LAB 1 NOTES: JAVA AND HADOOP ENCODING OF MAPREDUCE FUNCTIONS

BIG DATA PROCESSING

Félix Cuadrado

felix.cuadrado@qmul.ac.uk

Queen Mary University of London

School of Electronic Engineering and Computer Science

Mapper code

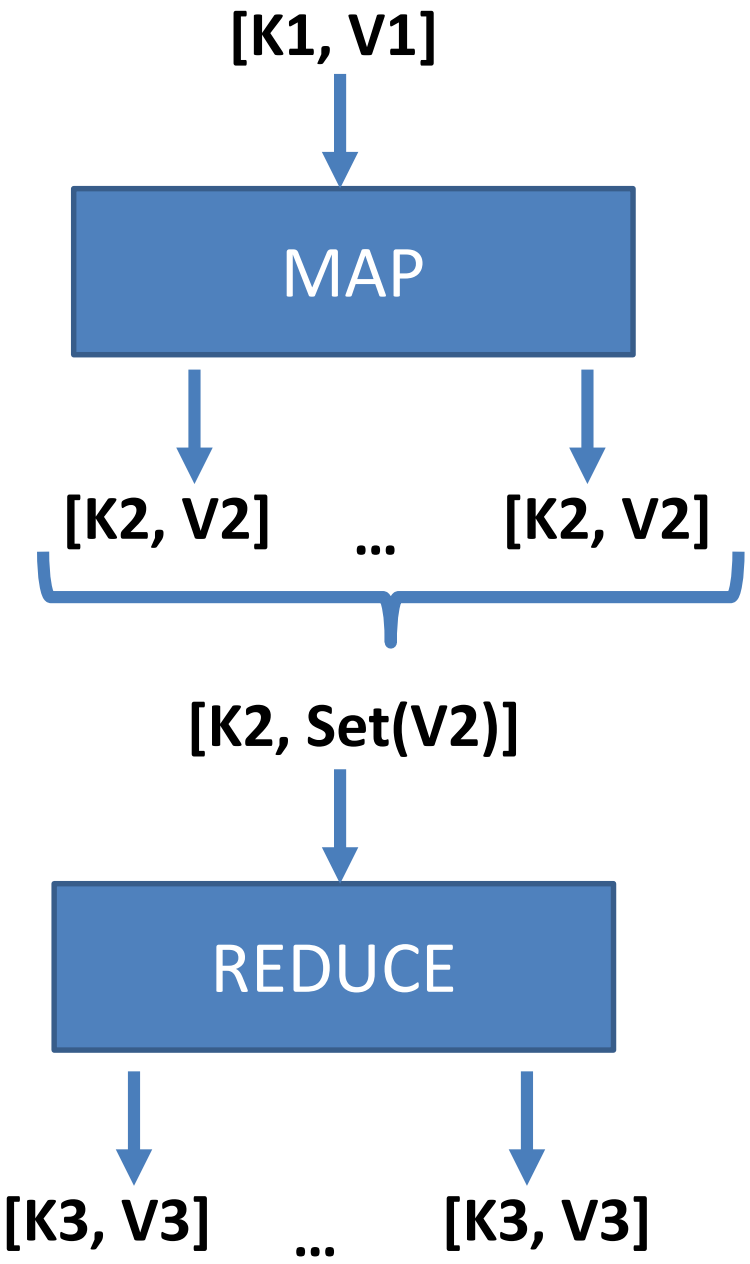
```
1. public class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
2.     private final IntWritable one = new IntWritable(1);
       private Text data = new Text();
3.     public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
         StringTokenizer itr = new StringTokenizer(value.toString(), "-- \t\n\r\f,.:;?![]'\"");
         while (itr.hasMoreTokens()) {
           data.set(itr.nextToken().toLowerCase());
           context.write(data, one);
         }
       }
   }
```

Reduce code

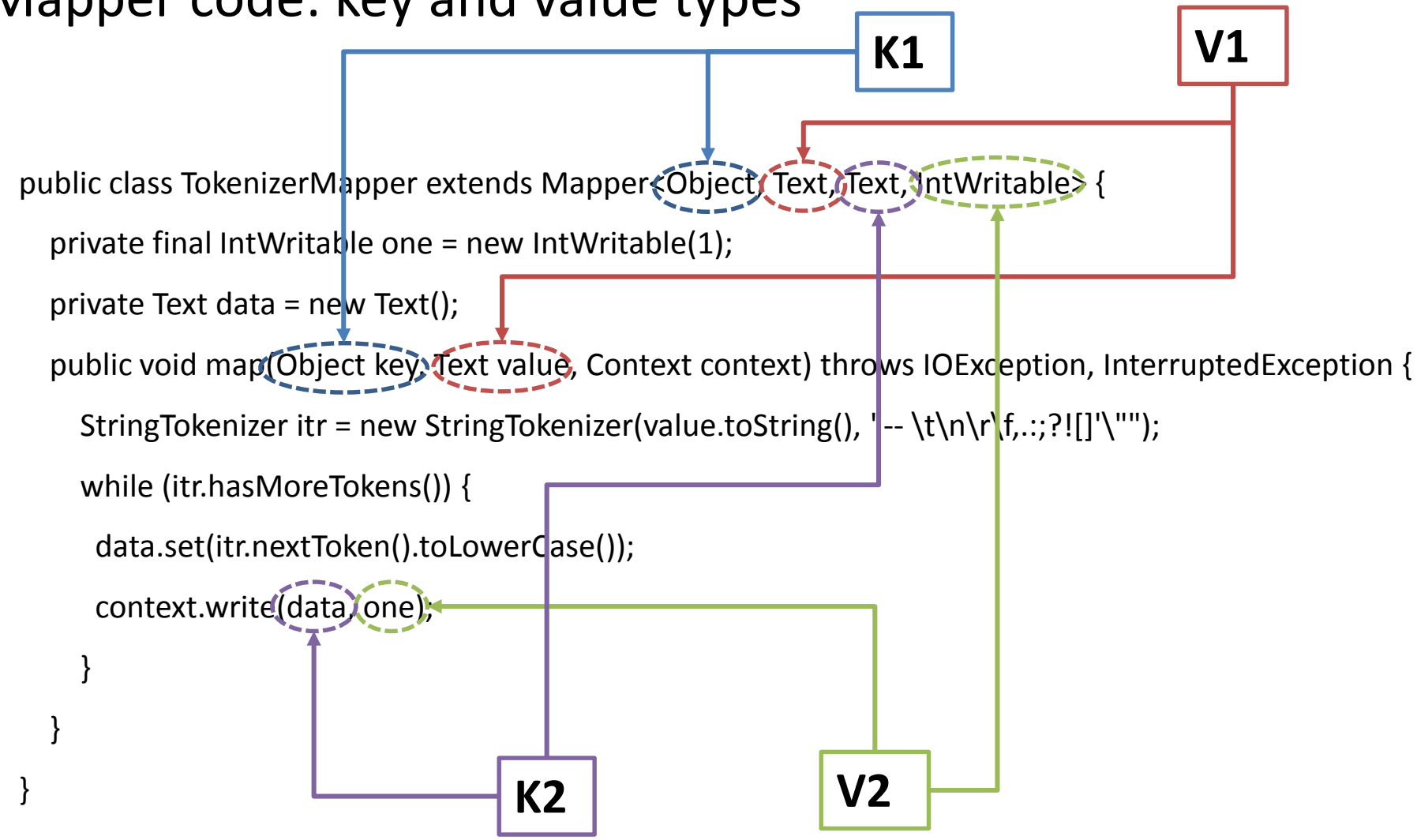
```
1. public class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
2.     private IntWritable result = new IntWritable();
3.     public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int sum =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

1. Declaring Writable types for Map and Reduce functions

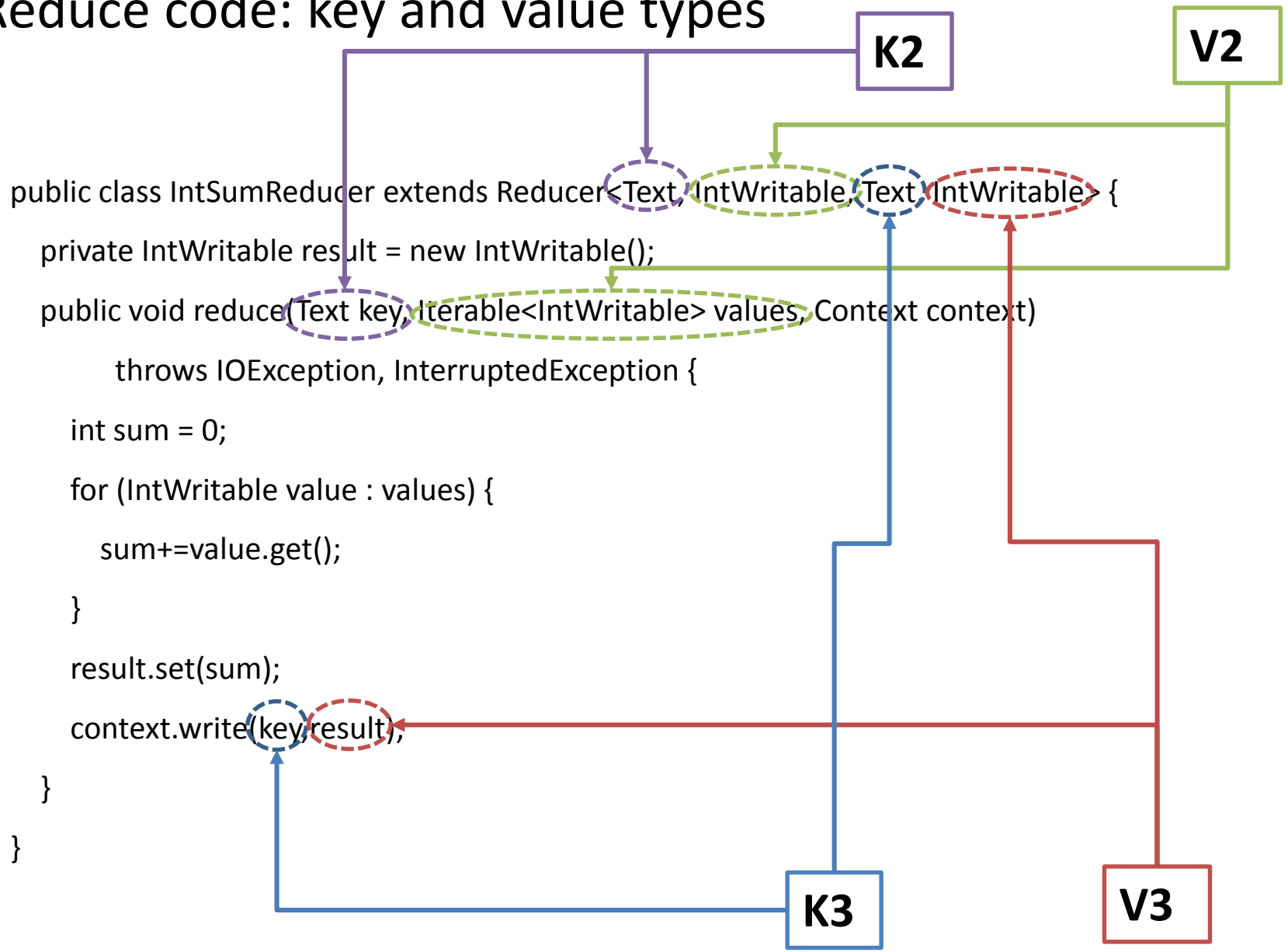
- Hadoop code is embedded into Java classes. A Java class has the following parts: Class declaration signature (first line), attributes (internal variables of the class), and methods (functions that can be invoked by objects of the class.
  - Note the color code is the same of the three highlighted parts in the Mapper code
- Map/Reduce is a functional programming paradigm, but being written in Java forces it to be shoehorned into the Java class structure
  - This is done the following way: The Map function is declared as a method of a class that extends the type Mapper<k1,v1,k2,v2>. You can pick any class name (this one is called TokenizerMapper, but the name itself is meaningless)
  - Analogously, the Reduce function is declared as a method of a class that extends the type Reducer<k2,v2,k3,v3>
  - The <> symbols enclose four types, determining the writable types for the input/output keys and values generated at the Mapper/Reducer
  - These types have to be consistent in the rest of the Java class: the map method has two initial parameters that provide the k1 and v1 input pair. Types need to be consistent with the ones declared in the first line of the class. For the Reducer, input types will be k2, v2.
  - Also, when invoking context.write(), the two provided variables must have types that match k2,v2 in the case of the Mapper (and k3,v3 for the Reducer).


[K1, V1]
MAP
[K2, V2]
...
[K2, V2]
[K2, Set(V2)]
REDUCE
[K3, V3]
...
[K3, V3]

Mapper code: key and value types



Reduce code: key and value types



2. Taking advantage of Java when writing Hadoop jobs

- The blue section of the Mapper identifies the declared class attributes. These are variables that exist in the context of all methods invoked by the class (in this case, the map function). They can be accessed inside the Map function, and for both the Mapper and the Reducer provided in this exercise, they hold the actual values that will be emitted using the context.write method.
- The reason why these variables are declared as class attributes, instead of regular variables of the method has to do with performance: When Hadoop invokes a Mapper, it creates an instances of the class (TokenizerMapper in this case), and reuses it thousands of times, invoking each of them the map method we have implemented. By declaring "one" and "data" as class attributes rather than method variables, these two objects are only created once; the first time the class object is created. Class creation in Java is a costly operation, and when repeated thousands of times the overhead would add up. Because of that, this code avoids having to create these objects every time the map function is invoked.
- The code would be functionally correct if we moved both inside the method declaration, but performance would suffer a slight penalty.

3. Implementation of the Map function

- **Parsing the input**: The key/value pairs that are being received by this Map function are in this case a LongWritable with the bytes offset for this specific input (how far within the data block we are). We ignore the k1 element for counting words, and here it is referred as a generic Object that we do not use later on. The value v1 is a Text Writable, which holds a String of text with exactly one line of information. Having this specific combination of k1 and v1 is because if we do nto specify otherwise, Hadoop uses a TextInputFormat to parse the input blocks.
  https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/TextInputFormat.html

- **Process the input:** In order to count words we need to split each one of these lines of text into individual words. In order to do so, we use the Java functionality of the Stringtokenizer class. We configure the tokenizer with two arguments: the String we want to split into parts (in our case extracted from the input value, using toString() to get the String value stored inside the Text Writable), and as second argument a list of symbols that are going to be word delimiters for us. If we do not specify any delimiter the method will only use regular spaces. By adding commas, colons and additional punctuation symbols we aim to split the words without any additional symbol.
- In order to retrieve the tokens (words) that are detected by the tokenizer we have to use a while loop. The method nextToken() returns a String with one of the words found in the input String, and hasMoreTokens() returns true if there are more tokens left (as each token is only returned once).
- http://docs.oracle.com/javase/7/docs/api/java/util/StringTokenizer.html
- **Emitting k2,v2 pairs:** The Mapper and the Reducer use the additional input object Context to emit key value pairs. The method context.write(k2,v2) sends one pair of values from the Mapper to shuffle and sort, or from the Reducer to the output. Both k2, and v2 objects must be of Writable types, which usually requires to invoke the method set() of a previously created Writable with the actual value we want to send across the network.

3. Implementation of the Reduce function

- **Parsing the input**: The reducer receives two input parameters: a key (in this case a Text object), and a collection of all the values that have been emitted by all the Mappers under that key (here Iterable<IntWritable> values. The Iterable collection is one Java object that holds multiple elements of the same type (IntWritable) in our case. The easiest way to go over each of these elements is by using a for each loop: for (IntWritable value : values) , which will execute once for each individual value inside the collection.
- **Emitting k3,v3 pairs:** The Mapper and the Reducer use the additional input object Context to emit key value pairs. The method context.write(k3,v3) sends one pair of values from the Reducer to to the output. Both k3, and v3 objects must be of Writable types, which usually requires to invoke the method set() of a previously created Writable with the actual value we want to send across the network. That is the reason why this code has an internal int variable sum, plus an actual IntWritable result, which is only used at the end to hold the total sum, and be emitted through context.write()