

Compléments en Programmation Orientée Objet

TP n° 3 (Correction)

Indication : avec le cours sous les yeux les trois premiers exercices doivent être fait rapidement.

1 Les bases du polymorphisme

Exercice 1 : Définir le sous-typage entre types objet

On suppose déjà définies :

```
1 class A {}
2 class B {}
3 interface I {}
4 interface J {}
5
```

Voici une liste de déclarations :

```
1 class C extends I {}
2 interface K extends B {}
3 class C implements J {}
4 interface K implements B {}
5 class C extends A implements I {}
6 interface K extends I, J {}
7 class C extends A, B {}
8 class C implements I, J {}
9
```

Lesquelles sont correctes ?

Correction : Sont correctes les lignes 3, 5, 6 et 8.

Ligne 1 : incorrecte car une classe n'étend pas une interface.

Ligne 2 : incorrecte car une interface n'étend pas une classe.

Ligne 4 : incorrecte car une interface n'implémente pas une classe.

Ligne 7 : incorrecte car l'héritage de classes multiples est interdit.

Exercice 2 : Transtypage

<pre>1 2 class A { } 3 4 class B extends A { } 5 6 class C extends A { } 7 8 public class Tests { 9 public static void main(String[] args) { 10 System.out.println((int>true); 11 System.out.println((int) 'a'); 12 System.out.println((byte) 'a');</pre>	<pre>13 14 System.out.println((byte) 257); 15 System.out.println((char) 98); 16 System.out.println((double) 98); 17 System.out.println((char) 98.12); 18 System.out.println((long) 98.12); 19 System.out.println((boolean) 98.); 20 System.out.println((B) new A()); 21 System.out.println((C) new B()); 22 System.out.println((A) new C()); 23 } 24 }</pre>
--	--

Dans la méthode `main()` ci-dessus,

1. Quelles lignes provoquent une erreur de compilation ?

Correction : 10 (conversion de booléen), 18 (conversion de booléen), 20 (conversion sans qu'un type soit sous-type ou supertype de l'autre).

Les autres lignes ne posent pas de problème à la compilation.

2. Après avoir supprimé ces-dernières, quelles lignes provoquent une exception à l'exécution ?

Correction : (en gardant la numérotation actuelle)

Seules les conversions d'objet vers une sous-classe (*downcasting*) sont susceptibles de provoquer une erreur à l'exécution. Donc seule la ligne 19 peut être concernée (on convertit de A vers B).

Et effectivement, à la ligne 19, on voit que ça va nécessairement provoquer une erreur car la valeur de `new A()` à l'exécution sera toujours un objet de type A, donc non-utilisable en tant qu'objet de type C (il pourrait, par exemple, manquer des attributs).

3. Après les avoir enlevées, elles aussi, quels affichages provoquent les lignes restantes ?

Correction : Exécutez pour voir. Les seules valeurs qu'une conversion de type est susceptible de modifier "physiquement" sont celles des types valeur. En cas de *downcasting*, on va perdre de l'information, observez les conséquences.

Enfin, que ce soit une conversion vers le haut ou vers le bas, la façon d'afficher peut changer énormément d'un type à l'autre (cas le plus criant : la conversion de/vers un nombre vers/de un `char`).

Pour les types référence, si l'objet "converti" est le récepteur (à gauche du point) de la méthode appelée (c'est le cas ici : on appelle implicitement `toString()`), à cause de la liaison dynamique, ça ne change absolument pas la méthode qui sera appelée (elle dépend uniquement de l'objet réel, pas du type que le compilateur associe à son expression). Si l'expression objet apparaît en tant qu'argument d'une méthode, en cas de surcharge, le résultat peut changer (la surcharge est résolue en regardant le type statique de l'expression).

Exercice 3 : Surcharge

```

1 class A {};
2 class B extends A {};
3 class C extends B {};
4 class D extends B {};
5
6 public class Dad {
7     public static void f(A a, A aa) { System.out.println("Dad : A : A"); }
8     public static void f(A a, B b) { System.out.println("Dad : A : B"); }
9 }
10 public class Son extends Dad {
11     public static void f(A a, A aa) { System.out.println("Son : A : A"); }
12     public static void f(C c, A a) { System.out.println("Son : C : A"); }
13
14     public static void main(String[] args) {
15         f(new B(), new A());
16         f(new D(), new A());
17         f(new B(), new D());
18         f(new A(), new C());
19     }
20 }

```

Dans la méthode `main()` ci-dessus,

1. Quels affichages provoquent les lignes 15 à 18?

Correction : Les lignes 15 et 16 provoquent l’affichage *Son : A : A* et les autres lignes provoquent l’affichage *Dad : A : B*.

2. Que se passe-t-il si on appelle `f(new C(), new C())` ? `f(new C(), new B())` ?

Correction : Dans les deux cas, les trois signatures possibles de `f` (`(A,A)`, `(A,B)`, `(C,A)`) subsument la signature d’appel, mais `(A,B)` ne subsume pas `(C,A)` et inversement, ce qui entraîne une erreur : appel ambigu.

3. Dans la classe `Son` comment être sûr d’appeler les méthodes `f` de la classe `Dad` ? Quels types de paramètres permettent d’appeler la fonction `f` avec signature `(A,A)` ?

Correction : Vu que les fonctions `f` sont statiques il suffit d’écrire `F.f(x,y)`. Pour appeler la fonction avec la signature `(A,A)`, il faut que le type de `x` et `y` soit un sous-type de `A` mais pas de `B`.

2 Programmer à l'interface

Exercice 4 : Tris

Le tri à bulles est un algorithme classique permettant de trier un tableau. Il peut s'écrire de la façon suivante en Java :

```

1  static void triBulles(int tab[]) {
2      boolean change = false;
3      do {
4          change = false;
5          for (int i=0; i<tab.length - 1; i++) {
6              if (tab[i] > tab [i+1]) {
7                  int tmp = tab[i+1];
8                  tab[i+1] = tab[i];
9                  tab[i] = tmp;
10                 change = true;
11             }
12         }
13     } while (change);
14 }
```

Cette implémentation du tri à bulles permet de trier un tableau d'entiers. Maintenant on veut pouvoir utiliser le tri à bulles sur tout autre type de données représentant une suite (séquence) d'objets comparables.

Pour cela, il faut programmer à l'interface. Ainsi, notre tri sera programmé pour les interfaces suivantes :

```

1  public interface Comparable {
2      public Object value(); // renvoie le contenu
3      public boolean estPlusGrand(Comparable i);
4  }
5
6  public interface Sequencable {
7      public int longueur(); // Renvoie la longueur de la sequence
8      public Comparable get(int i); // Renvoie le ieme objet de la sequence
9      public void echange(int i, int j); // Echange le ieme object avec le jieme objet
10 }
11
```

1. Écrivez une méthode `affiche()` dans l'interface `Sequencable` permettant d'afficher les éléments de la séquence du premier au dernier. (Utilisez la fonction `toString()` de `Object`.)

Correction :

```

1  default void affiche() {
2      String s=contenu(1).toString();
3      for(int i=2; i< longueur(); i++) {
4          s+=" " + contenu(i).toString();
5      }
6      System.out.println("[ " + s + " ]");
7  }
8
```

2. Écrivez une méthode `triBulle` dans l'interface `Sequencable` qui effectue un tri à bulles sur la séquence.

Correction :

```

1  default void triBulle() {
2      boolean change = false;
3      do {
4          change = false;
5          for (int i = 1; i < longueur()-1; i++) {
6              if ( contenu(i).estPlusGrand(contenu(i+1))) {
7                  echange(i, i+1);
8              }
9          }
10         change = !change;
11     } while (change);
12 }
```

```
8         change = true;
9     }
10 }
11 } while (change);
12 }
13
```

3. Écrivez une classe `MotComparable` représentant un mot et implémentant l'interface `Comparable` de tel sorte que `estPlusGrand(Comparable i)` :
- quitte sur une exception (`throw new IllegalArgumentException();`) si `i.value()` n'est pas un sous-type de `String`,
 - retourne vrai si le contenu est plus grand lexicographiquement que `i.value()`, faux sinon.
- N'oubliez pas les constructeurs () et la méthode `toString()`.

Correction : Voir le fichier « MotComparable.java ».

4. Écrivez une classe `SequenceMots` qui représente une séquence de `MotComparable` et qui implémente `Sequencable`.
- Écrivez un constructeur prenant un tableau de `String`.

Correction : Voir le fichier « SequenceMots.java ».

5. Testez votre code.

Vous pouvez passer en paramètre un tableau de chaînes aléatoires générées avec l'instruction `Integer.toString((int)(Math.random()*50000))` (ou utilisez un des générateurs de l'exercice précédent).

Exercice 5 : Générateurs de nombres

Attention : pour faire cet exercice, il faut savoir implémenter une interface (`Generateur`) et comprendre le polymorphisme par sous-typage (toute méthode retournant un `Generateur` a le droit de retourner une instance de classe implémentant `Generateur`).

Objectif : écrire la classe-bibliothèque `GenLib`, permettant de créer des générateurs de toute sorte (entiers au hasard, suites arithmétiques, suites géométriques, fibonacci, etc.), sans pour autant fournir d'autres types publics que la classe `GenLib` elle-même ainsi qu'une interface `Generateur` dans le package que vous livrez à vos utilisateurs.

Pour commencer, fixons l'interface générateur :

```
1 interface Generateur { int suivant(); }
```

Méthode : utiliser le schéma suivant : `GenLib` (classe non instanciable) contient une série de fabriques statiques permettant de créer les générateurs. Chaque appel à une fabrique instancie une classe imbriquée en utilisant les paramètres passés.

Attention : la classe `GenLib` n'implémente pas elle-même l'interface `Generateur` (ça n'aurait pas de sens, puisqu'elle n'est pas instanciable). Ses méthodes ne renvoient pas de `int` !

Exemple d'utilisation : pour afficher les 10 premiers termes de la suite de Fibonacci

```
1 Generateur fib = GenLib.nouveauGenerateurFibonacci();
2 for (int i = 0; i < 10; i++) System.out.println(fib.suivant());
```

Questions :

- Des 4 genres de classes imbriquées vues en cours (membre statique, membre non statique, locale, anonyme), l'un ne peut pas être utilisé ici, lequel ?

Correction : Peuvent convenir :

- les classes membres statiques (elles marchent comme les classes "normales", de premier niveau... qui pourraient d'ailleurs aussi convenir ici, bien qu'on perde l'avantage de l'encapsulation)
- les classes locales (et leur variante anonyme) : elles peuvent être créées dans toute méthode.

Pour utiliser les classes membres non statiques, il faudrait connaître une instance englobante. Or le genre de classe/bibliothèque que nous programmons n'est habituellement pas destinée à être instanciée.

Observons tout de même que les méthodes demandées sont de toute façon statiques et ne connaissent donc pas d'instance englobante implicite (`this`). La seule façon d'accéder à une classe membre non statique serait de fournir une instance explicite d'une façon où d'une autre (paramètre de la méthode, ou bien attribut)...

- Programmez les méthodes statiques permettant de créer les générateurs suivants :
 - générateur d'entiers aléatoires (compris entre 0 et $m - 1$, m étant un paramètre)
 - suite arithmétique : $0, r, 2r, 3r, \dots$ (r étant un paramètre)
 - suite géométrique : $1, r, r^2, r^3, \dots$ (r étant un paramètre)
 - suite de Fibonacci : $1, 1, 2, 3, 5, 8, 13, \dots$

Variez les techniques : montrez un exemple pour chaque genre de classe imbriquée. Si vous vous rappelez comment on utilise les lambda-expressions, tentez aussi cette approche pour implémenter `Generateur` sans définir de classe.

Correction :

```
1  public class Generateurs {
2      //avec le constructeur pas testé:
3      private Generateurs(){};
4
5      interface Generateur {
6          int suivant();
7      }
8
9      public static Generateur nouveauGenerateurEntiersAleatoires(int m) {
10         return new Generateur() {
11             @Override
12             public int suivant() {
13                 return (int) (m * Math.random());
14             }
15         };
16     }
17
18     public static Generateur nouveauGenerateurGeometrique(int m) {
19         class GenerateurGeometrique implements Generateur {
20             int prochain = 1;
21
22             @Override
23             public int suivant() {
24                 int courant = prochain;
25                 prochain *= m;
26                 return courant;
27             }
28         }
29         return new GenerateurGeometrique();
30     }
31
32     public static Generateur nouveauGenerateurArithmetique(int m) {
33         class GenerateurArithmetique implements Generateur {
34             int prochain = 0;
35
36             @Override
37             public int suivant() {
38                 int courant = prochain;
39                 prochain += m;
40                 return courant;
41             }
42         }
43         return new GenerateurArithmetique();
44     }
45
46     static class Fibo implements Generateur {
47         int eltSuivant = 1, eltCourant = 0;
48
49         @Override
50         public int suivant() {
51             int nouveau = eltSuivant + eltCourant;
52             eltCourant = eltSuivant;
53             eltSuivant = nouveau;
54             return eltCourant;
55         }
56     }
57
58     public static Generateur nouveauGenerateurFibonacci() {
59         return new Fibo();
60     }
61
62     public static void main(String args[]) {
63         Generateur fibo = nouveauGenerateurFibonacci();
64         for (int i = 0; i < 10; i++)
65             System.out.println(fibo.suivant());
66     }
67 }
68
```

Correction : Réécriture du générateur aléatoire avec lambdas :

```
1      public static Generateur nouveauGenerateurEntiersAleatoires(int m) {  
2          return () -> (int) (m * Math.random());  
3      }  
4
```

On ne peut pas écrire les autres exemples de cette façon, car la syntaxe lambda-expression ne permet que de définir la méthode (unique) de l'interface implémentée par la classe anonyme. En outre, on ne peut pas ajouter/utiliser d'attribut. Donc cette syntaxe ne permet pas d'écrire un générateur où l'on a besoin de sauvegarder un état.

3. Écrivez une méthode `int somme(Generateur gen, int n)` qui retourne la somme des `n` prochains termes du générateur `gen`.
4. Écrivez un `main()` qui demande à l'utilisateur de choisir entre les différents types de suite (et éventuellement d'entrer un paramètre), puis instancie le générateur de suite correspondant et en affiche ses 10 premiers termes et la somme des 5 suivants.
5. Maintenant vous vous mettez dans la peau de l'utilisateur de `GenLib`.

Vous voulez utiliser la méthode `somme` pour calculer la somme des termes d'une collection Java (implémentant `java.util.Collection`)... qui évidemment n'implémente pas `Generateur`.

Écrivez un adaptateur de `Collection` à `Generateur`, puis un programme utilisant l'adaptateur pour afficher la somme des termes de la liste `List.of(56, 23, 78, 64, 19)`.

3 Les dessous du transtypage

Exercice 6 : Transtypes primitifs

Voici un programme ([TranstypesPrimitifs.java](#) sur Moodle) :

```

1 public class TranstypesPrimitifs {
2     public static void main(String[] args) {
3         int vint = 1234567891;
4         short vshort = 42;
5         float vfloat = 9.2E11f;
6         System.out.println("vint = " + vint +
7             ", vshort = " + vshort +
8             ", vfloat = " + vfloat);
9     }
10 }
11 }

```

1. Compilez et exécutez ce programme (assurez-vous de comprendre la notation `9.2E11f`).
2. Nous allons regarder superficiellement le code-octet produit : dans un terminal, allez dans le répertoire où se trouve `TranstypesPrimitifs.class` et tapez la commande `"javap -c -v TranstypesPrimitifs"`. Le code-octet apparaît ainsi sous une forme désassemblée quasi lisible. Nous nous intéresserons en particulier au début de la partie `Code` :, qui correspond à la déclaration et l'initialisation de nos trois variables. On peut repérer l'appel à l'instruction suivante, `println`, par l'instruction `getstatic` dans le code-octet.

Il n'y a donc que 6 ou 7 lignes à regarder. Constatez que certaines variables sont initialisées par une séquence d'instructions comme : `bipush 42;istore_2`, alors que d'autres ont la séquence `ldc` suivie de `istore` ou `fstore` (le i ou le f désigne clairement un type)

3. Nous allons nous intéresser à la façon dont sont fait les transtypes. Ajoutez une ligne avant l'instruction d'affichage : `vint=vshort;` et interpréter les opérations `load`, `store` qui apparaissent.

Avec les 3 variables présentes il y a théoriquement 6 transtypes, certains qu'il faut rendre explicites. Essayez les tous et complétez le tableau ci-dessous avec vos remarques.

Le tableau :

	=	vint	vshort	vfloat
vint		XXX		
vshort			XXX	
vfloat				XXX

Relevez, notamment, dans chaque cas :

- si ça compile directement, s'il ajouter un *cast* explicite, etc. ;
- la nature des instructions ajoutées dans le code-octet (notez que les instructions de la forme `f2i` expriment un changement de type) ;
- l'affichage produit après conversion.

Exercice 7 : Transtypes d'objets (sur machine)

Même exercice que le précédent mais sur le programme suivant :

```
1 public class TranstypesObjets {
2     public static void main(String[] args) {
3         Object vObject = new Object();
4         Integer vInteger = 42;
5         String vString = "coucou";
6         System.out.println("vObject = " + vObject + ", vInteger = "
7             + vInteger + ", vString = " + vString);
8     }
9 }
```

Différence, vous ne verrez plus l'ajout de l'instruction `u2t` mais parfois celle de `checkcast`. Dans quels cas ?

Dans certains cas vous aurez eu besoin, pour compiler, d'un `cast` explicite. Lesquels ? Est-ce les-mêmes que dans la question précédente ?

Dans certains cas, le programme quittera sur `ClassCastException`, lesquels ?

Correction : Tout est écrit explicitement dans le cours.

Exercice 8 : Transtypes mixtes (sur machine)

Faites le même travail sur le programme suivant. Remarquez les instructions, insérées par le compilateur, qui correspondent au boxing et à la vérification de types.

```
1 public class TranstypesMixtes {
2     public static void main(String[] args) {
3         Object vObject = Integer.valueOf(9);
4         Integer vInteger = 42;
5         int vint = 111;
6         System.out.println("vObject = " + vObject +
7             ", vInteger = " + vInteger +
8             ", vint = " + vint);
9     }
10 }
11 }
```

Correction : Idem : ceci est couvert par le cours. On peut observer ici les appels de méthode que Java insère pour réaliser le *boxing* et l'*autoboxing*.