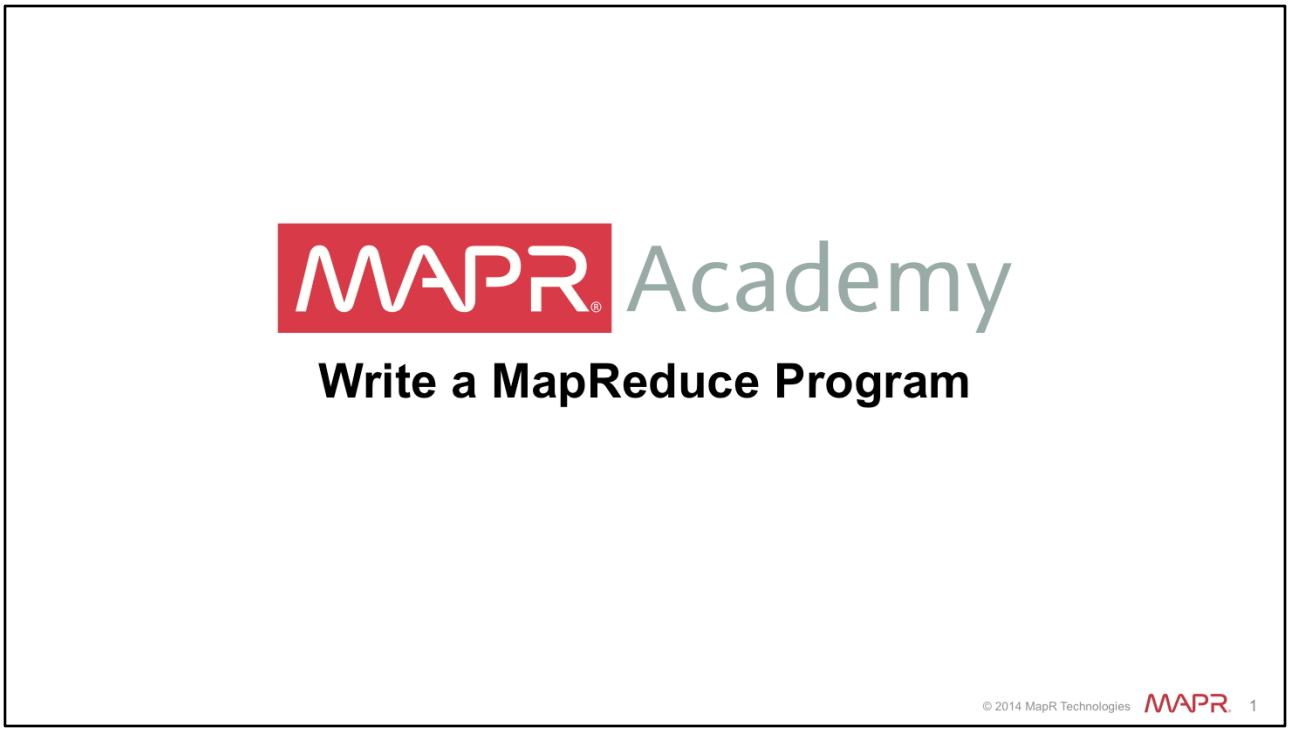
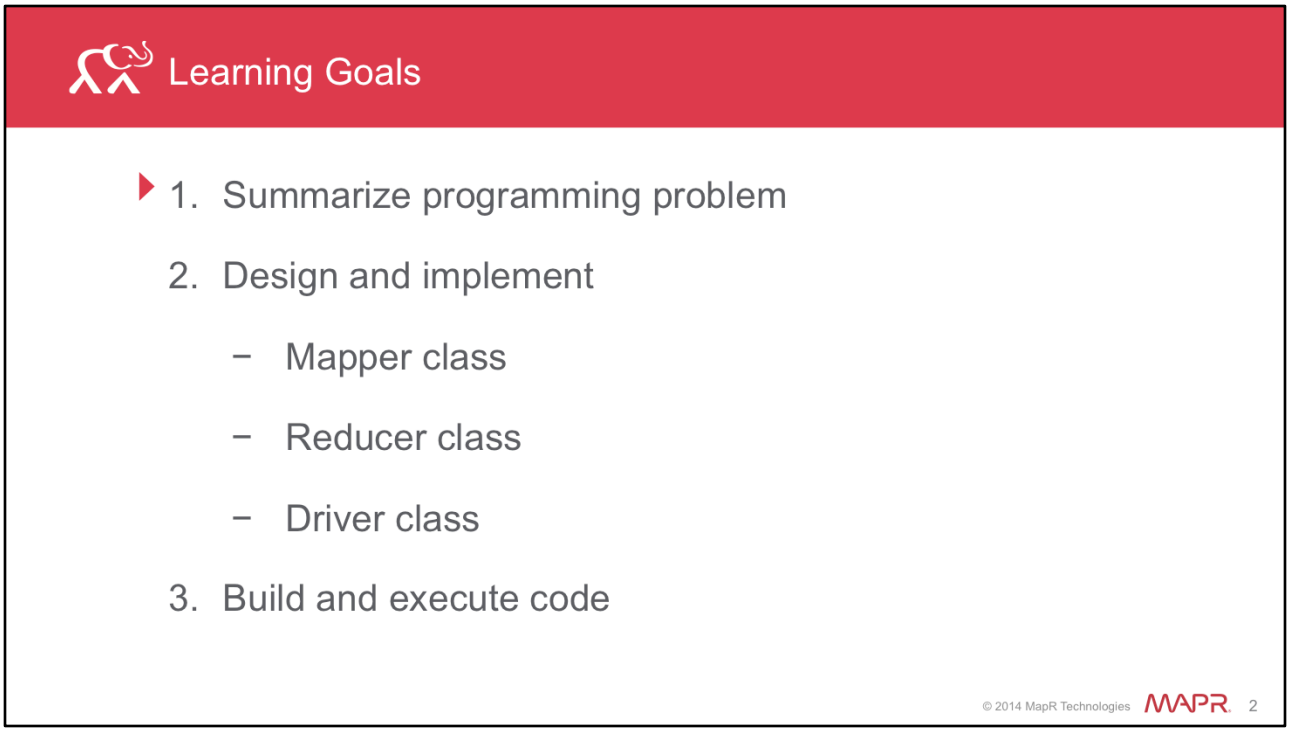
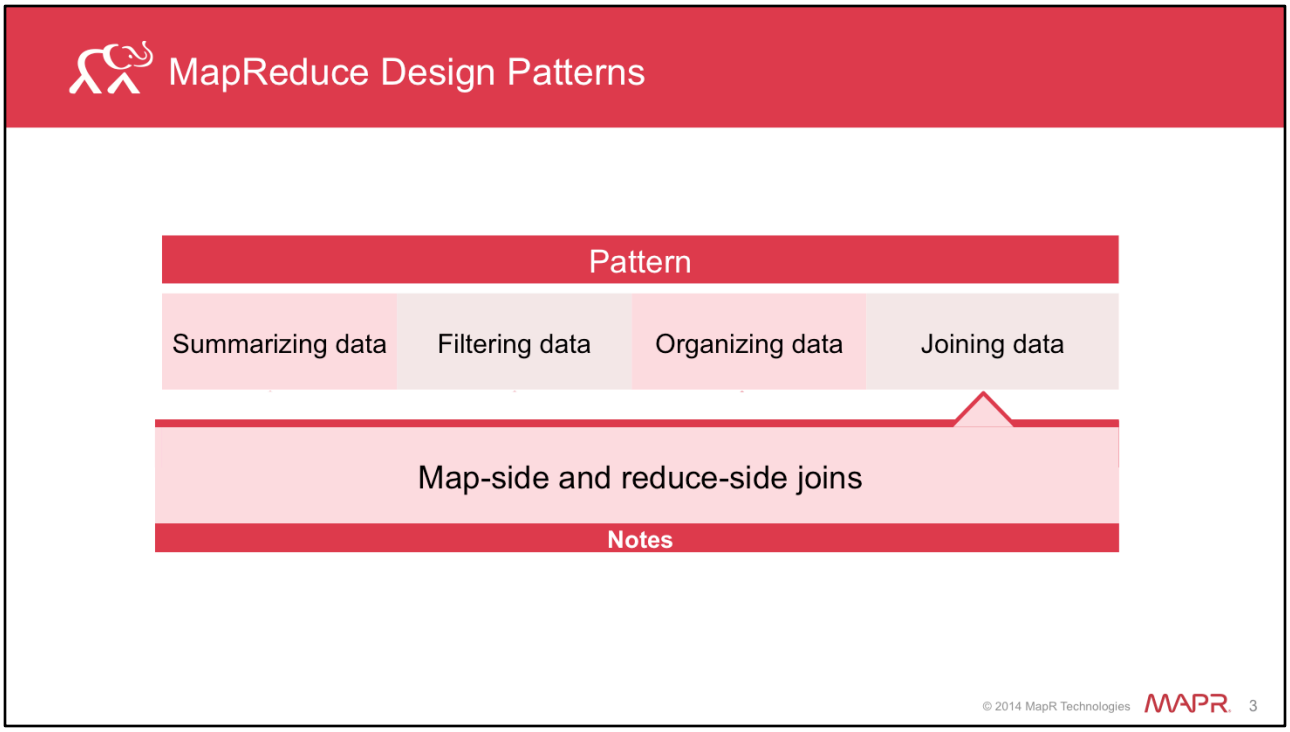


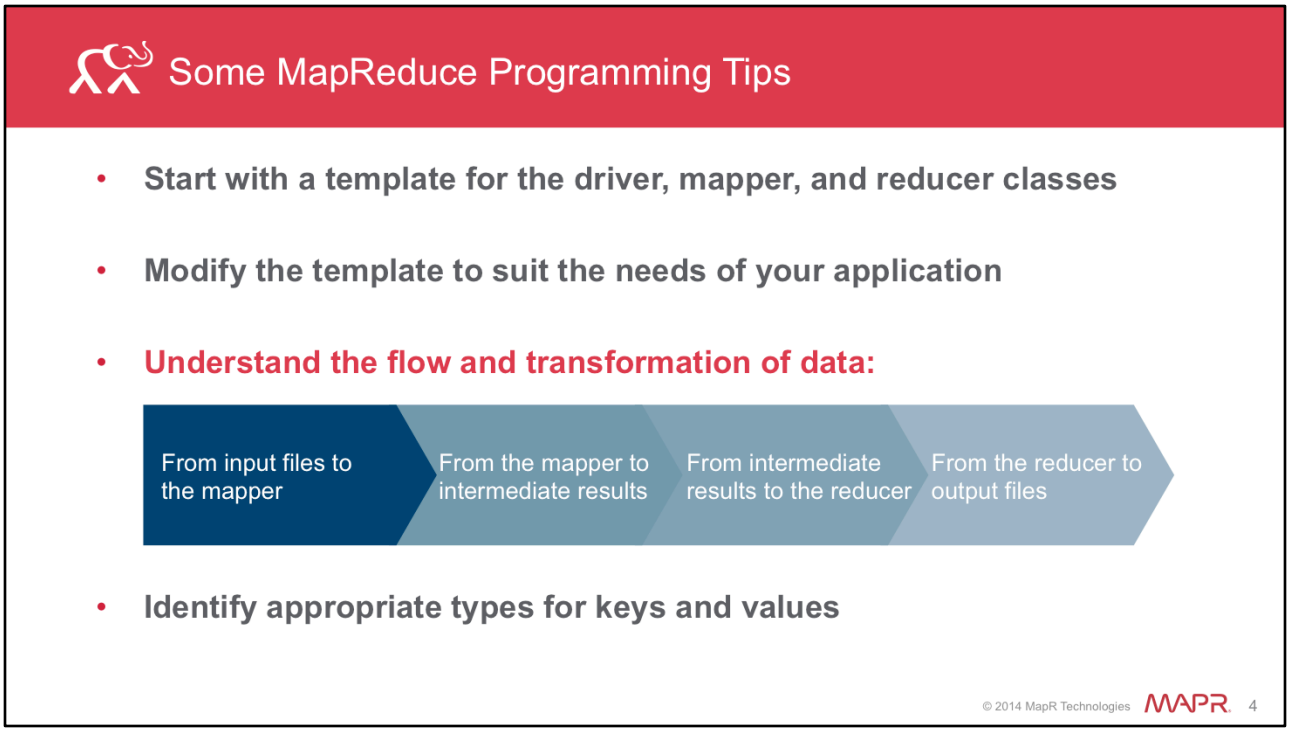
Welcome to MapR Academy. Use the navigation in the top right hand corner to move through the lesson. You can also jump to specific topics by selecting them from the table of contents on the left side of the screen.



Welcome to Lesson Three. In this lesson we will learn about the programming problem, how to design, implement the Mapper, Reducer, and Driver classes. We will also build, execute code, and examine the output.



MapReduce may be used for a wide variety of batch-processed tasks. Click on each to learn more.



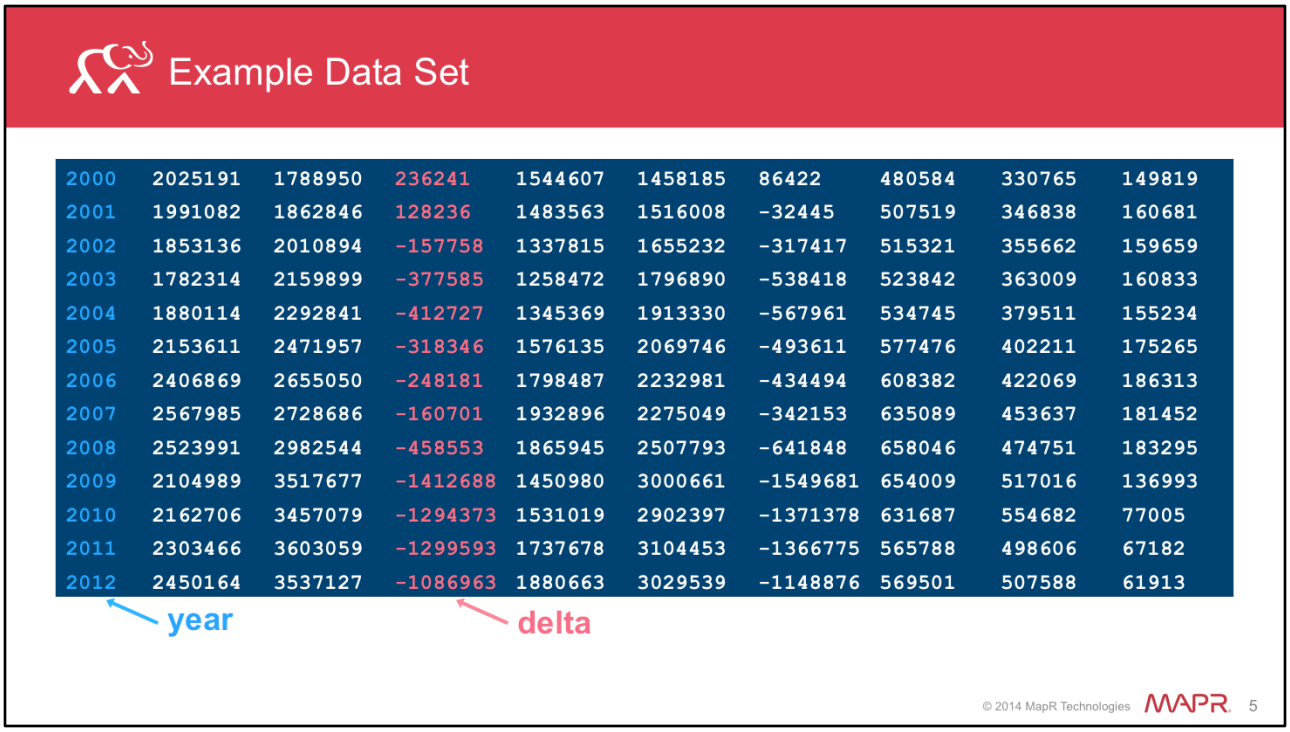
Let's look at some tips for MapReduce programming.

Most of the non-logic part of your MapReduce code is the same across all your applications. This includes import statements, class definitions, and method signatures. Construct a template and modify it according to the needs of each application rather than starting from scratch every time.

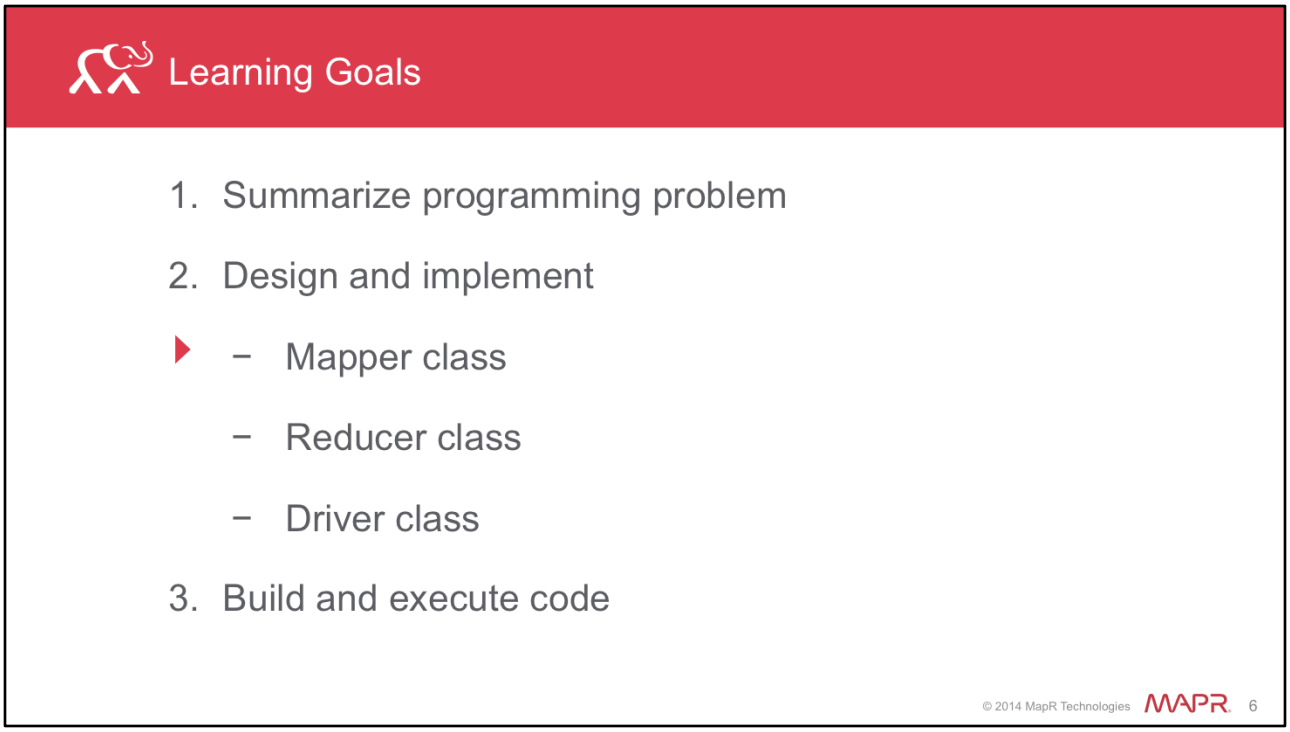
The most important aspect of writing MapReduce applications is understanding how data is transformed as it executes in the MapReduce framework. There are essentially four transformations from start to finish:

First, how data is transformed from the input files and fed into the mappers
Then how is data transformed by the mappers
Next how data is sorted, merged, and presented to the reducer
Last, how the reducers transform the data and write to output files

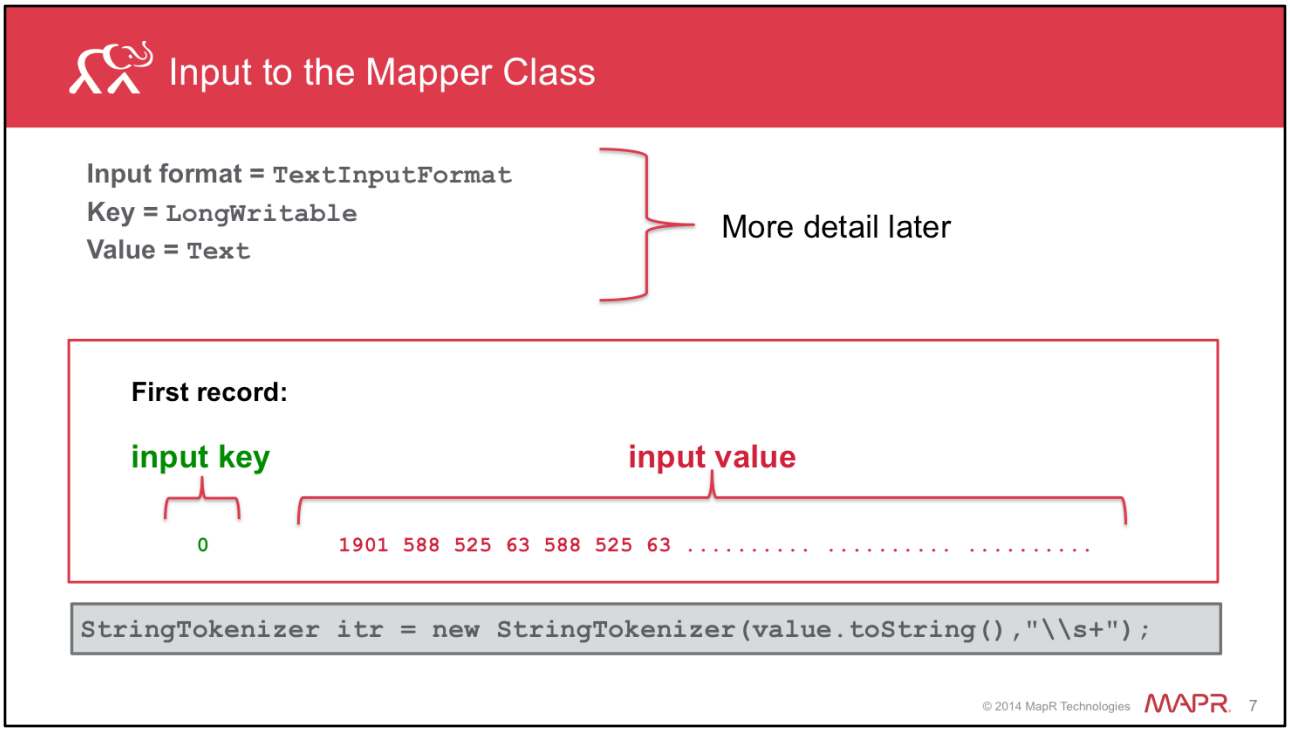
Similarly, it is important to use the appropriate types for your keys and values. There are several discussions in this course which provide details about which type of data to use for your input and output. Above all else, you must ensure that your input and output types match up, or



We will use this data set for our programming exercise example. There are a total of 10 fields of information in each line. Our programming objective is to find the minimum value in the delta column and the year associated with that minimum. We will ignore all the other fields of data.



In this section how to implement the mapper class for our programming problem.

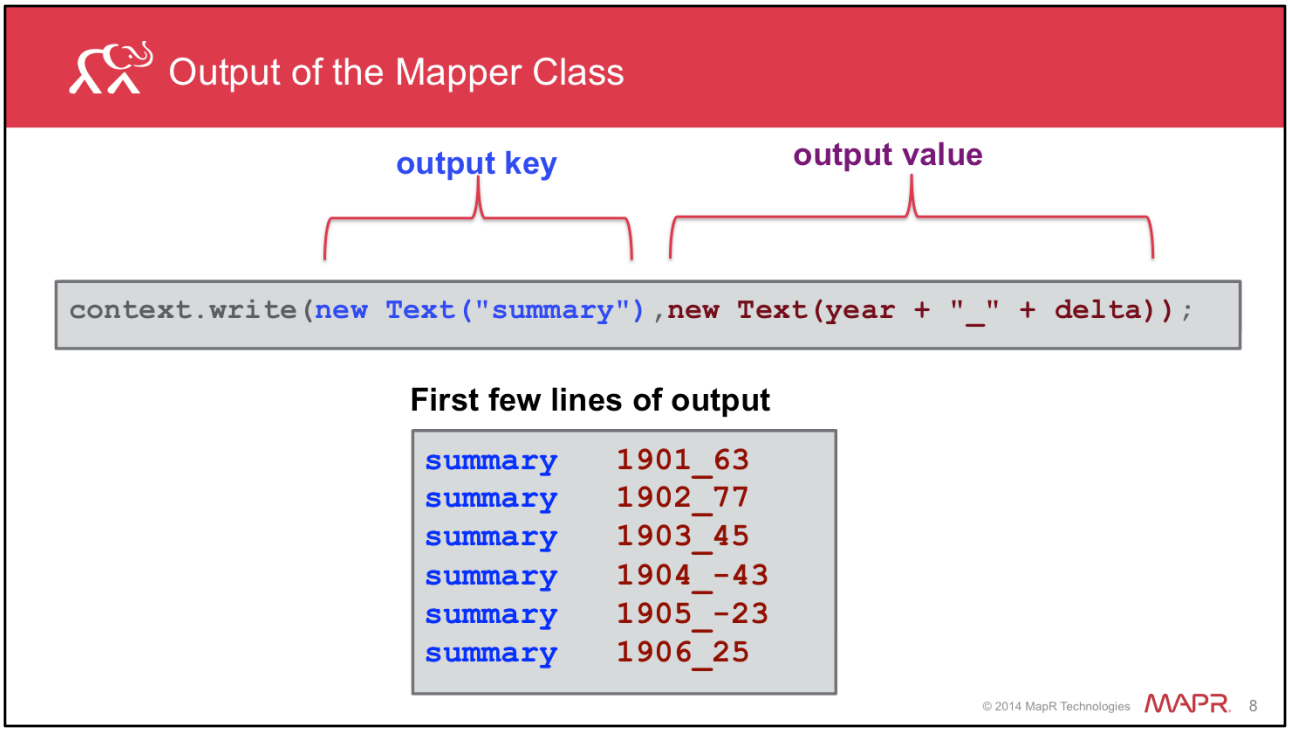


Now let's define and implement the Mapper class to solve the programming problem.

In this example, we will be using the TextInputFormat class as the type of input to the Mapper class. Using this format, the key from the default record reader associated with TextInputFormat is the byte offset into the file (LongWritable). We won't be using this key for anything in our program, so we will just ignore it. The value from the default record reader is the line read from the input file, in Text.

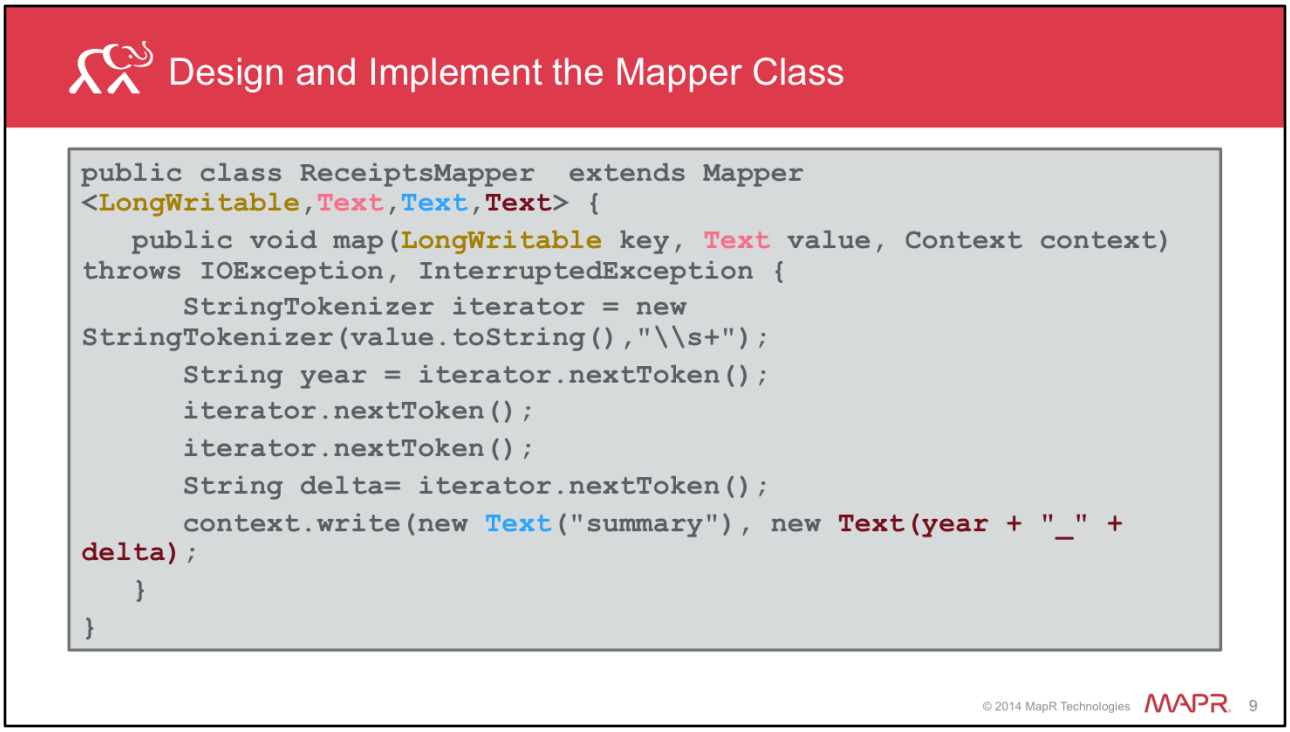
Let's look at the first record. The key of the first record is the byte offset to the line in the input file (the 0^{th} byte). The value of the first record includes the year, number of receipts, outlays, and the delta (receipts – outlays). Recall that we are interested only in the first and fourth fields of the record value.

Since the record value is in Text format, we will use a StringTokenizer to break up the Text string into individual fields.



After grabbing fields 1 and 4 from a record, the Mapper class emits the constant key ("summary") and a composite output value of the year (field 1) followed by an underscore followed by the delta (field 4). The first few lines of output are shown. From the first record in the data set, the mapper emits the key "summary" and the value 1901 underscore 63. From the second record in the data set, the mapper emits the same key "summary" and the value 1902 underscore 77. The mapper continues writing output for each record it reads from the file. Our file is rather small – it only goes from the year 1901 to the year 2012.

Note that since we hard-coded the key to always be the string "summary", there will be only one partition (and therefore only one reducer) when this mapreduce program is launched.



Note that the code shown omits the import statements in order to conserve space in the text area.

The four arguments to the Mapper class represent input key type, input value type, output key type, and output value type. In this case, everything is of type Text except for the input key to the mapper, which is of type LongWritable to represent the byte offset of the record in the file.

The first two arguments to the map() method are the key and value which must match the types defined in the Mapper class definition. The third argument is the context which encapsulates the Hadoop job running context (configuration, record reader, record writer, status reporter, input split, and output committer). You can dereference those objects if you wish (e.g. update the task status through the status reporter or get a parameter value from the job configuration).

We create a StringTokenizer to iterate over the values in the record, separated by white space.

We call nextToken() which assigns the first field in the record to the year variable.

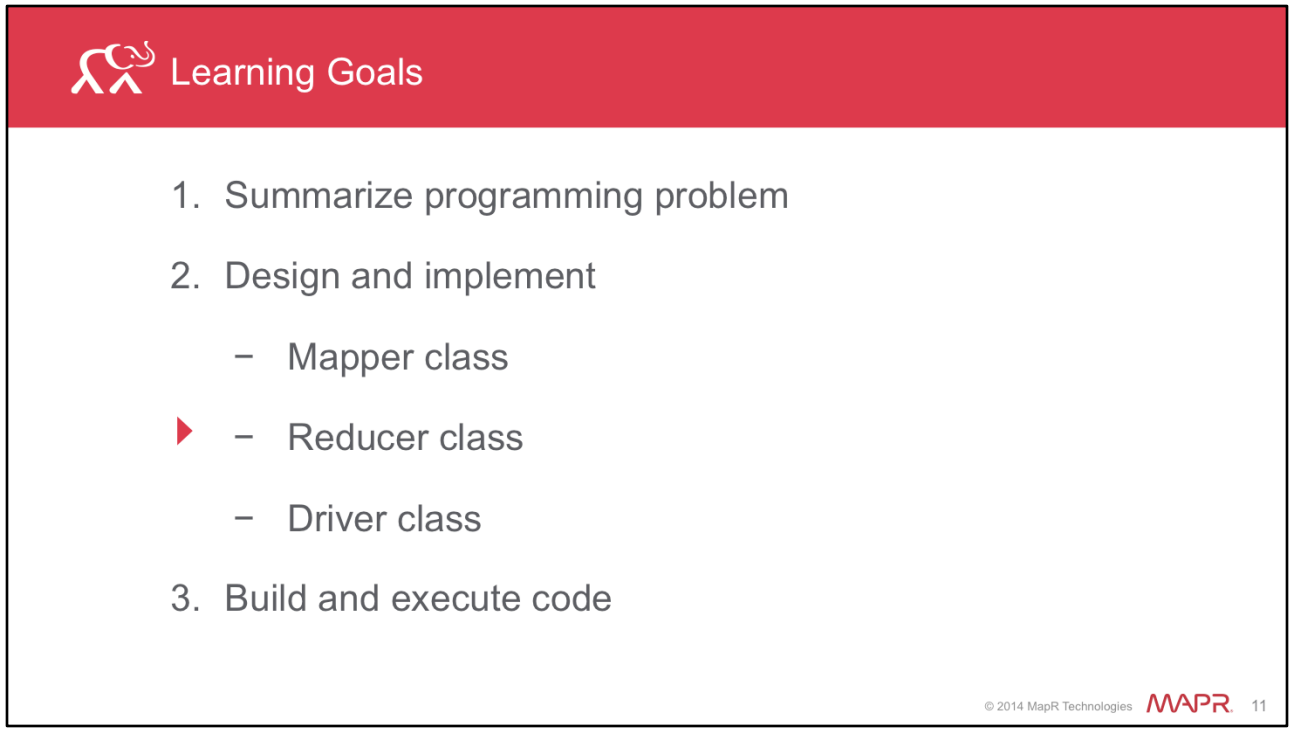
We then call nextToken() twice so that we can get to field 4, which is the delta.

Last, we emit the key ("summary") and composite value (year_delta).

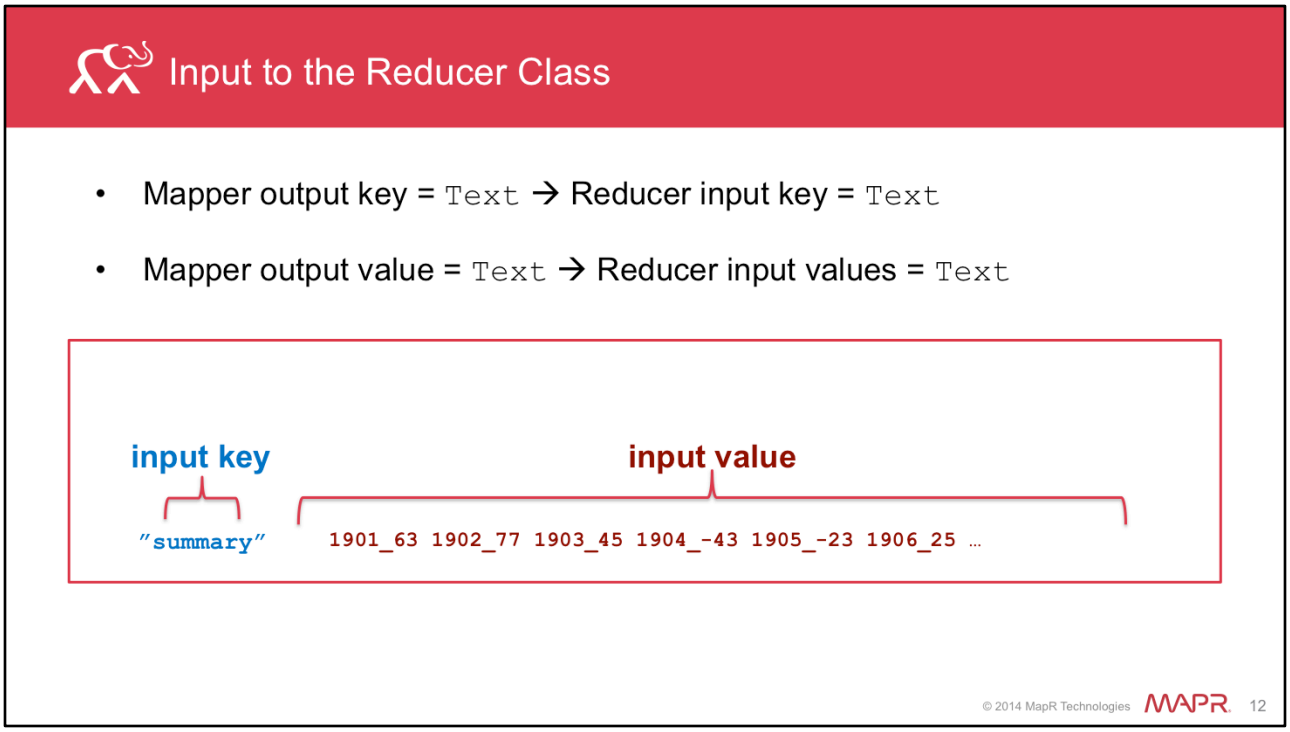
Knowledge Check

Which statements are true of Mapper class?

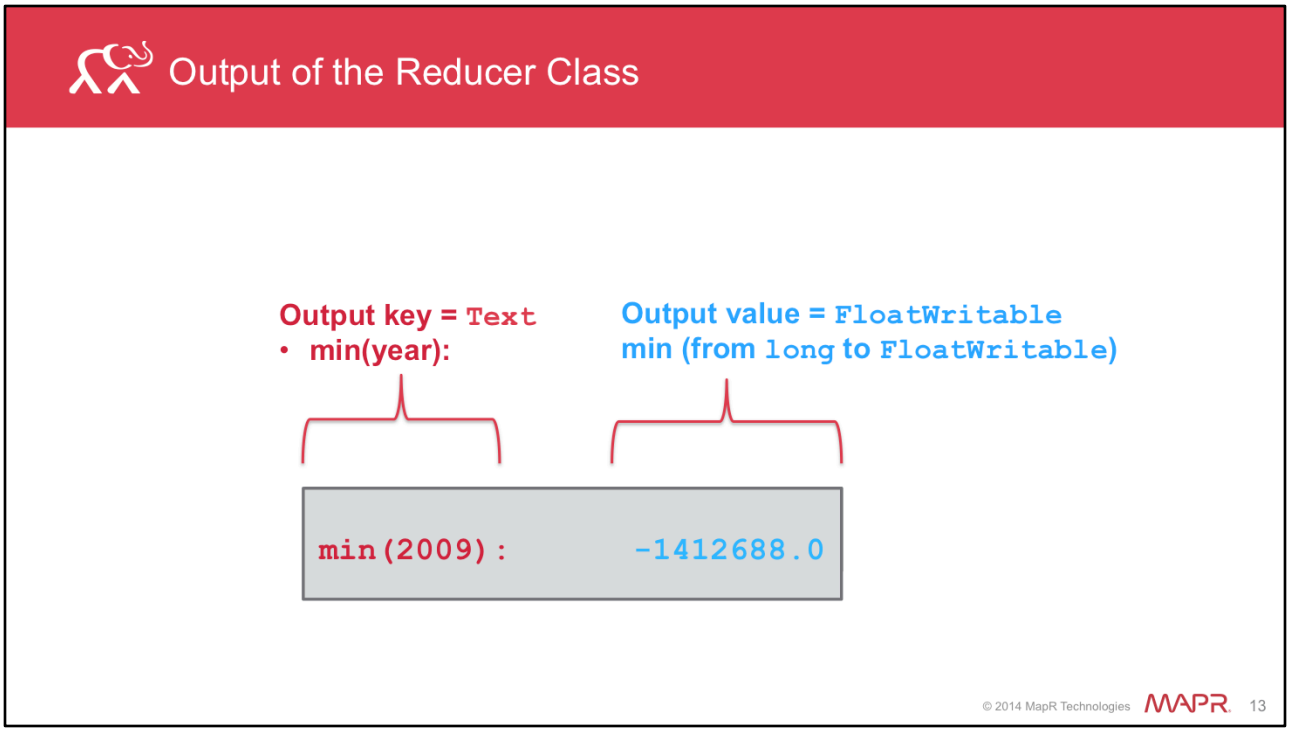
1. The four arguments to the Mapper class represent input key type, input value type, output key type, and output value type.
2. The mapper class calls the map() method
3. The first two arguments to the map() method are the key and value which must match the types defined in the Mapper class definition
4. All of the above



Now let's design and implement the reducer class for our simple MapReduce program.

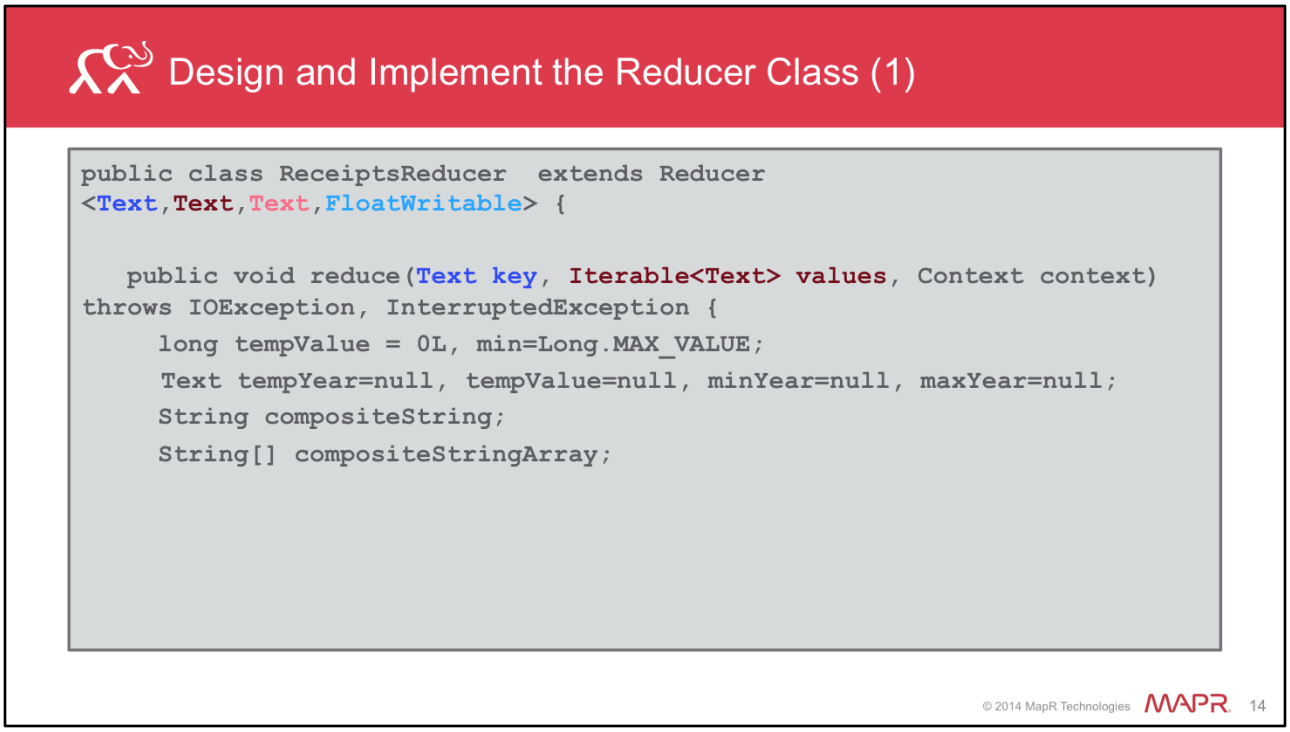


Let's look at the input for the Reducer class. Recall that the output of the Mapper must match the input to the Reducer (both key and value types). Since the output key from the Mapper class is Text, the input key to the Reducer class must also be Text. Likewise, since the output value from the Mapper class is Text, the input value to the Reducer class must also be Text. Note there is a distinction between what is output from a single map() call and the whole set of intermediate results that the all the calls to map() produces. Specifically, the output of a single map() call is a single key-value pair. The Hadoop infrastructure performs a sort and merge operation on all those key-value pairs to produce a set of one or more partitions. When a call to reduce is made, it is made with all the values for a given key.



The output of the reducer class must both conform to our solution requirements as well as match up to the data types specified in the source code. Recall that we defined both the output key and output value type for the Reducer as Text. In the output shown here, the minimum delta was reported in the year 2009 as -1412688.

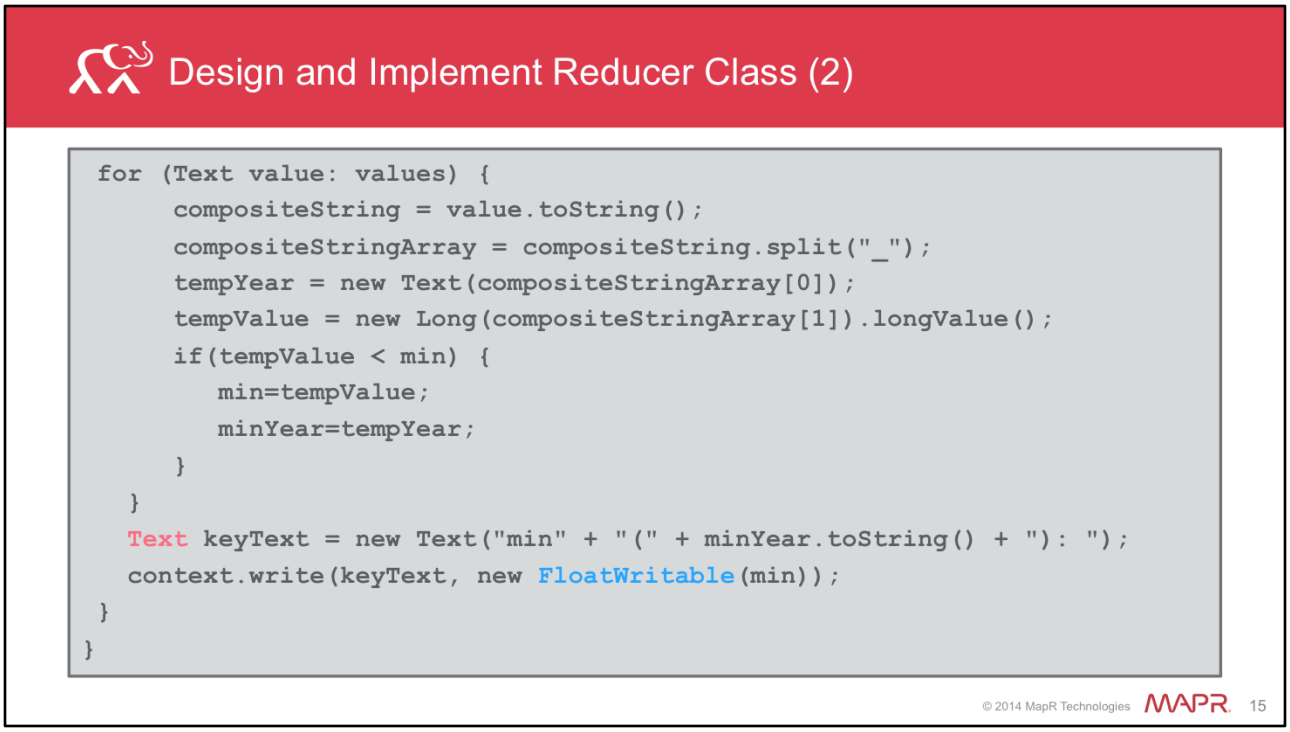
Note we are using FloatWritable as the output value type here because you will be calculating the mean value of the delta (a rational number) in the exercise associated with this lesson.



Note in this example, code omits import statements in order to conserve space.

The four arguments to the Reducer class represent the type for input key, input value, output key, and output value. Recall that the output key and output value from the mapper must match the reducer's input key and input value by type.

The first two arguments to the reduce() method are the key and value, which match in type from the class definition. The third argument is the context which encapsulates the Hadoop job running context (configuration, status reporter, and output committer). As in the Mapper class, you can dereference those objects if you wish.



We iterate over the all values associated with the key in the for loop.

We convert the Text value to a string (compositeString) so we can split out the year from the value (delta) for that year. We then convert that string into a string array (compositeStringArray) which splits out the compositeString variable based on the "_" character.

We pull out the year from the 0^{th} element of the string array, and then we pull out the value as the "1th" element of the array.

We determine if we've found a global minimum delta, and if so, assign the min and minYear accordingly.

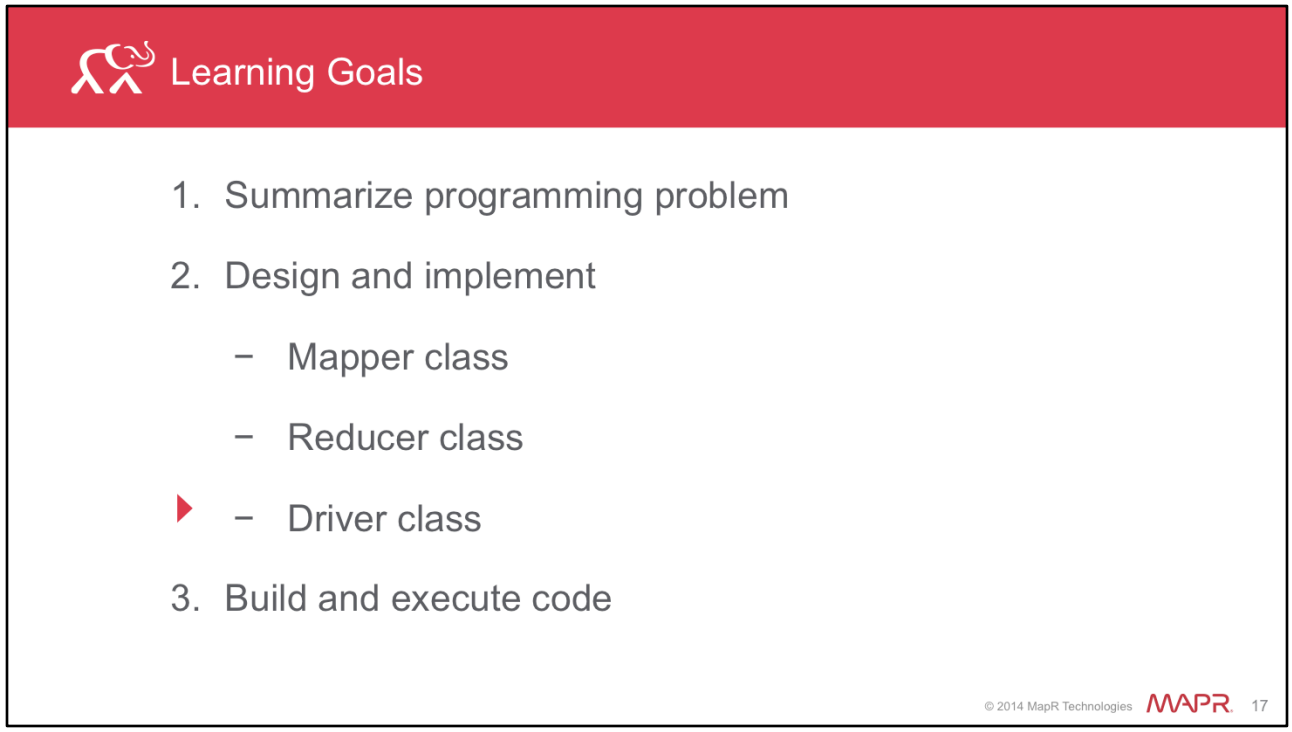
When we pop out of the loop, we have the global min delta and the year associated with the min. We emit the year and min delta.

Note that if the data set is partitioned into more than one partition, then we will have multiple output files, each with its "local" minimum calculated. In that case, we would need to do further processing to calculate the global minimum over the whole data set. It's for this reason we hardcoded the key – to guarantee we only have one partition and therefore one reducer.

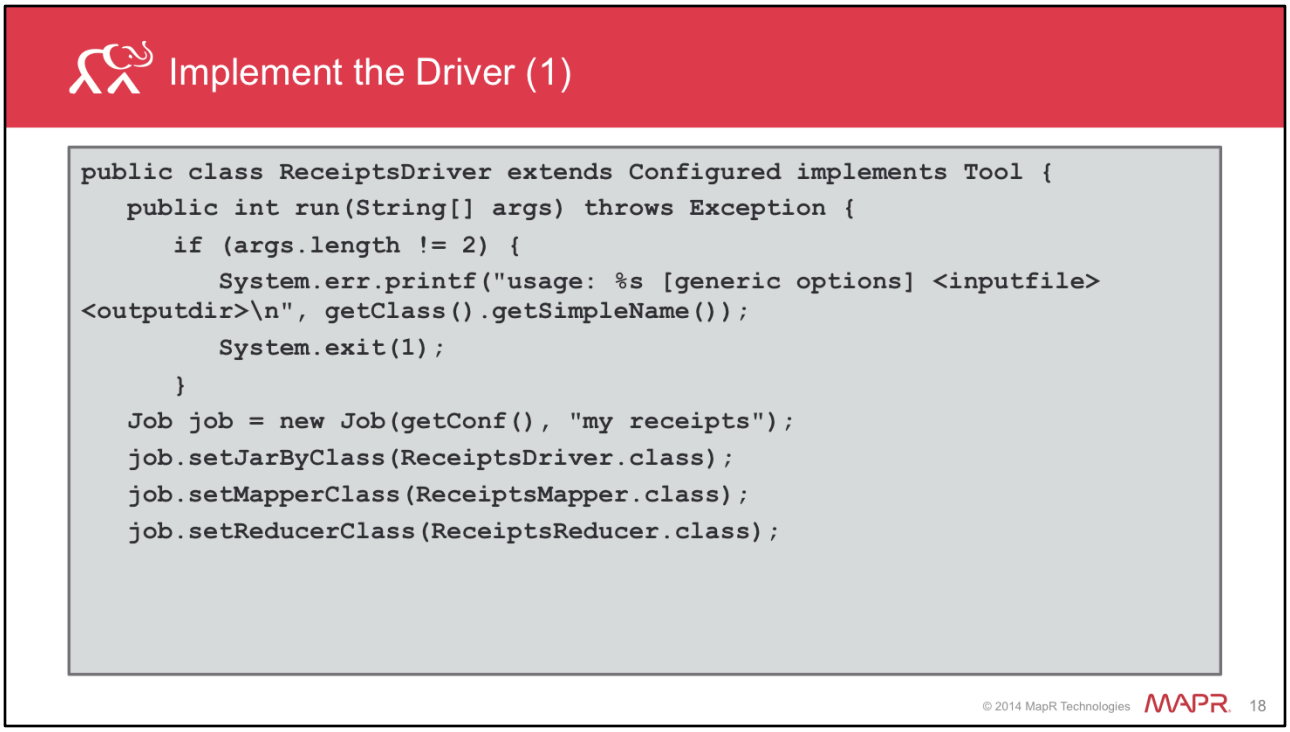
Knowledge Check

Which statements are true of the Reducer class?.

1. The four arguments to the Reducer class represent the type for input key, input value, output key, and output value
2. If the output value of the Mapper is Text, the input value to the Reducer can be LongWritable
3. If the output key of the Mapper is Text, the input key to the Reducer class must also be Text.
4. All of the above
5. 1 & 3 only



Now let's design and implement the driver for our simple MapReduce program.

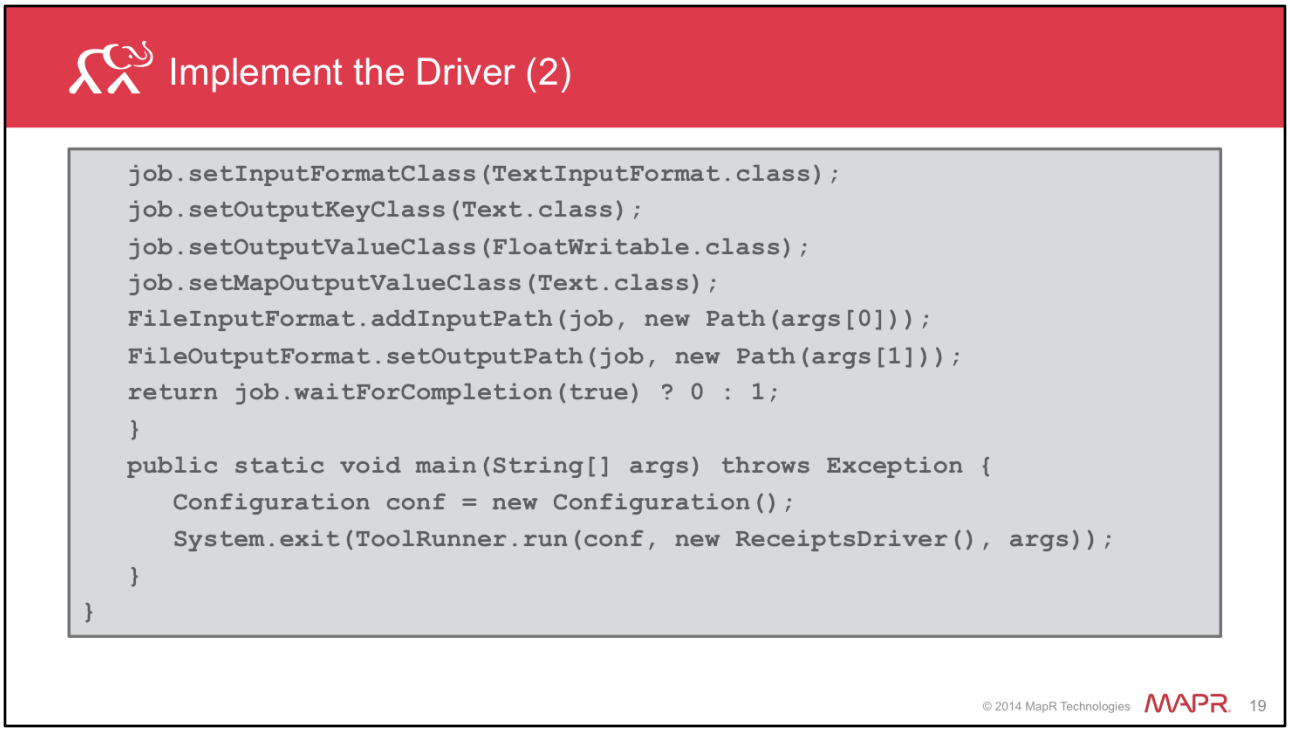


Let's look at the first part of the driver class.

It first checks the count of the command-line arguments provided and prints a usage statement if the argument count is not 2. Note that we don't check for the existence of the input file or output directory. The Hadoop framework will fail if either the input file doesn't exist or the output directory exists and is non-empty.

After checking the invocation of the command, the code instantiates a new Job object with the existing configuration and names the job "my receipts".

The rest of the code above sets values for the job, including the driver, mapper, and reducer classes used.



Now let's look at the second part of the driver implementation as shown here.

We define the types for output key and value in the job as Text and FloatWritable respectively. If the mapper and reducer classes do NOT use the same output key and value types, we must specify for the mapper. In this case, the output value type of the mapper is Text, while the output value type of the reducer is FloatWritable.

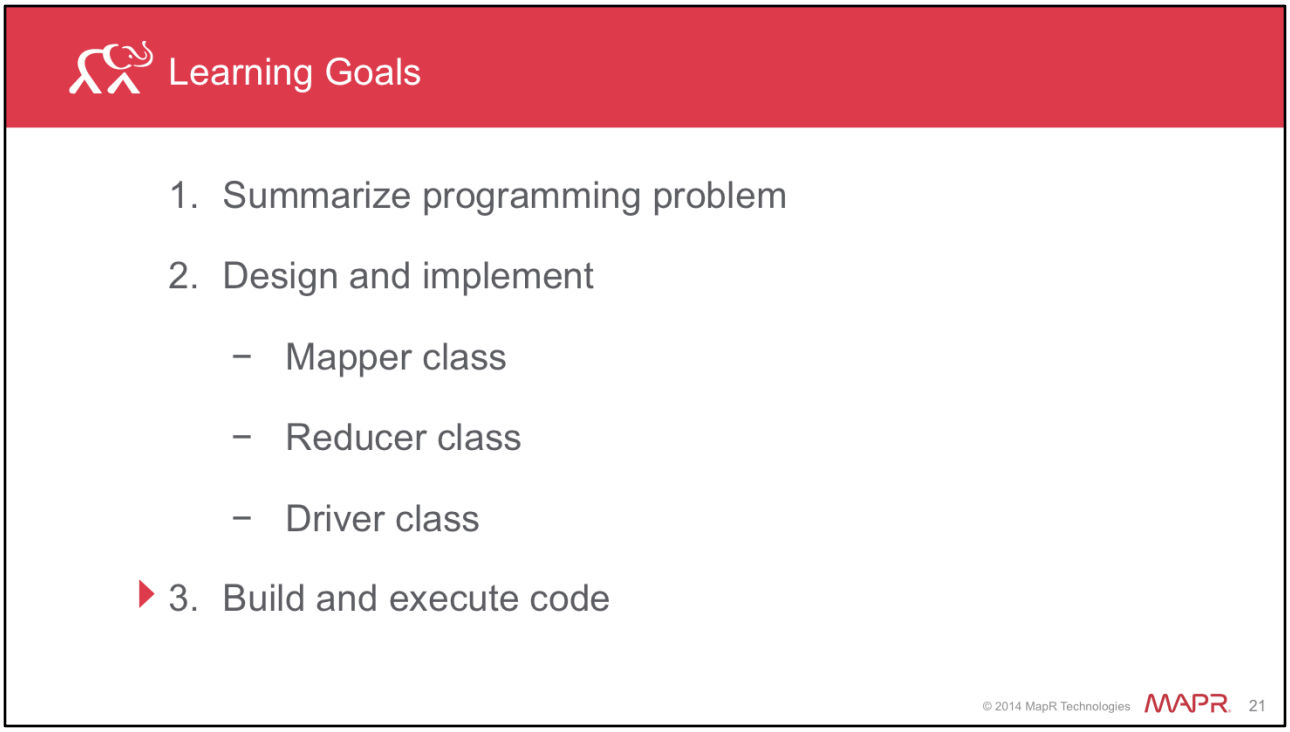
There are 2 ways to launch the job – syncronously and asyncronously. The job.waitForCompletion() launches the job syncronously. The driver code will block waiting for the job to complete at this line. The true argument informs the framework to write verbose output to the controlling terminal of the job.

The main() method is the entry point for the driver. In it, we instantiate a new Configuration object for the job. We then call the ToolRunner static run() method.

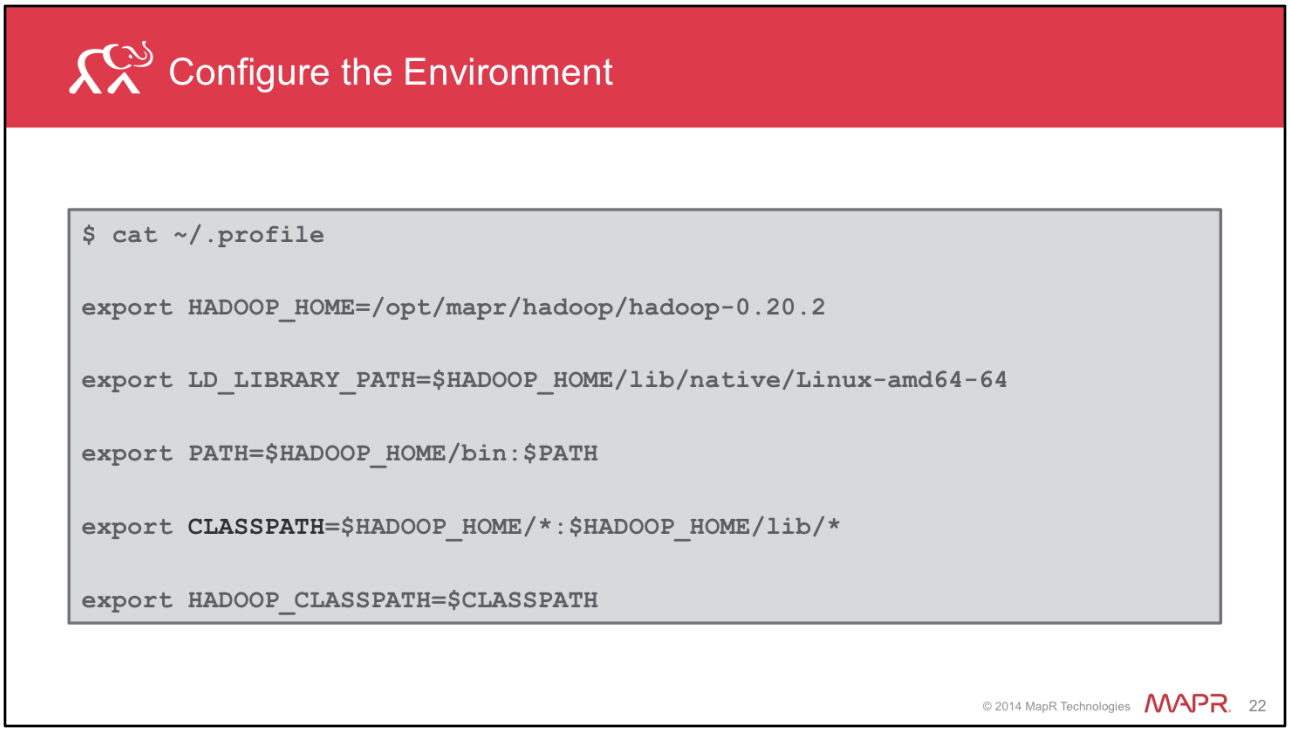
Knowledge Check

Which statements are true of the Driver class?

1. The Driver class first checks the invocation of the command (checks the count of the command-line arguments provided)
2. It sets values for the job, including the driver, mapper, and reducer classes used.
3. In the Driver class, we can specify how we want to launch the job – synchronously or asychronously
4. All of the above



Now let's learn how to build and execute our simple MapReduce program.



This example assumes you are a bash user for which the .profile file is the initialization file. If you use another shell, there is a different shell initialization file.

You are not obliged to define the HADOOP_HOME environment variable, but setting it as shown here allows you to reference the value of the HADOOP_HOME variable when defining other variables.

The LD_LIBRARY_PATH environment variable defines the path to your library files for executables. You are not obliged to define the LD_LIBRARY_PATH environment variable when running Hadoop jobs on a MapR cluster, but setting it as shown here uses libraries that are specifically compiled for the MapR distribution. Using Hadoop native libraries improves the performance of your MapReduce jobs by using compiled object code rather than Java byte codes.

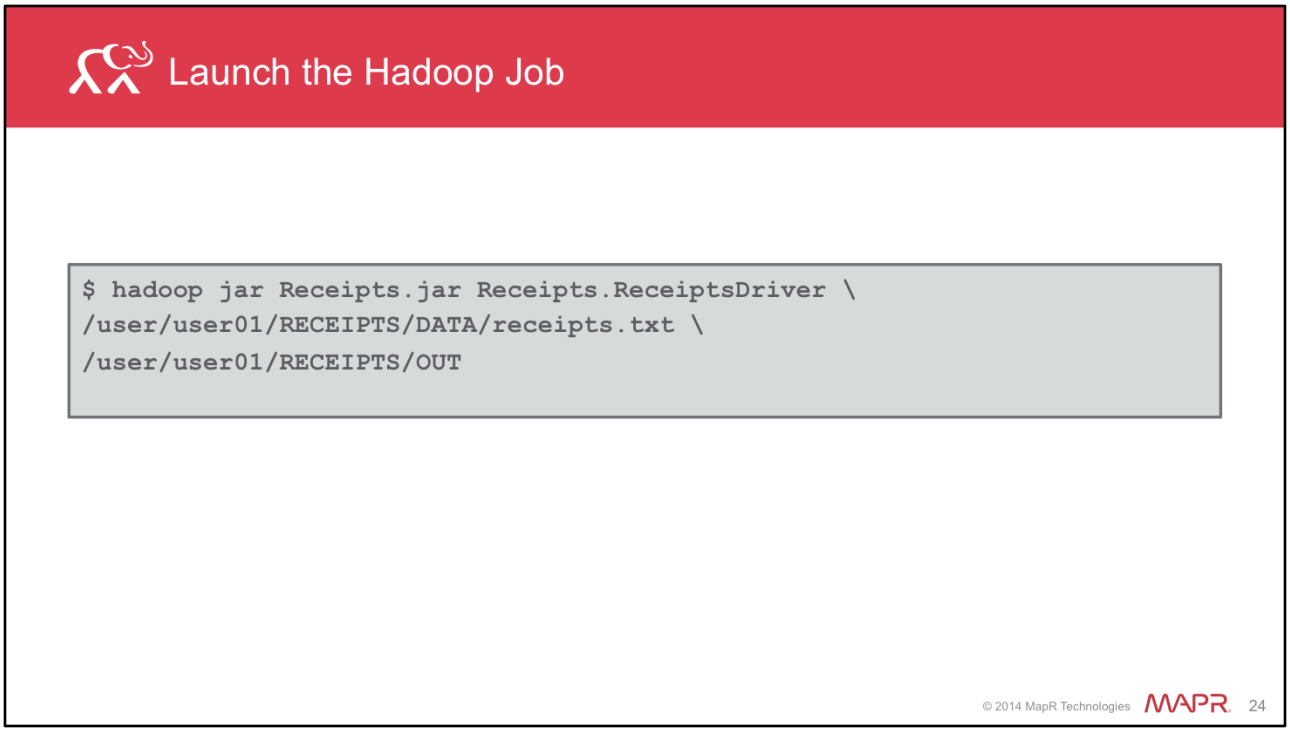
The PATH environment variable defines a list of directories where your executables are located. You are not obliged to define the PATH variable, but setting it as shown in the slide above allows you to reference commands directly without providing the absolute path to the executable. The order in the PATH environment variable is important. If you wish to execute the hadoop command from the HADOOP_HOME (and not the one installed in your default PATH), you must put $HADOOP_HOME/bin first in your PATH statement.

You could optionally define your CLASSPATH environment variable in your shell environment to point to all the jars in the Hadoop distribution required to compile and run your MapReduce programs. If you don't specify the CLASSPATH variable in your shell environment, then you'll need to specify it at compjile time and run time. Similarly, you can optionally define the HADOOP_CLASSPATH variable to make it easier to run MapReduce applications with the hadoop command.

This example shows the commands used to compile our 3 classes. First we'll create the directory "classes" to contain our class files.

The javac commands compile the mapper and reducer classes, and puts the compiled class files into a directory called "classes".

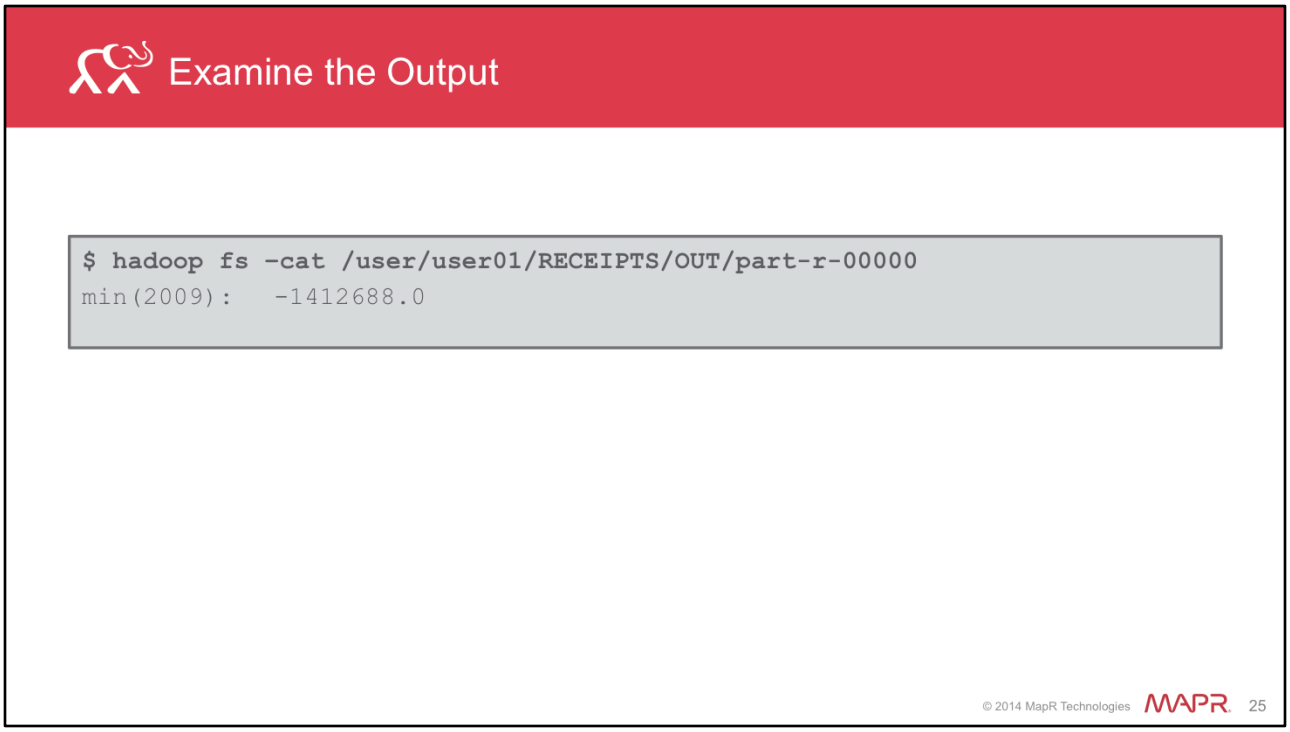
The jar command puts the mapper and reducer classes into a jar file. We include the new jar file in the classpath when we build the driver, after which we add the driver class to the existing jar file.



The hadoop command launches the Hadoop job for our MapReduce example.

The hadoop command will use the "Receipts.jar" file we just created which contains the driver, mapper, and reducer classes.

The input to the MapReduce program is the first argument (/user/user01/RECEIPTS/DATA/receipts.txt) and the output to the MapReduce program is the second argument (/user/user01//RECEIPTS/OUT). These are the values associated with args[0] and args[1] in the driver, respectively.



This output file was created by our Reducer. It contains the statistics that our solution asked for – minimum delta and the year it occurred.