Université Paris Cité

Compléments en Programmation Orientée Objet
TP n° 3

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 :

```
class A {}
class B {}
interface I {}
interface J {}

```

Voici une liste de déclarations :

```
class C extends I {}
interface K extends B {}
class C implements J {}
interface K implements B {}
class C extends A implements I {}
interface K extends I, J {}
class C extends A, B {}
class C implements I, J {}

```

Lesquelles sont correctes ?

Exercice 2 : Transtypage

```

class A { }

class B extends A { }

class C extends A { }

public class Tests {
    public static void main(String[] args) {
        System.out.println((int)true);
        System.out.println((int) 'a');
        System.out.println((byte) 'a');
        System.out.println((byte) 257);
        System.out.println((char) 98);
        System.out.println((double) 98);
        System.out.println((char) 98.12);
        System.out.println((long) 98.12);
        System.out.println((boolean) 98.);
        System.out.println((B) new A());
        System.out.println((C) new B());
        System.out.println((A) new C());
    }
}

```

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 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 ?
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)` ?

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 :

```java
    static void triBulles(int tab[]) {
        boolean change = false;
        do {
            change = false;
            for (int i=0; i<tab.length - 1; i++) {
            if (tab[i] > tab [i+1]) {
                int tmp = tab[i+1];
                tab[i+1] = tab[i];
                tab[i] = tmp;
                change = true;
            }
            }
        } while (change);
    }
```

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 :

```java
public interface Comparable {
    public Object value(); // renvoie le contenu
    public boolean estPlusGrand(Comparable i);
}

public interface Sequencable {
    public int longeur(); // Renvoie la longueur de la sequence
    public Comparable get(int i); // Renvoie le ieme objet de la sequence
    public void echange(int i, int j); // Echange le ieme object avec le jieme objet
}

```

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.)
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 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 :

1. 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 ?
2. 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 : $0, r, 2r, 3r, \ldots$ (r étant un paramètre)
   - suite géométrique : $1, r, r^2, r^3, \ldots$ (r étant un paramètre)
   - suite de Fibonnacci : $1, 1, 2, 3, 5, 8, 13, \ldots$

   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.
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 : 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.

   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 : Transtypages d'objets (sur machine)

Même exercice que le précédent 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 ?

Exercice 8 : Transtypages 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 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 }
```