TD et TP n° 10 : Généricité et *wildcards*
(Correction)

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>
```

Correction : `? extends T` : le type argument de la référence étend `T` et `? super T` : le type argument de la référence est étendu par `T`. À partir de là on en déduit les solutions.

1. Les types interdits sont `G<Object, Object>`, `G<Object, Base>`, `G<? super Object, ? super Object>` et `G<? super Object, ? super Base>`
2. Si on pose
   1. `G<Base,Object>`
   2. `G<Derive,Object>`
   3. `G<? extends Object ,? extends Object>`
   4. `G<? extends Object ,? extends Base>`
   5. `G<?,?>`
   6. `G<? extends Derive,? extends Object>`
   7. `G<? extends Base ,? extends Object>`
   8. `G<? extends Base ,? extends Derive>`
   9. `G<? super Base ,? super Object>`
   10. `G<? super Base ,? super Derive>`

   5 et 3 représentent un même type.

   Les conversions possibles sont :

   $1 \rightarrow 8 \rightarrow 4 \rightarrow 3, 5$
   $8 \rightarrow 7 \rightarrow 3, 5$
   $2 \rightarrow 6 \rightarrow 7$
   $1 \rightarrow 9 \rightarrow 10 \rightarrow 3, 5$

   Remarque :

- 9 est le type `G<? super Base, Object>`
- 4 est le type `G<?, ? extends Base>`
- 6 est le type `G<? extends Derive, ?>`

Exercice 2 :

1. Y a-t-il une différence entre les signatures des deux méthodes suivantes ? (laquelle ?)
   - `static void dupliquePremierA(List<?> l){}`
   - et `static <T> void dupliquePremierB(List<T> l){}`

   Correction : Pour ce qui est de l'utilisation des deux méthodes, on ne verra aucune différence : de l'extérieur, ces deux signatures sont équivalentes.
   La première dit qu'elle prend une liste de n'importe quoi et qu'elle ne retourne rien. La seconde dit que pour tout type `T`, elle peut prendre une list de ce type et ne rien retourner. Ces deux énoncés sont logiquement équivalents.
   Néanmoins, dans la deuxème signature, on a donné un nom au paramètre de type. Pour l'extérieur, c'est une « variable muette », mais ce nom de type est utilisable (= dénotable) dans le corps de la méthode, ce qui permet de faire plus de choses que les éventuelles captures du *wildcard* dans l'autre cas.

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 ?

   Correction : Non. Voir plus haut.

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 ?

   Correction : Une implémentation serait :

```java
static void dupliquePremierA(List<?> l) {
   if (!l.isEmpty()) l.add(l.get(0));
}

static <T> void dupliquePremierB(List<T> l) {
   if (!l.isEmpty()) l.add(l.get(0));
}
```

   Ce corps ne compile pas avec la première signature, mais seulement la deuxième.
   Explication : dans la première méthode, `l.get(0)` retourne une certaine capture de `?` et `l.add()` prend un argument qui est aussi une (autre) capture de `?`. Ces deux captures sont des types sans lien de parenté, donc l'analyse de bon typage échoue.
   Normalement, le paramètre de type de `l` n'a pas changé entre temps, mais le compilateur ne sait pas le détecter, donc une nouvelle capture est faite à chaque occurrence. Remarque : il est en fait tout à fait possible d'affecter à `l` une liste d'éléments d'un autre type entre temps !
   Dans la seconde méthode, `l.get(0)` retourne `T` et `l.add()` accepte (le même) `T`. Donc là ça marche (remarque : même si on affectait, entre les deux accès à la variable `l`, une autre liste à `l`, ce serait nécessairement une liste de `T`).

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 ?

Correction : Il suffit d'écrire :

```
1 static void dupliquePremierA(List<?> l) {
2    dupliquePremierB(l);
3 }
```

Ici, ça compile bien. En effet : lors de l'appel interne (ligne 2), une (seule) capture de `?` est faite. Elle est immédiatement « affectée » à `T` pour l'appel de `dupliquePremierB()`.
Cette technique, consistant à utiliser une méthode auxiliaire (privée) qui introduit un paramètre de type, est souvent utilisée quand on implémente une interface qui impose d'utiliser des *wildcards*, mais qu'on a besoin de nommer ou fixer un des paramètres pour pouvoir programmer la méthode (on ne peut pas redéfinir une méthode ayant la signature de `dupliquePremierA()` par une méthode ayant la signature de `dupliquePremierB()`).

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.

Correction :

```
1  public class Paire<X, Y> {
2     public X gauche;
3     public Y droite;
4
5     public Paire(X gauche, Y droite) {
6        this.gauche = gauche;
7        this.droite = droite;
8     }
9
10    public X getGauche() { return gauche; }
11    public Y getDroite() { return droite; }
12    public void setGauche(X gauche) { this.gauche = gauche; }
13    public void setDroite(Y droite) { this.droite = droite; }
14 }
```

Remarque : cette classe ne correspond pas aux pratiques habituelles. En général, les attributs seraient privés. Dans de rares cas (tuple utilisé localement, en privé), on simplifie en utilisant des attributs publics, mais dans ce cas on ne mettrait pas de getteurs et setteurs.
Ici, la classe `Paire<X, Y>` est programmée ainsi dans le seul but de pouvoir répondre aux questions d'après.

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 ?).

Correction :

```
1   static <U extends Number, V extends Number> Paire<Double, Double>
        somme(List<Paire<U, V>> aSommer) {
2      double sommeGauche = 0, sommeDroite = 0;
3      for (Paire<U, V> p : aSommer) {
4         sommeGauche += p.gauche.doubleValue();
5         sommeDroite += p.droite.doubleValue();
6      }
7      return new Paire<>(sommeGauche, sommeDroite);
8   }
```

Le type de retour est `Paire<Double, Double>` au lieu de `Paire<U, V>` pour la raison suivante :

- soit on décide de faire les calculs intermédiaires dans des acculumateurs de type `U` et `V`, mais à ce moment-là, on on ne sait pas comment initialiser les accumulateurs : via une méthode non statique, ce n'est pas possible car on ne connait a priori pas d'instance de `U` et `V` ; via une méthode statique (de `Number` ou de ses sous-classes), ce n'est pas possible car on ne peut pas savoir quelle méthode choisir : ce sont des méthodes différentes pour générer des valeurs chaque sous-type de `Number`, qu'il faudrait donc choisir à l'exécution en vérifiant une condition sur `U` et `V`, ce qui n'est pas possible à cause de l'effacement de type.
- soit on fait les calculs intermédiaires avec un type primitif (**`double`** par exemple), mais on est coincé quand on doit retransformer le résultat en `U` et `V` (pour les mêmes raisons que ci-dessus). On aurait pu essayer de se reposer sur l'autoboxing, mais ça ne marche que si le compilateur peut connaitre le type cible exact (ce n'est pas le cas vu que le type n'est qu'une variable).

Une façon de contourner serait de restreindre la méthode aux listes non vides. À ce moment-là, il suffit d'initialiser les accumulateurs avec le premier élément.
Une autre façon consisterait à ajouter deux paramètres de types `U` et `V` destinés à recevoir les zéros des deux types et initialiser ainsi les deux accumulateurs.

3. – Écrivez le type d'une paire pouvant contenir, à gauche comme à droite, tout objet instance de `Number` ou de l'un de ses sous-types (et seulement cela).

   Correction : `Paire<Number, Number>` (en effet, avec `p` déclaré avec ce type, on peut ensuite faire par exemple `p = new Paire<Number` Number>(5L, 12.),).

   – Écrivez le type d'une variable à laquelle on peut affecter n'importe quelle paire du type `Paire<M, N>` où `M` <: `Number` et `N` <: `Number`.

   Correction : `Paire<? extends Number, ? extends Number> p;`

   – Expliquez la différence entre les deux déclarations précédentes.

   Correction : La différence : dans le premier cas, une fois qu'on affecte un objet à `p`, cet objet, via les méthodes `set` appelées sur `p`, peut se voir affecter à sa gauche ou à sa droite n'importe quelle instance de `Number`.

Dans le deuxième cas, si on veut modifier cet objet via la variable `p`, on ne pourra jamais qu'affecter la valeur `null` à ses attributs : `null` est la seule valeur qui est garantie appartenir à tous les sous-types de `Number`. L'accès en lecture, lui, fonctionnera bien (on sait qu'on récupère des instances, directes ou indirectes, de `Number`).
"Normalement", l'objet affecté à `p` sera une paire dont les attributs gauche et droite ne peuvent contenir que des instances des 2 sous-types de `Number` choisis, une fois pour toute, lors de l'instanciation de la paire.
Cependant, cela n'est vrai que "normalement". En effet, l'objet qui existe dans la JVM n'a, lui, aucune limitation, à cause de l'effacement de type. Il suffit donc qu'un `Pair<Float, Short>` ait été casté en `Pair<Short, Float>` au préalable pour que ça ne tienne plus, ce qui heureusement provoque normalement un avertissement du compilateur !

4. – Si on écrit

```
1 Paire<? extends Number, ? extends Number> p1 = new Paire<Integer, Integer>(15, 12);
```

quelles méthodes, appelables sur `p1`, voient leur utilité fortement réduite, par rapport à une déclaration du type `Paire<Integer, Integer> p1 = ...` ? Lesquelles restent assez utiles ? (discutez sur les signatures)

Correction : Méthodes peu utiles : les setteurs. En effet, on ne pourra que leur passer la valeur `null`, qui appartient à tous les types référence (voir corrigé question précédente). De façon générale, toute méthode prenant `U` ou `V` en paramètre aura ce problème (utilisation en position contravariante, alors que les paramètres sont bornés par le haut).
Méthodes utiles : les getteurs (et toute méthode retournant `U` ou `V`). En effet, on sait qu'ils retournent des instances d'un sous-type de `Number`... donc des instances de `Number`.

– Si on écrit

```
1 Paire<? super Integer, ? super Integer> p2 = new Paire<Number, Number>(15, 12)
```

quelles méthodes de la classe `Paire` voient leur utilité réduite ? Lesquelles restent utiles ?

Correction : Méthodes peu utiles : les getteurs. En effet, on aura aucune information sur leur type de retour (le seul type contenant tous les supertypes de `Integer` est `Object`). Cela dit, cela n'empêche pas, par exemple de faire `System.out.println(p2.getDroite())` et d'avoir un affichage exploitable, car la méthode `toString` est déclarée dans `Object` et la liaison dynamique assure que c'est la bonne version qui est appelée, même si le compilateur connait mal le type de retour de cet appel à `getDroite`.
De façon générale, toute méthode retournant `U` ou `V` aura ce problème (utilisation en position covariante, alors que les paramètres sont bornés par le bas).
Méthodes utiles : les setteurs (et toute méthode prenant `U` ou `V` en paramètre). En effet, on sait qu'ils prennent des instances d'un supertype de `Integer` (même si on ne sait pas lequel), donc, en particulier, toute instance de `Integer` est acceptée.

– 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`) ?

Correction : Du point de vue des vérifications de type, les accès aux attributs en lecture se comportent comme les getteurs, et ceux en écriture comme les setteurs. Ainsi, dans le premier cas (`extends`), seuls les accès en lecture fonctionnent bien, alors que dans le second cas, ce sont les accès en écriture (`super`).

– 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>(x, y);`). Pour que cette variable soit utile, doit-elle plutôt être déclarée avec un type comme celui de `p1` ou comme celui de `p2` ?

Correction : Les accès à un objet de type immuable se font en lecture seule. Or dans le cas de la paire, les paramètres de type servent à typer le contenu de la paire, donc ils ne vont apparaître qu'en position covariante (retour de méthode). Donc une déclaration utile pour une variable générique de paire immuable utilisera des *wildcards* bornés par le haut (mot-clé `extends`), comme `p1`.

Exercice 4 : Streams maison

On veut écrire 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.

À faire :

1. Avant de faire les autres méthodes, réécrivez `limit` en utilisant une lambda-expression au lieu de la classe intermédiaire explicite `LimitStream`.
2. Implémentez les autres méthodes (avec la technique de votre choix).
3. **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".).

Correction : Version non-optimisée (sans le bonus), utilisant systématiquement des lambda-expressions plutôt que des classes explicites (regardez pour l'opération `limit`, la différence avec l'exemple de l'énoncé : la concision est sans comparaison !) :

```
1  package streams;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.function.Function;
6
7  public interface MyStream<T> {
8    static <T> MyStream<T> makeStream(List<T> sourceL) {
9      return () -> new ArrayList<>(sourceL);
10   }
11
12   List<T> toList();
13
14   default MyStream<T> limit(int limitNum) {
15     return () -> new ArrayList<>(toList().subList(0, limitNum));
16   }
17
18   default MyStream<T> skip(int skipNum) {
19     return () -> {
20       List<T> sourceL = MyStream.this.toList();
21       return new ArrayList<>(sourceL.subList(skipNum, sourceL.size()));
22     };
23   }
24
25   default <U> MyStream<U> map(Function<T, U> ope) {
26     return () -> {
27       List<T> sourceL = MyStream.this.toList();
28       List<U> targetL = new ArrayList<>(sourceL.size());
29       for (T x : sourceL) targetL.add(ope.apply(x));
30       return targetL;
```

```
31          };
32      }
33  }
```

Version optimisée qui ne fait la copie défensive qu'une seule fois dans la réduction (ici on n'utilise pas un booléen, contrairement à ce que l'énoncé suggère, mais on ajoute une méthode auxiliaire qui fait la réduction sans copie défensive ; `toList` se contente alors de faire une copie défensive du résultat de cette dernière. Toute la difficulté, c'est de réussir à rendre privée la méthode auxiliaire alors qu'elle est membre d'une interface...) :

```
1   import java.util.ArrayList;
2   import java.util.List;
3   import java.util.function.Function;
4
5
6   public interface MyStream<T> {
7
8       static <T> MyStream<T> makeStream(List<T> sourceL) {
9           /* on force l'inférence vers UnsafeStream<T>, qui contient toutes les
                implémentations par
10          * défaut qui nous intéressent (de toute façon MyStream n'est pas une interface
11          * fonctionnelle). */
12          return (__HiddenStuff.UnsafeStream<T>) () -> new ArrayList<>(sourceL);
13      }
14
15      List<T> toList();
16      MyStream<T> limit(int limitNum);
17      MyStream<T> skip(int skipNum);
18      <U> MyStream<U> map(Function<T, U> ope);
19
20      /* Astuce pour que unsafeToList soit "comme private" : on la déclare dans une
            sous-interface,
21       * elle-même encapsulée dans une classe membre (UnsafeStream directement membre
             privé de
22       * MyStream n'est pas possible car Java n'autorise pas encore les types membres
             privés dans
23       * les interfaces).
24       *
25       * Alternative : utiliser comme intermédiaire une classe abstraite plutôt qu'une
             interface,
26       * et déclarer unsafeToList package-private (mais ça protège moins... et les
             classes
27       * abstraites ne peuvent pas être instanciées via des lambda-expressions).
28       * */
29      class __HiddenStuff {
30          @FunctionalInterface private interface UnsafeStream<T> extends MyStream<T> {
31
32              /* on aurait aimé private, mais les méthodes privées abstraites n'existent
                    pas (note :
33              * Java 9 autorise les méthodes private dans les interfaces, mais elles
                    doivent y être
34              * implémentées) */
35              List<T> unsafeToList();
36
37              default List<T> toList() {
38                  return new ArrayList<>(unsafeToList()); // copie défensive, une seule
                        fois !
39              }
40
41              default UnsafeStream<T> limit(int limitNum) {
42                  return () -> unsafeToList().subList(0, limitNum);
43              }
44
45              default UnsafeStream<T> skip(int skipNum) {
46                  return () -> {
47                      List<T> sourceL = UnsafeStream.this.unsafeToList(); // appel de toList
```

```
                    sur objet englobant (sinon, récursion jusqu'à  StackOverflowError
                    !)
48                  return sourceL.subList(skipNum, sourceL.size());
49               };
50            }
51
52            default <U> UnsafeStream<U> map(Function<T, U> ope) {
53               return () -> {
54                  List<T> sourceL = UnsafeStream.this.unsafeToList();
55                  List<U> targetL = new ArrayList<>(sourceL.size()); // là , il faut
                        construire une nouvelle liste (nouveau type d'éléments)
56                  for (T x : sourceL) targetL.add(ope.apply(x));
57                  return targetL;
58               };
59            }
60         }
61      }
62   }
```