Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2020-2021**

TD - Séance n°4 - Correction

Héritage

Exercice 1 *Héritage et surcharge*
On définit les classes `A` et `B` de la manière suivante :

```
1  public class A {
     public void f(A x) {
3      System.out.println("A.f(A)");
     }
5    public void g() {
       f(new A());
7    }
   }
9  public class B extends A {
     @Override
11   public void f(A x) {
       System.out.println("B.f(A)");
13   }
     // Surcharge
15   public void f(B x) {
       System.out.println("B.f(B)");
17   }
   }
```

On construit des objets `a`, `b` et `c` :

```
A a = new A();
B b = new B();
A c = new B();
```

Qu'affichent les instructions suivantes ?

```
a.g();
b.g();
c.g();
```

```
a.f(a);
a.f(b);
a.f(c);
b.f(a);
b.f(b);
b.f(c);
```

```
c.f(a);
c.f(b);
c.f(c);
```

Que deviennent les questions précédentes si les classes `A` et `B` sont définies comme suit ?

```
1  public class A {
     public void f(A x) {
3      System.out.println("A.f(A)");
     }
5    //Surcharge
     public void f(B x) {
7      System.out.println("A.f(B)");
     }
9    public void g() {
       f(new A());
11   }
}
13 public class B extends A {
     @Override
15   public void f(A x) {
       System.out.println("B.f(A)");
17   }
     @Override
19   public void f(B x) {
       System.out.println("B.f(B)");
21   }
}
```

Correction : Cet exercice illustre la notion de "Dynamic binding" pour les méthodes et "static binding" pour les variables :

```
A.f(A)
2 B.f(A)
B.f(A)
```

```
1  a.f(a) -> A.f(A)
   a.f(b) -> A.f(A)
3  a.f(c) -> A.f(A)
   b.f(a) -> B.f(A)
5  b.f(b) -> B.f(B)
   b.f(c) -> B.f(A)
7  c.f(a) -> B.f(A)
   c.f(b) -> B.f(A)
9  c.f(c) -> B.f(A)
```

```
1  a.f(a) -> A.f(A)
   a.f(b) -> A.f(B)
```

```
a.f(c) -> A.f(A)
b.f(a) -> B.f(A)
b.f(b) -> B.f(B)
b.f(c) -> B.f(A)
c.f(a) -> B.f(A)
c.f(b) -> B.f(B)
c.f(c) -> B.f(A)
```

Exercice 2 *Constructeurs et héritage*
On définit les classes suivantes :

```
class C {
    public C() { System.out.println("Hello"); }
}
class D extends C {
    private int x;
    public D(int x) { this.x = x; }
}
```

1. Quel est le résultat de `new C();` ? De `new D()` ?

   Correction : Le premier fonctionne normalement (affiche "Hello").
   Le deuxième pose problème parce que le nombre d'arguments ne correspond pas au nombre d'arguments du constructeur de D :

```
Test.java:4: error: constructor D in class D cannot be
applied to given types;
        new D();
        ^
  required: int
  found: no arguments
  reason: actual and formal argument lists
          differ in length
1 error
```

2. Peut-on définir les classes suivantes ? Pourquoi ?

```
class E extends C {}
class F extends D {}
```

   Correction : F pose problème tel quel (même message d'erreur pendant la compilation) à cause du constructeur de D : il faut à F un constructeur qui permette de remplir le champ x de la classe D.

Exercice 3 *Héritage de méthodes statiques*
Le but de cet exercice est de modéliser une serre. Une serre est un tableau d'objets de type `Plante`. Certaines plantes ont besoin d'un apport en engrais supplémentaire et seront de type `PlanteFleurie`. Le début des classes est donné par :

```java
public class Plante {
    private static int nbPlanteSoif;
    private boolean arrosee=false;
```

```java
public class PlanteFleurie extends Plante {
    private static int nbPlanteSansEngrais;
    private boolean engrais=false;
```

1. Écrivez les constructeurs adéquats.
2. Écrivez les méthodes `toString` de chacune des classes.
3. Écrivez dans chaque classe une méthode `Etat` qui renseigne sur le nombre de plantes à arroser ou auxquelles il manque de l'engrais.
4. Dans chaque classe écrivez une méthode **statique** pour arroser un objet de la classe même. Lors de l'arrosage, contrairement à une `Plante` simple, une `PlanteFleurie` reçoit également de l'engrais.
5. Qu'affiche le code suivant ?

```java
Plante[] serre=new Plante[5];
serre[0]=new Plante();
serre[1]=new PlanteFleurie();
serre[2]=new Plante();
serre[3]=new PlanteFleurie();
serre[4]=new Plante();
System.out.println(Plante.Etat());
System.out.println(PlanteFleurie.Etat());
for(int i=0;i<5;i++){
    Plante.arrosage(serre[i]);
    System.out.println(serre[i]);
}
System.out.println(Plante.Etat());
System.out.println(PlanteFleurie.Etat());
```

Correction : Toutes les plantes ont été arrosées, mais les deux fleuries n'ont pas eu leur engrais.

Que se passe-t-il si `Plante.arrosage(serre[i])` est remplacé par `PlanteFleurie.arrosage(serre[i]);` ?

Correction : Idem. En effet la type déclaré de `serre[i]` est `Plante`. Donc la méthode PlanteFleurie.arrosage(Plante) (héritée par la classe mère) est invoquée.

6. Proposez une solution pour que la méthode `arrosage` ait l'effet attendu (c'est-à-dire pour qu'elle arrose toutes les plante et donne de l'engrais à toutes les plantes fleuries).

Correction : Il faut écrire une méthode d'instance. Cf. fichiers .java

Correction : Voir les fichiers Plante.java et PlanteFleurie.java.

Exercice 4 *Héritage - Immobilier*

Correction : Cf. fichiers .java

On cherche à modéliser un patrimoine immobilier. Tous les attributs seront privés, on créera des accesseurs selon les besoins.

1. Écrivez une classe `Batiment` avec deux variables `adresse` et `surfaceHabitable` (un `double`) et son constructeur `Batiment(String adresse, double surface)`. Implémentez la méthode `String toString()`.
2. Écrivez une classe `Maison` héritant de `Batiment` avec les attributs `nbPieces` et `surfaceJardin`. Écrivez le constructeur `Maison(String adresse, int surfaceH, int surfaceJ, int nbPieces)`. Écrivez aussi la méthode `String toString()` en utilisant un appel à `super.toString()`.
3. Écrivez une méthode `main` dans une classe `TestBatiment`. Ce programme doit instancier un bâtiment, deux maisons et les afficher.

   Ensuite, créez un tableau de 10 bâtiments. Est-ce que les bâtiments sont instanciés ?

   Correction : Non

   Remplacez 2 éléments du tableau par les deux maisons précédemment créées. Qu'affichera `System.out.println` appelée sur chaque élément du tableau ?
4. On va maintenant implémenter les méthodes `getSurfaceHabitable()` ainsi que `getSurfaceJardin()`. Dans quelles classes les implémente-t'on ?
5. Écrivez une méthode `surfaceHabitableTotale(Batiment[] tabBat)` dans la classe `TestBatiment` qui prend en argument un tableau de bâtiments (avec éventuellement des cases vides) et calcule la surface totale habitable des bâtiments .

   Ajouter une méthode `surfaceJardinTotale(Batiment[] tabBat)` qui calcule de la même façon la surface totale des jardins.

   Qu'ont de particulier ces deux méthodes ?
6. L'impôt local d'un bâtiment est calculé selon la formule

   $$\text{Impot} = \text{TauxA} \times \text{surfaceHabitable} + \text{TauxB} \times \text{surfaceJardin}$$

   Les valeurs de cette année étant TauxA $= 5.6$ et TauxB $= 1.5$.

   Où déclarer les variables TauxA et TauxB ? et comment ? Dans quelle(s) classe(s) faut-il écrire la méthode `impot` ?

   Correction : TauxA dans Batiment, TauxB dans Maison (et statiques toutes les deux) et impot dans les deux (surcharge)

Exercice 5 *Héritage - Immobilier (suite)*

Correction : Cf. fichiers.java

1. Écrivez une classe Personne. Une personne a un nom et possède une certaine somme d'argent (en centimes d'euros).
2. Tous les bâtiments ont un propriétaire. Par contre, seuls les maisons peuvent être louées, mais ce n'est pas automatique (une maison peut être `louable` ou non).

   Modifiez vos classes pour qu'on puisse trouver le propriétaire d'un bâtiment, le locataire d'une maison ainsi que les différents biens immobiliers dont est propriétaire ou locataire une personne.
3. Chaque maison a un loyer. Écrire une méthode qui fait payer à un locataire son loyer et le reverse au propriétaire. (On permettra aux personnes d'avoir des dettes)

   Correction : La méthode sera dans la classe MaisonLouable car il faut pouvoir accéder à la fois au locataire et au propriétaire
4. Écrivez une méthode qui fait payer à un propriétaire ses impôts sur ses biens. L'impot sur un bien depend de la catégorie du bien : une maison louable a un impôt plus élevé qu'une maison non louable qui a un impôt plus élevé qu'un bâtiment autre qu'une maison.

   Correction : La méthode sera dans la classe Personne car on a les références aux biens
5. Dans une classe de test écrivez une méthode `chercherLocation` qui reçoit en paramètre un tableau de maisons louables, un futur locataire et un budget. La méthode loge la personne dans une maison louable sans locataire actuel dont le loyer est inférieur au budget, ou renvoie `false` s'il n'y a pas de telle maison.