Cours LIFI - 2004

TD n°1

29 septembre 2004

1 Calcul du PPCM et du PGCD

On veut disposer en Java d'une fonction donnant le PPCM et le PGCD de deux nombres. Le PPCM de a et b, noté $a \vee b$, est le *Plus Petit Commun Multiple*. Par exemple, pour 18 et 45, il s'agit de $90 = 5*18 = 2*45$. Le PGCD, noté $a \wedge b$, est le *Plus Grand Commun Diviseur*. Par exemple, pour 18 et 45, il s'agit de $9 = 18/2 = 45/5$. On peut calculer le PGCD en utilisant la propriété que :

$$a \wedge b = \begin{cases} a & \text{si b=0} \\ b \wedge (a \mod b) & \text{où } a \mod b \text{ est le reste de } a \text{ par } b \end{cases} \quad (1)$$

Pour calculer le PPCM on utilisera la propriété que :

$$(a \vee b) \times (a \wedge b) = a \times b \quad (2)$$

Question 1 (Calcul du PGCD - méthode 1). *Écrire une fonction récursive pour calculer le PGCD de deux entiers. On définira cette fonction comme une* méthode de classe *d'une classe* `Algebra`. □

Question 2 (Calcul du PGCD - méthode 2). *Bien que concise, l'approche récursive peut poser un problème pour le PGCD de très grands nombres. À votre avis, pourquoi ? Proposez une version non récursive.* □

Question 3 (Calcul du PPCM). *Compléter la classe* `Algebra` *pour ajouter la fonction PPCM. Écrivez un programme de test qui prend deux nombres en paramètre sur la ligne de commande (ex :* `java Test 18 45`*) et affiche leur PPCM et PGCD.* □

2 Design patterns

Les *design patterns* sont des motifs de programmation qui proposent des modèles de conception objet pour des cas de figure bien définis. Ces modèles ont fait leur preuve et un bon programmeur doit les connaître pour pouvoir les appliquer quand le programme qu'il cherche à faire ressemble aux cas de figure du *pattern*. L'ouvrage de référence est le livre du "Gang des quatres" [1]. Bien que les *design patterns* ne soit pas liés à un langage en particulier, on pourra aussi consulter une version orientée Java[2]. Dans cet exercice, nous étudions deux *patterns* en particulier.

[1] *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, Addison-Wesley, 1994

[2] *Patterns in Java : A Catalog of Reusable Design Patterns Illustrated With Uml* by Mark Grand, Wiley,1999

Question 4 (Le *pattern* Singleton). *Comment faire pour définir une classe dont on ne pourra créer qu'une seule instance ? Indication : utiliser un constructeur privé et une* méthode de classe *publique.* □

Le deuxième *pattern* est le *pattern* Strategy. On va étudier son utilité à travers un exemple : le tri d'un tableau. Il existe de nombreuses méthodes de tris qui sont à connaître. Pour une très bonne analyse et toutes les implémentations en Java, voir le site `http://www.iut-orsay.fr/~astier/divers/des_tris.html`. Ici, on utilisera un tri *sélection*. Le principe est le suivant :

- on cherche le plus grand élément dans les n premiers du tableau ;
- on l'échange avec le dernier élément ;
- on cherche le plus grand élément dans les $n-1$ premiers du tableau ;
- on l'échange avec l'avant-dernier élément ;
- et ainsi de suite...

Question 5 (Trier un tableau de `int`). *Écrire une fonction qui trie dans l'ordre croissant un tableau d'entiers donné en paramètre. On définira cette fonction comme une méthode de classe de la classe* `Sorter` *comme montré ci-dessous :*

```
class Sorter {
    static int[] select_sort(int[] v) {
        // ...
        // Compléter cette fonction
        // ...
    }
}
```

Compléter la fonction `select_sort` *pour qu'elle fasse ce tri.* □

Question 6 (Trier un *Vector* de `int`). *Modifier la fonction précédente pour qu'elle s'applique sur un* `Vector`*. On supposera que le vecteur contient des* `Integer` *mais on lèvera une exception si ce n'est pas le cas.* □

Le problème de la méthode ci-dessus est qu'elle n'est pas généralisable à d'autres types. Si on veut maintenant trier un vecteur de `Point`, il faut modifier le code de la fonction. De plus, si on veut modifier l'ordre de tri (par exemple pour trier par ordre décroissant), il faut modifier le code. Pour s'affranchir de ces limitations, on va *déléguer* la fonction de comparaison à un objet. On dira que cet objet implémente une *stratégie* de comparaison.

Question 7 (Une classe `Comparator`). *Définir une classe abstraite* `Comparator` *avec une fonction membre* `compare()` *qui prend deux* `Object` *et renvoie un* `boolean`*. Modifier ensuite la fonction* `select_sort` *pour qu'elle utilise une instance de cette classe (passée comme paramètre).*

On dispose maintenant d'une méthode de tri générale. Supposons par exemple qu'on ai un vecteur de `Point` avec

```
class Point {
    public int x,y;
    public Point(int x,int y) {
        this.x = x;this.y = y;
    }
    public String toString() { return "("+x+","+y+")"; }
}
```

Il suffit de dériver un comparateur qui sait comparer des points et d'appeller notre fonction de tri avec une instance de cette classe.

Question 8 (Une classe `PointComparator`). *Dériver une classe `PointComparator` qui compare des points en utilisant l'ordre lexicographique sur x et sur y.*
ex : $(2,5) < (3,4)$ *et* $(2,3) < (2,6)$

3 Structure de données : les listes

Une liste est une structure de données qui permet de représenter une séquence d'objets. Dans la séquence, chaque objet a un prédecesseur et un suivant. On accède aux éléments d'une liste en partant du premier et en passant au suivant jusqu'à arriver à l'élément voulu. Une bonne façon de modéliser une liste en objet est de dire (comme en Caml par exemple) qu'une liste est :

- soit la liste vide ;
- soit un élément suivi d'une autre liste

Question 9 (La classe abstraite `List`). *Définir une classe abstraite `List` qui a 3 fonctions membres, une qui renvoie vrai ou faux si la liste est vide ou non, une qui renvoie le premier élément (un `Object`) et une autre qui renvoie la sous-liste (éventuellement vide) constitué par tous les éléments après le premier. La première méthode s'appellera `empty()`, la seconde `head()` et la troisième `tail()`.*

Question 10 (La liste vide). *Dériver une classe `EmptyList` qui implémente la fonction `head()` en levant une exception, et la fonction `tail()` en retournant une liste vide. Utiliser le pattern Singleton pour assurer qu'on ne puisse créer qu'une seule instance de `EmptyList` (on appellera `get()` la méthode de classe permettant d'accéder à cette instance).*

Question 11 (La liste non vide). *Dériver une classe `NonEmptyList` qui stocke une référence sur un objet et une référence sur une liste. Fournir le constructeur approprié et surcharger les méthodes nécessaires.*

À ce stade, il est possible de créer des listes en combinant les constructeurs. Par exemple, la liste `"xavier" : :"sabine" : :"juliette" : :[]` est créée avec le code Java

```
List l = new NonEmptyList("xavier",
                          new NonEmptyList("sabine" ,
                                           new NonEmptyList("juliette",
                                                            EmptyList.get())));
```

Comme cette écriture est rapidement lourde, on va proposer un autre méchanisme.

Question 12 (Fonction `prepend`). *On veut définir la méthode `List prepend(Object o)` tel que `l.prepend("jean")` retourne une liste composée de `"jean"` et de `l`. Attention, `l` ne doit pas être modifiée par l'appel ! Définir cette méthode pour les 3 classes introduites précédemment.*

Question 13 (Fonction `append`). *Définir de la même façon la méthode `append()` qui construit une nouvelle liste en lui ajoutant un élément à la fin.*

Avec ces 2 nouvelles méthodes, il est désormais possible de créer facilement des listes comme dans l'exemple suivant :

```
List k = EmptyList.get().prepend("juliette").prepend("sabine").prepend("xavier");
List j = EmptyList.get().append("xavier").append("sabine").append("juliette");
```

Question 14 (Fonction `concat`). *Définir une méthode* `List concat(List k)` *qui construit une nouvelle liste en lui ajoutant une liste à la fin.*

Question 15 (Fonction `reverse`). *Définir une méthode* `List reverse()` *qui construit une nouvelle liste en inversant la liste appellante.*