TD de CPOO n° 3 et 4

TP 3

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 ?
2. Après avoir supprimé ces-dernières, quelles lignes provoquent une exception à l'exécution ?
3. Après les avoir enlevées, elles aussi, quels affichages provoquent les lignes restantes ?

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 ?

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

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

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 }
```

TP 4

Exercice 5 : 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 ?
2. Que se passe-t-il si on appelle `f(new C()),new C())` ? `f(new C(),new B())` ?
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)` ?

Exercice 6 : 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 séquence d'objets comparables. Pour cela, on considère 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 longeur(); // 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      }
```

1. Écrivez une méthode `affiche()` dans l'interface `Sequencable` permettant d'afficher les éléments de la séquence du premier au dernier (utiliser la fonction `toString()` de Object).
2. Écrivez une méthode `triBulle` dans l'interface `Sequencable` qui effectue un tri à bulles sur la séquence.
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()`.
4. Écrivez une classe `SequenceMots` qui représente une séquence de `MotComparable` et qui implémente `Sequencable`.
   Écrivez un constructeur prenant un tableau de `String`.
5. Testez votre code.
   Vous pouvez passer en paramètre un tableau de chaînes aléatoires générérées avec l'instruction `Integer.toString((int)(Math.random()*50000))`.

Si vous ne les avez pas fait/terminé, faites les exercices 4 et 5 du TP3 avant de faire le suivant.

Exercice 7 : Transtypages d'objets (sur machine)

Même exercice que l'exercice 4 et 5 du TP3 mais sur le programme suivant :

```
1  public class TranstypagesObjets {
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 ?