TP 3 : Interfaces, types génériques. Application aux piles.

1 Les piles : Définition

Une pile est une structure de données dont les caractéristiques sont les suivantes :
- tous les éléments sont de même type ;
- le nombre d'éléments peut être éventuellement limité ;
- le seul **élément accessible est celui du sommet** de la pile : on peut le lire, l'enlever, ou poser un nouvel élément au sommet.

Dans ce TP, nous chercherons à réaliser une pile qui puisse stocker indifféremment des chaînes de caractères, des entiers, et des rationnels (comme ceux définis au projet 1). Nous utiliserons donc un *type générique*. On donne ci-dessous l'interface `Pile` qui devra être implémentée par toute classe censée représenter une pile :

L'interface `Pile.java`

```
public interface Pile<E> {
    int MAX_ELEMENTS = 100 ;              // nombre maximal d'éléments
    boolean vide() ;                      // teste si la pile est vide
    boolean pleine() ;                    // teste si la pile est pleine
    boolean peutEmpiler(E x) ;           // teste si la pile peut empiler x
    E sommet() ;                          // référence de l'objet au sommet
    E depile() ;                          // enlève et retourne l'objet au sommet
    void empile(E x) ;                    // place un objet au sommet
    void vider() ;                        // vide la pile
    int nbElements() ;                    // compte le nombre d'éléments empilés
    void deplacerUnElementVers(Pile<E> p) ;// déplace un élément de la pile courante vers p
    String toString() ;                   // affichage de tous les éléments depuis le sommet
}
```

2 Travail demandé

Q 1. Écrivez une classe `PileTableau<E>` qui implémente l'interface `Pile<E>` au moyen des attributs suivants :

```
private E [] elements ;          // les éléments contenus dans la pile
private int nbElem = 0 ;         // le nombre d'éléments dans la pile
private String nom ;             // nom de la pile : information supplémentaire propre à PileTableau
```

N'oubliez pas que pour compiler `PileTableau.java`, vous devrez avoir écrit **toutes les méthodes** de l'interface `Pile` : pour pouvoir les tester une par une, le plus simple est donc d'écrire dans un premier temps des méthodes « vides » (qui ne font rien ou retournent une valeur arbitraire), puis de compléter chaque méthode successivement.

Remarque sur les tableaux de types génériques

Attention ! Pour instancier le tableau `elements`, il est interdit d'écrire : `elements = new E[MAX_ELEMENTS]` car au moment de l'instanciation le type `E` doit être remplacé par un type réel !
Il faut donc en pratique procéder comme suit : `elements = (E[]) new Object[MAX_ELEMENTS]`

Q 2. Testez la classe `PileTableau`, en créant des instances dans lesquelles vous placerez respectivement des entiers, des chaînes de caractères et des rationnels.

Q 3. On veut maintenant que la méthode `deplacerUnElementVers` affiche le nom de la pile de départ et le nom de la pile d'arrivée. Quels problèmes se posent ? Comment procéder ?

Q 4. Modifiez ensuite la classe `Rationnel` pour qu'elle implémente l'interface `Comparable` (du package `java.lang`). On donne page suivante la description de l'interface `Comparable<T>` qui est utilisée lorsque l'on souhaite pouvoir comparer des instances d'une même classe (consultez la documentation de l'API pour voir ce qu'il en est des classes `Integer` et `String` par exemple).

java.lang.Comparable

```
public interface Comparable<T> {
  public int compareTo(T o) ;
}
```

La méthode `compareTo` retourne :
- un nombre négatif si l'instance courante est « plus petite » que l'objet `o` ;
- 0 si elle est « égale » ;
- un nombre positif si l'instance courante est « plus grande » que l'objet `o`.

Par exemple : `(new Integer(5)).compareTo(new Integer(3))` renvoie 2.

Q 5. Peut-on créer une instance de `PileTableau<Comparable>` ? Si oui, est-il possible d'y placer à la fois des entiers, des chaînes de caractères et des rationnels ?

Q 6. En recopiant le code déjà écrit pour `PileTableau`, écrivez une classe `PileTableauComp` qui ne stocke que des objets comparables.

Q 7. Écrivez dans `PileTableauComp` une méthode `public void trier()` qui range les éléments de la pile dans l'ordre croissant en partant du sommet (le plus petit au sommet, le plus grand à la base de la pile). Testez-la, par exemple avec `TestPileTableauComp.java` disponible sur Moodle. Que se passe-t-il si votre pile contient des instances de classes différentes (entiers et rationnels par exemple) ? Pourquoi ?

Complément : gestion mémoire des objets

Nous avons vu en cours que les objets sont persistants : une fois instanciés, ils restent présents dans la mémoire jusqu'à ce qu'ils ne soient plus référencés. En Java, le *garbage collector* (ou « ramasse-miettes ») se charge alors de désallouer l'espace mémoire que les objets non référencés occupent.
Par défaut, le *garbage collector* fonctionne de façon asynchrone, lorsque la machine virtuelle estime qu'il lui faut faire du nettoyage. On peut toutefois le déclencher explicitement via la méthode `System.gc()`.
Pour illustrer ce mécanisme, nous utiliserons la classe `ObjetMortel` (une variante de `ObjetNumerote`), comportant une méthode `protected void finalize()`. Cette méthode est appelée par le *garbage collector* juste avant que l'espace mémoire de l'instance ne soit désalloué. Les affichages réalisés dans `finalize` permettent donc de visualiser à quel moment l'objet a été détruit.

Q 8. Téléchargez sur Moodle les classes `ObjetMortel` et `TestMemoire1`. Dans cette dernière, on crée 10 instances dont la première seule est référencée par une variable (elle le reste donc dans toute la méthode `main` alors que les autres ne sont référencées que pendant le `println`). Expliquez l'affichage obtenu. L'objet 1 a-t-il été désalloué ? Si oui, quand ? Si non, pourquoi ?

Q 9. Essayez maintenant `TestMemoire2`. Dans cette dernière, on crée 10 000 instances non référencées par une variable. Ont-elles été désallouées ? Si oui, quand ? Si non, pourquoi ? (N'hésitez pas à vous servir des commandes unix `grep` et `wc` ou de la redirection de la sortie standard !)

Q 10. Essayez maintenant `TestMemoire3`. Dans cette dernière, on crée 100 000 instances non référencées par une variable. Ont-elles été désallouées durant l'exécution de la méthode `main` ? Si oui, quand ? Si non, pourquoi ?