# TD et TP de Compléments en Programmation Orientée Objet n° 8 : 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 : 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` ?