# Part 1 - A Map-Reduce framework exploiting the Java Stream API

The Map-Reduce paradigm is widely used for processing huge amounts of data in a parallel and distributed setting. In this assigment, students are required to implement a simple software framework providing the functionalities of Map-Reduce, but ignoring the aspects of parallelism and distribution. As a proof of concept, two simple working instances of the framework should be implemented as well.

For an introduction to the Map-Reduce framework see the paper MapReduce: Simplified Data Processing on Large Clusters. For a presentation of Map-Reduce as Software Framework, identifying the *hot spots*, see https://en.wikipedia.org/wiki/MapReduce#Dataflow (since we ignore the distribution aspects, you can ignore the *Partition function*.)

**Solution format:** An archive `MapReduce-<yourSurname>.zip` containing the Java files implementing Exercises 1, 2, and (optionally) 3. If you use `NetBeans`, please send in the archive the entire project.

### Exercise 1 - The framework

Following the guidelines presented in the lesson of October 23, 2020 (see http://pages.di.unipi.it/corradini/Didattica/AP-20/index.html#framework), and more specifically the *Template Method design pattern*, implement in Java a Map-Reduce software framework providing the functionalites described in the above documentation and respecting the following constraints:

1. For `key/value` pairs, the framework must use the attached class `Pair.java` (you can change its package, but nothing else).
2. The hot spots of the framework are the methods `read`, `map`, `compare`, `reduce` and `write`.
3. The framework must use, when possible, the Stream API. For example, `map` takes a stream of key-value pairs as argument and returns a stream of key-value pairs (types of argument and result may differ, of course).

### Exercise 2- Counting words

By instantiating the framework, implement a program that counts the occurrences of words of length greater than 3 in a given set of documents, respecting the following constraints:

1. The program should ask the user for the absolute path of the directory where documents are stored. Only files ending in `.txt` should be considered.
2. The `read` function must return a stream of pairs `(fileName, contents)`, where `filename` is the name of the text file and `contents` is a list of strings, one for each line of the file. For the `read` function you can exploit the enclosed class `Reader.java` in the way you prefer.
3. The `map` function must take as input the output of `read` and must return a stream of pairs containing, for each word (of length greater than 3) in a line, the pair `(w, k)` where `k` is the number of occurrences of `w` in that line.
4. The `compare` function should compare strings according to the standard alphanumeric ordering. (The result should adhere to the standard Java conventions, see the `compareTo` method of interface `Comparable`.)
5. The `reduce` function takes as input a stream of pairs `(w, lst)` where `w` is a string and `lst` is a list of integers. It returns a corresponding stream of pairs `(w, sum)` where `sum` is the sum of the integers in `lst`.
6. The `write` function takes as input the output of `reduce` and writes the stream in a CSV (Comma Separated Value) file, one pair per line, in alphanumeric ordering. For the `write` function you can exploit the enclosed class `Writer.java` in the way you prefer.

For testing the program you can use the enclosed archive `Books.zip` which contains parts of some famous books as downloaded from the pages of the *Gutenberg Project*. (before the site became inaccessible from Italy, see *Raffaele Angius, Perché il Progetto Gutenberg sarà sotto sequestro per sempre*).

**Exercise 3 - [Optional] Producing an Inverted Index**

By instantiating the framework, implement a program that generates an *Inverted Index* (for words of length greater than 3). That is, given as input the absolute path of a directory, the program prints in a CSV file for each word `w` (of length greater than 3) appearing in the `.txt` documents of the directory, a line `w, filename, line` if `w` appears in line number `line` of file `filename`. The lines should be sorted in the natural way.