

# Master 1 Bioinformatique

## Object-oriented programming

### Hierarchical clustering

Emmanuelle Becker, Olivier Dameron, Marine Louarn

February 19, 2018

Version 1.4

## 1 Objective

This project's goal is to classify a set of students according to their grades, and to generate the corresponding dendrogram.

From the object-oriented programming point of view, this is an excellent opportunity to practice:

- data encapsulation
- method and constructor overloading within a class
- class inheritance and method overriding
- static methods
- iterators
- and more generally:
  - reuse existing code
  - generate javadoc
  - use git

## 2 Set up your environment

### 2.1 Cloning the project

The project's description, the Java source files and some example datasets can be retrieved from its git repository<sup>1</sup>.

---

<sup>1</sup><https://gitlab.com/odameron/javaHierarchicalClustering>

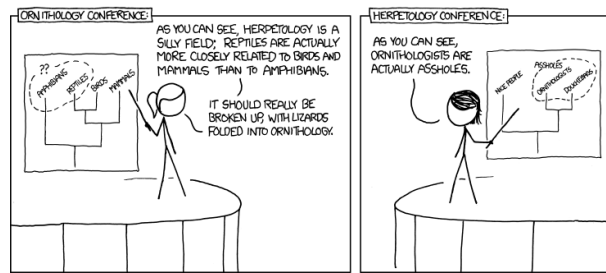


Figure 1: xkcd 867 (<http://xkcd.com/867/>)

**FIXME:** We should definitely do a fork instead. Change git remote origin

**FIXME:** Encourage the students to create their own repository on gitlab or github so that we can monitor their progress

**Step 1** *Set up your working environment:*

1. create an empty local directory;
2. move to this directory;
3. clone the project with `git clone <gitURL>` (you have to retrieve `gitURL` from the web page);
4. create your own branch with `git branch -b devLastnameFirstname` (obviously, adapt `Lastname` and `Firstname`). Reminder: you might want to read the section on branches from a git tutorial<sup>2</sup>

From now on, you are strongly encouraged to use git profusely and commit at least at each step.

## 2.2 Configure your editor

**FIXME:** instructions for importing in Eclipse or writing an ant file

**FIXME:** instructions for running javadoc in the doc directory

# 3 Representing a class of students

## 3.1 Class Student: methods overload

**Step 2** Create a class *Student* that represents the set of students. Each student has an (assumed) unique identifier (a string) and a grade (a double).

**Step 3** Add a first constructor having for parameters an identifier and a grade. Then add a second constructor having an identifier as single parameter. This is an excellent opportunity to use method overload...

**Step 4** Add the methods `getIdent()`, `getGrade()` et `setGrade(double newGrade)`.

<sup>2</sup><https://bioinfo-fr.net/git-usage-collaboratif>

**Step 5** In the `main(...)` method, create the following instances and check that the methods from step 4 still work correctly:

```
1 Student riri = new Student("riri", 12.5);
2 Student fifi = new Student("fifi", 14.0);
3 Student loulou = new Student("loulou", 18.5);
4 Student geo = new Student("geo", 19.5);
5 Student donald = new Student("donald", 10.5);
```

**Step 6** *FIXME: run javadoc and commit*

### 3.2 Class GroupOfStudents: inheritance and static methods

**Step 7** Create a class `GroupOfStudents` that represents a set of students. `GroupOfStudents` is a sub-class of `java.util.ArrayList`<sup>3</sup>. Make sure to read the documentation for `ArrayList`, you will need it soon. Please note that `ArrayList` is a generic class, whereas all the elements of a `GroupOfStudents` instance are composed of instances of `Student`, so you will need to state that `GroupOfStudents` is a subclass of `ArrayList<Student>`.

**Step 8** In the `GroupOfStudents`'s `main(...)` method, create an object `m1bioinfo` as an instance of `GroupOfStudents`, and add the members `riri`, `fifi`, `geo`, `donald` and `loulou` (respect this order so that the highest and lowest grades are in the middle of the list. The idea here is to avoid having the students almost sorted for the clustering).

**Step 9** Add the methods `getMinGrade()`, `getMaxGrade()` et `getAverageGrade()` to the class `GroupOfStudents`. The class `java.lang.Math`<sup>4</sup> has several useful methods. For iterating over all the students that compose a promotion, you can seek inspiration from the article "Traversing collections"<sup>5</sup>. Feel glad to have declared `GroupOfStudents` as a subclass of `ArrayList`.

**Step 10** In the `main(...)` method of `GroupOfStudents`, add the code for printing the lowest grade, the highest and the average for `m1bioinfo`.

**Step 11** The `GroupOfStudentsLoader` class (provided in the `src` directory) has a method `loadTsvFile(...)` that takes as argument a text file (one student per line; its identifier, a tabulation, its grade) for creating an instance of `GroupOfStudents`. Why is the method `loadTsvFile(...)` declared as "static"? Draw the parallel with the methods `min(...)`, `max(...)` and `abs(...)` from `java.lang.Math`.

**Step 12** *FIXME: run javadoc and commit*

<sup>3</sup><http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

<sup>4</sup><http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>

<sup>5</sup><http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>

## 4 Hierarchical clustering

### 4.1 Principle

#### 4.1.1 Agglomerative vs. divisive approaches

**Classifying** consists in organizing a set of elements in groups based on the elements' similarities or differences.

**Hierarchical clustering** consists in organizing the sets of elements into subsets included in to each others in a tree-like structure. There are two main approaches for determining this organization:

- the **agglomerative approach** (also called ascending) starts by creating one (atomic) cluster for each element, and then iteratively generates new clusters composed of the most similar two, until there only remains one cluster;
- the **divisive approach** (also called descending) starts by gathering all the elements into a single cluster, and then iteratively decompose the clusters into subclusters until each of them is only composed of a single element.

The divisive approach requires more operations than the agglomerative one and is therefore longer... except when we only need the most general clusters (e.g. to separate a sample into two groups).

#### 4.1.2 Distance measures between elements and between clusters

For both the agglomerative and ther divisive approaches, clustering depends on two main parameters:

- a **distance measure between elements** (also simply called *distance*). There are several classical ones: euclidian distance, Manhattan distance... In our case, we will consider that the distance between two students is the absolute value of the difference of their grades;
- a **distance measure between clusters** (also called *linkage*) that relies on the *distance* between elements of the two clusters. There are several classical linkage measures: the average of the distances between all the combinations of elements, their maximum, their minimum... In our case, we will consider that the distance between two clusters of students is the average of the distances between all the elements of the first cluster and all the elements of the second cluster.

### 4.2 Class ClusterOfStudents

This section aims at implementing the `ClusterOfStudents` class for representing a cluster of `Student` instances. A simple cluster is composed of a single instance of `Student`. A complex cluster is composed of several sub-clusters which can themselves be either simple or complex clusters. A complex cluster has a tree-like structure where all the leaves are simple clusters.

Initially, a complex cluster is only composed of simple clusters (Fig. 2). After clustering, a complex cluster is composed of sub-clusters that are intermediate complex clusters (Fig. 3)



Figure 2: Complex cluster in its initial state: it is composed of five simple sub-clusters, each composed of a student.

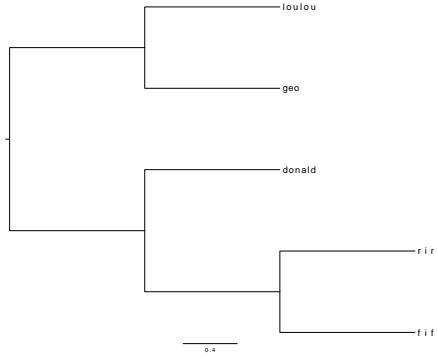


Figure 3: Complex cluster after clustering. It is composed of two intermediate complex sub-clusters. The first is itself composed of two simple clusters (geo and loulou). The second is composed of a simple sub-cluster (donald) and a complex sub-cluster composed of two simple sub-clusters (riri and fifi).

#### 4.2.1 Initialization

**Step 13** Create a *ClusterOfStudents* with a *subclusters* attribute that represents the list of its sub-clusters. For simplifying the clustering step (when marshalling the students composing the cluster), add an attribute *students* that represents the set of students constituting the leaves of the cluster.

**Step 14** Add the following constructors:

- a default constructor *ClusterOfStudents()* that creates an empty cluster (we do not really need it, but I find it cleaner to have a default constructor);
- a constructor for simple clusters *ClusterOfStudents(Student aStudent)*;
- a constructor for complex clusters before clustering *ClusterOfStudents(GroupOfStudents aGroupOfStudents)*.

**Step 15** In *ClusterOfStudents*' *main(...)* main method, create an instance of a simple cluster *geoCluster* initialized with *geo*, and an instance of complex cluster *bioCluster* initialized with *m1bioinfo*.

**Step 16** *FIXME*: run javadoc and commit

#### 4.2.2 Visualization

The Newick format<sup>6</sup> provides a straightforward representation of trees and dendrograms, and is supported by most visualization tools. You can use the Tree Viewer web server<sup>7</sup> or T-REX<sup>8</sup> or the dedicated softwares FigTree<sup>9</sup>, dendroscope<sup>10</sup> (free use in an academic context; getting a licence is not required for the basic functions). FigTree seems to give the best results.

The dendrogram from Fig. 2 can be represented by  
`((loulou,geo),(donald,(riri,fifi)));`.

NB : for visualizing dendrograms, we could as well have used the R functions via the Java-R binding, but it is more complicated, and writting Newick files makes for an interesting exercise anyway.

**Step 17** Add a *getNewick()* method to the class *ClusterOfStudents* that returns a string representing the dendrogram in the Newick format. Because of the final semicolon, you may need to introduce an intermediate function (aptly named *getNewickIntermediate()*). For marshalling the tree, you will make your life easier by considering a recursive approach (but this is not mandatory). Should these methods' visibility be public, protected or private?

**Step 18** Generate a Newick representation of *bioCluster* and check (for example with T-REX or dendroscope) whether you get something similar to Fig. 2.

**Step 19** *FIXME*: run javadoc and commit

#### 4.2.3 Clustering

**Step 20** Add a method *linkage(ClusterOfStudents anotherCluster)* that returns the distance between the current cluster and *anotherCluster*. Choosing the average of the absolute value of the grade differences for each combination of students from each cluster is probably the easiest solution.

**Step 21** In *ClusterOfStudents*'s *main(...)* method, create two simple clusters *loulouCluster* and *donaldCluster* and check whether the distance between *geoCluster*, *loulouCluster* and *donaldCluster* are what you expect them to be (check the six combinations).

**Step 22** In *ClusterOfStudents*'s *main(...)* method, create the complex cluster *geoLoulouCluster* and check whether its distance with *donaldCluster* and *geoLoulouCluster* (and conversely).

<sup>6</sup><http://evolution.genetics.washington.edu/phylip/newicktree.html>

<sup>7</sup><http://www.proweb.org/treeviewer/>

<sup>8</sup><http://www.trex.uqam.ca/>

<sup>9</sup><http://tree.bio.ed.ac.uk/software/figtree/>

<sup>10</sup><http://ab.inf.uni-tuebingen.de/software/dendroscope/>

**Step 23** Add a method `clusterizeAgglomerative()`. Perform clustering on `bigCluster` and display the result as a Newick string.

**Step 24** *FIXME*: run javadoc and commit

Figure 4 shows the classification result for a set of students. Notice that because all the branches have the same length, the dendrogram seems to display two main clusters. Figure 5 shows that by making the length of each branch proportional to the distance separating the two clusters it joins, the dendrogram reveals three main clusters (cf. section 5.1).

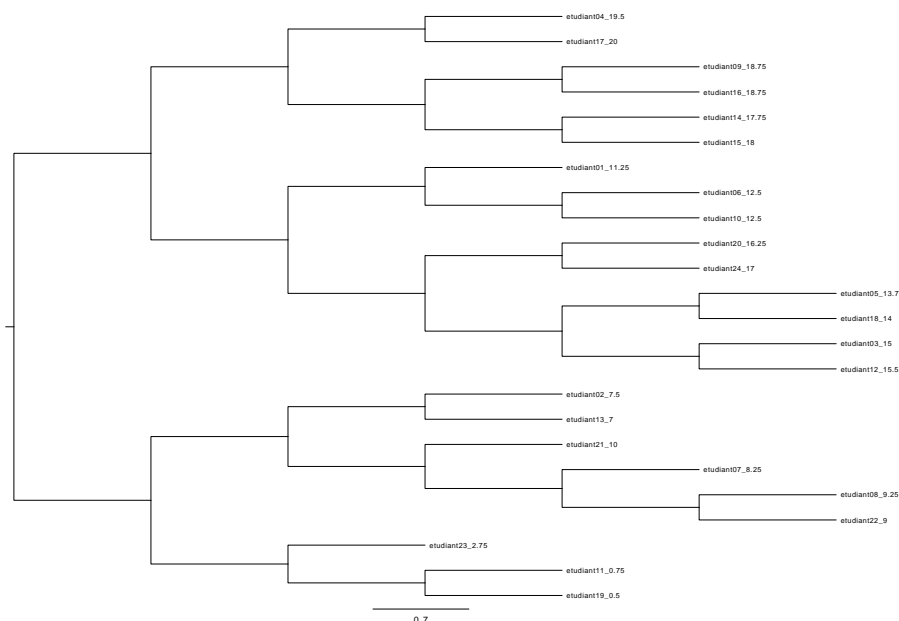


Figure 4: Complex cluster after clustering. The length of each branch is constant.

## 5 Optional extensions

### 5.1 Dendrogram improvement

The Newick format allows to specify the branches' length. For visualizing the result, not all the tools mentioned previously support this feature. Rather use FigTree<sup>11</sup> or the Tree Viewer<sup>12</sup> website.

**Step 25** Improve the `getNewick()` method so that all the leaves are at the same level (i.e. aligned on the right, contrary to Fig. 4).

**Step 26** Improve the `getNewick()` method so that all the leaves are at the same level and the branches' length are proportional to the (absolute value of) the difference between the grades.

<sup>11</sup><http://tree.bio.ed.ac.uk/software/figtree/>

<sup>12</sup><http://www.proweb.org/treeviewer/>

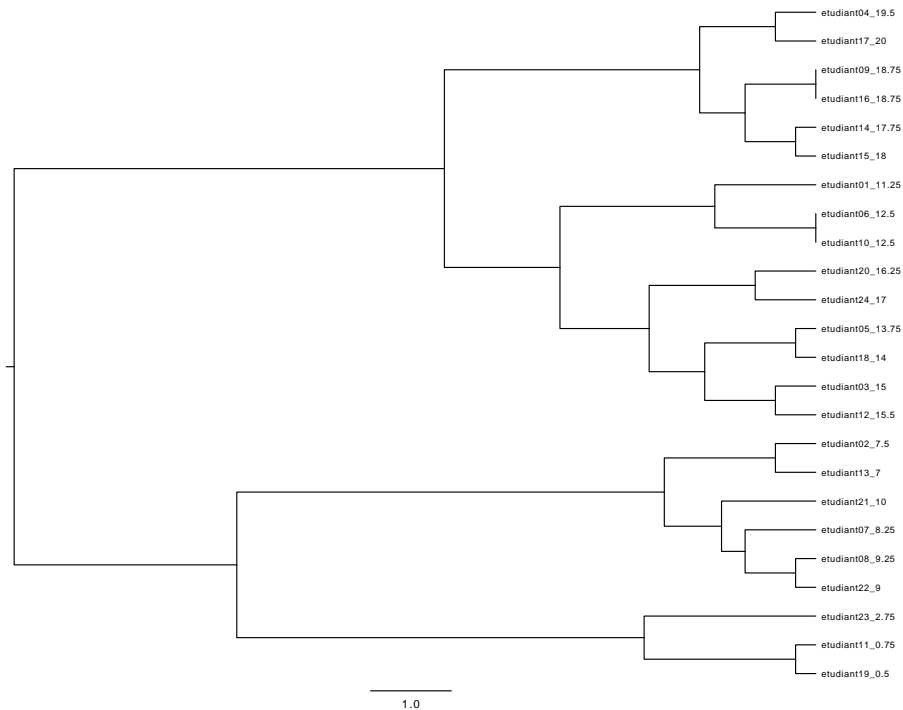


Figure 5: Complex cluster after clustering. The length of each branch is proportional to the distance between the two clusters it unites.

## 5.2 Divisive approach

**Step 27** Add a method `clusterizeDivisive()` to the class `ClusterOfStudents`. Perform clustering on `bioCluster` and compare with the agglomerative approach.

## 5.3 Modeling considerations for `ClusterOfStudents`

**Step 28** Should we have declared `ClusterOfStudents` as a subclass of `GroupOfStudents`? Are these classes' internal structures compatible? Are there `GroupOfStudents`'s attribute or methods for which such an inheritance would make sense?

In the class `ClusterOfStudents`, each instance of `Student` appears twice:

- in the attribute `subClusters` because the dendrogram has as many sub-cluster leaves as students;
- in the attribute `students` qui permet un parcours plus simple de la liste des étudiants d'un cluster en évitant de devoir parcourir récursivement tout le dendrogramme à chaque fois.

One could have the impression that this results in doubling the memory usage (even if in our case the overhead would be perfectly acceptable, as each instance takes up a small space in memory and there are few students). However, Java obviously does not duplicate the `Student` instances in both attributes. Each



attributes only contains references to the **Student** instances (i.e. their address). In addition to avoiding unnecessary object duplication, the second benefit is that it preserves consistency: changing a student's grade in the **students** attribute will result in the change being visible if you later access this student through the **subClusters** attribute (and conversely).

Overall, using two attributes seemingly redundant because they contain (references to) the same objects:

- has the main advantage of improving processing performances by avoiding to traverse the dendrogram when retrieving the list of students (which happens often during clustering). This was actually the motivation for introducing the **students** attribute.
- has the secondary advantage of dispensing you from writing the dendrogram traversal function that would have been necessary for retrieving the list of students.
- has the drawback of increasing the memory footprint.

**Step 29** *Create a class **ClusterOfStudentsBis** that only contains the attribute **subClusters**. Compare the respective clustering time of **ClusterOfStudents** and **ClusterOfStudentsBis**.*