INTERFACES ET TEMPLATES

Il y a une multitude de tris dans la littérature. Dans le cadre de ce TP nous allons voir le tri pair-impair (ou plus précisément tri pair-impair par transposition). C'est un algorithme de tri assez simple qui est basé sur le tri à bulles. Il opère en comparant tous les couples d'éléments aux positions paires et impaires consécutives dans une liste et, si un couple est dans le mauvais ordre (le premier élément est supérieur au second), il en échange les deux éléments. Se qui donne en Java :

```
public void triPairImpair(int [] tab){
  boolean done = false;
  while(done == false){
    done = true;

    // comparaisons impaires-paires :
    for(int i = 1 ; i<tab.length-1 ; i+=2){
      if(tab[i] > tab[i+1]){
        int tmp = tab[i];
        tab[i] = tab[i+1];
        tab[i+1] = tmp;
        done = false;
      }
    }

    // comparaisons paires-impaires :
    for(int i = 0 ; i<tab.length-1 ; i+=2){
      if(tab[i] > tab[i+1]){
        int tmp = tab[i];
        tab[i] = tab[i+1];
        tab[i+1] = tmp;
        done = false;
      }
    }
  }
}
```

Comme nous pouvons le voir cette implémentation ne fonctionne que sur des tableaux d'entiers. Ainsi, si nous souhaitons trier des chaînes de caractères nous devons écrire un algorithme dédié. Afin d'éviter cela, nous allons utiliser une interface de manière à utiliser tout type de données (muni d'une relation d'ordre) sans avoir à réécrire l'algorithme à chaque fois. Pour cela on va supposer définie comme suit, l'interface `Triable` :

```
public interface Triable {
  // échange les éléments en positions i et j
  void echange(int i, int j);

  // retourne vrai si l'élément en i est plus grand que l'élément en j
  int plusGrand(int i, int j);

  // retourne le nomnre d'éléments
  int nbElement();

  // toString
  String toString();
}
```

Les objets des classes implémentant cette interface devront donc implémenter ces méthodes.

Question 1. Écrivez la méthode `void triPairImpair(Triable t)` qui met en oeuvre le tri pair-impair pour les objets `Triables`.

Question 2. Tester votre méthode avec un objet de type `EntierTriable` qui permet de manipuler des entiers.

Question 3. Tester votre méthode avec un objet de type `StringTriable` qui permet de manipuler des chaînes de caractères.

Le problème avec la solution décrite précédemment est que nous sommes obligés d'avoir la gestion du tableau dans une classe dédiée. En fait, nous aurions sûrement préféré réaliser notre tri sur un tableau quelconque ! Plus précisément, nous aurions aimé avoir des objets comparables (la propriété est sur les objets du tableau et pas sur le tableau). Ainsi nous aimerions avoir comme entête pour la fonction `triPairImpair` quelque chose comme cela :

```
public void triPairImpair(ObjetComparable [] tab){
  // le code
}
```

Par conséquent, notre interface `ObjetComparable` implemente une seule méthode permettant de comparer deux à deux les éléments implémentant cette dernière.

```
public interface ObjetComparable{
  int compareTo(ObjetComparable t);
}
```

À présent supposons que nous souhaitons trier des entiers. Nous voudrions implémenter l'interface `ObjetComparable`. Pour cela, il suffit de créer une nouvelle classe pour faire le travail . . .

```
public class EntierComparable implements ObjetComparable{
  int val;
  public EntierComparable(int v){val = v;}
  public int compareTo(ObjetComparable t){
    // ??????
  }
}
```

Nous voyons clairement qu'il est impossible d'implémenter la fonction `compareTo` car nous ne pouvons par comparer l'objet courant avec un objet de type `ObjetComparable` (nous ne connaissons pas sa définition) ! Afin de palier ce problème nous allons rapidement introduire la notion de `template` en Java. Les templates permettent de produire des classes génériques. En fait, la généricité permet de définir des classes, ou des méthodes paramétrées par une ou plusieurs autres classes. Considérons un exemple simple, supposons que nous souhaitons créer un objet pour la gestion de couples d'objets identiques. Nous avons donc :

```
public class Couple{
  Object v1, v2;
  public Couple (Object a, Object b){v1 = a; v2 = b;}

  public Object getPremierElt(){return v1;}
  public Object getSecondElt(){return v2;}
}
```

Dans la configuration actuelle de la classe `Couple` rien n'empêche de créer des couples hétérogène !

```
Couple c = new Couple("toto", 51);
```

Dans ce cas il n'y aurait pas d'erreur à l'exécution du programme. Cependant, nous pouvons voir que notre solution ne satisfait pas entièrement l'objectif que nous nous sommes fixé. Pour résoudre ce problème, nous allons paramétrer la classe avec un type générique que nous ne connaissons pas à la création de la classe. Pour cela il suffit d'utiliser la notation `< >`. Par exemple, nous avons :

```
public class Couple<T>{
  T v1, v2;
  public Couple (T a, T b){v1 = a; v2 = b;}

  public T getPremierElt(){return v1;}
  public T getSecondElt(){return v2;}
}
```

C'est à la création de l'objet que nous allons instancier le type de `T`. Donc, si nous voulons une paire d'entiers il suffit d'écrire :

```
Couple<int> c = new Couple<int>(1664, 51);
```

Dans cette configuration si l'utilisateur ne donne pas de paramètres du bon type il y aura une erreur dès la phase de "compilation".

```
Couple<int> c = new Couple<int>("toto", 51); // pas content !!!
```

Dans le contexte qui nous intéresse, c'est-à-dire la gestion d'une classe pour le tri d'objets quelconques, nous avons l'interface comparable qui devient :

```
public interface ObjetComparable<T>{
  int compareTo(T t);
}
```

Puisque le type de `T` peut être quelconque nous avons le droit de l'instancier avec le nom de classe qui implemente l'interface `ObjetComparable`. Nous avons donc :

```
public class EntierComparable implements ObjetComparable<EntierComparable>{
  // du code
}
```

Question 4. Remplissez le code nécessaire à la classe `EntierComparable` ci-dessus.

Question 5. Proposez une classe `StringComparable` qui permet de comparer deux objets de type chaîne de caractères.

Pour conclure, nous allons utiliser la généricité pour l'implantation de la fonction de tri `triPairImpair`. En effet, en Java il est possible de définir une méthode générique dans une classe de la façon suivante :

```
public class Test {
  public <T> void affiche(Couple<T> p){
    System.out.println(p);
  }
  public static void main(String [] a){
    Paire<String> ps = new Couple<String>("un", "deux");
    Test x = new Test();
    x.affiche(ps);
  }
}
```

Il est même possible de spécifier des propriétés sur ce que doit être l'objet `T`. Par exemple, nous pouvons vouloir que `T` hérite d'une certaine classe ou implémente une certaine interface. Pour cela nous allons simplement utiliser le mot clef `extends` de la manière suivante :

```
public class Test {
  public <T extends ObjetComparable<T>> void compare(T p1, T p2){
    System.out.println(p1.compareTo(p2));
  }
  public static void main(String [] a){
    EntierComparable e1 = new EntierComparable(2);
    EntierComparable e2 = new EntierComparable(3);

    Test x = new Test();
    x.compare(e1, e2);
  }
}
```

Question 6. Écrivez la méthode `triPairImpair` générique qui prend en paramètre un tableau quelconque d'objets qui implémentant l'interface `ObjetComparable`.