TD 2 : références et héritage

1 Références

Considérons le fichier `References.java` suivant :

```
class Point
{
    int x,y;
    void afficher()
    {
        System.out.println("x: "+x+"\ty:"+y);
    }
    Point(int x0,int y0) { x=x0; y=y0; }
}

class References
{
    public static void main(String [] args)
    {
        int i1=5;
        int i2=10;
        int i3=i2;
        System.out.println("i1: "+ i1 + " i2:"+ i2 + " i3:" + i3);
        i3=100;
        System.out.println("i1: "+ i1 + " i2:"+ i2 + " i3:" + i3);

        Point p1=new Point(5,5);
        Point p2=new Point(10,10);
        Point p3=p1;
        p1.afficher(); p2.afficher(); p3.afficher();
        p3.x=100;
        p1.afficher(); p2.afficher(); p3.afficher();
    }
}
```

- Combien d'entiers sont crées en mémoire ?
- Combien d'objets `Point` sont crées en mémoire ?
- Qu'est-ce que ce programme affiche ?
- Ajoutez une méthode `addition` à la classe `References` qui ajoute 5 à la coordonnée d'un point.

2 Classes et objets

Créez une classe `Lapin` qui contient les champs suivants :
- nom,
- age,
- position (un entier),
- longueur des poils.

Ajoutez également à la classe `Lapin` une méthode `avancer` qui ajoute une valeur aléatoire comprise entre -4 et 5 à la position. Créez une classe `Course` dans laquelle on trouvera la méthode `main`. Dans cette méthode, créez plusieurs lapins ayant des caractéristiques différentes. Créez ensuite un tableau pour y ranger les lapins. Faites ensuite en sorte que les tous lapins avancent 100 fois de suite, en affichant à chaque fois la nouvelle position du lapin.

Reprenez votre programme, et décrivez chacun de ces éléments. Il ne s'agit pas de décrire le comportement du programme, mais le rôle de chaque élément syntaxique. Ainsi pour

```
int a = 1;
```

on ne dira pas *on met la valeur 1 dans a*, mais *on déclare un entier a initialisé à 1.*

3 Héritage

On veut créer une classe `Tortue` qui partage certaines caractéristiques avec le lapin. Il y aura cependant des différences, notamment une caractéristique `tailleCarapace` à la place de la longueur des poils. Donnez deux manières de le faire, avec les avantages et les inconvénients.

Créez une classe `Animal` qui regroupe les caractéristiques communes au lapin et à la tortue. Un lapin *est un* animal, de même qu'une tortue *est un* animal. Cette notion est celle d'héritage, exprimée en Java par le mot clé `extends` :

```
class Lapin extends Animal
{
}

class Tortue extends Animal
{
}
```

Toutes les données de la classe `Animal` sont alors automatiquement intégrées dans les classes `Lapin` et `Tortue`. Écrivez les classes `Lapin` et `Tortue` en utilisant l'héritage. La méthode `avancer` de la classe `Tortue` fera avancer la tortue de 1 ou de 2.

Dans le `main`, ajoutez un tableau de tortues, et faites les avancer en même temps que les lapins, en affichant leurs positions respectives.

4 Polymorphisme

Dans le `main` précédent, vous pouvez remarquer que des parties du code sont redondantes : les déclarations des tableaux, de même que la "course" sont les mêmes pour les tortues et les lapins. Cette redondance n'est pas souhaitable du point de vue du génie logiciel, et peut être supprimée.

Créez deux méthodes vides dans la classe `Animal`, `avancer` et `afficher`. Dans la fonction `main`, créez un tableau d'objet `Animal`, et rangez les lapins *et* les tortues dans ce tableaux. Qu'est-ce que cette opération a de particulier ? Faites ensuite avancer chque animal 100 fois de suite, en l'affichant à chaque fois. Même question que précédemment, qu'est-ce que cette opération a de particulier ? À quoi servent les méthodes `avancer` et `afficher` de la classe `Animal` ?

5 Bilan

À quoi sert l'héritage ? À quoi sert le polymorphisme ?

Pour conclure, l'héritage et le polymorphisme servent à *réduire* la complexité d'un logiciel. Les hiérarchies de classes peuvent rapidement devenir elles-mêmes très complexes, et il s'agit alors de bien réfléchir avant de les créer, et surtout d'essayer de les faire les plus *simples* possibles.