Project ABP-`light`

The goal of the project is to apply most of the notions you learnt during the course. The topic of the project is about allocating students to schools. In France, the system APB (Admission Post Bac) allocates the students who terminate high school to higher institution. This project should give you an idea about the difficulty of the problem, though you will work on a very small (synthetic) data set.
You will be provided two files :

— one containing the program of 21 institutions in Ile-de-France. You will find only two kinds of higher institutions : universities and preparatory schools. For each institution, we provide an id (which will be an integer), the maximum number of students that is can accept, the type ("`U`" if it is a university, "`P`" if it is a preparatory school), the name of the institution, the name of the program, and a "mention".

— another containing a list of candidates students with their preferences over institutions.

You need to write code to answer the following questions.

1. For each institution, print on the console how many times the institution is ranked first, second and third by all candidates.
2. Do the same questions but only among the universities. For instance, let p_1, p_2 and p_3 be preparatory schools and u_1, u_2 be universities. Let us consider candidate Alice who has the following preferences $p_1 \succ u_1 \succ p_2 \succ p_3 \succ u_2$, where $a \succ b$ means that Alice prefers institution a over institution b. Then we will consider that u_1 is ranked by Alice first among the universities and u_2 is ranked second.
3. Find a fair allocation of the candidates to the institutions. The constraint of the maximum number of students per institution must be satisfied. Of course, there are many solutions to this problem, some are fairer than others. What could be considered as optimal could be difficult to compute. I do not expect you to find such an optimal solution. I want you to find at least one simple solution. Of course, the fairer it gets, the more bonus points you will get.
4. Print in the console all the candidates using the alphabetic order with their accepted school.
5. Print in the console all institutions by order of number of candidates accepted. You should write in each line the name of the institution followed by the number of candidates accepted.

The project should be done in pairs. Many architectures are possible and you should choose one that uses well the concepts of an object oriented langage. We advise to write the code and comments in English (at least, do not use any accents in the code).
what needs to be submitted : If the name of the authors of the project are Astérix and Obélix, store in a directory named `Asterix_Obelix`

— all the source files (i.e. all files with extension `.java`). Please comments your code so that I can understand what the methods do.

— A pdf file containing a brief explanation/description of your method to allocate the candidates and why you think it is fair.

The directory should then be compressed in a `zip` file.
submission : **by email** to `stephane.airiau@dauphine.fr`
deadline May 10th

Sorting in Java

In the last course, we saw how to manage lists, sets, etc. We did not have time to present two slides about sorting. The good news is that Javaprovides tools to sort automatically.

Natural order

To sort something, one need to use a notion of ordering for that something. If you rank runners of a marathon, you usually rank them with their times. If you rank movies, you usually rank them by number of viewers, etc. The idea is then simply to specify that the objects stored in your list have an ordering. To do so, we just need to make them implement the interface `Comparable`. It contains *one* method only

```
public int compareTo(T o)
```

The method returns

— a negative `int` if the current obect is "smaller" than the object `o`.

— `0` if the two objects are "equally large"

— a positive `int` if the current object is "larger" than the object `o`.

So, if a class `E` implements the `Comparable` interface, and if a and b are two objects of class `E`, you can use `a.compareTo(b)` to know whether `a` is smaller, equal, or larger than `b`. Note that `String`, `Integer`, `Double`, `Date`, already implement the interface `Comparable`. In the following examples, we decided that the natural order for Gauls is the number of boars they eat.

```
public class Gaul extends Character
                implements Comparable<Gaul>{
  String name;
  int numBoars;
    ...

  public int compareTo(Gaul ixis) {
    return this.numBoars - ixis.numBoars;
  }
}
```

```
Gaul astérix = new Gaul("Astérix");
astérix.numBoars=12;
Gaul Obélix = new Gaul("Obélix");
astérix.numBoars=52;
if (obelix.compareTo(asterix) > 0)
    System.out.println("Qui est gros??");
```

Alternative orderings

If we get back to the example of runners of a marathon, indeed it is natural to rank them by their running time, but one may also want to rank them using the alphabetic order. To allow the use of alternative notion of ordering, one can create a *class* that will represent this alternative notion of ordering, and this class will then need to implements the interface `Comparator`. Basically, you only need to implement the method `compare` that compares the two objects passed in argument.

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

The following class is designed to compare Gauls, or even any Character using their height.

```
public class OrderByHeight implements Comparator<Character> {
    public int compare(Character left, Character right){
      return left.height < right.height ? -1:
            (left.height == right.height ? 0 : 1);
    }
  }
```

```
OrderByHeight compH = new OrderByHeight();
Personnage obelix = new IrreductibleGaulois("Obelix", 1.81);
Gaulois asterix = new IrreductibleGaulois("Astérix", 1.60);
if (compH(asterix, obelix) > 0)
    System.out.println("Astérix is taller than Obélix!”);
```

Sorting made easy

To sort, you must simply use the interface Collections (note the plural). It contains two methods that you can call to order a list. The signatures are a bit intimidating, but they make sense. Actually, as you will only use the methods, they usage is really straightforward. Imagine you want to sort a list l of elements of type T (so l is of type List<T>). If the class T has a natural order, you simply need to call Collections.sort(l);. If the class T does not have a natural order, or if you want to use an alternative ordering comp, you will use Collections.sort(l,comp);

For the brave, here is a short description of the signatures.

— The first method sort is used when the class T has a natural ordering (so it implements the interface Comparable). The signature reads as follows : sort is a public and static method that manipulates a generic class that we will name T. It is not any class, it is a class that must extends a class that implements the intereface Comparable[1]. It takes only one argument, a list of objects of class T. It is a procedure (i.e. it does not return anything), but the method will re-arrange the list in increasing order.

— The second sort method works on a list of elements of type T, but this time, there is no constraints on T (the class T does not need to implements a notion of ordering). But we must specify a second argument of type Comparator that provides a way to compare elements of type T[2].

```
// interface Collections
public static <T extends Comparable<? super T»
                void sort(List<T> list)
public static <T> void sort
                (List<T> list, Comparator<? super T> c)
```

1. and the notion of ordering can be used to compare any objects that are parents of class T. For example, one can use the ordering of Character to order Gauls. This is why we have T **extends** Comparable<? super T>, but I admit this is more advanced.

2. There is again a subtlety as we want to ensure that the notion of order for the elements of type T can be a notion that can order elements of a class parent of T

Here is a small examples where we print Characters by height (and Character does not implement Comparable).

```java
public static void main(String[] args){
   Character obelix = new IndomitableGaul("Obelix", 1.81);
   Gaul asterix = new IndomitableGaul("Astérix", 1.60);
   Character cesar = new Character("César", 1.75);

   List<Character> cast = new ArrayList<Character>();
   cast.add(asterix);
   cast.add(obelix);
   cast.add(cesar);

   for (Character p: cast)
     System.out.println(p.presentation());

   Comparator<Character> order = new OrderByHeight();
   Collections.sort(cast, order);

   for (Character p: cast)
     System.out.println(p.presentation());
```

In the following, you will find some help.

Example Code to read a file

You do not need to understand all the technical details to use this code (of course, you can read the following chapters about input-output and Exception handling). Very briefly :

— The block `try...catch` is used to handle exceptions. Whenever you access a file or communicate through the network, something may go wrong (the file is not present, you cannot write on the disk as it is full, etc). To prevent the application from closing, Javaprovides a mechanism to handle some failures. In this code, if a failure occurs in the block `try{...}`, the execution of the application is suspended and the block `catch` is executed.

— The object of type `FileReader` handles a file stored on your drive. The object of type `BufferedReader` is just a tool to help one read the file.

```
public static void listSchoolNames(String filename){
  try{
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    String line = reader.readLine();
    while (line != null){
      String[] chunks = line.split(";");
      int id= Integer.parseInt(chunks[0]);
      int capacity = Integer.parseInt(chunks[1]);
      String name =chunks[3];
      String degreeName = chunks[4];
      String mention = chunks[5];
      System.out.println("["+id+"]␣" + name);
      line = reader.readLine();
    }
    reader.close();
  }
  catch(IOException e){
    e.printStackTrace();
  }
}
```