TD et TP de Compléments en Programmation Orientée Objet n° 12 : Généricité et *wildcards*

Exercice 1 :

Soit le code suivant.

```
1  class Base { }
2  class Derive extends Base { }
3  class G<T extends Base, U> { public T a; public U b; }
```

Ci-dessous, plusieurs spécialisations du type `G`.

1. Certaines ne peuvent exister, dites lesquelles.
2. Des conversions sont autorisées entre les types restants. Quelles sont-elles ? Donnez-les sous forme d'un diagramme.

Voici les types :

```
G<Object, Object>                    G<Object, Base>
G<Base, Object>                      G<Derive, Object>
G<? extends Object, ? extends Object>  G<? extends Object, ? extends Base>
G<?, ?>                              G<? extends Derive, ? extends Object>
G<? extends Base, ? extends Object>  G<? extends Base, ? extends Derive>
G<? super Object, ? super Object>    G<? super Object, ? super Base>
G<? super Base, ? super Object>      G<? super Base, ? super Derive>
```

Exercice 2 :

1. Y a-t-il une différence entre les signatures des deux méthodes suivantes ? (laquelle ?)
   — `void dupliquePremierA(List<?> l){}`
   — et `<T> void dupliquePremierB(List<T> l){}`
2. Écrivez des programmes appelant ces deux versions de méthode sur des arguments `l` différents. Existe-t-il des cas où l'une des signatures fonctionne mais pas l'autre ?
3. Programmez ces deux méthodes. Ce qu'elles doivent faire : si la liste `l` est non vide, insérer le premier élément une seconde fois dans la liste.
   Est-ce que vous y arrivez dans les deux cas ? Pourquoi ?
4. Implémentez la méthode que vous n'avez pas réussi à implémenter à la question précédente par un appel à l'autre méthode. Cette fois-ci, est-ce que ça marche ?

Exercice 3 : Paires

Qui n'a jamais voulu renvoyer deux objets différents avec la même fonction ?

1. Implémenter une classe (doublement) générique `Paire<X,Y>` qui a deux attributs publics `gauche` et `droite`, leurs getteurs et setteurs respectifs et un constructeur, prenant un paramètre pour chaque attribut.
2. Application : programmez une méthode

```
1  static <U extends Number, V extends Number> Paire<Double, Double> somme(List<Paire<U, V>> aSommer);
```

qui retourne une paire dont l'élément gauche est la somme des éléments gauches de `aSommer` et l'élément droit la somme de ses éléments droits (pour une raison technique, le résultat est typé `Paire<Double, Double>`, mais quelle est cette raison ?).

3. — Écrivez la déclaration d'une variable à laquelle on peut affecter toute paire de nombres de type `Paire<Number, Number>` (contenant donc des instances de `Number` où d'un de ses sous-types).
   — Écrivez la déclaration d'une variable à laquelle on peut affecter toute paire du type `Paire<M, N>` où `M <: Number` et `N <: Number`.
   — Expliquez la différence entre les deux déclarations précédentes.
4. — Si on écrit `Paire<? extends Number, ? extends Number> p1 = new Paire<Integer, Integer>(15, 12)`, quelles méthodes de la classe `Paire` seront inutiles, appelées sur l'expression `p` ? Lesquelles seront utiles ? (discutez sur les signatures)
   — Si on écrit `Paire<? super Integer, ? super Integer> p2 = new Paire<Number, Number>(15, 12)`, quelles méthodes de la classe `Paire` seront inutiles, appelées sur l'expression `p` ? Lesquelles seront utiles ?
   — Dans les 2 cas précédents, peut-on, sans *cast*, accéder aux attributs de `p1` ou `p2` en lecture (essayez de copier leurs valeurs dans une variable déclarée avec un type de nombre quelconque) ? et en écriture (essayez de leur affecter une valeur autre que `null`) ?
   — Du coup, supposons qu'on écrive une version immuable de `Paire` (ou n'importe quelle classe générique immuable), et qu'on veuille en affecter une instance à une variable (`Paire<XXX, XXX> p = new Paire<A, B>();`). Pour que cette variable soit utile, doit-elle plutôt être déclarée avec un type comme celui de `p1` ou comme celui de `p2` ?

Exercice 4 : Streams maison

Écrivez une interface `MyStream<T>` servant réaliser des opérations d'agrégation paresseuses sur des `List<T>` sans utiliser les *streams* de Java 8.

Fonctionnalités :

— opérations intermédiaires à implémenter : `<U> MyStream<U> map(Function<T, U> f)`, `MyStream<T> skip(int n)` et `MyStream<T> filter(Predicate<T> p)`.
— opération terminale à implémenter : `List<T> toList()` (directement en tant que méthode de l'interface `MyStream<T>`, sans chercher à programmer l'équivalent de `Collector`)
— méthode `static <T> MonStream<T> MyStream.makeStream(List<T> liste)` pour créer un `MyStream<T>` depuis une `List<T>`.

Les objets retournés par les méthodes `map`, `skip`, `filter` et `makeStream` seront des nouvelles instances de classes (a priori différentes : une par opération) implémentant toutes `MyStream<T>`.

On insiste bien sur le coté paresseux : un objet de type `MyStream<T>` ne contient pas le résultat de l'opération effectuée. Le résultat concret n'est calculé que lors de l'appel à `toList()`.

Vous pouvez prendre exemple sur l'interface suivante (`MyStream` munie de l'opération `limit`), à laquelle vous ajouterez les opérations demandées :

```
1  public interface MyStream<T> {
2     List<T> toList();
3
4     public class LimitStream<T> implements MyStream<T> {
5        private final MyStream<T> source;
6        private final int limit;
7
```

```
8         public LimitStream(MyStream<T> source, int limit) {
9             this.source = source;
10            this.limit = limit;
11        }
12
13        @Override
14        public List<T> toList() {
15            return new ArrayList<>(source.toList().subList(0, limit));
16        }
17    }
18
19    default MyStream<T> limit(int n) {
20        return new LimitStream<>(this, n);
21    }
22 }
```

Note : `subList()` retourne une vue de la liste sur laquelle on l'appelle, et non une copie (partielle) indépendante. C'est pour cela qu'on en fait une copie (`new ArrayList<>(...)`), afin que les modifications à la liste obtenue à la fin soient indépendantes de celles sur la liste initiale.

Bonus : optimiser le traitement de telle sorte que la copie de liste n'ait lieu qu'une seule fois pour un *pipeline* donné (Au début de la réduction ; ensuite on s'assure grâce à un attribut booléen et une exception qu'on ne puisse appeler qu'une seule fois une méthode sur une instance de `MyStream` donnée. Cela empêche que plusieurs références vers l'unique copie de la liste ne "s'échappent dans la nature".).