Université de Paris **Programmation Orientée Objet**
L2 **Année 2020-2021**

TD - Séance n°10

Généricité

Exercice 1 *Paires génériques*

On définit l'interface générique suivante

```
public interface AUneClef<K> {
    K getClef();
}
```

Écrivez une classe générique `Paire<K, V>` qui implémente `AUneClef<K>`. Un objet de `Paire<K, V>` contiendra un objet de type `K` (la clef) et un objet de type `V` (la valeur).

En plus d'un constructeur, on écrira les méthodes `getValeur()` et `toString()`, ainsi qu'une méthode `renverse` qui transforme une paire (clef, valeur) en une paire (valeur, clef).

Exercice 2 *Généricité et héritage*

On rappelle qu'il existe en Java une classe paramétrique Vector<E>, qui correspond à un tableau potentiellement extensible, et qui a en particulier les méthodes suivantes :

boolean add(E e)
E get(**int** index)

On suppose avoir défini deux classes A et B, chacune avec un constructeur 0-aire, tel que B hérite de A.

— Le code suivant ne compile pas :

```
Vector<A> l;
Vector<B> m = new Vector<B>();
l = m;
```

Que peut on en déduire pour la relation d'héritage entre Vector<A> et Vector<B> ? Donner un exemple illustrant la nécessité que ce code soit rejeté par le compilateur.

— En revanche, le code suivant compile :

```
Vector<? extends A> l;
l = new Vector<B>();
```

Que peut on en déduire pour la relation d'héritage entre Vector<? **extends** A> et Vector<B> ?

— On veut écrire une méthode qui prend en argument n'importe quel objet de type C, et renvoie un élément de type C, où C est une classe qui hérite de A. Quelle doit être la signature d'une telle méthode ?

— On considère les deux morceaux de code suivants : le code ci-dessous ne compile pas.

```
Vector<A> m = new Vector<A>();
m.add(new A());
Vector<? extends A> l = m;
l.add(new A());
```

Le code ci-dessous compile.

```
        Vector<A> m = new Vector<A>();
2       m.add(new A());
        Vector<? extends A> l = m;
4       A a = l.get(0);
```

Expliquer pourquoi.

— On peut utiliser ? **super** pour indiquer l'ensemble des classes dont une certaine classe hérite, ainsi que l'ensemble des interfaces qu'elle implémente (borne supérieure). On considère la méthode définie par :

```
static void affiche(Vector<? super A> vector){
2   for(Object v : vector)
      System.out.print(v.toString());
4 }
```

Lesquelles de ces instructions compilent ?

affiche(**new** Vector<Object>());
affiche(**new** Vector<A>());
affiche(**new** Vector<B>());

— Le code suivant compile :

```
        Vector<? super A> l2 = new Vector<A>();
2       l2.add(new A());
```

Tandis que le code suivant ne compile pas :

```
    Vector<? super A> l2 = new Vector<A>();
2   A a = l2.get(0);
```

Pourquoi ?

Exercice 3 On veut écrire un certain nombre de méthodes statiques, spécifiées ci-dessous, dans une classe Test qui n'est pas paramétrique. Les en-têtes des méthodes sont volontairement incomplets. On veut qu'on puisse en particulier leur passer comme argument un `Vector` de `Paire<Integer, String>`, et qu'ils permettent la plus grande généralité possible.

— Réécrire la méthode affiche du deuxième exercice, de telle sorte qu'elle puisse être appliqué à tout élément de type Vector<C>, pour n'importe quelle classe C.

— Écrivez une méthode `compteElement(K clef, ...)` qui prend en argument une clef et un `Vector<>` d'éléments dont la classe implémente `AUneClef<K>` et qui retourne le nombre d'éléments du vecteur qui ont `clef` comme clef.

— Écrivez une méthode `double sommeClefs(...)` qui prendra en argument un Vector d'éléments dont la classe implémente `AUneClef<K>` pour n'importe quelle classe (ou interface) `K` étendant (ou implémentant) `Number`.

— Écrivez une méthode `convertit(...)` qui prend un `Vector` d'éléments de type `T` et les transfère tous dans un `Vector` d'éléments de type `U`. Pour que ce soit possible, il faut bien sûr que `T` étende ou implémente `U`.

— Écrivez `ajoute(K clef, V val, Vector <? super Paire<K, V>> tab)` qui ajoute une paire (clef, val) au `Vector` donné en argument. Donner des exemples de type acceptés par cette méthode.

Exercice 4 On veut écrire une classe `Pile<T>` correspondant à une pile générique. On veut représenter l'ensemble des éléments empilés par un tableau. Comme il est impossible d'écrire `new T[10]`, on est obligé d'utiliser un tableau de `Object`. On notera que cette implémentation

suppose l'utilisation d'un *cast* générique ce qui provoquera lors de la compilation avec l'option `-Xlint` des avertissements de type`unchecked cast`. Une pile est une structure de type LIFO (Last In First Out). Écrire les méthodes et constructeur suivants :

- `public Pile(int taille)`
- `public T depile()`
- `public void empile(T e)`
- `public T getSommet()`
- `public boolean estVide()`
- `public boolean estPleine()`

On pensera à gérer les cas extremum (lorsqu'on dépile une pile vide, par exemple), en levant des exceptions appropriées.