TD - Séance n° 4 - Correction

Héritage (suite)

Exercice 1 *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. Pour chacune des expressions suivantes, cette expression est-elle acceptée par le compilateur ? Si oui, quel est l'effet de son évaluation ? : `new C()` ; `new D()` ; `new D(0)`.

   Correction : La première crée une instance de `C` et affiche `"Hello"`. La deuxième n'est pas compilable : la classe `D` contient un constructeur (donc plus de constructeur par défaut) et pas de constructeur sans arguments. La dernière crée une instance de `D`, invoque implicitement le constructeur sans arguments de la classe parente (donc affiche `"Hello"`), puis initialise le champ `x` à `0`.

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

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

   Correction : La première définition ne pose pas problème : la classe `E` est définie sans constructeur explicite, donc est munie d'un constructeur par défaut, qui appelle lui-même implicitement le constructeur sans arguments de la classe `C`.

   La seconde n'est pas compilable : le constructeur par défaut de `F` ne peut appeler le constructeur sans arguments de la classe `D`, puisque ce constructeur n'existe pas.

Exercice 2 *Héritage et surcharge*
On définit les classes `A` et `B` de la manière suivante :

```
public class A {
  public void f(A x) {
    System.out.println("A : f(A)");
```

```
  }
  public void g() {
    f(new A());
  }
}
public class B extends A {
  public void f(A x) {
    System.out.println("B : f(A)");
  }
  public void f(B x) {
    System.out.println("B : f(B)");
  }
}
```

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();
```

Correction :

```
  A : f(A)
  B : f(A)
  B : f(A)
```

L'invocation de `g` lance : dans le premier cas, l'implémentation de `g()` dans `A` ; dans les deux suivants, l'implémentation de `g()` dans `B`, qui est celle de `A` héritée par `B` (en l'absence d'implémentation de `g()` dans `B`). Dans chaque cas, on atteint dans `g()` l'instruction `this.f(new A())` :

1. `this.f(new A())` atteint par `a.g()` : `a` désigne une instance de `A`, donc l'implémentation de `f` invoquée est celle de `f(A a)` dans `A`.
2. `this.f(new A())` atteints par `b.g()` et `c.g()` : `b` et `c` désignent des instances de `B` et l'argument de `f` est une référence de type `A`, donc l'implémentation de `f` invoquée est celle de `f(A a)` dans `B`,.

Même question pour les instructions qui suivent.

```
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);
```

Correction :

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

Détails des appels :

1. `a.f(a)`, `a.f(b)`, `a.f(c)`, `c.f(a)`, `c.f(b)`, `c.f(c)` :
   - Les références `a` et `c` sont de type `A`, donc l'implémentation de `f` invoquée ne peut être dans chaque cas que l'une des implémentations de `f(A a)`, dont la signature est la seule visible pour `f` avec une `A`-référence.
   - Dans les trois premiers cas, `a` désigne une instance de `A`, donc l'implémentation invoquée est celle de `f(A a)` dans `A`. Noter que dans le second cas, l'appel est compatible avec le type de la référence `b`, qui est convertie en `A`-référence.
   - Dans les trois derniers cas, `c` désigne une instance de `B`, donc l'implémentation invoquée est celle de `f(A a)` dans `B`.
2. `b.f(a)`, `b.f(b)`, `b.f(c)` :
   - `b` est une référence de type `B` désignant une instance de `B`, donc l'implémentation de `f` invoquée dans chaque cas ne peut être que l'une des implémentations de `f(A b)` ou `f(B b)` de la classe `B`.
   - Les références `a` et `c` sont de type `A`, donc dans le premier et le troisième cas, l'implémentation invoquée est celle de `f(A a)` dans `B`.
   - La référence `b` est de type `B`, donc dans le second cas, l'implémentation invoquée est celle de `f(B a)` dans `B`.

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

```
public class A {
  public void f(A x) {
    System.out.println("A : f(A)");
  }
  public void f(B x) {
    System.out.println("A : f(B)");
  }
  public void g() {
    f(new A());
  }
}
public class B extends A {
```

```
  public void f(A x) {
    System.out.println("B : f(A)");
  }
  public void f(B x) {
    System.out.println("B : f(B)");
  }
}
```

Correction : Chaque type de référence voit à présent deux signatures de `f` : celle de `f(A a)` et celle de `f(B b)`. Dans chaque cas, l'implémentation choisie est celle dont la signature est spécifiée par le type de la référence donnée en argument, dans la classe de l'objet sur laquelle `f` est invoquée :

```
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 3 *Héritage de méthodes statiques*
On considère les trois classes suivantes.

```
class A {
  static void m(A a) {
    System.out.println("A.m(A)");
  }
}
```

```
class B extends A {
  static void m(B b) {
    System.out.println("B.m(B)");
  }
}
```

```
public class Test {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    A c = new B();

   A.m(a);
   A.m(b);
   A.m(c);
```

```
    B.m(a);
    B.m(b);
    B.m(c);
  }
}
```

Qu'affiche ce programme, et pourquoi ?
Correction :

```
  A.m(a);   // A.m(A)
  A.m(b);   // A.m(A)
  A.m(c);   // A.m(A)

  B.m(a);   // A.m(A)
  B.m(b);   // B.m(B)
  B.m(c);   // A.m(A)
```

La classe `A` ne voit qu'une seule méthode statique de nom `m`, celle en `m(A a)`. Les trois premières instructions ne donc invoquer que cette méthode – dans le second cas, la signature de `m` est compatible avec le type de la référence `B`.

La classe `B` voit deux exemplaires de `m` : une version en `m(B b)` définie dans `B`, et une version en `m(A a)` définie dans `A` mais aussi vue dans `B` par héritage. Le choix de la version exécutée est alors déterminé par le type de la référence argument.

Exercice 4 *Héritage - Immobilier*
Correction : Cf. fichiers .java

On cherche à modéliser un patrimoine immobilier.

1. Tout bâtiment est caractérisé par une certaine surface habitable. Définir une classe `Batiment` contenant un champ `surfaceH` de type `double` représentant la surface habitable d'un bâtiment, un constructeur permettant d'initialiser ce champ, et un accesseur.
   Correction :

```
public class Batiment {
  private double surfaceH;

  public Batiment(double surfaceH) {
    this.surfaceH = surfaceH;
  }

  public double getSurfaceH() {
    return surfaceH;
  }
}
```

2. Définir une extension `Maison` de la classe `Batiment` contenant deux champs supplémentaires : `nbPieces` représentant le nombre de pièces d'une maison,

`surfaceJ` représentant la surface de son jardin. Écrire un constructeur pour cette classe ainsi qu'un accesseur pour la surface de jardin.

Correction :

```
public class Maison extends Batiment {
  private int nbPieces;
  private double surfaceJ;

  Maison(double surfaceH, double surfaceJ, int nbPieces)
      {
        super(surfaceH);
        this.nbPieces = nbPieces;
        this.surfaceJ = surfaceJ;
      }
  public double getSurfaceJ() {
    return surfaceJ;
  }
}
```

3. Écrire une méthode statique

   `static double surfaceH(Batiment[] tab)`

   renvoyant la somme de toutes les surfaces habitables des bâtiments référencés dans le tableau `tab`.

   Correction :

```
public static double surfaceH(Batiment[] tab){
    double res = 0.0;
    for(Batiment b : tab) {
      if (b != null) {
        res += b.getSurfaceH();
      }
    }
    return res;
}
```

4. Si l'on souhaitait écrire une méthode statique

   `static double surfaceJ(Batiment[] tab)`

   calculant la somme des surfaces des jardins de tous les bâtiments référencés dans le tableau `tab` qui sont aussi des maisons, quel problème rencontrerait-on ? Quelle solution proposeriez-vous ?

   Correction : L'accesseur `getSurfaceJ()` n'est pas défini dans la classe `Batiment`, mais seulement dans la classe `Maison`. Une solution consisterait à le définir également dans `Batiment`, où il renverra par défaut `0`.

   Dans `Batiment.java` :

```
  public double getSurfaceJ() {
        return 0;
  }
```

Dans le `main` :

```
public static double surfaceJ(Batiment[] tab) {
  double res = 0;
  for (Batiment b : tab) {
    if (b != null) {
      res += b.getSurfaceJ();
    }
  }
  return res ;
 }
```

5. L'impôt local d'un bâtiment est calculé selon la formule

$$impot = tauxA \times (surface\ habitable) + tauxB \times (surface\ jardin)$$

Les valeurs de cette année étant $tauxA = 5.6$ et $tauxB = 1.5$.

Où et sous quelle forme déclarer les champs `tauxA` et `tauxB` représentant ces taux ? Dans quelle(s) classe(s) faut-il implémenter la méthode `double impot()` calculant cet impôt local, et comment ?

Correction : On peut soit définir `tauxA` et `tauxB` dans `Batiment`, soit `tauxA` dans `Batiment` et `tauxB` dans `Maison` (dans ce dernier cas, il sera nécessaire de redéfinir `impot()` dans `Maison`). `tauxA` et `tauxB` doivent être `static` et `final`.

Une des solutions :

Dans `Batiment` :

```
static final double tauxA = 5.6;
public double impot() {
  return tauxA * surfaceH;
}
```

Dans `Maison` :

```
static final double tauxB = 1.5;
public double impot() {
  return super.impot() + tauxB * surfaceJ;
}
```