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

TD - Séance n°10 - Correction

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).

Correction :

```
public class Paire<K, V> implements AUneClef<K> {
    private K clef;
    private V valeur;

    public Paire(K clef, V valeur) {
        this.clef = clef;
        this.valeur = valeur;
    }

    public K getClef() {
        return this.clef;
    }

    public V getValeur() {
        return this.valeur;
    }

    public String toString() {
        return "Paire : " + clef + " -> " + valeur;
    }

    public Paire<V, K> renverse() {
        return new Paire<V, K>(this.valeur, this.clef);
    }
}
```

Il n'est pas nécéssaire d'utiliser `toString()` sur `clef` et `valeur`, elle est appellée automatiquement. Aussi, dans `Objet` il y a toujours une implémentation par défaut, qui va renvoyer quelque chose comme `NomDeLaClasse@f6f4d33`.

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.

Correction : On en déduit que Vector<B> n'hérite pas de Vector<A>. Si c'était le cas, on pourrait alors faire l'opération suivante :

```
l.add(new A());
```

mais le vecteur l contiendrait alors des éléments qui ne sont pas des B (alors que l est une instance de Vector<B>).

— 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> ?

Correction : `Vector<? extends A>` est un super-type de `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 ?

Correction :

```
<C extends A> C methode(C objet) {...}
```

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

Expliquer pourquoi.

Correction : À la ligne 4 le compilateur considère l comme de type Vector<T> avec T un type quelconque qui hérite de A (capture du *wild card*).
Dans le premier cas, le compilateur attend donc que add prenne un argument de type T. T pourrait être un sous-type strict de A donc add ne doit pas prendre un argument de type déclaré A.
Dans le deuxième cas, le type de retour de get est T. Or il est implicitement converti vers le type plus haut A (*upcast*).

— 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>());

Correction : Les deux premières instructions compilent car Object et A sont des super-types de A. La troisième instruction ne compile pas car ce n'est pas le cas de 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 ?

Correction : À la ligne 2 le compilateur considère l2 comme de type Vector<T> avec T un type quelconque dont A hérite.
Dans le premier cas, le compilateur attend donc que add prenne un argument de type T. Ici on a un argument de type A, il est implicitement converti vers le type plus haut T (*upcast*), donc il n'y a pas d'erreur.
Dans le deuxième cas, le type de retour de get est T. Or on ne peut le convertir implicitement vers le type plus bas A (pas de *downcast* implicite).

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.

Correction :

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

mais puisque uniquement `toString` est invoquée sur les elements du `Vector`, nous n'avons pas besoin de les lire comme objets de type `C`, `Object` suffit. Alors on peut aussi avoir

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

— 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.

Correction :

```
static <K, T extends AUneClef<K>>
int compteElements(K clef, Vector<T> vecteur) {
    int compteur = 0;
    for (T e : vecteur) {
        if (e.getClef().equals(clef))
            compteur++;
    }
    return compteur;
}
```

Mais il n'y a pas besoin de donner un nom au type T, on peut aussi avoir :

```
static <K> int compteElement(K clef, Vector<? extends AUneClef<K>> v) {
    int compteur = 0;
    for(AUneClef<K> elem : v) {
        if (clef.equals(elem.getClef()))
            compteur++;
    }
    return compteur;
}
```

— 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`.

Correction :

```
static <K extends Number, T extends AUneClef<K>>
double sommeClefs(Vector<T> vecteur) {
    double somme = 0;
    for (T e : vecteur)
        somme += e.getClef().doubleValue();
    return somme;
}
```

ou bien, si on ne veut pas devoir déclarer les types génériques :

```
static
double sommeClefs(Vector<? extends AUneClef<? extends Number>> v) {
    // Remarque : pas besoin de nommer le type des clef,
    // il n'est en relation avec aucun autre type

    double somme = 0;

    // Par contre, comme on n'a pas nomme les types, il faut repeter
    // ici '? extends Number' :
    for (AUneClef<? extends Number> elem : v) {
        // Remarque : apres capture, le type des elements de v
        // est AUneClef<C> pour un certain type C qui etend
        // Number, toutefois AUneClef<Number> n'est pas un
        // super-type de AUneClef<C> donc on ne peut pas utiliser
        // une variable AUneClef<Number> pour lire les elements
        // du tableau. Il faut utiliser une variable de type
```

```
17          // AUneClef<? extends Number> (qui est un super-type de
            // AUneClef<C> pour tout C qui etend Number).
19          somme += elem.getClef().doubleValue();
        }
21      return somme;
    }
```

— 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`.

Correction :

```
static <U, T extends U> Vector<U> convertit(Vector<T> s) {
2       // Remarque : on doit nommer les types pour exprimer
        // la relation entre eux.
4       Vector<U> v = new Vector<U>();
        for (T e : s)
6           v.add(e);
        return v;
8   }
```

— 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.

Correction :

```
static <K, V> void
2   ajoute(K clef, V val, Vector<? super Paire<K, V>> tab) {
        tab.add(new Paire<K, V>(clef, val));
4   }
```

Exemples de types acceptés par `ajoute` :

```
Vector<Paire<Integer,String>> v1 = new Vector<>();
2   ajoute (1, "un", v1);

4   Vector<Paire<Integer,Double>> v2 = new Vector<>();
    ajoute (1, 1.2, v2);

    Vector<AUneClef<Integer>> v3 = new Vector<>();
8   ajoute (1, "un", v3);
    ajoute (1, 1.2, v3);

    Vector<Object> v4 = new Vector<>();
12  ajoute(1, "un", v4);
    ajoute("deux", 1.2, v4);
```

Pour le troisième exemple, le type des clefs est détérminé à `Integer` par `AUneClef<Integer>`, par contre le type des valeurs n'est pas détérminé au moment de la déclaration de `v3`. Le type `V` des valeurs est détérminé par le deuxième paramètre donné à `ajoute`.

Pour le quatrième exemple, ni `K` ni `V` sont détérminés au moment de la déclaration de `v4`. Les deux sont détérminés par les deux premiers paramètres donnés à `ajoute`.

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.

Correction : cf. fichiers Pile.java PileVideException.java et PilePleineException.java