

# Master 1 Bioinformatique

## Object-oriented programming

Olivier Dameron

olivier.dameron@univ-rennes1.fr

February 5, 2018

Version 1.3

### 1 Objective

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

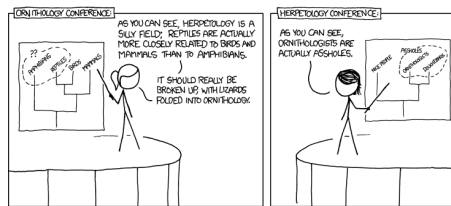


Figure 1: xkcd 867 (<http://xkcd.com/867/>)

### 2 Representing a class of students

#### 2.1 Class Student: methods overload

**Step 1** 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 2** 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 3** Add the methods *getIdent()*, *getGrade()* et *setGrade(double newGrade)*.

**Step 4** In the *main(...)* method, create the following instances and check that the methods from step 3 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);

```

## 2.2 Class Promo: inheritance and static methods

**Step 5** Create a class *Promo* that represents a set of students. *Promo* is a sub-class of `java.util.ArrayList`<sup>1</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 *Promo* instance are composed of instances of *Student*, so you will need to state that *Promo* is a subclass of `ArrayList<Student>`.

**Step 6** In the *Promo*'s `main(...)` method, create an instance `m1bioinfo` of *Promo*, 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 7** Add the methods `getMinGrade()`, `getMaxGrade()` et `getAverageGrade()` to the class *Promo*. The class `java.lang.Math`<sup>2</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>3</sup>. Feel glad to have declared *Promo* as a subclass of *ArrayList*.

**Step 8** Dans le `main(...)` de *Promo*, ajoutez le code pour afficher la plus basse note, la plus haute note et la moyenne de `m1big`.

**Step 9** La classe *PromoLoader* (**FIXME: fournie sur l'ENT**) fournit une méthode `loadTsvFile(...)` qui utilise un fichier texte (un étudiant par ligne ; l'identifiant et la note sont séparés par une tabulation) pour construire une instance de *Promo*. Pourquoi la méthode `loadTsvFile(...)` est-elle déclarée "static" ? Faites le parallèle avec les méthodes `min(...)`, `max(...)` et `abs(...)` de la classe *Math*.

## 3 Clustering hiérarchique

### 3.1 Principe

#### 3.1.1 Approches agglomérative et divisive

La classification consiste à organiser des éléments en groupes basés sur leurs similarités ou leurs différences.

Le clustering hiérarchique consiste à organiser ces éléments en sous-groupes imbriqués les uns dans les autres de façon arborescente. Il y a deux façon de procéder :

<sup>1</sup><http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

<sup>2</sup><http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>

<sup>3</sup><http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>

- l'approche agglomérative (aussi appelée ascendante) commence par créer un cluster par élément, puis crée successivement un nouveau cluster regroupant les deux clusters les plus similaires jusqu'à ce qu'il ne reste plus qu'un seul cluster ;
- l'approche divisive (aussi appelée descendante) commence par rassembler tous les éléments dans un seul cluster puis à le décomposer successivement en sous-clusters jusqu'à ce que tous les clusters ne soient plus constitués que d'un seul élément.

L'approche divisive nécessite plus d'opérations que l'approche agglomérative et est donc plus longue... sauf si on n'a besoin que des clusters de plus haut niveaux (par exemple pour séparer un échantillon en deux groupes).

### 3.1.2 Mesures de distance entre éléments et entre clusters

Pour l'approche agglomérative comme pour l'approche divisive, le clustering dépend de deux paramètres principaux :

- une **mesure de distance entre éléments** (appelée simplement *distance*). Il en existe plusieurs classiques : la distance euclidienne, la distance de Manhattan... Dans notre cas, on considérera que la distance entre deux étudiants est la valeur absolue de la différence de leurs notes ;
- une **mesure de distance entre clusters** (appelée *linkage*), qui utilise la distance entre les éléments des deux clusters. Là encore il en existe plusieurs classiques : la moyenne des distances de toutes les combinaisons d'éléments, le maximum, le minimum... Dans notre cas, on considérera que la distance entre deux clusters d'étudiants est la moyenne des distances entre tous les éléments du premier cluster et tous les éléments du second cluster.

## 3.2 Classe ClusterOfStudents

Le but de cette section est de mettre en œuvre la classe `ClusterOfStudents` qui représente un cluster d'instances de la classe `Student`. Un cluster simple n'est constitué que d'une seule instance de `Student`. Un cluster complexe est constitué de plusieurs sous-clusters qui peuvent être chacun des clusters simples ou des clusters complexes. Un cluster complexe a donc une structure d'arbre dont les feuilles sont toujours des clusters simples. À l'état initial, un cluster complexe est constitué uniquement de clusters simples (Fig. 2). Après clustering, un cluster complexe est constitué de sous-clusters qui peuvent être des clusters complexes intermédiaires (Fig. 3)

### 3.2.1 Initialisation

**Step 10** Créez la classe `ClusterOfStudents` avec un attribut `subclusters` qui représente la liste des sous-clusters. Pour simplifier l'étape de clustering où il est nécessaire de parcourir la liste des étudiants composant un cluster, on ajoute l'attribut `students` qui représente la liste des étudiants constituant les feuilles du cluster.

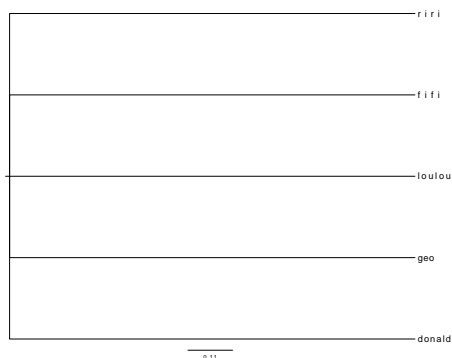


Figure 2: Cluster complexe dans l'état initial : il est composé de cinq sous-clusters simples, composés chacun d'un étudiant.

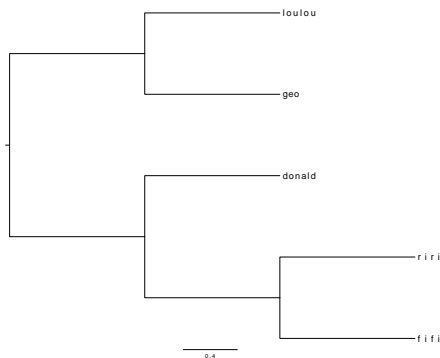


Figure 3: Cluster complexe après clustering. Il est composé de deux sous-clusters complexes. Le premier est à son tour composé de deux sous-clusters simples (geo et loulou). Le second est composé d'un sous-cluster simple (donald) et d'un sous-cluster complexe composé de deux sous-clusters simples (riri et fifi).

**Step 11** Ajoutez un constructeur par défaut `ClusterOfStudents()` qui crée un cluster vide (on ne devrait pas en avoir besoin, mais c'est plus propre), un constructeur `ClusterOfStudents(Student aStudent)` qui crée un cluster simple, et un constructeur `ClusterOfStudents(Promo aPromo)` qui crée un cluster complexe dans l'état initial.

**Step 12** Dans le `main(...)` de `ClusterOfStudents`, créez une instance de cluster simple `geoCluster` initialisée avec `geo`, et une instance de cluster complexe `bigCluster` initialisée avec `m1big`.

### 3.2.2 Visualisation

Le format Newick<sup>4</sup> permet une représentation simple des arbres et des dendrogrammes, qui est compris par de nombreux outils de visualisation. Vous

<sup>4</sup><http://evolution.genetics.washington.edu/phylip/newicktree.html>

pourrez par exemple utiliser le serveur web Tree Viewer<sup>5</sup> ou T-REX<sup>6</sup> ou les programmes FigTree<sup>7</sup> dendroscope<sup>8</sup> (utilisation académique libre, il n'est pas nécessaire d'obtenir une licence pour les fonctions de base). FigTree est celui qui semble donner les meilleurs résultats.

Le dendrogramme de la figure 2 peut être représenté par  
`(riri,fifi,loulou,geo,donald);`  
et celui de la figure 3 par  
`((loulou,geo),(donald,(riri,fifi))):`.

NB : pour la visualisation, on aurait aussi pu utiliser les fonctions de R via le binding Java-R, mais c'est plus compliqué, et puis l'exercice de générer des fichiers au format Newick était intéressant.

**Step 13** Ajoutez une méthode `getNewick()` qui renvoie une chaîne de caractères représentant le dendrogramme au format Newick. À cause du point-virgule final, vous aurez peut-être besoin d'introduire une fonction intermédiaire `getNewickIntermediate()`. Puisqu'il s'agit d'un parcours d'arbre, vous vous simplifierez la vie en adoptant une approche récursive. Quelle devra être la visibilité de ces méthodes (publique, protégée, privée) ?

**Step 14** Générez la représentation de `bigCluster` au format Newick et vérifiez (par exemple avec T-REX ou dendroscope) que vous obtenez quelque chose de similaire à la figure 2.

### 3.2.3 Clustering

**Step 15** Ajoutez une méthode `linkage(ClusterOfStudents anotherCluster)` qui renvoie la distance entre deux clusters. Le plus simple est sans doute de choisir la moyenne des différences des notes entre toutes les combinaisons d'étudiants du cluster courant et de `anotherCluster`.

**Step 16** Dans le `main(...)` de `ClusterOfStudents`, créez les clusters simples `loulouCluster` et `donaldCluster` et vérifiez que les distances entre `geoCluster`, `loulouCluster` et `donaldCluster` correspondent à ce que vous attendez (pensez à vérifier les six combinaisons).

**Step 17** Dans le `main(...)` de `ClusterOfStudents`, créez le cluster complexe `geoLoulouCluster` et vérifiez que sa distance avec `donaldCluster` correspond à ce que vous attendez. Vérifiez également la distance entre `donaldCluster` et `geoLoulouCluster`.

**Step 18** Ajoutez la méthode `clusterizeAgglomerative()`. Clusterisez `bigCluster` et affichez le résultat au format Newick.

La figure 4 montre le résultat de la classification pour un ensemble d'étudiants. Observez que puisque toutes les branches ont une longueur constante, on a l'impression de distinguer deux clusters principaux. La figure 5 montre qu'en rendant la longueur des branches proportionnelle à la distance séparant les deux clusters que l'on fusionne, ce sont en fait trois clusters principaux qui apparaissent (voir section 4.1).

---

<sup>5</sup><http://www.proweb.org/treeviewer/>

<sup>6</sup><http://www.trex.uqam.ca/>

<sup>7</sup><http://tree.bio.ed.ac.uk/software/figtree/>

<sup>8</sup><http://ab.inf.uni-tuebingen.de/software/dendroscope/>

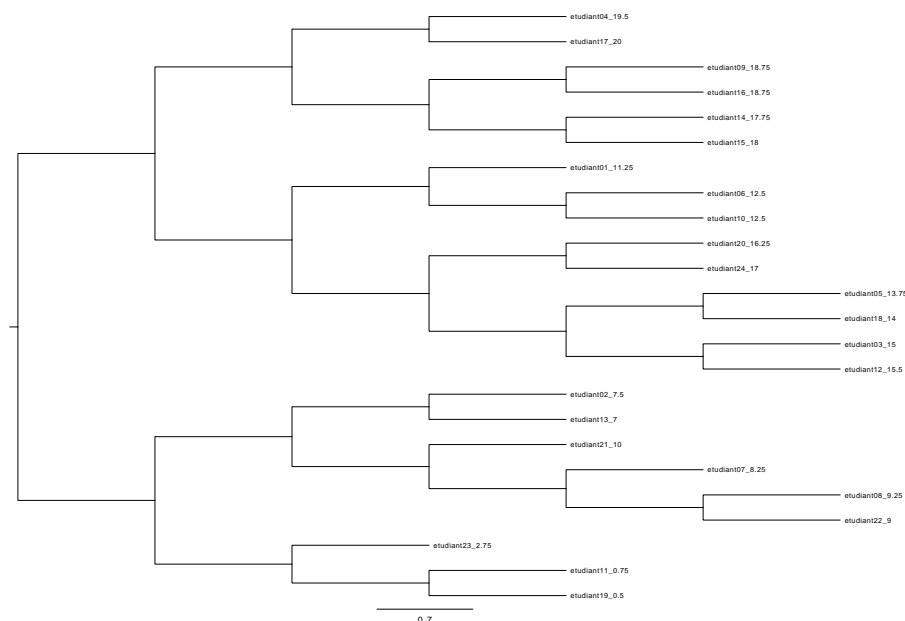


Figure 4: Cluster complexe après clustering.

## 4 Extensions facultatives

### 4.1 Amélioration du dendrogramme

Le format Newick permet d'indiquer la longueur de chacun des branches. Utilisez plutôt l'application FigTree<sup>9</sup> ou le site de Tree Viewer<sup>10</sup> pour la visualisation, les autres outils semblent avoir des problèmes.

**Step 19** Améliorez la méthode `getNewick()` pour que toutes les feuilles soient au même niveau.

**Step 20** Améliorez la méthode `getNewick()` pour que toutes les feuilles soient au même niveau et que la longueur des branches soit proportionnelle à l'écart des notes.

### 4.2 Approche divisive

**Step 21** Ajoutez une méthode `clusterizeDivisive()` à la classe `ClusterOfStudents`. Clusterisez `bigCluster` et comparez avec l'approche ascendante.

### 4.3 Modélisation de `ClusterOfStudents`

Dans la classe `ClusterOfStudents`, chaque instance d'étudiant apparaît deux fois :

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

<sup>10</sup><http://www.proweb.org/treeviewer/>

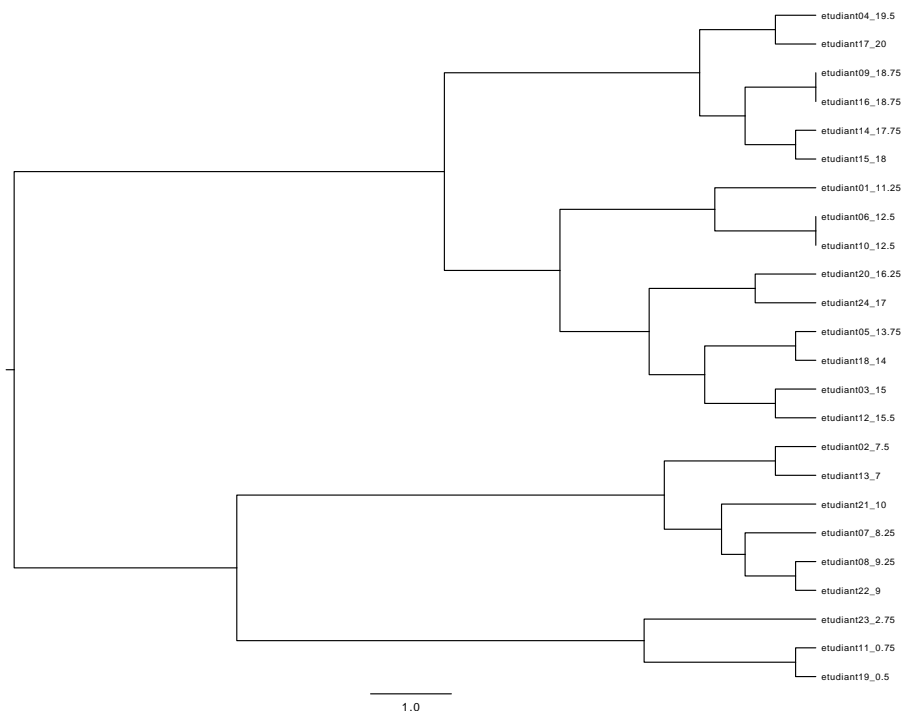


Figure 5: Cluster complexe après clustering. La longueur des branches est proportionnelle à la distance séparant les deux clusters fusionnés.

- dans l'attribut **subClusters** puisque le dendrogramme a autant de sous-clusters feuilles que d'étudiants ;
- dans l'attribut **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.

On pourrait penser que cela occupe donc deux fois plus de place en mémoire que nécessaire, même si dans notre cas ce surcoût est acceptable dans la mesure où chaque instance occupe peu de place mémoire et qu'il y a relativement peu d'étudiants. Néanmoins, Java ne duplique évidemment pas les instances de **Student** dans les deux attributs. Chaque attribut ne contient en fait que les adresses des instances de **Student** (on appelle ça un *passage d'objet par référence*). Ainsi, si vous modifiez la note d'un étudiant dans **students**, cette modification apparaîtra également dans **subClusters**, et inversement.

Au final, le fait d'utiliser deux attributs qui semblent redondants parce qu'ils contiennent les mêmes objets :

- a l'avantage principal d'améliorer le temps de traitement en évitant un parcours de l'arborescence du dendrogramme chaque fois que l'on souhaite parcourir les étudiants (et cela arrive souvent durant l'étape de clustering) ;
- a l'avantage secondaire de simplifier l'écriture de la classe en vous dispensant justement d'écrire la fonction de parcours de l'arborescence du

dendrogramme ;

- a l'inconvénient d'augmenter légèrement la consommation de la mémoire.

**Step 22** *Écrivez une classe `ClusterOfStudentsBis` qui ne contient que l'attribut `subClusters`. Comparez les temps de clusterisation de `ClusterOfStudents` et de `ClusterOfStudentsBis`.*