**Understanding Data Transformations**

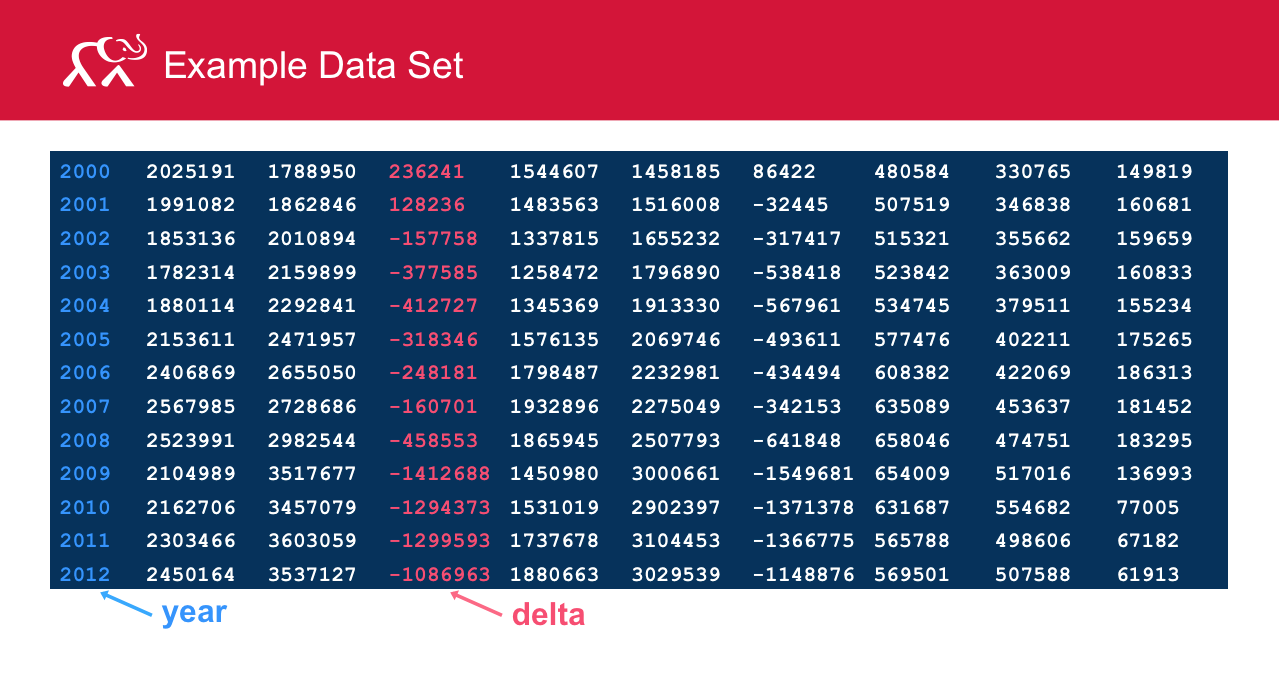
In order to write MapReduce applications you need to have an understanding of how data is transformed as it executes in the MapReduce framework.

From start to finish, there are four fundamental transformations. Data is:

1. Transformed from the input files and fed into the mappers
2. Transformed by the mappers
3. Sorted, merged, and presented to the reducer
4. Transform by reducers and written to output files

**Tip:** It is important to use the appropriate type for your keys and values. You must ensure that your input and output types match up, or your MapReduce code will not work. Since the input and output formats are derived from the same base class, your code may compile well but then fail at runtime.

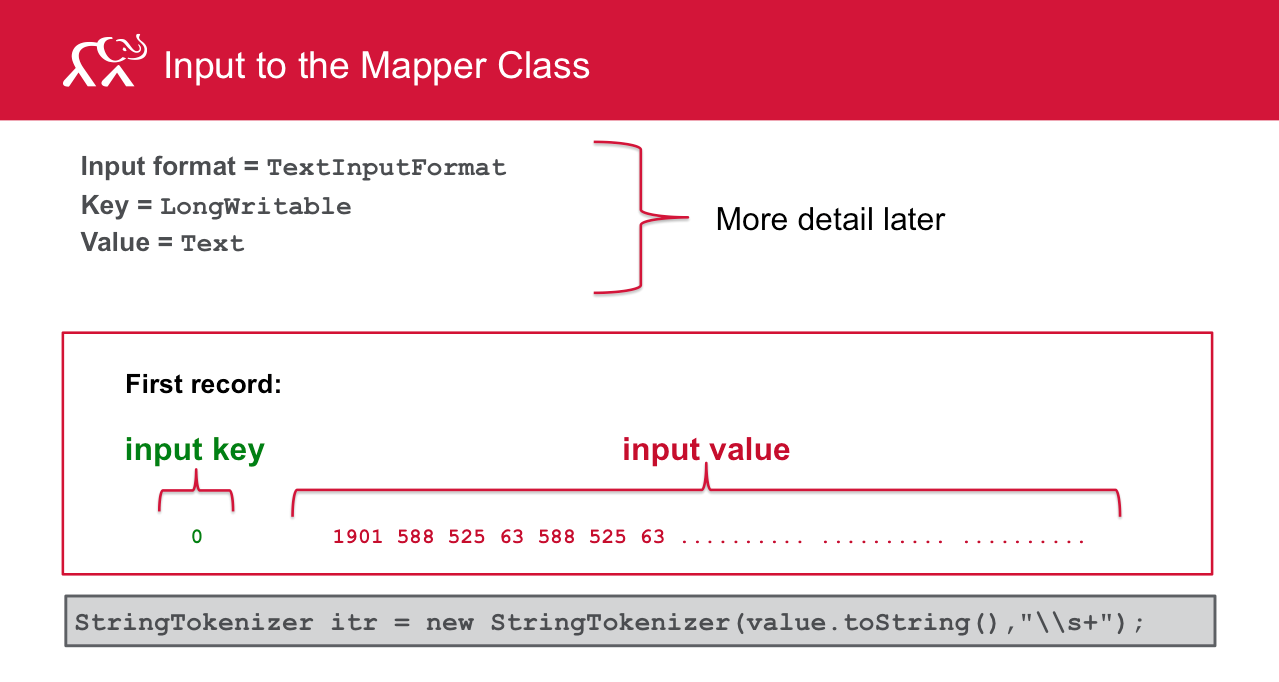
**Solving a Programming Problem using MapReduce**



The above image shows a data set that is the basis for our programming exercise example. There are a total of 10 fields of information in each line. Our programming objective uses only the first and fourth fields, which are arbitrarily called "year" and "delta" respectively. We will ignore all the other fields of data.

We will need to define a Mapper class, Reducer class and a Driver class.

**Designing and Implementing the Mapper Class**



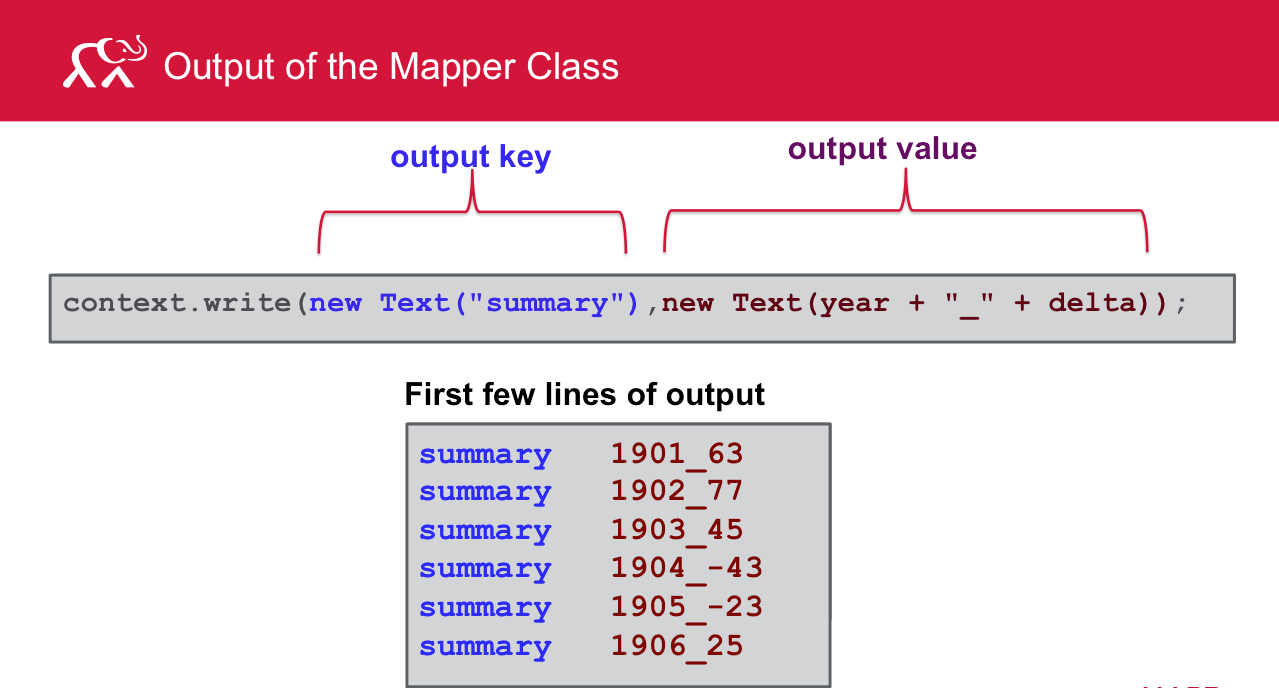
Let us define and implement the Mapper class to solve the programming problem.

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.

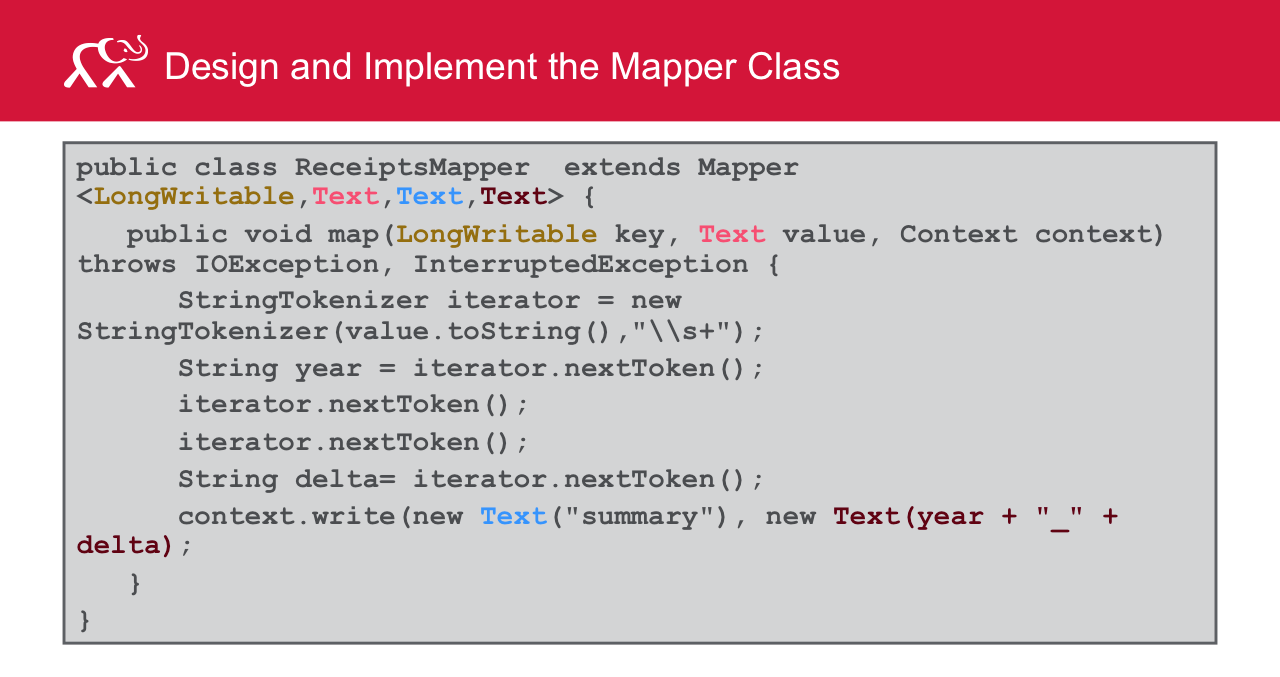
The key of the first record is the byte offset to the line in the input file (the 0th byte). The value of the first record includes the year, number of receipts, outlays, and the delta (receipts – outlays).

Remember – 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.

Here we construct the StringTokenizer using white space as the delimiter.

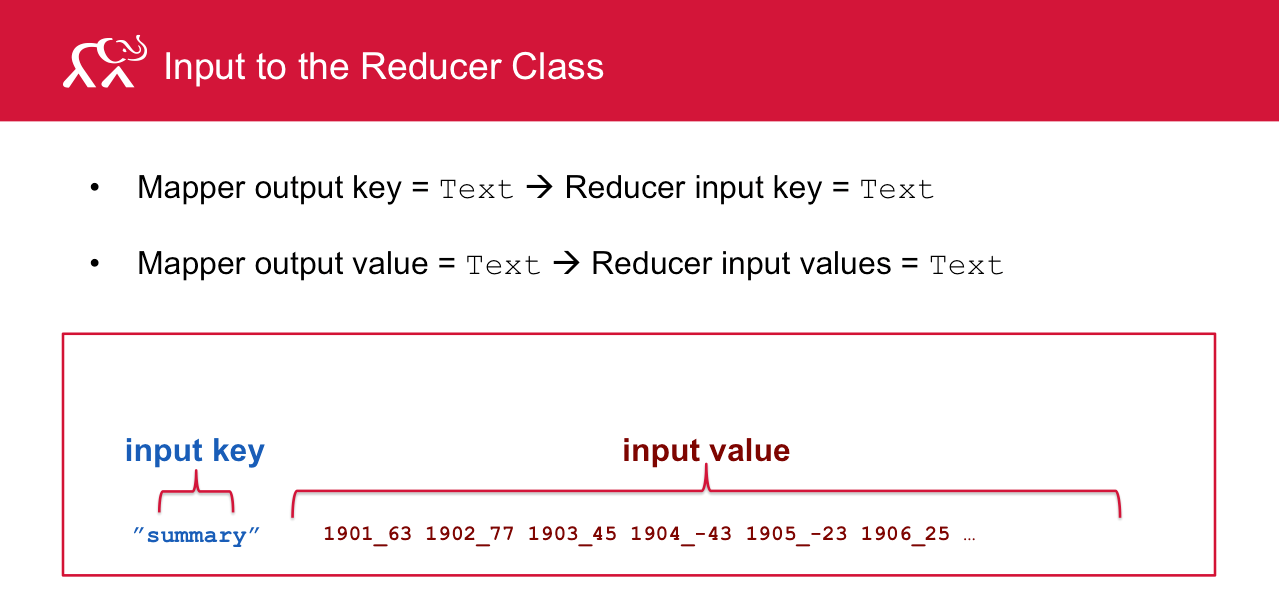


After grabbing fields 1 and 4 from a record, the Mapper class emits the constant key (“summary”) and a composite output value of the year (field 1) followed by an underscore followed by the delta (field 4).

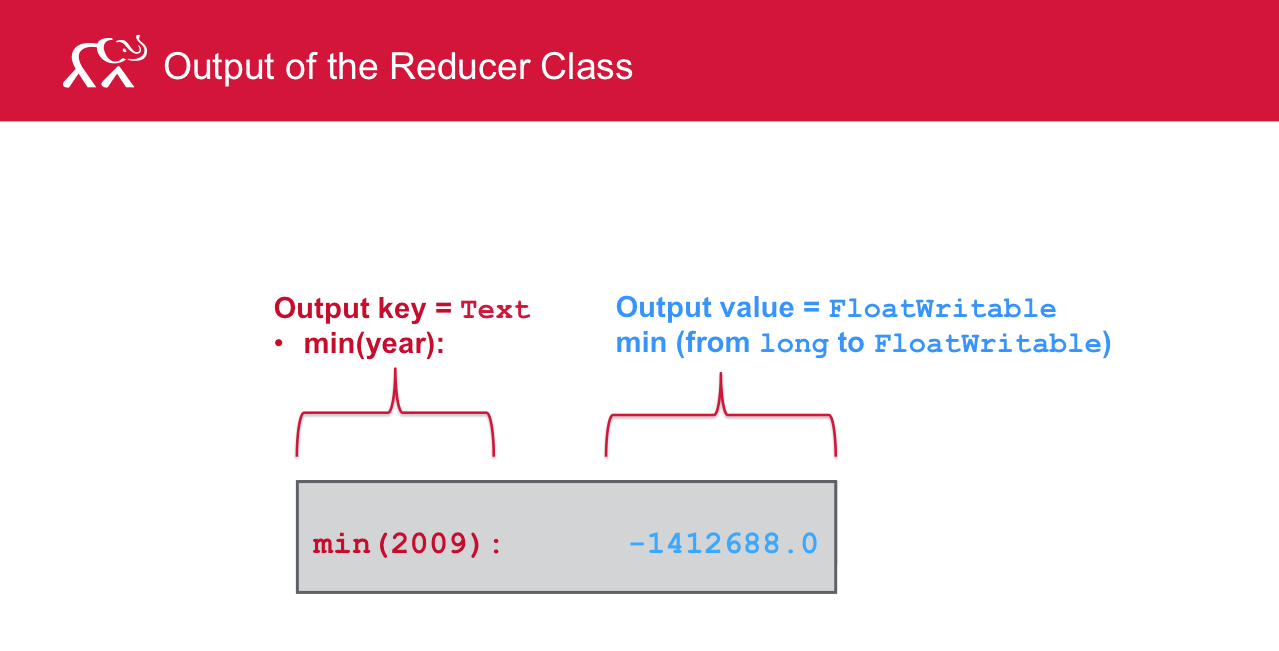


Since we hard-coded the key to always be the string “summary,” there will be only one partition (and therefore only one reducer) when this mapreduce program is launched.

**Designing and Implementing the Reducer Class**

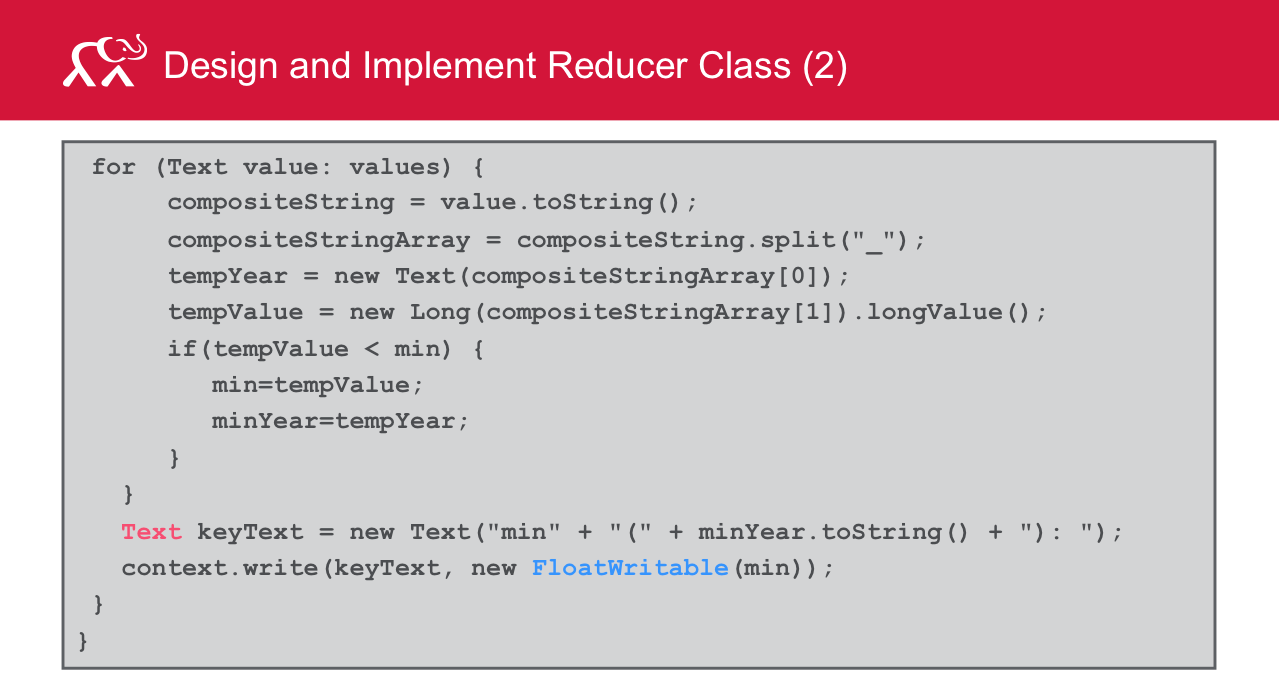


Let’s look at the input for the Reducer class. Recall that the output of the Mapper must match the input to the Reducer (both key and value types). Since the output key from the Mapper class is Text, the input key to the Reducer class must also be Text. Likewise, since the output value from the Mapper 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 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 the output key as Text. 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.

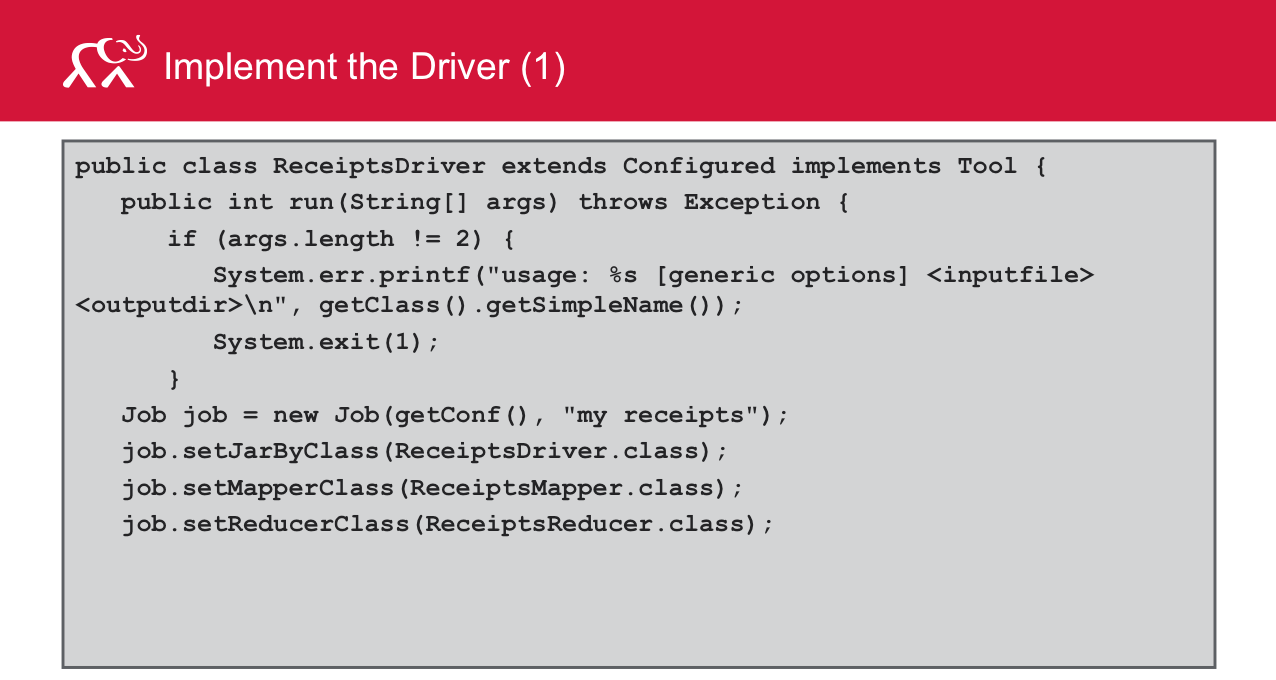
In the output shown here, the minimum delta was reported in the year 2009 as -1412688.



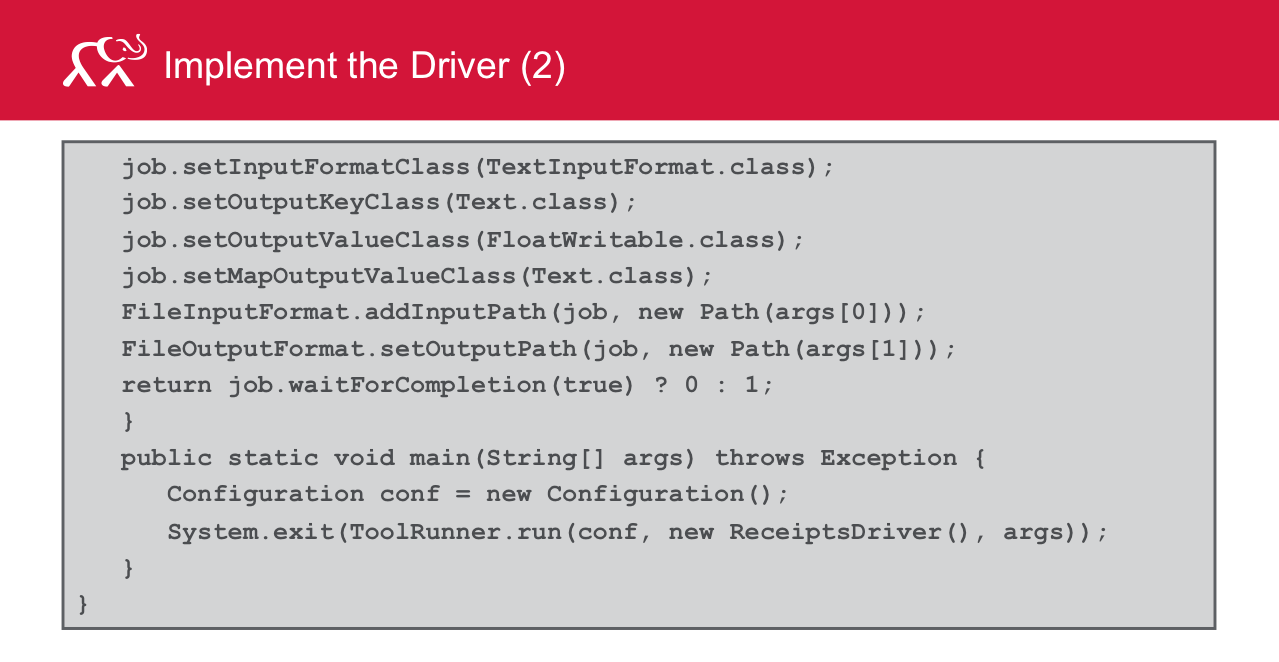
In the Reducer code,

* 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 0th 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.

**Design and Implement The Driver**



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.



1. In the Driver class, we also 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.
2. 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 this method, we instantiate a new Configuration object for the job. We then call the ToolRunner static run() method.

**Build and Execute a Simple MapReduce Program**

You have to compile the three classes and place the compiled classes into a directory called “classes”. Use the jar command to put the mapper and reducer classes into a jar file the path to which is included in the classpath when you build the driver. After you build the driver, the driver class is also added to the existing jar file.

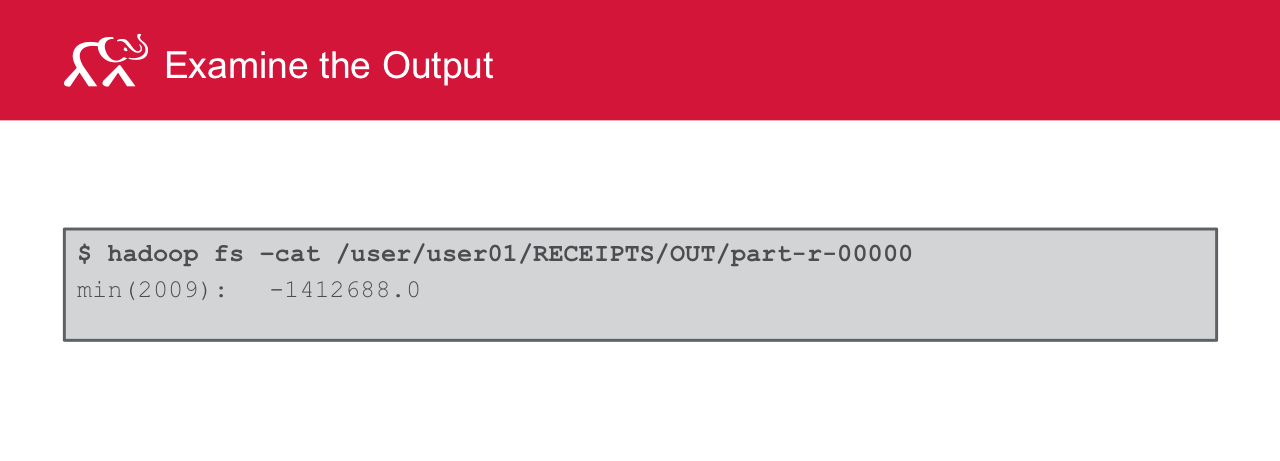
Make sure that you delete the reduce output directory before you execute the MapReduce program.



Use the hadoop command to launch the Hadoop job for the MapReduce example.



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



**Notes on the Data Used Here**

The data set we used for this example is publicly available from the US federal government. It includes a set of spreadsheets that summarize the federal budget (including income and expenses) from 1793 to projected 2018.

The data as provided by the US government is structured data with a well-defined schema. The set we are using can be downloaded as a ZIP file of XLS spreadsheets from the following URL: <http://www.whitehouse.gov/omb/budget/Historicals>. The actual file is hist01z1.xls, titled “Summary of Receipts and Outlays, Surpluses and Deficits 1789-2018.” It can be downloaded directly from <http://www.whitehouse.gov/sites/default/files/omb/budget/fy2014/assets/hist01z1.xls>

For the purpose of this exercise, the data was stripped down, cleansed, and converted. We did this as follows:

* Convert the file from XLS to a text file that is usable on the Linux operating system. In this case, the operating system used to save the file in Unicode only supports UTF-16, and the Linux cluster on which the job will run only supports UTF-8. So here we save as UTF-16 on the desktop, copy that file to the cluster, and then use the iconv(1)utility to convert the file from UTF-16 to UTF-8.
* When the file format is readable by the cluster operating system, we need to remove records that our MapReduce program will not know how to digest. Specifically, we remove the headers and footers that describe the data, remove records that do not have a single year in the first column, and remove records that don’t have numerical values in the surplus or deficit field.