Université de Paris **Programmation Orientée Objet**
L2 **Année 2022-2023**

TD - Séance2 n°3

Héritage

1 Héritage

Exercice 1 On définit les classes `A`,`B`,`C` de la manière suivante :

```
1  public class A {
2    public void g() {
3      System.out.println(0);
4    }
5  }
6  public class B extends A {
7    public void g() {
8      System.out.println(1);
9    }
10 }
11 public class C extends A {}
```

et on écrit un programme de test qui contient le code suivant :

```
1  public class Test {
2    public static void main(String[] args) {
3      A[] tab = new A[3];
4      tab[0] = new A();
5      tab[1] = new B();
6      tab[2] = new C();
7      for (int i=0; i<3; i++) { tab[i].g(); }
8    }
9  }
```

Que se passe-t-il à l'exécution ? Pourquoi n'a-t-on pas besoin de definir les constructeurs ?

2 Modélisation

Exercice 2 On définit une classe `Personne` de la manière suivante :

```
1  public class Personne {
2    private String name;
3    public Personne(String name) {
4      this.name = name;
5    }
6
7    public String toString() {
8      return "Je m'appelle " + this.name + ". ";
9    }
10 }
```

On veut ici illustrer la notion d'héritage en modélisant la structure de la société française au moyen-âge. Cette structure reposait sur une division en trois ordres : la noblesse (les nobles), le clergé (les prêtres) et le tiers-état (les roturiers).

1. Définir des classes `Noble`, `Pretre` et `Roturier`, qui héritent de `Personne`, de telle sorte que l'exécution du code suivant produise :
   Je m'appelle Louis. Je suis un noble.

```
1 public static void main(String args[]) {
2   Personne n = new Noble("Louis");
3   System.out.println(n);
4 }
```

2. On ajoute maintenant à la classe `Personne` :

```
1 private int argent = 0;
2 public void recevoirArgent(int i) {
3   this.argent += i;
4 }
```

   Ajouter une fonction `boolean donnerArgent(int i)`, renvoyant `false` lorsque la personne n'a pas assez d'argent sur elle.

3. On considère maintenant que les nobles ont le droit de contracter des dettes (avoir une somme d'argent négative). Que doit-on rajouter au code, et où ?

4. Le tiers-état est un ordre très hétérogène socialement, qui comprend à la fois des paysans, des artisans et des bourgeois. Quelles classes doit-on créer pour modéliser cela ? De quelle classe doivent-elles hériter ?

5. On considère maintenant une classe `Societe`, qui a comme attribut un tableau de `Personne`.
   — Écrire un constructeur de `Societe`, qui prend en argument un entier n et qui crée une société de n personnes, de rôle social choisi aléatoirement. Le nom de i-ème personne de la societé peut être `Personne_i`. On peut utiliser `Math.random()` (qui se trouve dans le package `java.lang`), qui renvoie un `double` entre `0.0` (inclus) et `1.0` (exclus), ou un objet de la classe `Random` (qui se trouve dans le package `java.util`), dont une utilisation possible est la suivante :

```
1   Random r = new Random();
2   int i = r.nextInt(5); // i est pris dans {0,1,2,3,4}
```

   — Écrire une méthode : `public int nbPaysan()`, qui renvoie le nombre de paysans dans la société. On peut utiliser pour cela l'opérateur `instanceof`, qui permet de déterminer si un objet est une instance d'une classe, ou d'une de ses classes dérivées. Exemple :

```
1 class Felin{}
2 class Chat extends Felin{}
3 public class Test{
4   public static void main(String[] args){
5     Felin f1 = new Felin();
6     Felin f2 = new Chat();
7     Chat f3 = new Chat();
```

```
8       Chat f4 = null;
9       System.out.println(f1 instanceof Felin); // true
10      System.out.println(f2 instanceof Felin); // true
11      System.out.println(f3 instanceof Felin); // true
12      System.out.println(f4 instanceof Felin); // false
13      System.out.println(f1 instanceof Chat); // false
14      System.out.println(f2 instanceof Chat); // true
15      System.out.println(f3 instanceof Chat); // true
16      System.out.println(f4 instanceof Chat); // false
17    }
18  }
```

— Écrire une méthode : `public int argentTotal()`, qui renvoie la somme de l'argent que chaque roturier membre de la société possède. (On peut avoir besoin d'une méthode supplémentaire dans une autre classe.)

Exercice 3 On crée maintenant une classe `Percepteur`, qui correspond à un collecteur d'impôt. Il a comme attribut une société (l'ensemble des gens qu'il peut taxer). Néanmoins, seuls les roturiers doivent payer des impôts (attention, historiquement c'est faux).

1. Écrire un constructeur `Percepteur(int n)`, qui crée un percepteur avec une quantité d'argent initiale égale à 0 et ayant comme attribut une société de taille n, initialisée aléatoirement.
2. Ajouter à la classe `Percepteur` une méthode `public void impot()` telle que : de chaque membre de sa société, le percepteur prend pour sa propre poche une pièce d'argent si c'est un roturier – et qu'il a encore de l'argent. (On peut avoir besoin d'une méthode supplémentaire dans une autre classe.)
3. Les percepteurs ont une certaine autonomie dans la collecte des impôts. Écrire une classe `PercepteurProportionnel`, qui hérite de `Percepteur`, pour que l'imposition soit proportionnelle à l'argent que possède la personne.
4. Écrire une classe `PercepteurInegalitaire`, avec le même comportement qu'un `PercepteurProportionnel`, mais dont le taux d'imposition dépend de sa position sociale : paysan, artisan ou bourgeois. De quelle classe doit hériter `PercepteurInegalitaire` ?
5. Écrire une classe `PercepteurAgricole`, avec le même comportement qu'un `PercepteurProportionnel`, mais qui ne prend de l'argent qu'aux paysans uniquement. De quelle classe doit hériter `PercepteurAgricole` ?

Exercice 4 On va maintenant légèrement changer l'implémentation de `Societe` pour qu'elle évolue avec de nouvelles naissances et des morts des membres de la société. On va pour cela ajouter un champ `age` à la classe `Personne`, et le tableau de `Personne` de la classe `Societe` va devenir une `LinkedList<Personne>`.

1. Ajouter une méthode `boolean anniversaire()` à la classe `Personne` faisant vieillir une personne d'un an et renvoyant `true`.

2. Redéfinir `anniversaire()` pour les classes héritées de `Personne` de façon à ce que la méthode renvoie `false` si l'individu a atteint un âge trop avancé (on donnera des espérances de vie différentes aux différentes classes).
3. Ajouter à la classe `Personne` une méthode `Personne enfanter()` renvoyant `null` et la redéfinir dans les différentes classes de façon à ce que cette méthode :
   — Renvoie `null` si l'individu est trop jeune.
   — Renvoie une nouvelle personne sinon. Le type de cette personne depend de la classe dont on appelle la méthode (un noble engendrera un noble ou un pretre, un prêtre sera sans enfant, et les roturiers engendre des prêtres ou des roturiers, de professions variées)
4. Ajouter une méthode `void anniversaire()` à la société qui fait s'écouler un an. Pendant cette année :
   — Chaque membre de la société vieillit d'un an. Ayant atteint son âge limite, la personne est retirée de la société.
   — Chaque membre adulte de la société a une petite chance d'engendrer un enfant, ajouté à la société.

   Remarque : on peut utiliser dans cet exercice le fait qu'une fonction redéfinie peut renvoyer un type plus précis, e.g :

```
1  class A{
2    public A create(){
3      return new A();
4    }
5  }
6  class B extends A{
7    public B create(){ //on redéfinit create(), cela ne pose
8      return new B(); //pas de problèmes comme B extends A
9    }
10 }
```