Chapter 7

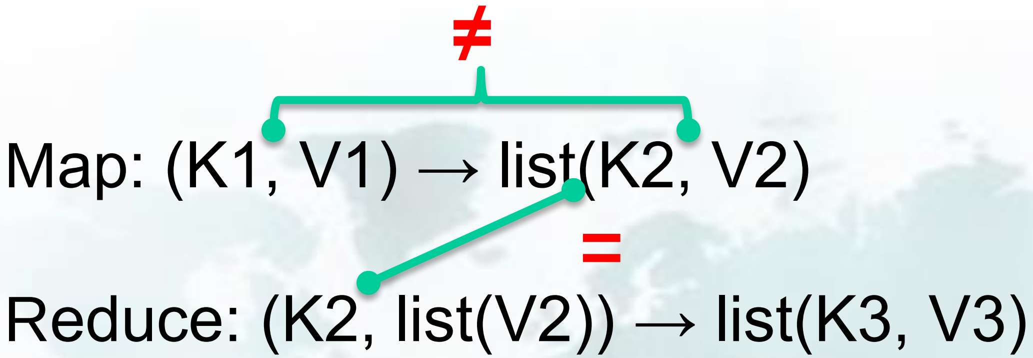
MapReduce
Types and Formats

Learning Objectives

- MapReduce Types
- Input Formats
- Output Formats

General Form of MapReduce

- General form:



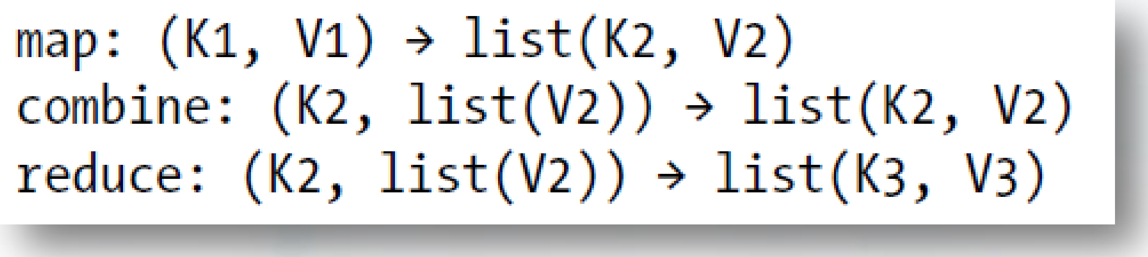
≠

Map: (K1, V1) → list(K2, V2)

=

Reduce: (K2, list(V2)) → list(K3, V3)

- Combine function:



```
map: (K1, V1) → list(K2, V2)
combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

Configuration of MapReduce Types – 1

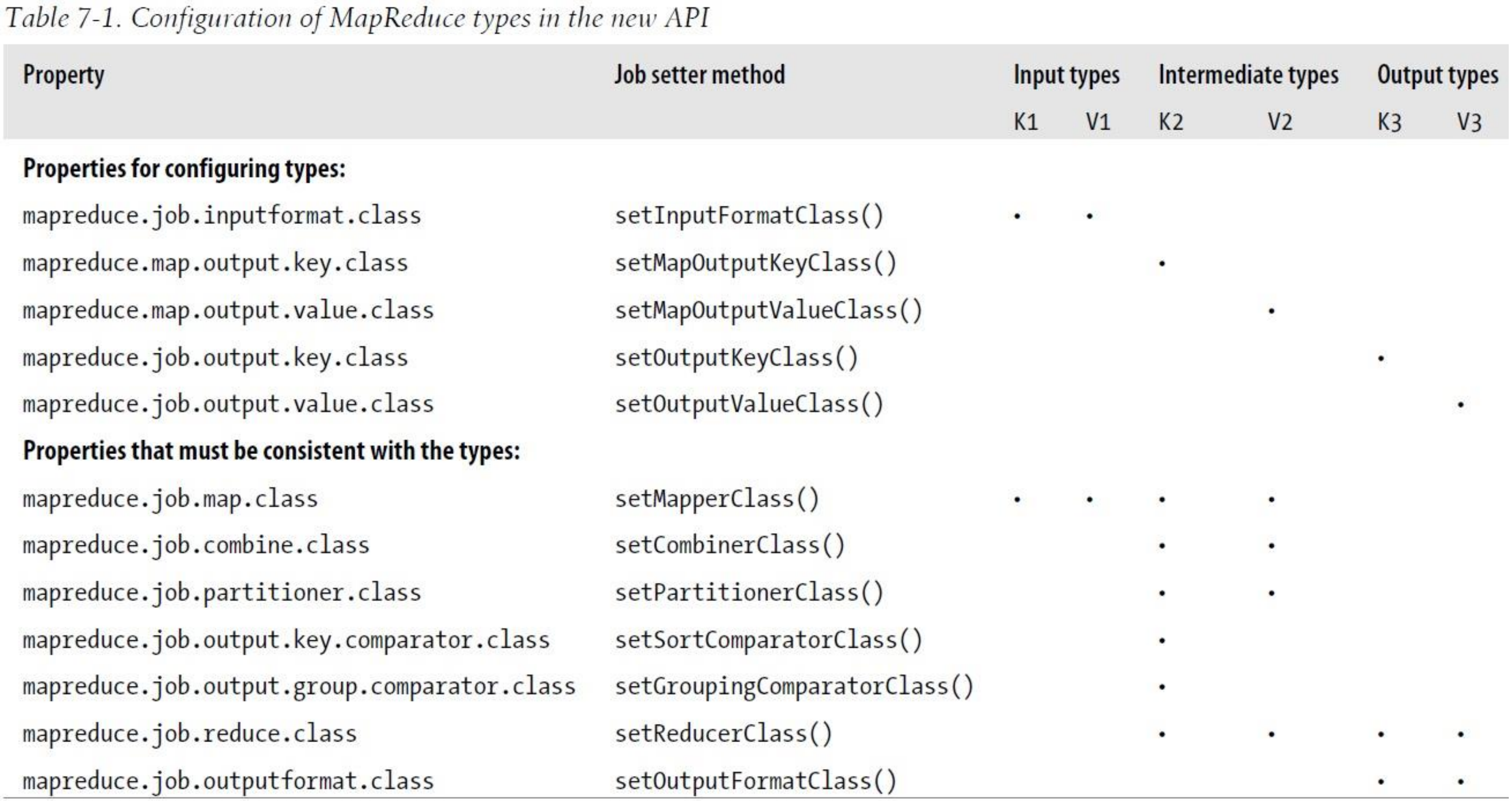


Table 7-1. Configuration of MapReduce types in the new API

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
`mapreduce.job.inputformat.class`	`setInputFormatClass()`	•	•				
`mapreduce.map.output.key.class`	`setMapOutputKeyClass()`			•			
`mapreduce.map.output.value.class`	`setMapOutputValueClass()`				•		
`mapreduce.job.output.key.class`	`setOutputKeyClass()`					•	
`mapreduce.job.output.value.class`	`setOutputValueClass()`						•
Properties that must be consistent with the types:							
`mapreduce.job.map.class`	`setMapperClass()`	•	•	•	•		
`mapreduce.job.combine.class`	`setCombinerClass()`			•	•		
`mapreduce.job.partitioner.class`	`setPartitionerClass()`			•	•		
`mapreduce.job.output.key.comparator.class`	`setSortComparatorClass()`			•			
`mapreduce.job.output.group.comparator.class`	`setGroupingComparatorClass()`			•			
`mapreduce.job.reduce.class`	`setReducerClass()`			•	•	•	•
`mapreduce.job.outputformat.class`	`setOutputFormatClass()`					•	•

Configuration of MapReduce Types – 2

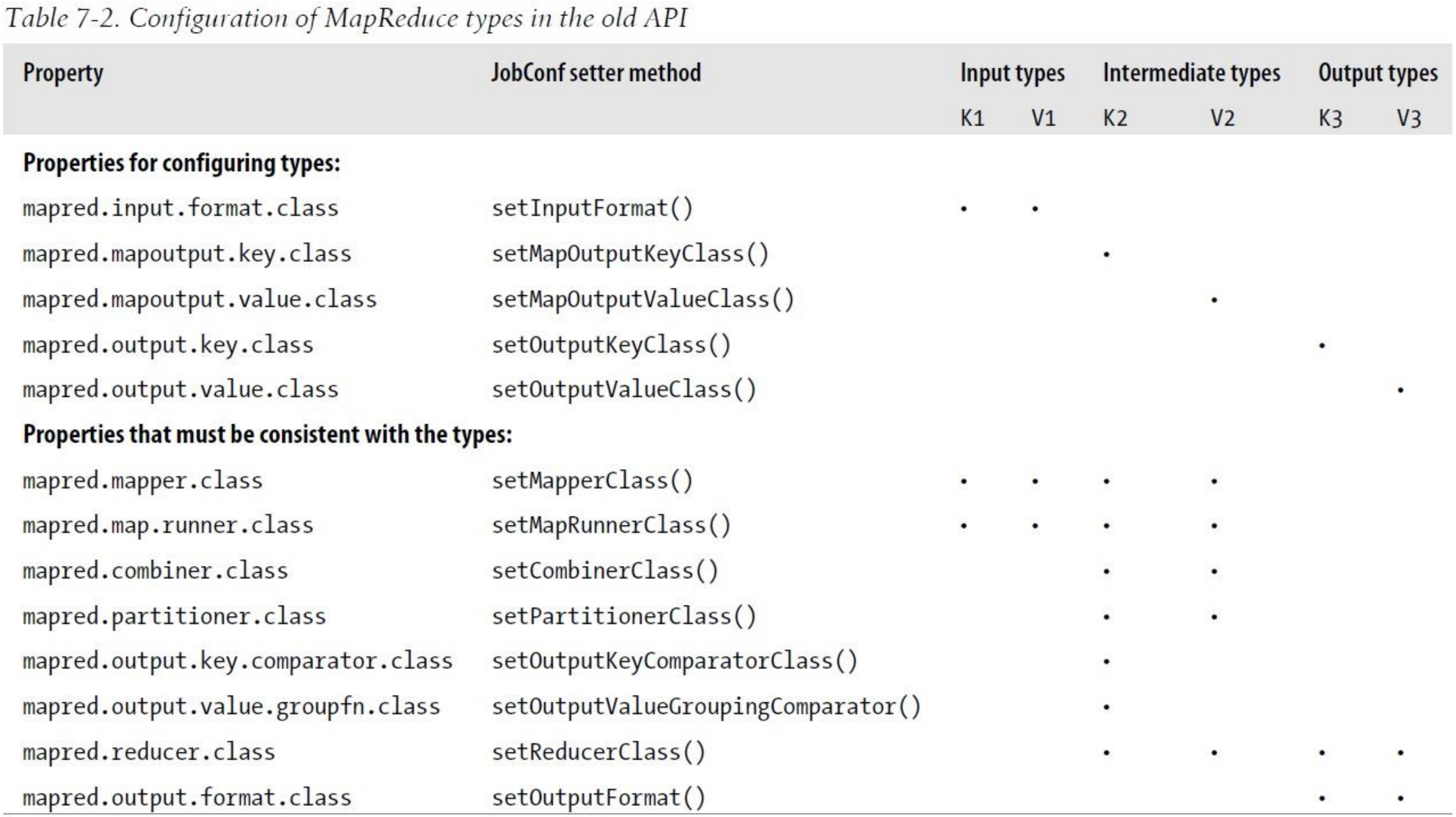
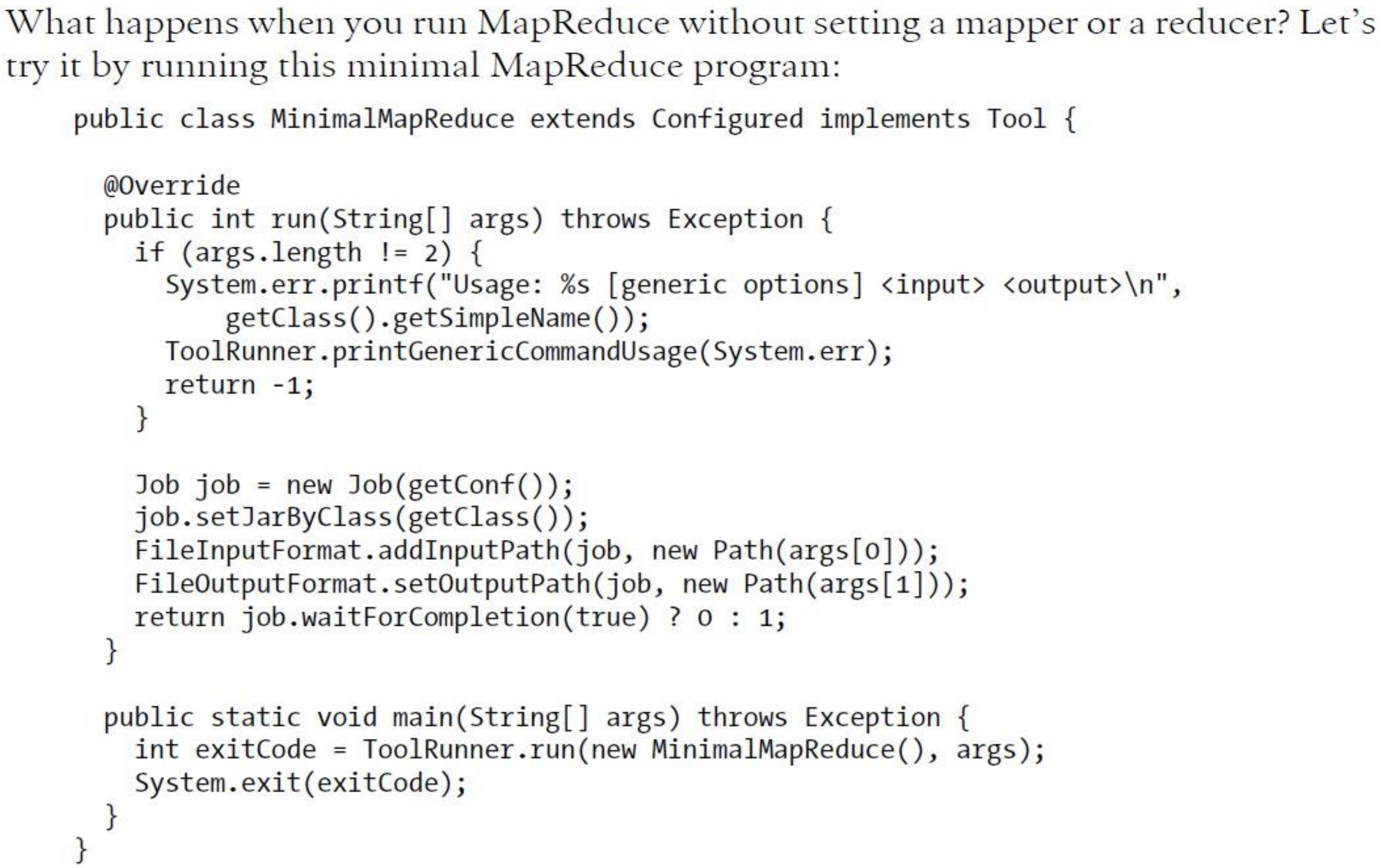


Table 7-2. Configuration of MapReduce types in the old API

Property	JobConf setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapred.input.format.class	setInputFormat()	•	•				
mapred.mapoutput.key.class	setMapOutputKeyClass()			•			
mapred.mapoutput.value.class	setMapOutputValueClass()				•		
mapred.output.key.class	setOutputKeyClass()					•	
mapred.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapred.mapper.class	setMapperClass()	•	•	•	•		
mapred.map.runner.class	setMapRunnerClass()	•	•	•	•		
mapred.combiner.class	setCombinerClass()			•	•		
mapred.partitioner.class	setPartitionerClass()			•	•		
mapred.output.key.comparator.class	setOutputKeyComparatorClass()			•			
mapred.output.value.groupfn.class	setOutputValueGroupingComparator()			•			
mapred.reducer.class	setReducerClass()			•	•	•	•
mapred.output.format.class	setOutputFormat()					•	•

Without Setting a Mapper or a Reducer – 1



What happens when you run MapReduce without setting a mapper or a reducer? Let's try it by running this minimal MapReduce program:

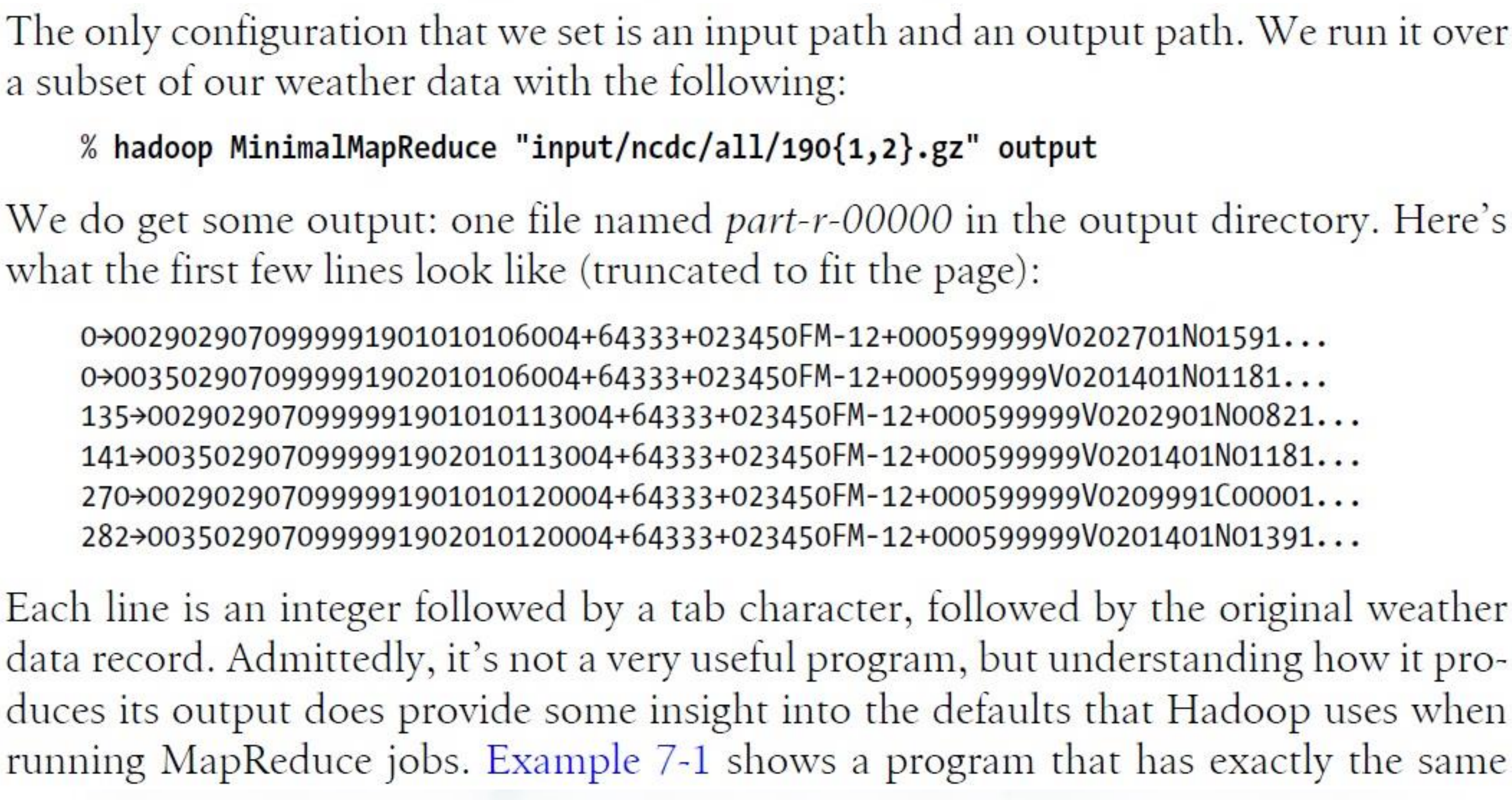
```
public class MinimalMapReduce extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.printf("Usage: %s [generic options] <input> <output>\n",
          getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }

    Job job = new Job(getConf());
    job.setJarByClass(getClass());
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    return job.waitForCompletion(true) ? 0 : 1;
  }

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

Without Setting a Mapper or a Reducer – 2



The only configuration that we set is an input path and an output path. We run it over a subset of our weather data with the following:

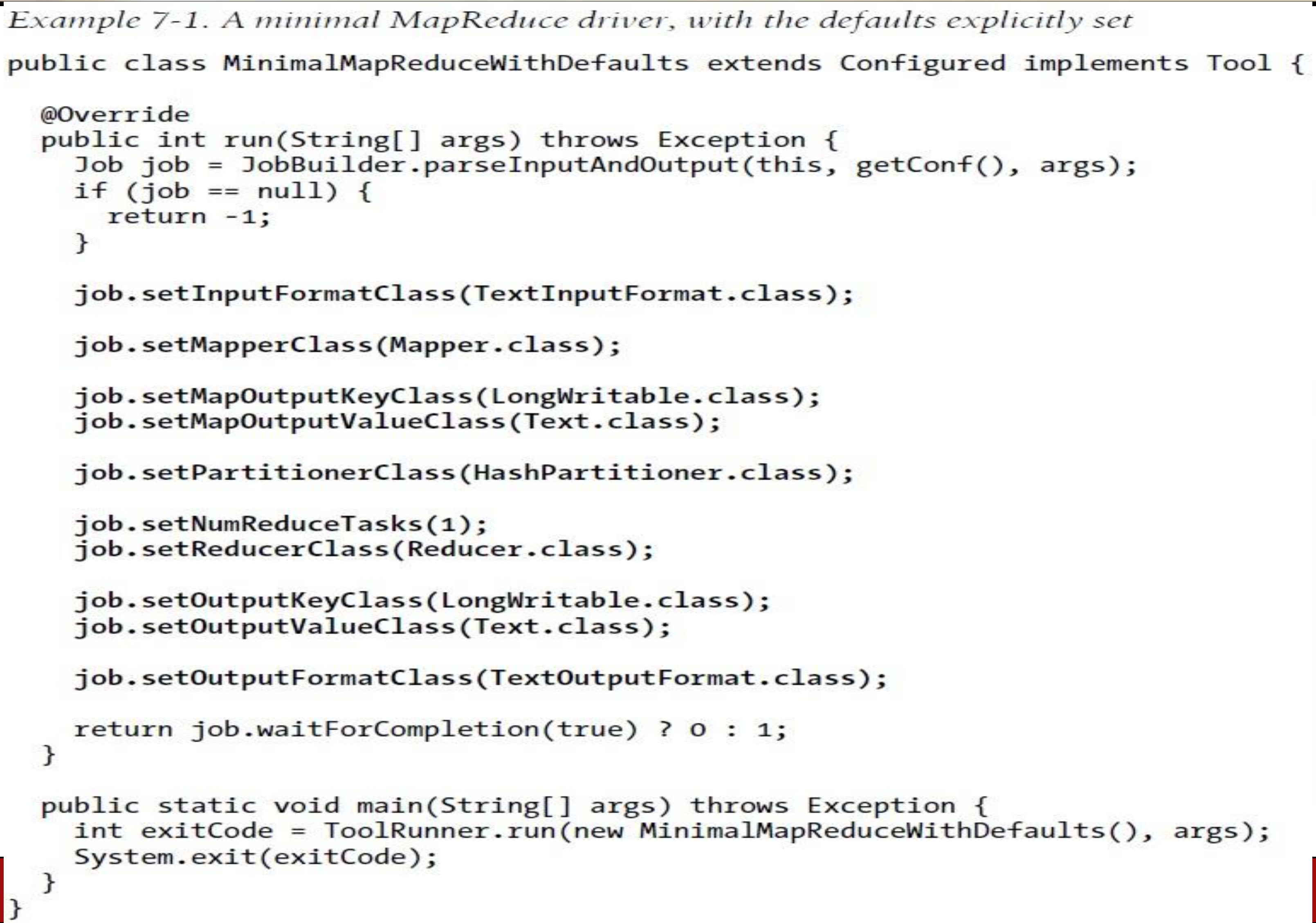
```
% hadoop MinimalMapReduce "input/ncdc/all/190{1,2}.gz" output
```

We do get some output: one file named *part-r-00000* in the output directory. Here's what the first few lines look like (truncated to fit the page):

```
0→0029029070999991901010106004+64333+023450FM-12+000599999V0202701N01591...
0→0035029070999991902010106004+64333+023450FM-12+000599999V0201401N01181...
135→0029029070999991901010113004+64333+023450FM-12+000599999V0202901N00821...
141→0035029070999991902010113004+64333+023450FM-12+000599999V0201401N01181...
270→0029029070999991901010120004+64333+023450FM-12+000599999V0209991C00001...
282→0035029070999991902010120004+64333+023450FM-12+000599999V0201401N01391...
```

Each line is an integer followed by a tab character, followed by the original weather data record. Admittedly, it's not a very useful program, but understanding how it produces its output does provide some insight into the defaults that Hadoop uses when running MapReduce jobs. Example 7-1 shows a program that has exactly the same

MapReduce Default Settings



Example 7-1. A minimal MapReduce driver, with the defaults explicitly set

```
public class MinimalMapReduceWithDefaults extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
      return -1;
    }

    job.setInputFormatClass(TextInputFormat.class);

    job.setMapperClass(Mapper.class);

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

    job.setPartitionerClass(HashPartitioner.class);

    job.setNumReduceTasks(1);
    job.setReducerClass(Reducer.class);

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

    job.setOutputFormatClass(TextOutputFormat.class);

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

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

Choosing the Number of Reducers

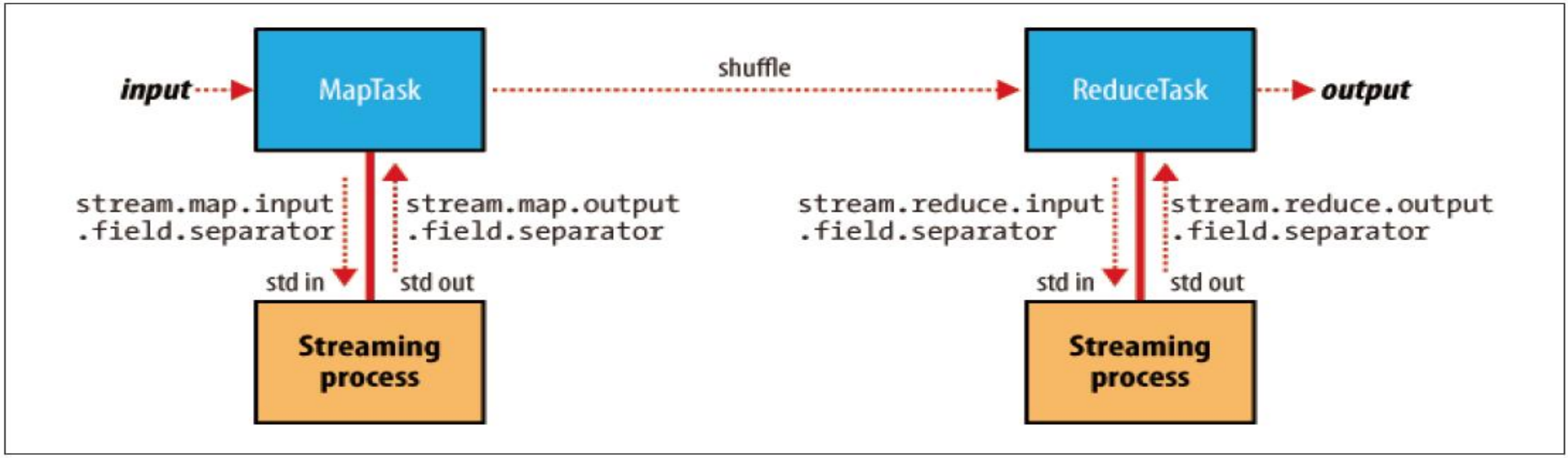
- The single reducer default is no good and needs to be set to a larger number.
- The optimal number of reducers is related to the total number of available reducer slots, which can be calculated as:

  Total nodes * tasktracker.reduce.tasks.maximum (default 2)
- To have slightly fewer reducers than total slots.
  - Tolerate a few failures without extending job execution time.

Keys and values in Streaming

A streaming application can control the separator.

- Default separator: tab character;
- Separators may be configured independently for maps and reduces;
- A key can be made up of the first n fields:
  - Ex) Output was a,b,c (and separator is a comma), n=2
    - Key: a,b   Value:c


input
MapTask
shuffle
ReduceTask
output
stream.map.input .field.separator
std in
stream.map.output .field.separator
std out
Streaming process
stream.reduce.input .field.separator
std in
stream.reduce.output .field.separator
std out
Streaming process

Input Splits

- InputSplit
  - A chunk of the input processed by a single map;
  - Each split is divided into records;
  - The map processes each record—a key-value pair;
  - Process the largest split first to minimize the job runtime;
  - The client sends the calculated splits to the JobTracker.

InputFormat Class Hierarchy

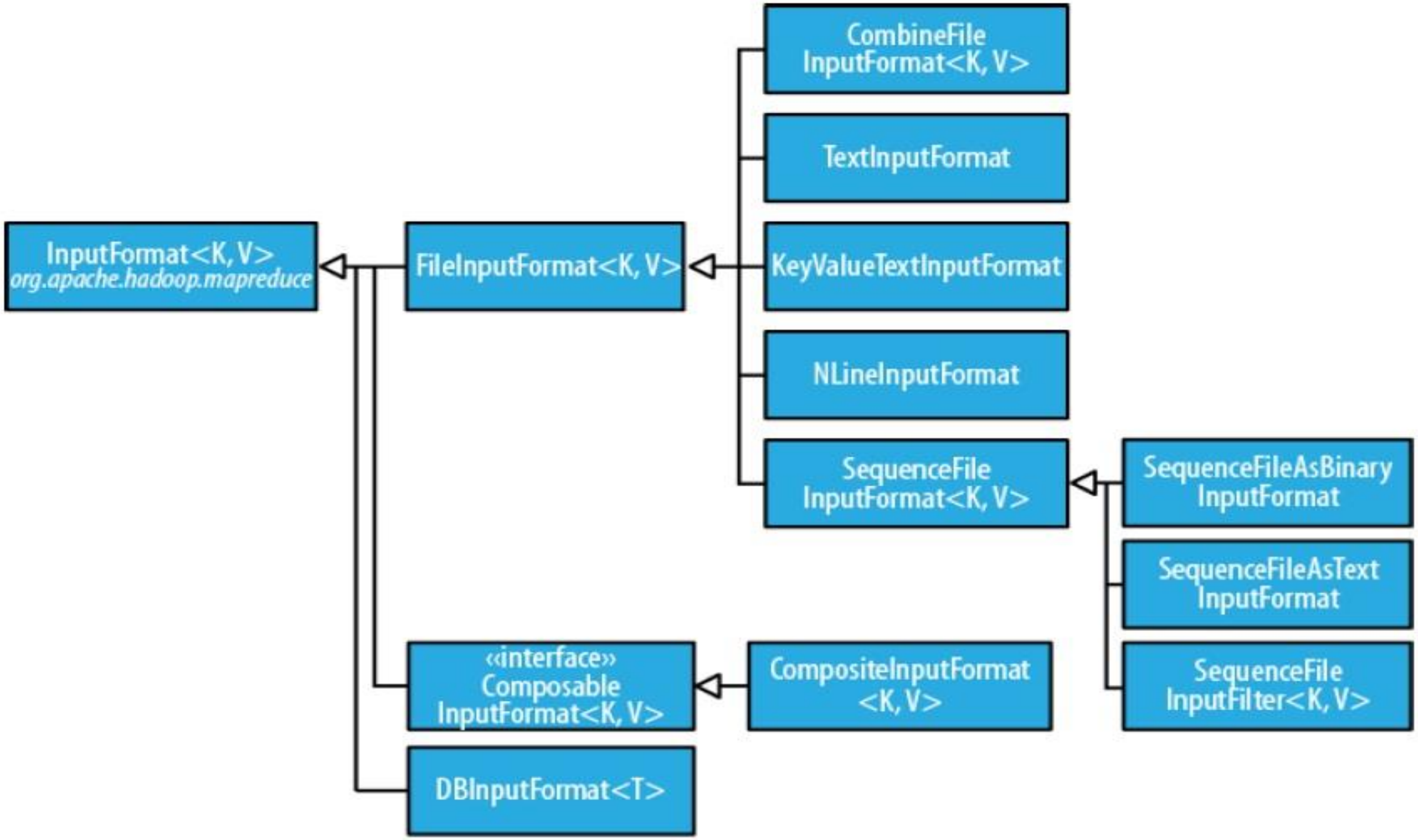
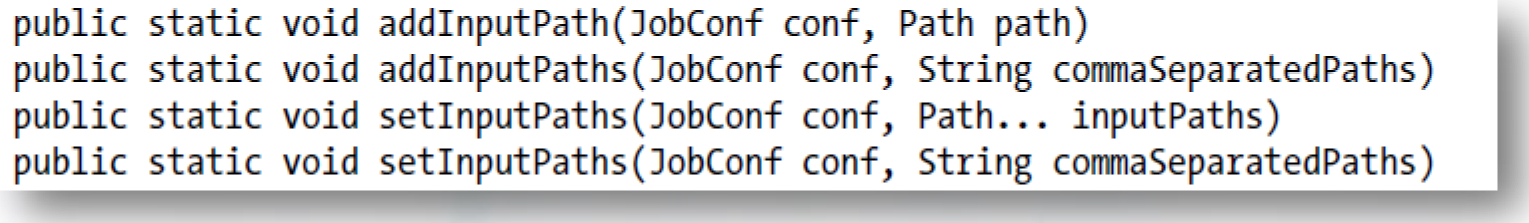

InputFormat<K, V>
org.apache.hadoop.mapreduce
FileInputFormat<K, V>
CombineFile InputFormat<K, V>
TextInputFormat
KeyValueTextInputFormat
NLineInputFormat
SequenceFile InputFormat<K, V>
SequenceFileAsBinary InputFormat
SequenceFileAsText InputFormat
SequenceFile InputFilter<K, V>
«interface» Composable InputFormat<K, V>
CompositeInputFormat <K, V>
DBInputFormat<T>

Figure 7-2. InputFormat class hierarchy

Input Paths

FileInputFormat offers 4 static methods for setting a Job’s input paths:

- addInputPath() and addInputPaths()
  - Add a path or paths to the list of inputs
  - Can call these methods repeatedly
- Two setInputPaths()
  - Set entire list of paths in one time (Replacing any paths that were set in previous calls)
- A path may represent
  - A file or a directory (consider all files in the directory as input)
  - A collection of files and directories by using a glob



```
public static void addInputPath(JobConf conf, Path path)
public static void addInputPaths(JobConf conf, String commaSeparatedPaths)
public static void setInputPaths(JobConf conf, Path... inputPaths)
public static void setInputPaths(JobConf conf, String commaSeparatedPaths)
```

FileInputFormat

- FileInputFormat uses a default filter that excludes hidden files.
- FileInputFormat splits only large files that larger than an HDFS block.
- The split size is normally the size of an HDFS block.
- The minimum split size is usually 1 byte.
- The maximum split size defaults to the maximum value of Java long type.

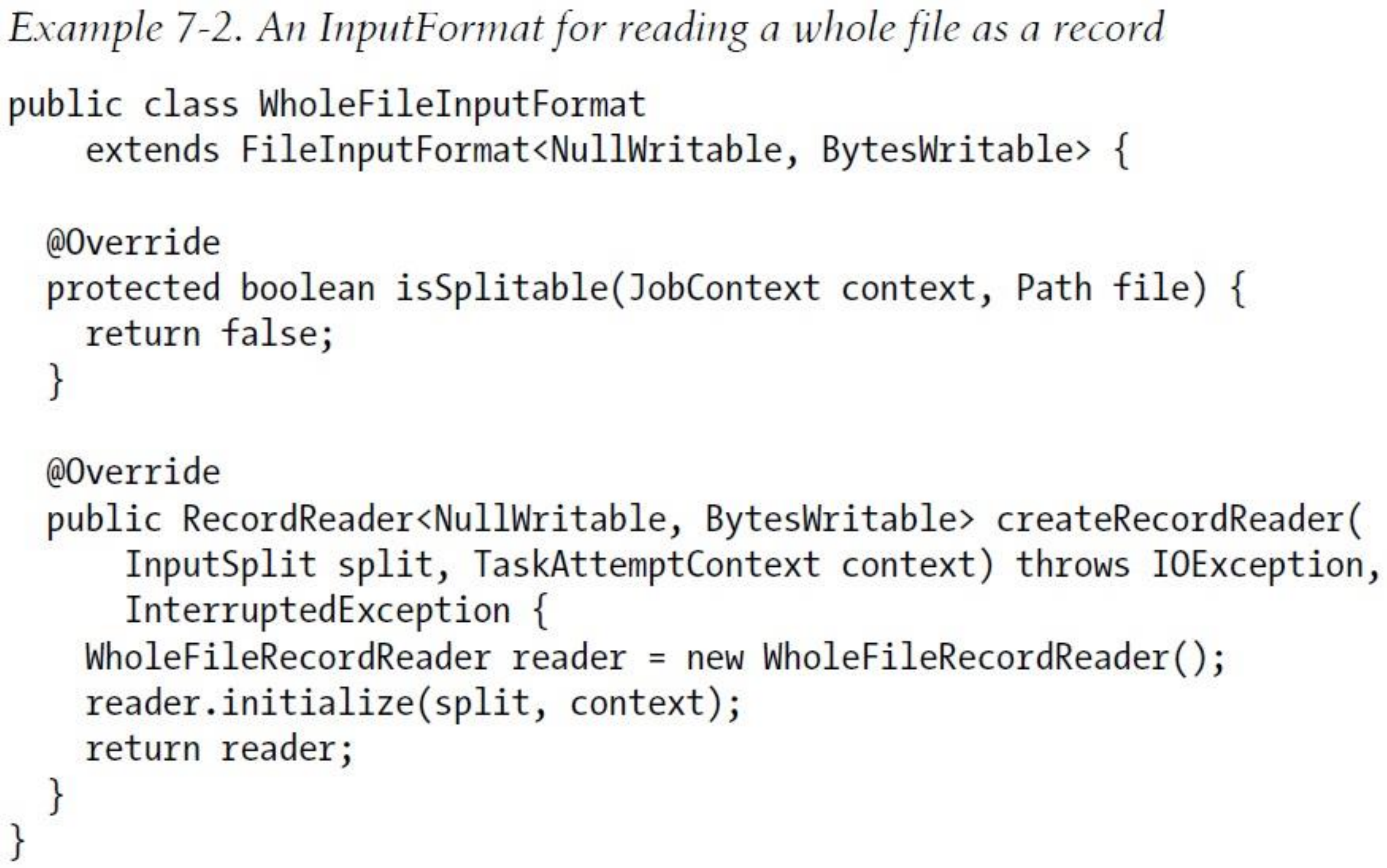
Small Files and CombineFileInputFormat

- Hadoop works better with a small number of large files than a large number of small files.
  - FileInputFormat generates splits that each split is all or part of a single file
- Use CombineFileInputFormat to pack many files into splits.
  - Designed to work well with small files
  - Take node and rack locality when packing blocks into split
  - Worth when already have a large number of small files in HDFS
- Avoiding the many small files is a good idea.
  - Reduce the number of seeks
  - Merge small files into larger files by using a SequenceFile

Preventing Splitting

- Some application don’t want files to be split.
  - Want to process entire data by a single mapper
- Two ways of ensuring an existing file is not split:
  - Set the minimum split size larger than the largest file size
  - Override the isSplitable() method to return **false**

Processing a Whole File As a Record – p.242



Example 7-2. An InputFormat for reading a whole file as a record

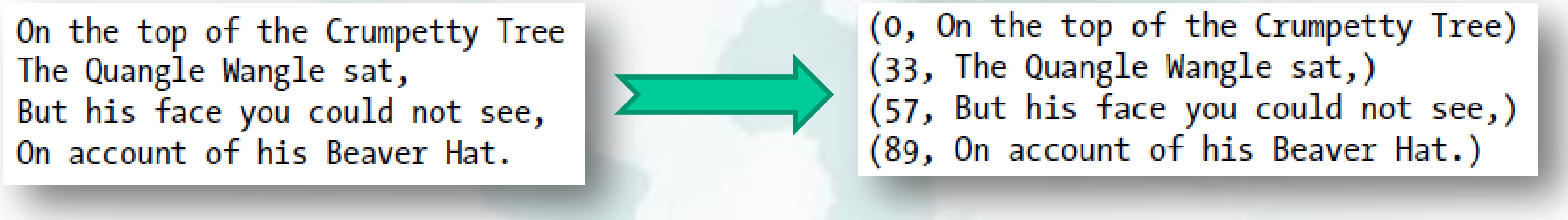
```
public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

  @Override
  protected boolean isSplitable(JobContext context, Path file) {
    return false;
  }

  @Override
  public RecordReader<NullWritable, BytesWritable> createRecordReader(
      InputSplit split, TaskAttemptContext context) throws IOException,
      InterruptedException {
    WholeFileRecordReader reader = new WholeFileRecordReader();
    reader.initialize(split, context);
    return reader;
  }
}
```

TextInputFormat – 1

- TextInputFormat is the default InputFormat
  - Key: The byte offset of the beginning of the line (LongWritable) ; Not line number
  - Value: The contents of the line excluding any line terminators (Text)

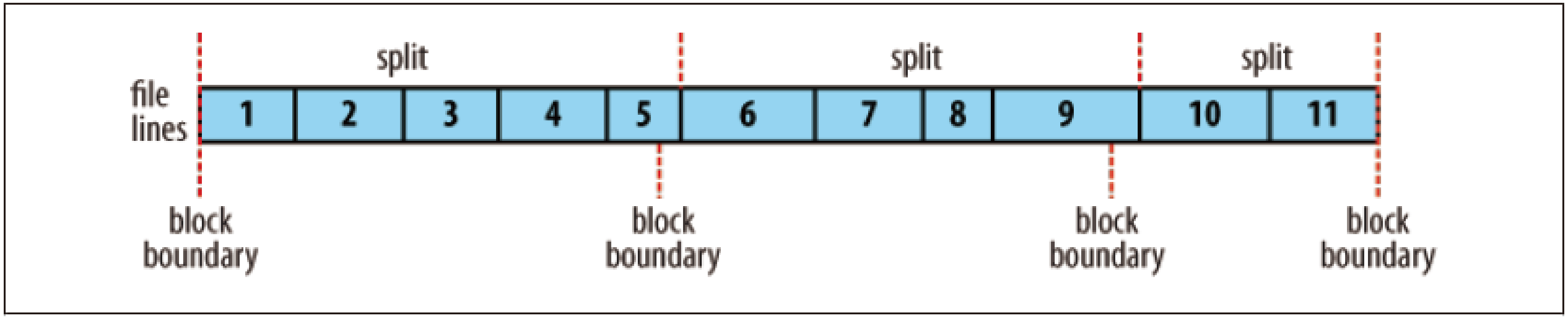


```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

TextInputFormat – 2

- The offset within the file of each line is known by each split independently of the other splits;
- Each split knows the size of the preceding splits;
- Add previous split size onto the offsets within the split to produce a global file offset.



NLineInputFormat

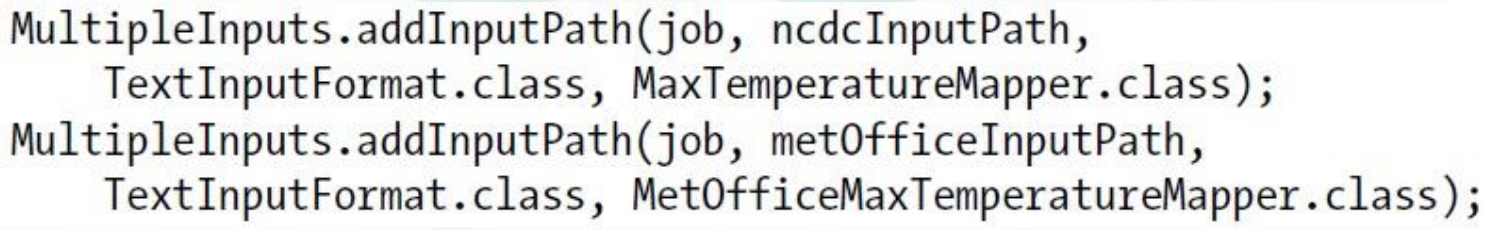
- **TextInputFormat, KeyValueTextInputFormat** – each mapper receives a variable number of lines of input.
- **NLineInputFormat** – receive a fixed number of lines of input.
  - N: The number of lines of input.
  - Control N in Mapred.line.input.format.linespermap property
  - With N set to one, each mapper receives exactly one line of input.

Binary Input

- SequenceFileInputFormat
  - Hadoop’s sequence file format stores sequences of binary key-value pairs
  - Data is splittable (Data has sync points)
  - Use SequenceFileInputFormat
- SequenceFileAsTextInputFormat
  - Convert the sequence file’s keys and values to Text objects
  - Use toString() method
- SequenceFileAsBinaryInputFormat
  - Retrieve the sequence file’s keys and values as opaque binary objects

Multiple Inputs

- Use MultipleInput when have data sources that provide the same type of data but in different formats, for example:
    - One might be tab-separated plain text, the other a binary sequence file;
    - Need to be parsed differently;
    - Use different mappers;
    - The map outputs have the same types;
    - Reducers are not aware of the different mappers.



```
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

Output Formats

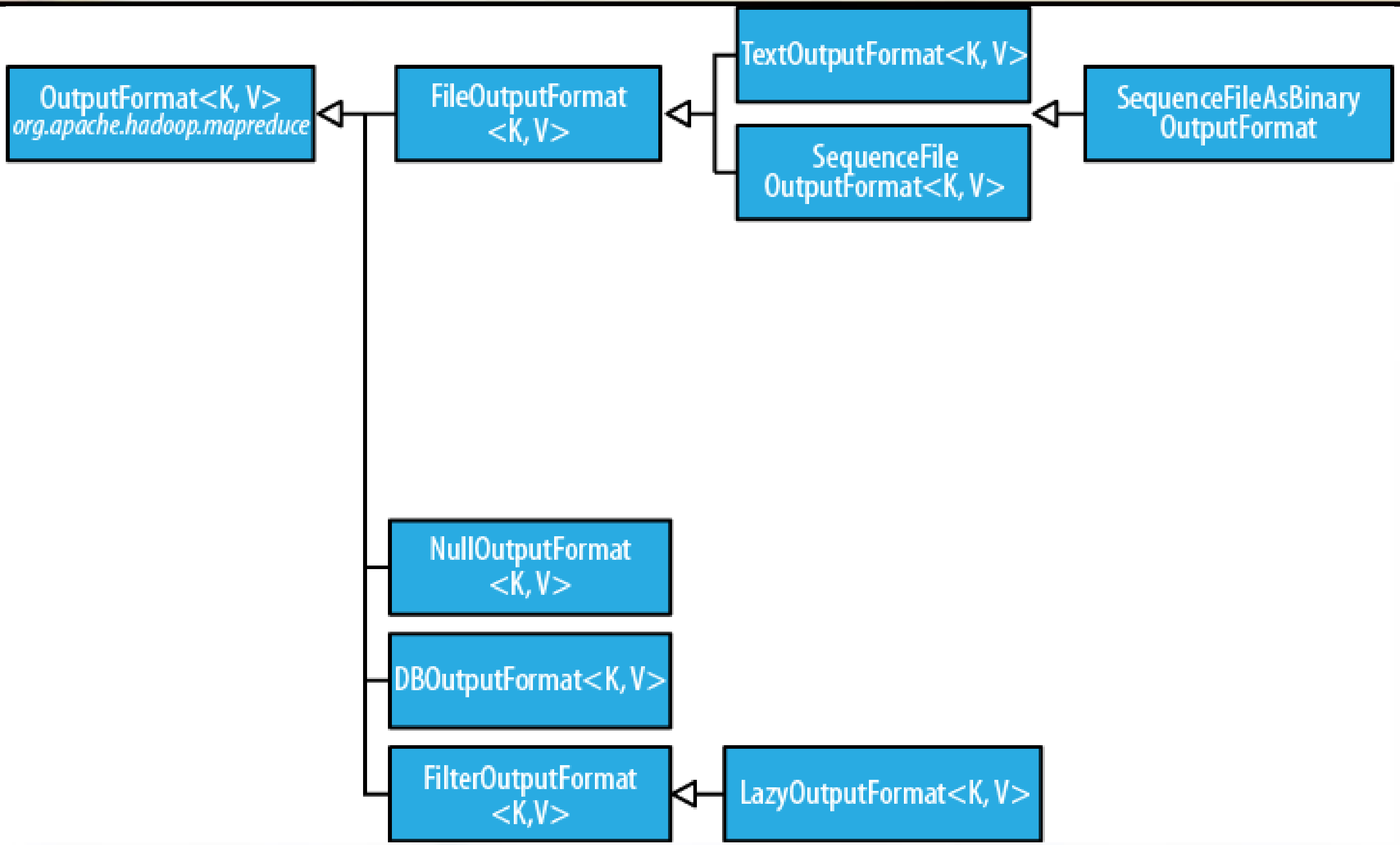


Figure 7-4. The OutputFormat class hierarchy

Text Output

TextOutputFormat (default)

- Write records as lines of text
- Keys and Values may be of any type
  - It calls toString() method
- Each key-value pair is separated by a tab character
  - Set the separator in **mapred.textoutputformat.separator** property

Binary Output

- SequenceFileOutputFormat
  - Write sequence files for its output.
  - Compact, readily compressed (Useful for a further MapReduce job).
- SequenceFileAsBinaryOutputFormat
  - Write keys and values in raw binary format into a SequenceFile container.
- MapFileOutputFormat
  - Write MapFiles as output.
  - The keys in MapFile must be added in order.

MultipleOutputs – 1

MultipleOutputs

- Output file names are derived from the output keys and values, or from an arbitrary string.
- Allows each reducer (or mapper in a map-only job) to create more than a single file.
- Filenames are of the form name-m-nnnnn for map outputs and name-r-nnnnn for reduce outputs.
- Name is arbitrary and set by the program, and nnnnn is an integer designating the part number, starting from zero.

MultipleOutputs – 2

- P. 255 Example 7-5