TP de Compléments en Programmation Orientée Objet n° 3
(Correction)

Indications : avec le cours sous les yeux les deux premiers exercices doivent être fait rapidement. Ne passez pas plus d'une heure sur le 3ème exercice afin qu'on puisse regarder ensemble l'exercice 4 lors de cette séance

Exercice 1 : Transtypage

```
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');
13         System.out.println((byte) 257);
14         System.out.println((char) 98);
15         System.out.println((double) 98);
16         System.out.println((char) 98.12);
17         System.out.println((long) 98.12);
18         System.out.println((boolean) 98.);
19         System.out.println((B) new A());
20         System.out.println((C) new B());
21         System.out.println((A) new C());
22     }
23 }
```

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 convetit 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 2 :

On suppose déjà définies :

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

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 {}
```

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 3 : Listes chaînées

On explore une façon particulière de programmer des listes chaînées pouvant contenir plusieurs types de données, mais sans utiliser la généricité.

Une liste chaînée est constituée de cellules à deux champs : un champ "contenu" (contenant un des éléments de la liste) et un champ "suivant", pointant sur une autre cellule, ou bien sur rien (fin de liste).

Pour notre mise en œuvre en Java, on va considérer que tout objet implémentant l'interface `Chainable`, ci-dessous peut servir de cellule de liste chaînée :

```
1 interface Chainable {
2     Chainable suivant();
3 }
```

Ainsi, un objet `Chainable` peut représenter une liste non-vide (l'objet est la première cellule, les suivantes sont obtenues par appels successifs à la méthode `suivant()`), alors que la liste vide est juste représentée par la valeur `null`.

1. Écrivez les classes `EntierChainable` et `MotChainable` implémentant l'inferface `Chainable` et dont les objets contiennent respectivement un élément entier et un élément chaîne de caractères.

Correction :

```
class EntierChainable implements Chainable {
  public final int elt;
  private final Chainable suiv;
  public EntierChainable(int elt, Chainable suiv) {
    this.element = elt;
    this.suiv = suiv;
  }
  @Override public Chainable suivant () {
    return suiv;
  }
}
```

(implémentations similaires pour `MotChainable`, remplacer `int` par `String`)

2. Écrivez pour chacune de ces classes le constructeur de types respectifs `public EntierChainable(int elt, Chainable suiv)`, et `public MotChainable(String elt, Chainable suiv)`, construisant une nouvelle cellule de contenu `elt` et de successeur `suiv`.

Correction : Cf. ci-dessus.

3. Programmez une méthode `int longueur()` qui donne la longueur d'une liste. Faites en sorte qu'il n'y ait pas besoin d'ajouter du code dans toutes les implémentations de `Chainable`, c'est à dire utilisez la possibilité d'écrire du code dans les interfaces sous certaines conditions.

Correction : Ajouter dans l'interface `Chainable` la méthode, précédée de `default`

```
default int longueur() {
    return 1 + ((suivant() == null)?0:suivant().longueur());
}
// ou
default int longueur() {
    if (suivant()==null) return 1;
    else return 1+suivant().longueur();
}
```

4. Écrivez les méthodes `toString()` de ces classes. Elles devront non seulement présenter la donnée stockée dans la cellule, mais aussi celles des cellules suivantes.

Correction : Dans chaque implémentation :

```
@Override
public String toString() {
    return elt + ((suivant() == null)?"":(", " + suivant()));
}
```

5. Pourrait-on programmer la méthode `toString()` de la même façon que la méthode `longueur()` ? Que faudrait-il changer/ajouter à l'interface `Chainable` ?

Correction : Non car on ne peut pas définir des méthodes de `Object` (dont `toString()`) dans une interface. La raison : Si on pouvait définir `toString()` dans une interface, toute classe l'implémentant en hériterait de deux versions de `toString()` (l'autre provenant d'une superclasse, au pire `Object`). Or en cas de conflit, c'est toujours la méthode héritée de la superclasse qui prime : l'implémentation par défaut

héritée de l'interface n'est là justement que... par défaut !
Fidèles à la philosophie de Java, les concepteurs de Java 8 ont donc choisi d'interdire ce qui est inutile.
→ Bref si on veut définir une telle méthode dans l'interface, elle doit porter un autre nom (p. ex. `recursiveToString()`).
L'autre problème c'est que cette méthode doit pouvoir appeler une méthode pour convertir en `String` le contenu du maillon courant. Cette méthode pourrait être ajoutée en tant que méthode abstraite à l'interface... ou bien il faut une méthode abstraite `contenu()` pour récupérer le contenu du maillon, sur lequel on appelle `toString()`. Dans tous les cas il manque une méthode abstraite.
Donc il faut 2 méthodes : `default String recursiveToString(){ ... }` et `Object contenu();`.
Si on tient à avoir une méthode appelée `toString()`, il faudra malheureusement l'écrire dans chaque classe (même si elle ne fait qu'appeler `recurviseToString()`).

On considère maintenant l'interface `Pesable` :

```
1  interface Pesable {
2      int poids();
3  }
```

On considèrera que le poids d'un entier est sa valeur absolue, le poids d'une chaîne sa longueur, et le poids d'une liste la somme des poids de ses cellules. Pour permettre de construire une liste qui contienne des éléments tous pesables, il sera utile de définir également une nouvelle interface `ChainablePesable` combinant les précédentes. Quel type retour de `suivant()` peut-on alors garantir ?

4. Écrivez les classes `EntierChainablePesable` et `MotChainablePesable`, implémentant à la fois l'inferface `Chainable` et l'interface `Pesable`. Complétez les classes de l'exercice précédent en leur ajoutant leurs méthodes `poids()`.

Correction : Ici, il est utile de définir une interface `ChainablePesable` qui soit l'union des deux interfaces `Chainable` et `Pesable` et force le maillon suivant à être aussi pesable :

```
1  public interface ChainablePesable extends Chainable, Pesable {
2      @Override ChainablePesable suivant();
3  }
```

Remarquer qu'on ne peut pas réécrire `suivant()` par défaut en utilisant `Chainable.super.suivant()` car celle ci est considérée comme à priori abstraite ...
On se contente donc de préciser le type de retour lors de la redéfinition de `suivant()` (qui reste abstraite). Cela n'est autorisé que pour les types référence.
Ainsi on peut écrire nos classes comme ceci :

```
1  public class EntierChainablePesable extends EntierChainable implements ChainablePesable {
2      public EntierChainablePesable(int elt, ChainablePesable suivant) {
3          super(elt, suivant);
4      }
5      @Override public int poids() {
6          return elt + (suivant()!=null?suivant.poids():0);
7      }
8      @Override public ChainablePesable suivant() {
9          return (ChainablePesable) super.suivant();
10     }
11 }
```

(on définit `MotChainablePesable` similairement)
Remarque : la conversion de type dans `suivant()` est garantie, car le champs suivant est initialisé via le constructeur de `EntierChainablePesable` seulement et n'est pas modifiable. Cela dit, il s'agit tout de même d'une conception fragile, car si on modifie la superclasse `EntierChainable` dans le futur, on risque de casser `EntierChainablePesable`. Probablement, il serait plus propre de ne pas utiliser l'héritage dans ce cas.

5. Pourrait-on programmer la méthode `poids()` de la même façon que la méthode `longueur()` ? Que faudrait-il changer/ajouter à l'interface `Pesable` ?

   Correction : Comme pour `toString()`, il faudrait pouvoir avoir accès au (poids du) maillon courant (différent du poids de toute la sous-liste qui démarre ici). Pour cela, on peut ajouter une autre méthode abstraite `int poidsMaillon()` et écrire poids ainsi :

```
1  default int poids() {
2    return poidsMaillon() + (suivant() == null)?0:suivant().poids();
3  }
```

Exercice 4 : Transtypages primitifs

Voici un programme (`TranstypagesPrimitifs.java` sur Moodle) :

```
1  public class TranstypagesPrimitifs {
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 `TranstypagesPrimitifs.class` et tapez la commande "`javap -c -v TranstypagesPrimitifs`". 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 transtypages. 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 transtypages, certains qu'il faut rendre explicites. Essayez les tous et complétez le tableau ci-dessous avec vos remarques. Notamment :

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

=	vint	vshort	vfloat
vint	XXX		
vshort		XXX	
vfloat			XXX

4. Vous pouvez regarder (sans vous attarder) le code-octet correspondant au premier exercice.
5. Faites le même travail sur le programme suivant. Remarquez les instructions qui correspondent au boxing et à la vérification de types.

```
1  public class TranstypagesMixtes {
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*.