

220

# Functional Ideas in Java 8

Dr Robert Chatley - [rbc@imperial.ac.uk](mailto:rbc@imperial.ac.uk)



@rchatley #doc220

In the last lecture we looked at how we could create abstractions in Java that allowed us to package up computation into an object, and separate that from the logic that performed the traversal of a data structure. In Java 8, these have been introduced into the language and standard libraries - here we see how.

## Streams

Stream.of(...)  
Arrays.asList(...).stream()

Create

stream.map(...)  
stream.filter(...)

Transform

stream.forEach(...)

Consume

stream.reduce(...)

Reduce

#doc220

Java 8 introduces the abstraction of Streams, which are often linked to the collections libraries, but allow us to do processing of data in batches. They are pretty powerful, and follow a functional style. The Stream library provides a lot of functionality, but we see here how to create, transform, consume, and reduce Streams.

## Lambdas

```
Function<Integer, Integer> square = new Function<Integer, Integer>() {  
    @Override  
    public Integer apply(Integer n) {  
        return n * n;  
    }  
};
```



```
Function<Integer, Integer> square = n -> n * n;
```

#doc220

Another major addition in Java 8 is the addition of lambdas (anonymous functions that are first class citizens) to the language. This means that we have a concise syntax for defining functions, and they can be passed as parameters to other functions.

# Method Refs

```
stream.forEach(System.out::println);
```

```
stream.forEach(this::square);
```

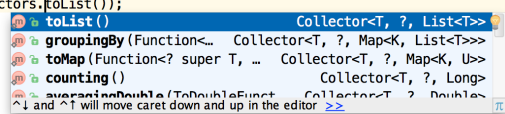
#doc220

We can also pass existing methods as parameters to other functions, using method references, notated with a double colon.

# Collectors

```
List<Integer> collected = stream.collect(Collectors.toList());
```

```
ct(Collectors.toList());
```



Lots of interesting things in the Collectors class

#doc220

To convert the contents of a stream back into a collection, we used a “terminal” such as a Collector. One of the most common things to do is to collect the contents of a stream into a List, so that it can be used with other code that works on collections. Do have a look at the Collectors class and the related APIs - there is a lot that you can do with these.

# Refactoring to Streams

```
public Collection<String> studentsWithAGrades(Collection<Submission> submissions) {  
    Collection<String> aGrades = new ArrayList<>();  
    for (Submission submission : submissions) {  
        if (submission.courseCode == 220 && submission.courseworkNumber == 3) {  
            if (submission.classCode.equals("c2")) {  
                if (submission.mark > 70 && submission.mark < 80) {  
                    aGrades.add(submission.login);  
                }  
            }  
        }  
    }  
    return aGrades;  
}
```

How would you refactor this code to use streams?

#doc220

Still, the vast majority of Java code in the world is written without these abstractions - as it comes from previous versions of Java. We can use small refactoring steps to gradually transform code to follow the new style.

For more examples, see <http://martinfowler.com/articles/refactoring-pipelines.html> and <https://github.com/dmccg/refactoring-to-streams>

220

# MapReduce

Dr Robert Chatley - [rbc@imperial.ac.uk](mailto:rbc@imperial.ac.uk)



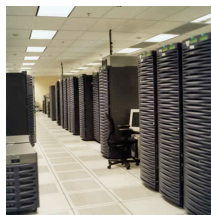
@rchatley #doc220

In the last lecture we looked at how we could create abstractions in Java that allowed us to package up computation into an object, and separate that from the logic that performed the traversal of a data structure. Although we might not use this approach all the time in day-to-day Java code, it can be used to solve some bigger computation problems. Today we look at how MapReduce works.

## MapReduce

MapReduce is a system that Google built to work with very large sets of data

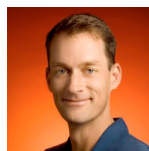
It builds on some of the techniques we have been discussing



#doc220

Although the concepts of mapping and reducing were not invented by Google - they have been in functional programming for a long time - Google's implementation of MapReduce for distributed computation over large data sets meant that the ideas have become more popular in software engineering recently. It also inspired other implementations such as Hadoop, which also uses the map-reduce technique.

Google wanted to index the entire world wide web



Jeff



Sanjay



#doc220

Google's MapReduce was implemented by two of their early engineers, Jeff Dean and Sanjay Ghemawat. They had a large problem to solve: they wanted to build an index of the web, which required processing a very large amount of data.

MapReduce is designed to work on large datasets,  
running in parallel across multiple machines

But at its simplest...

```
// compute sum of squares  
  
var square = function(x) { return x * x }  
var sum = function(x,y) { return x + y }  
[1,2,3,4].map(square).reduce(sum);
```

#doc220

This job couldn't possibly be run on one machine, so Dean and Ghemawat came up with a strategy for splitting the computation across many machines in parallel. They divided the computation into two parts, first applying a map function, and then reducing over the results to aggregate them.

[1,2,3,4]

map phase: square each number individually

[1,4,9,16]

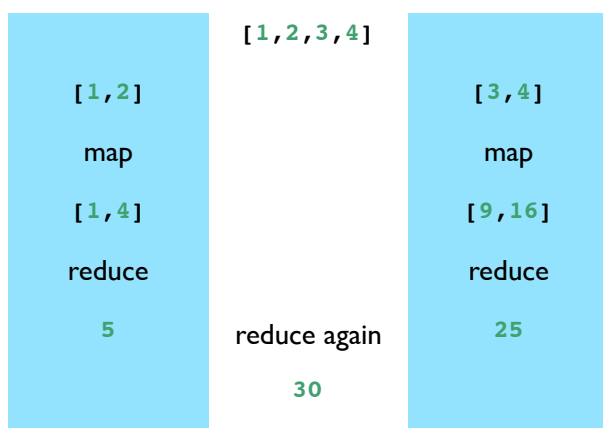
reduce phase: add up the numbers in the list

30

#doc220

So, in this example we want to find the sum of the squares of a set of numbers. First of all we map the square function across the input list, producing a list of squares. Then we reduce this this using the addition function to calculate the sum.

## Parallelisation



We can do this in parallel. Here we split the data in two, and map each of the smaller lists separately. Then we reduce each of the output lists, aggregate these results and in this case reduce again to produce a single result.

## Inputs and Outputs: Keys and Values

Rather than simple input values, MapReduce works with sets of keys and values

$\text{map}(k1, v1) \rightarrow \text{list}(k2, v2)$   
 $\text{reduce list}(k2, \text{list}(v2)) \rightarrow \text{list}(k2, v2)$

$k1$  (line num)  $v1$  (line of text)

```
1 I think that it's extraordinarily important that we in computer science keep fun in computing.  
2 When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every  
3 now and then, and after a while we began to take their complaints seriously. We began to feel as  
4 if we really were responsible for the successful, error-free perfect use of these machines. I don't  
5 think we are. I think we're responsible for stretching them, setting them off in new directions,  
6 and keeping fun in the house. I hope the field of computer science never loses its sense of fun
```

The actual implementation of MapReduce is slightly more complex to use, as it works with key-value pairs, but this makes in more powerful and more flexible.

## Example: Word Frequencies

$k1$  (line num)  $v1$  (line of text)

```
1 I think that it's extraordinarily important that we in computer science keep fun in computing.  
2 When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every  
3 now and then, and after a while we began to take their complaints seriously. We began to feel as  
4 if we really were responsible for the successful, error-free perfect use of these machines. I don't  
5 think we are. I think we're responsible for stretching them, setting them off in new directions,  
6 and keeping fun in the house. I hope the field of computer science never loses its sense of fun
```

the map ( $k1, v1$ ) => [ $(k2, v2)$ ]:

```
[ {'think':1}, {'fun':1},  
  {'in': 1}, {'in': 1} ... ]  
[ {'fun':1}, {'it':1}, {'was': 1},  
  {'it':1} ... ]
```

We take the example of counting the frequencies of words in a passage of text. Here  $k1$  is the line number,  $v1$  is the line of text. The map separates the line into words, and for each word emits a pair with the word as the key ( $k2$ ) and the number 1 as the value ( $v2$ )

## Example: Word Frequencies

$k2$  (word)  $v2$  (freq per line)

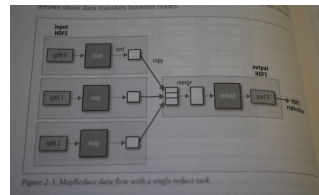
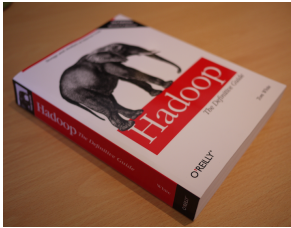
```
[ 'think': [1],  
  'fun':   [1,1],  
  'in':    [1,1,1,1],  
  'it':    [1]...]
```

reduce with + for each key ( $k2, [v2]$ ) => ( $k2, v2$ ):

```
[ {'think':1}, {'fun':2}, {'in':4} ... ]
```

MapReduce then collects together all the  $v2$ s for a given  $k2$ , and allows us to provide a reduce function. Here we simply use plus as our reduction function to sum up all the 1s, resulting in a count for that word.

# MapReduce and Hadoop



Although the concepts of mapping and reducing were not invented by Google - they have been in functional programming for a long time - Google's implementation of MapReduce for distributed computation inspired other implementations such as Hadoop. Hadoop is now widely used, especially in cloud environments, to allow large data-sets to be processed quickly using many machines in parallel.

## Java Code : Mapper (Hadoop)

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

If we were to write this in Java code, using Hadoop (Google's MapReduce is not open source), then we would write our mapper something like this - the type parameters on Mapper reflect the types of k1, v1, k2 and v2. context. We emit a pair from the mapper by calling `context.write(word, one)`. It is worth noting that the number of output pairs from the mapper does not have to be the same as the number of input pairs.

## Java Code : Reducer (Hadoop)

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

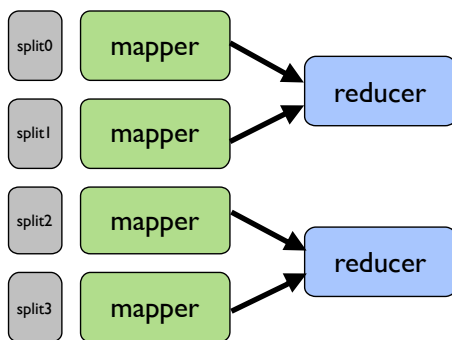
The reduce here takes a word (Text) and a sequence of numbers (Iterable<IntWritable>) which we simply sum up and emit paired with the original key.

## Java Code : main (Hadoop)

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
  
        Job job = new Job(conf, "wordcount");  
  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        job.setMapperClass(Map.class);  
        job.setReducerClass(Reduce.class);  
  
        job.setInputFormatClass(TextInputFormat.class);  
        job.setOutputFormatClass(TextOutputFormat.class);  
  
        FileInputFormat.addInputPath(job, new Path("/home/rbc/textfiles");  
        FileOutputFormat.setOutputPath(job, new Path("/home/rbc/wordcounts");  
  
        job.waitForCompletion(true);  
    }  
}
```

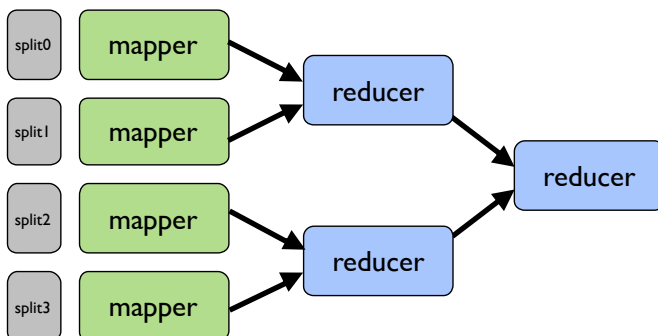
The last piece of the puzzle is a main class which sets up the job with the mapper and reducer, and tells MapReduce (or Hadoop in this case) where to find the input data, and where to write the results.

## Scaling Out



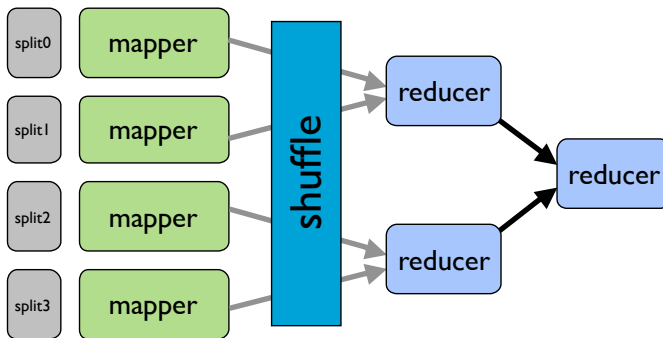
When MapReduce runs it splits the input data into a number of “splits” and processes each split with a separate mapper, most likely running on a separate computer. The outputs are then put through the reducer phase. There are normally fewer reducers than mappers, but it depends on the complexity of the various parts of the job.

## Scaling Out



Reducers can be chained if the reduce function is commutative and associative. This can lead to improved efficiency as the initial reductions can be done nearer to the mappers, so we don't have to move so much data around.

## Shuffle and Sort



The magic of MapReduce is in the shuffle. This is where all of the key-value pairs output from the map for a given key value are gathered up and supplied to the same reducer, so that it can do its work.

## Example: Distributed Grep

I want to grep for a certain word in a very large file: gigabytes or terabytes of data, or more...

The map function emits a line if it matches a supplied pattern.

The reduce function is an identity function that just copies the supplied intermediate data to the output.

This distributed grep example is taken from Dean and Ghemawat's original MapReduce paper, referenced on the last slide of this lecture.

## Computing the Web Index

For all words on the web, I want to index each webpage that mentions each word

The map function emits a pair of (word, documentId)

The reduce function sorts the documentIds (by some means) for each key word.

This web indexing example shows a simplified form of how the web index might be calculated. It doesn't take into account any particular ranking algorithm.



# References

## Some further resources on MapReduce

Jeff Dean and Sanjay Ghemawat's original MapReduce paper  
<http://research.google.com/archive/mapreduce.html>

Hadoop, an open source implementation of MapReduce  
<http://hadoop.apache.org/mapreduce/>

Tom White's *Hadoop - The Definitive Guide*  
<http://hadoopbook.com/>

---