

Define `OutputFormat` Class

```
public abstract class OutputFormat<K, V> {
   public abstract RecordWriter<K, V> getRecordWriter(TaskAttemptContext
context) throws IOException, InterruptedException;
   public abstract void checkOutputSpecs(JobContext context) throws
IOException, InterruptedException;
   public abstract OutputCommitter getOutputCommitter(TaskAttemptContext
context) throws IOException, InterruptedException;
}
```

Implementations include:

- `FileOutputFormat` (file-based wrapper for `OutputFormat`)
- `TextOutputFormat` (file-based format for plain-text files)
- `NullOutputFormat` (sends all output to /dev/null)
- `SequenceFileOutputFormat` (file-based output format for binary files)

The OutputFormat class is responsible for

1. Validating the output specification of the job using checkOutputSpecs().
2. Provide the RecordWriter (as defined in the OutputCommitter) to write output files.

FileOutputFormat is an abstract class base for file-based output. FileOutputFormat provides a contract for setting and getting the output path, setting and getting the compression type, and defining the OutputCommitter.

TextOutputFormat extends FileOutputFormat and provides a RecordWriter for text-based files.

SequenceFileOutputFormat extends FileOutputFormat and

Define `RecordWriter` Class

```
public abstract class RecordWriter<K, V> {

   public abstract void write(K key, V value) throws IOException,
InterruptedException;

   public abstract void close(TaskAttemptContext context) throws
IOException, InterruptedException;

}
```

Implementations include:

- `TextOutputFormat.LineRecordWriter` (writes key-value pairs to plain-text files)

 21

RecordWriter writes the key-value pairs to the output files.

The TextOutputFormat.LineRecordWriter implementation irequires a java.io.DataOutputStream object to write the key-value pairs to the HDFS/MapR-FS file system. If the output is configured to be compressed to storage, the the LineRecordWriter will invoke the configured codec to encode the data prior to writing to the stream. The separator between the key and value is configurable – the default is the tab character.

Define `OutputCommitter` Class

```
public abstract class OutputCommitter {
   public abstract void setupJob(JobContext jobContext) throws IOException;
   public void commitJob(JobContext jobContext) throws IOException;
   public void abortJob(JobContext jobContext, JobStatus.Stats stats) throws
IOException;
   public void cleanupJob(JobContext jobContext) throws IOException;
   public abstract void setupTask(TaskAttemptContext taskContext) throws
IOException;
   public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
throws IOException;
   public abstract void commitTask(TaskAttemptContext taskContext) throws
IOException;
   public abstract void abortTask(TaskAttemptContext taskContext) throws
IOException;
}
```

Implementations include:

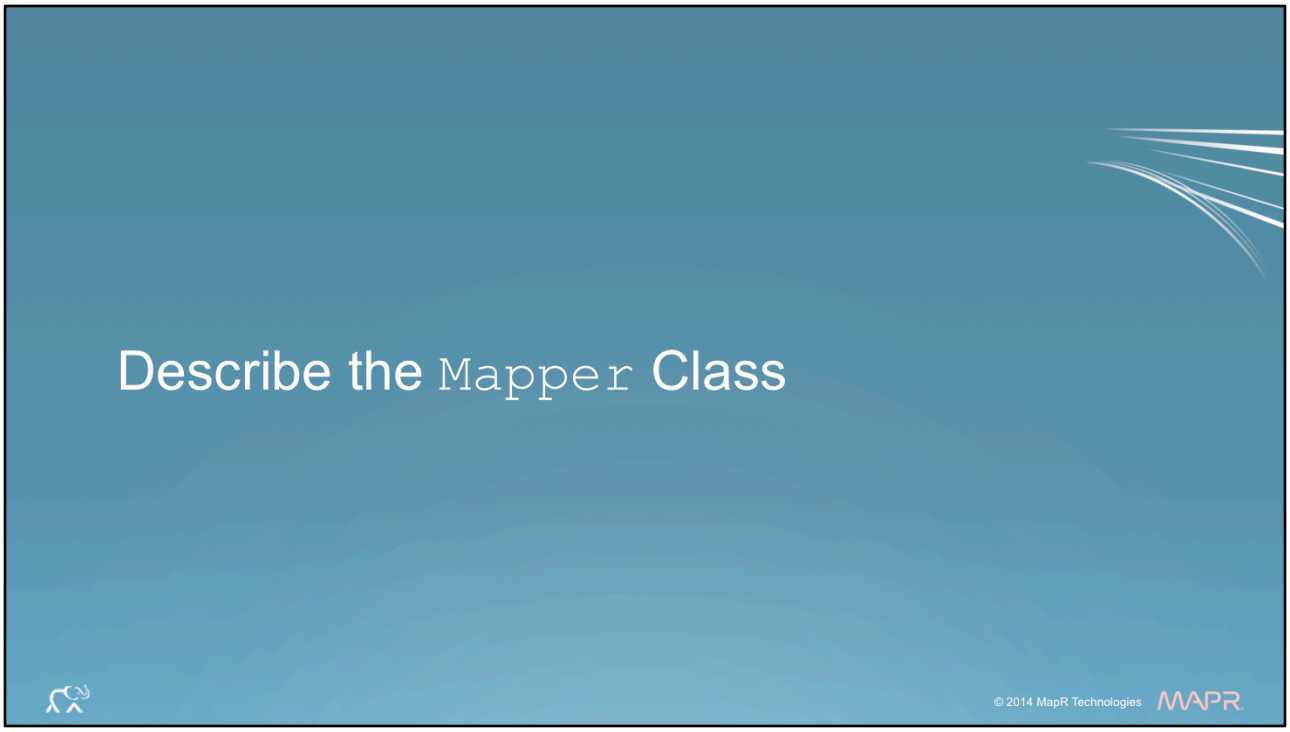
- `FileOutputCommitter` (commits files to job output directory)

 22

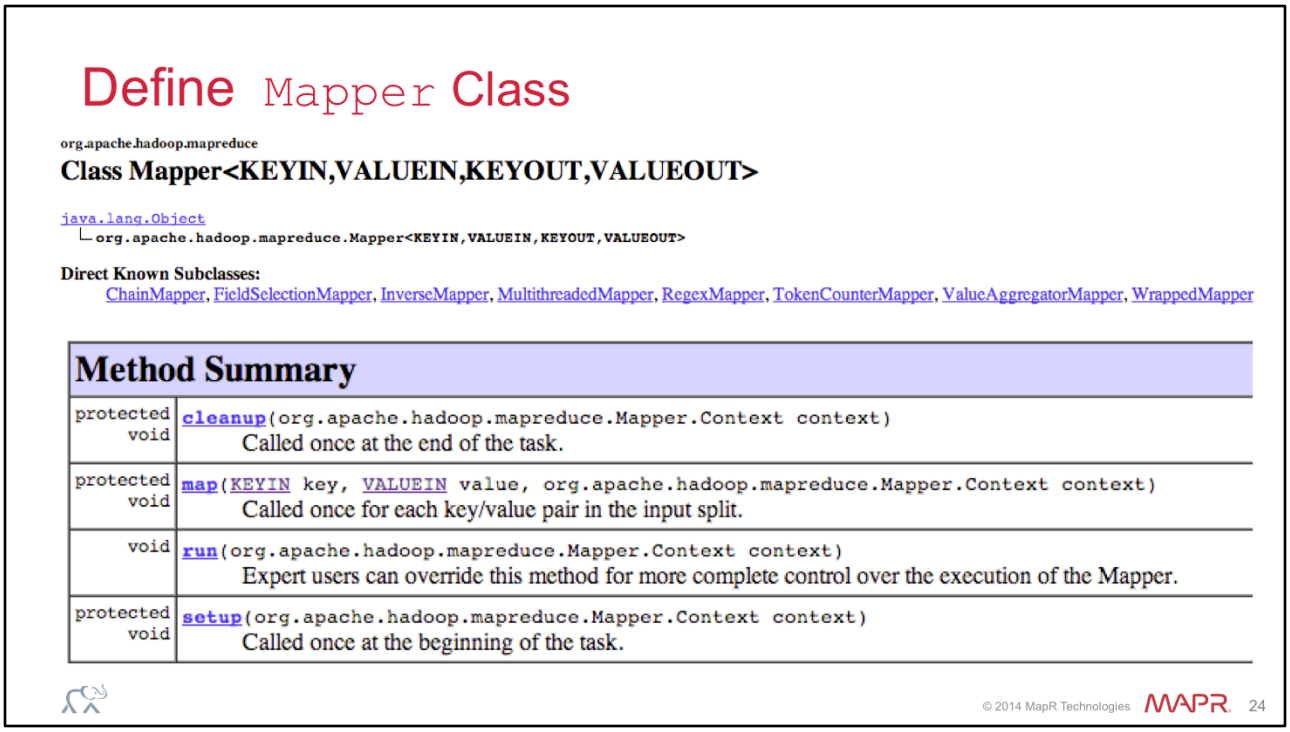
The OutputCommitter is responsible for:

1. Initializing the job at job start (e.g. create temp directories) in setupJob().
2. Cleaning up the job on job completion (e.g. remove temp directories) in cleanupJob().
3. Setup the task temporary output (in setupTask()).
4. Check whether a task needs a commit (in needsTaskCommit()).
5. Commit of the task output (in commitTask()).
6. Discard the task commit (abortTask()).

All methods throw IOException on error interacting with DataOutputStream.



In this section, we describe the Mapper class.



The slide above shows the class and method summaries for the Mapper class. The Mapper class is parameterized with the key and value types for input and output. Here are a few rules regarding input and output keys and values for the Mapper class:

1. The input key class and input value class in the Mapper class must match those defined in the job configuration
2. The input key class and input value class in the map() method must match those defined in the Mapper class
3. The output key class and output value class in the Mapper must match the input key class and input value class defined in the Reducer class

The primary method you implement in the Mapper class is the map() method. The other methods (cleanup(), run(), and setup()) are described below.

Discuss Mapper Example

```
public  class MyWordcountMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
   private Text word = new Text();
   private final static IntWritable one = new IntWritable(1);
   public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer iterator = new StringTokenizer(line);
      while (iterator.hasMoreTokens()) {
         word.set(iterator.nextToken());
         context.write(word, one);
      }
   }
}
```

The example Mapper above iterates over a line (list (k1, v1)) and emits each word (as a key) and a "1" (as a value) as part of a word count application (k2, v2).

The LongWritable input key class in this case is provided as the byte offset into the input file to the beginning of the record (line). The input value and output key classes are both defined as Text, and the output type is defined as IntWritable. In this word count case, the output of the mapper will be the key (word) and value (cardinality of word in record).

The Mapper.Context object passed to the map() method contains, among other things, a reference to the job and its configuration. It also contains information specific to this (map) task.

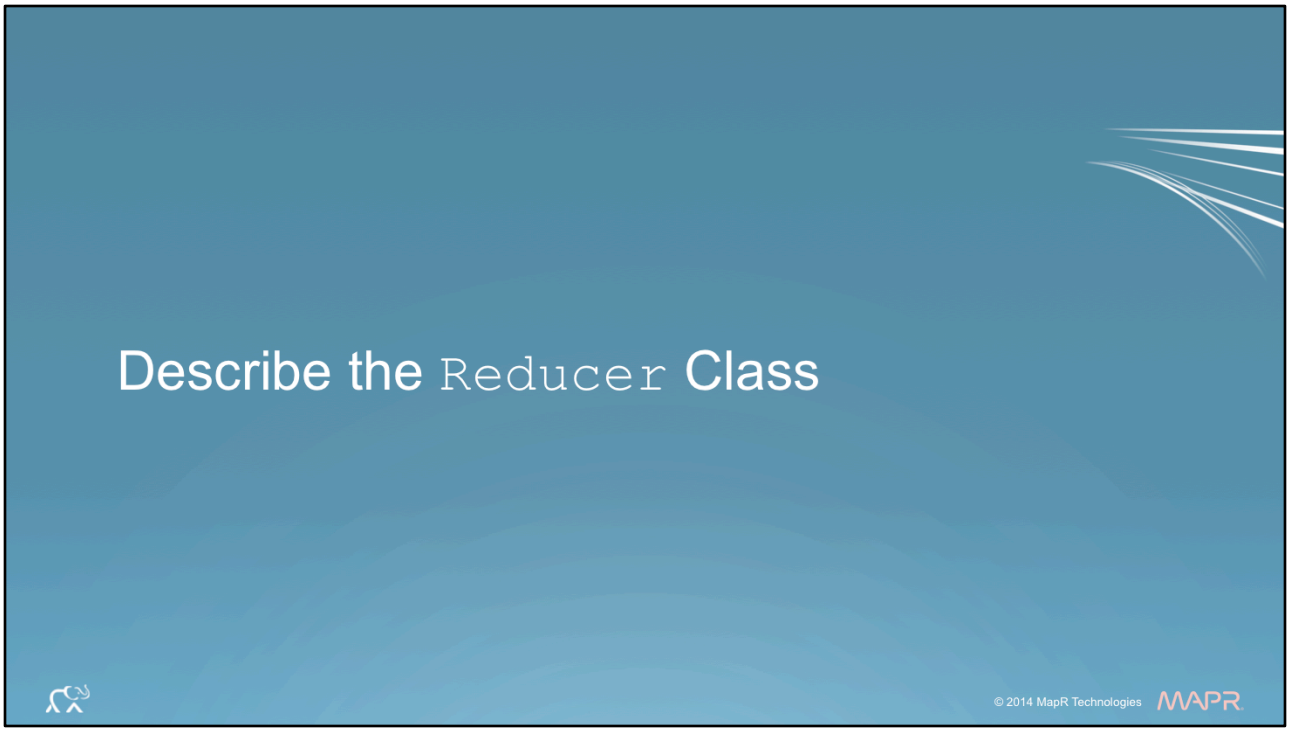
Discuss `setup()` and `cleanup()` Methods

```
public void run(Context context) throws IOException, InterruptedException {
   setup(context);
   try {
      while (context.nextKey()) {
         map(context.getCurrentKey(), context.getValues(), context);
      }
   }
   finally {
      cleanup(context);
   }
}
```

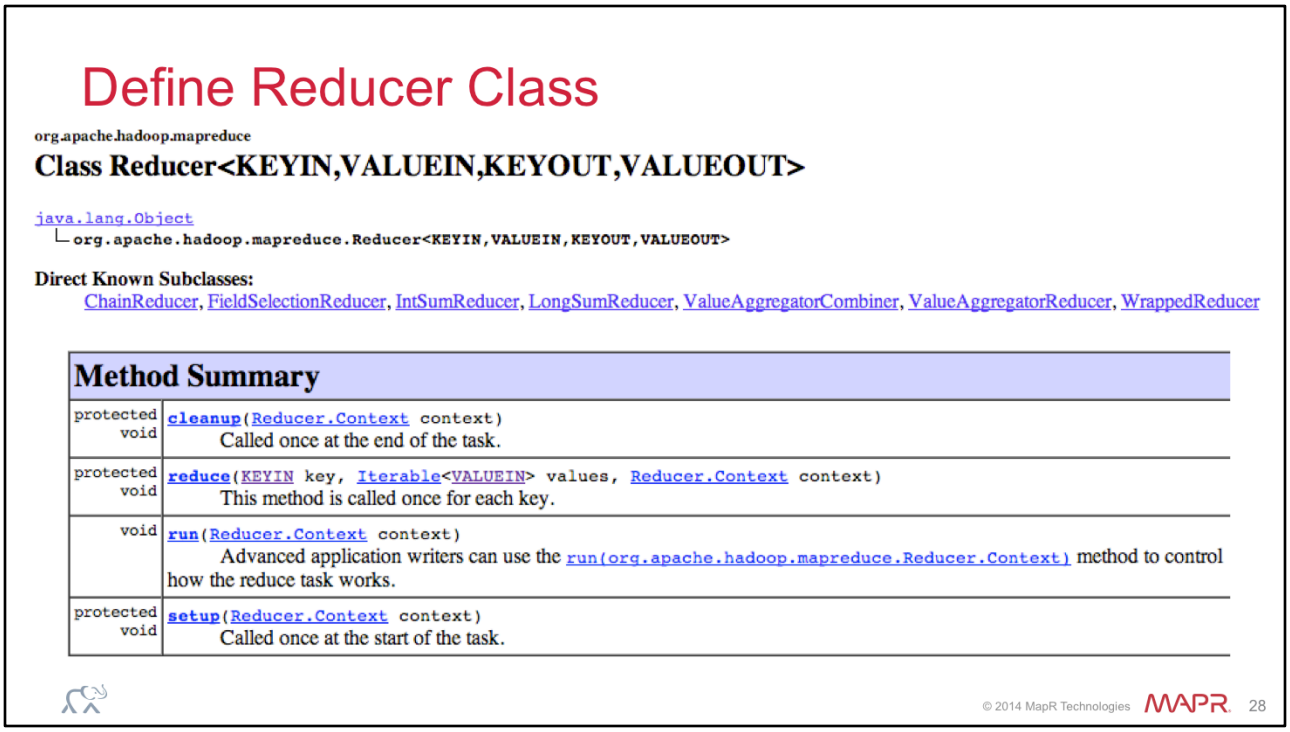
NOTE: the Reducer is called in the same fashion as the Mapper

As depicted in the Hadoop code snippet above for run(), the setup() and cleanup() methods are each called once for the task attempt. The context object is passed to each method so information may be passed to and from them.

One use case for using the setup() and cleanup() method is to open and close a file, HBase, or JDBC connection. This is a much more streamlined approach if I/O is performed during the task than to open and close a stream within the map() or reduce() method. Note however that the default run() method does not catch exceptions thrown in the map or reduce code.



In this section, we discuss the Reducer class.



The slide above shows the class and method summaries for the Reducer class. The Reducer class is parameterized with the key and value types for input and output. Here are a few rules regarding input and output keys and values for the Reducer class:

1. The input key class and input value class in the Reducer must match the output key class and output value class defined in the Mapper class
2. The output key class and output value class in the Reducer must match those defined in the job configuration

The behavior of the cleanup(), run(), and setup() methods are identical as those described for the Mapper class.

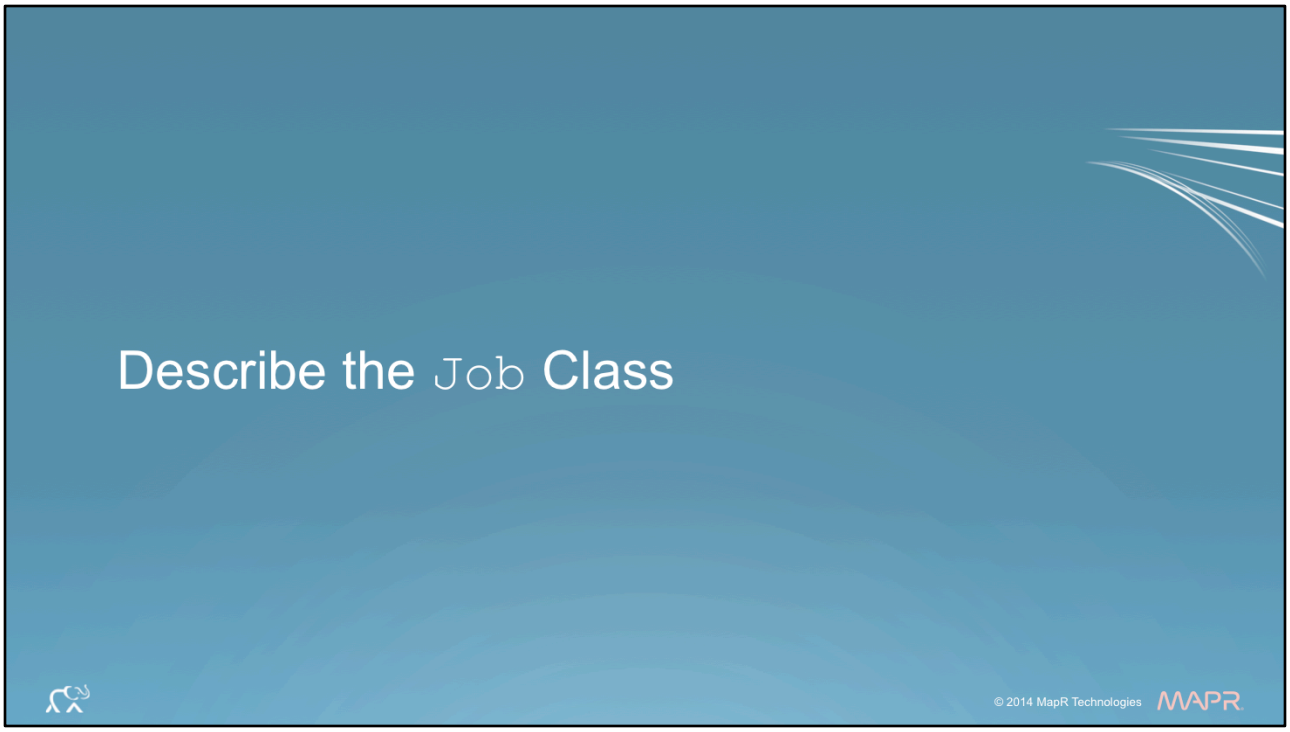
Discuss Reducer Example

```
public class MyWordcountReducer extends
Reducer<Text,IntWritable,Text,IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable value : values) {
         sum += value.get();
      }
      context.write(key, new IntWritable(sum));
   }
}
```

The example Reducer above iterates over a list of values for a given key (list(k2, v2)) and emits each word (as a key) and a sum (as a value) as part of a word count application (k3, v3).

The Text input (key) class and Iterable<IntWritable> input value class match the output key class and output value class, respectively, from the Mapper.

The Reducer.Context object passed to the reduce() method contains, among other things, a reference to the job and its configuration. It also contains information specific to this (reduce) task.



In this section, we discuss the Job class.

Define Job Class

org.apache.hadoop.mapreduce

Class Job

java.lang.Object
  └ org.apache.hadoop.mapreduce.JobContext
    └ **org.apache.hadoop.mapreduce.Job**

Constructor Summary

Job()
Job(Configuration conf)
Job(Configuration conf, String jobName)

```
Configuration conf = new Configuration();
Job job = new Job(conf, "mywordcount");
```

 MAPR 31

The job submitter's view of the Job. It allows the user to configure the job, submit it, control its execution, and query the state. The set methods only work until the job is submitted, afterwards they will throw an IllegalStateException.

The Job class and constructor summary are provided in the slide above. The example at the bottom of the slide constructs a new Job object using the string name “mywordcount”. If you construct and job without specifying the name, a default name will be given to your job (which is the name of the driver class which instantiates the Job object).

Define Job Methods (1 of 3)

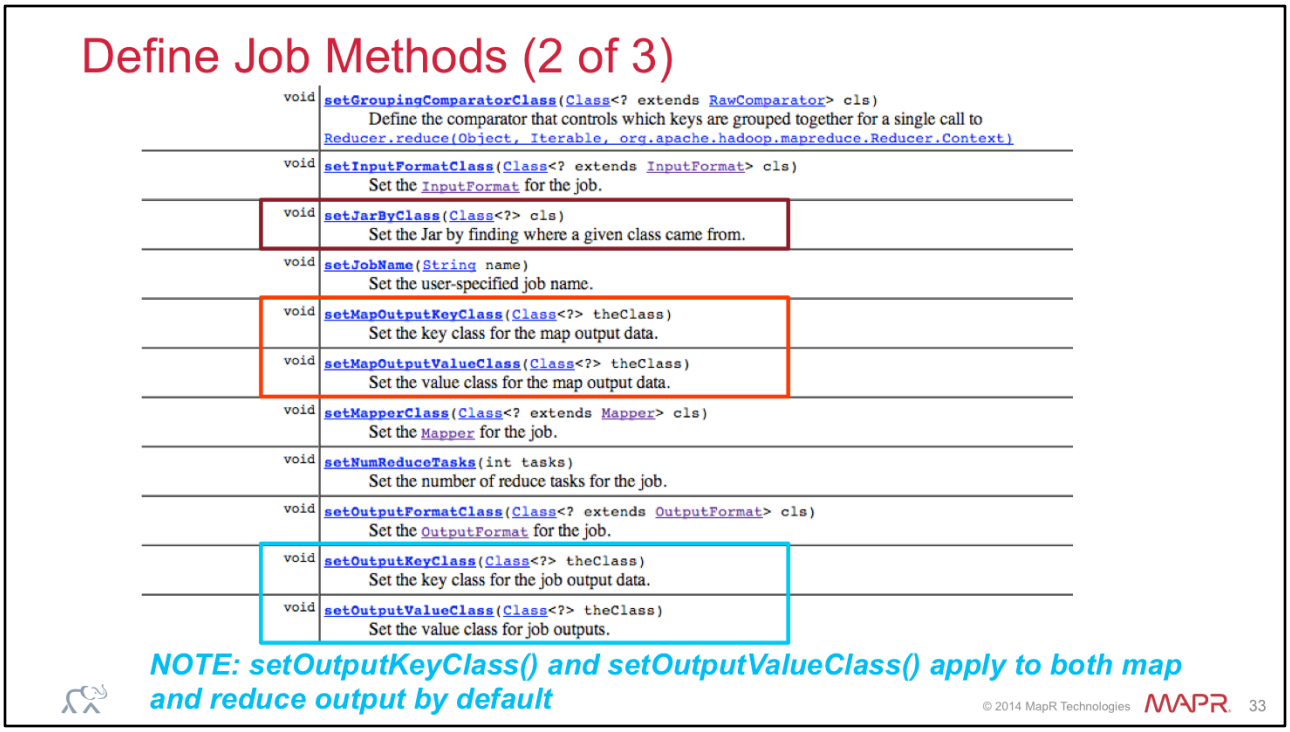
Method Summary	
void	**failTask**(TaskAttemptID taskId) Fail indicated task attempt.
Counters	**getCounters**() Gets the counters for this job.
String	**getJar**() Get the pathname of the job's jar.
TaskCompletionEvent[]	**getTaskCompletionEvents**(int startFrom) Get events indicating completion (success/failure) of component tasks.
String	**getTrackingURL**() Get the URL where some job progress information will be displayed.
boolean	**isComplete**() Check if the job is finished or not.
boolean	**isSuccessful**() Check if the job completed successfully.
void	**killJob**() Kill the running job.
void	**killTask**(TaskAttemptID taskId) Kill indicated task attempt.
float	**mapProgress**() Get the *progress* of the job's map-tasks, as a float between 0.0 and 1.0.
float	**reduceProgress**() Get the *progress* of the job's reduce-tasks, as a float between 0.0 and 1.0.
void	**setCombinerClass**(Class<? extends Reducer> cls) Set the combiner class for the job.

NOTE: none of these methods are required to use by default

The first set of methods in the Job class are described above.

The failTask() method

The setCombinerClass() is optionally called. Typically, if you use a combiner at all, you'll use the reduce() method as the combiner. There are certain rules regarding associativity and commutativity that must be obeyed in order to use a combiner. We'll discuss this later in the course.



The second set of methods in the Job class are defined above.

There are several setters defined here which allow the user to define the job configuration. A few of these configurables have default values if you don't explicitly provide one (e.g. the default job name is the name of the driver class, the default number of reduce tasks is equal to the number of partitions, …).

Define Job Methods (3 of 3)

void	**setPartitionerClass**(Class<? extends Partitioner> cls) Set the Partitioner for the job.
void	**setReducerClass**(Class<? extends Reducer> cls) Set the Reducer for the job.
void	**setSortComparatorClass**(Class<? extends RawComparator> cls) Define the comparator that controls how the keys are sorted before they are passed to the Reducer.
void	**setWorkingDirectory**(Path dir) Set the current working directory for the default file system.
void	**submit**() Submit the job to the cluster and return immediately.
boolean	**waitForCompletion**(boolean verbose) Submit the job to the cluster and wait for it to finish.

NOTE: submit() is asynchronous and waitForCompletion() is synchronous

The last set of methods in the Job class are defined above.

Besides the remaining setters, the submit() and waitForCompletion() methods are how the user launches the job from the driver. The difference is that submit() is non-blocking (background) while waitForCompletion() is blocking (foreground).

Discuss a Simplistic Driver Implementation

```
public class MyWordcountDriver {
   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "mywordcount");
      job.setJarByClass(MyWordcountDriver.class);
      job.setMapperClass(MyWordcountMapper.class);
      job.setReducerClass(MyWordcountReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TextOutputFormat.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

DRAWBACK: No way to dynamically pass configuration to job

The driver class contains a main method which instantiates a new job and job configuration. We set various components of the job configuration and then call waitForCompletion() to launch the job and block until it returns. Later in this lesson we will discuss a slight modification to this driver by using ToolRunner.

If you don't provide a name to the job, the name of the driver class (in this case, MyWordcountDriver) would be used as the name.

You can use the getClass() method when calling setJarByClass() rather than specifying the name of the class.

Input and output classes must be a subclass of Writable.

Discuss the Best Way to Implement the Driver

```
public class MyWordcountDriver extends Configured implements Tool {
   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      System.exit(ToolRunner.run(conf, new MyWordcountDriver (), args));
   }

   public int run(String[] args) throws Exception {
      Job job = new Job(conf, "mywordcount");
      . . .
      job.waitForCompletion(true) ? 0 : 1;
}
```

BENEFIT: `ToolRunner` implicitly calls `GenericOptionsParser` which permits dynamic configuration for jobs

 MAPR 36

Using ToolRunner allows you to make use of the GenericOptionsParser to pass Hadoop options as well as command-line arguments to your driver.

The generic Hadoop command-line options are:
-conf <configuration file> (config file must reside on every task tracker or in MapR-FS as XML configuration)
-D <property=value> (note this requires a space in between –D and the property name)
-fs <local|namenode:port>
-jt <local|jobtracker:port>

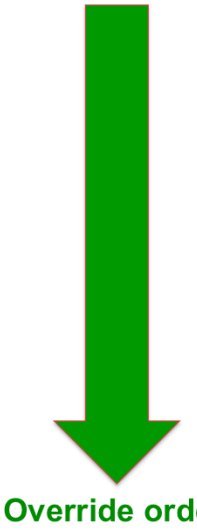
Summarize the `Configuration` Class

- Some are related to the file system (HDFS/MapR-FS)
  - `io.file.buffer.size=8192`
- Some are related to MapReduce
  - `mapred.map.child.log.level=DEBUG`
- Some may be custom to your application
  - `myfavoritecolor=green`
- Some are MapR-specific:
  - `mapr.centrallog.dir=logs`
- Some are meaningless
  - E.g. `hadoop.mapreduce.map.tasks=5`

Here's a complete list of the configuration parameters used for a Hadoop 0.20-2 job:

config files= Configuration: core-default.xml, core-site.xml, mapred-default.xml, mapred-site.xml
properties=
{"properties":
[{"key":"mapreduce.tasktracker.group","value":"mapr","isFinal":false,"resource":"mapred-site.xml"},
{"key":"textinputformat.record.delimiter","value":"))","isFinal":false,"resource":"Unknown"},
{"key":"mapred.local.dir","value":"/tmp/mapr-hadoop/mapred/local","isFinal":false,"resource":"mapred-site.xml"}]}
io.seqfile.compress.blocksize=1000000
keep.failed.task.files=false
mapreduce.reduce.input.limit=-1

How to Set Configuration Parameters



- Hadoop framework (jar files)
- Global XML files (`/opt/mapr/hadoop/hadoop-0.20.2/conf`)
  - `mapred-default.xml, mapred-site.xml`
  - `core-default.xml, core-site.xml`
- Local XML files (arbitrary app-specific user-defined)
  - `-conf mymapred.xml`
- Command-line arguments (job + JVM properties)
  - `-D param=value`
- Within the driver (last chance to override)
  - `conf.set("param", "value")`

 MAPR 38

See http://doc.mapr.com/display/MapR/MapR+Parameters for more information on user-configurable parameters.

Global XML files (in the order they are read)

1. mapred-default.xml
2. mapred-site.xml
3. core-default.xml
4. core-site.xml

Local XML files:

hadoop jar jarfile –conf /path/to/conf.xml …

```
<configuration>
<property>
   <name>mapred.map.child.log.level</name>
```

Lesson Summary

- Identify the correct MapReduce API for your environment
- Writable types require conversion in order to manipulate data
- Determine the proper scope and lifetime of variables you use
- Each mapper is assigned a single complete input split, and each map method processes a single record
- Each reducer is assigned one or more complete partitions, and each reducer method processes all records from those partitions
- Use the ToolRunner interface for your driver code