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

TD - Séance n°3 - Correction

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 ?

Correction : Réponse :

```
1  0
2  1
3  0
```

Au passage, mentionner le constructeur par defaut pour chaque des 3 classes ci-dessus :

```
1  public class A {
2    public A() {}
3    ...
4  }
5  public class B extends A {
6    public B() {} // super() implicite
7    ...
8  }
9  public class C extends A {
10   public C() {} // super() implicite
11   ...
12 }
```

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  Personne n = new Noble("Louis");
2  System.out.println(n);
```

Correction : À écrire : constructeur de la classe Noble ; redéfinition de la méthode toString :

```
1  public class Noble extends Personne {
2    public Noble(String name) {
3      super(name);
4    }
5
6    public String toString() {
7      return super.toString() + "Je suis un noble.";
8    }
9  }
```

```
1  public class Pretre extends Personne {
2    public Pretre(String name) {
3      super(name);
4    }
5    public String toString() {
6      return super.toString() + "Je suis un prêtre.";
7    }
8  }
```

```
1  public class Roturier extends Personne {
2    public Roturier(String name) {
3      super(name);
4    }
5    public String toString() {
6      return super.toString() + "Je suis un roturier.";
7    }
8  }
```

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

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

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

Correction :

```
  public boolean donnerArgent(int i) {
    if (this.argent < i) {
      return false;
    }
    this.argent -= i;
    return true;
  }
```

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 ?

Correction : argent étant un attribut privé de Personne, il n'est pas hérité par Noble. Il faut un setteur publique dans la classe Personne avant de redéfinir la méthode donnerArgent. Ecrivez aussi un getteur, on en aura besoin plus tard :

Personne.java :

```
  public void setArgent(int i) {
    this.argent = i;
  }

  public int getArgent() {
    return this.argent;
  }
```

Noble.java :

```
  public boolean donnerArgent(int i) {
    setArgent(getArgent() - i);
    return true;
  }
```

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 ?

Correction : On crée les classes `Paysan`, `Artisan`, `Bourgeois`, qui héritent de `Roturier`.

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 :

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

Correction :

```
import java.util.Random;

public class Societe {
  private Personne[] personnes;

  public Societe(int n) {
    double intervalle[] = {.1, .2, .5, .8};
    personnes = new Personne[n];
    for (int i = 0; i < n; i++) {
      double role = Math.random();
      Random r = new Random();
      int argent = r.nextInt(100);
      if (role <= intervalle[0]) {
        personnes[i] = new Noble("Noble " + i);
      }
      else if (role <= intervalle[1]) {
        personnes[i] = new Pretre("Pretre " + i);
      }
      else if (role <= intervalle[2]) {
        personnes[i] = new Paysan("Paysan " + i);
      }
      else if (role <= intervalle[3]) {
        personnes[i] = new Artisan("Artisan " + i);
      }
      else {
        personnes[i] = new Bourgeois("Bourgeois " + i);
      }
      personnes[i].setArgent(argent);
    }
  }
}
```

— 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 :

```
class Felin{}
class Chat extends Felin{}
public class Test{
  public static void main(String[] args){
    Felin f1 = new Felin();
    Felin f2 = new Chat();
    Chat f3 = new Chat();
    Chat f4 = null;
    System.out.println(f1 instanceof Felin); // true
    System.out.println(f2 instanceof Felin); // true
    System.out.println(f3 instanceof Felin); // true
    System.out.println(f4 instanceof Felin); // false
    System.out.println(f1 instanceof Chat); // false
    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  }
```

Correction :

```
1   public int nbPaysan() {
2     int res = 0;
3     for (int i = 0; i < personnes.length; i++) {
4       if (personnes[i] instanceof Paysan) {
5         res++;
6       }
7     }
8     return res;
9   }
```

— 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.)

Correction : On a besoin d'un getteur pour l'attribut argent dans la classe Personne.

Societe.java :

```
1   public int argentTotal() {
2     int res = 0;
3     for (int i = 0; i < personnes.length; i++) {
4       if (personnes[i] instanceof Roturier) {
5         res += personnes[i].getArgent();
6       }
7     }
8     return res;
9   }
```

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.

   Correction :

```
1  public class Percepteur {
2    protected Societe societe;
3    protected int argent;
4
5    public Percepteur(int n) {
6      argent = 0;
7      societe = new Societe(n);
8    }
9  }
```

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.)

   Correction : On a besoin d'un getteur pour le tableau de personnes de la societé dans la classe Societe. Attention de ne pas retourner directement le tableau -car cela donnerait accès aux utilisateurs des autres classes (par ex de la classe Perceptuer) de changer les valeurs de l'attribut privé de la classe Societe, mais plutôt une copie de l'attribut.

   Percepteur.java :

```java
  public void impot() {
    Personne[] personnes = societe.getPersonnes();
    for (int i = 0; i < personnes.length; i++) {
      if (personnes[i] instanceof Roturier) {
        if (personnes[i].donnerArgent(1)) {
          argent += 1;
        }
      }
    }
  }

  public int getArgent() {
    return argent;
  }
```

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.

   Correction : PercepteurProportionnel.java :

```java
public class PercepteurProportionnel extends Percepteur {
  protected double proportion;
  public PercepteurProportionnel(int n, double proportion) {
    super(n);
    // assumption: 0.0 <= proportion <= 1.0
    this.proportion = proportion;
  }

  public void impot() {
    Personne[] personnes = societe.getPersonnes();
    for (int i = 0; i < personnes.length; i++) {
      if (personnes[i] instanceof Roturier) {
        double impots = personnes[i].getArgent() * proportion
            ;
        if (personnes[i].donnerArgent((int)impots)) {
          super.argent += impots;
        }
      }
    }
  }
}
```

4. Écrire une classe `PercepteurInegalitaire`, tel que le taux d'imposition dépende de sa position sociale : paysan, artisan ou bourgeois. De quelle classe doit hériter `PercepteurInegalitaire` ?

   Correction : Directment de la classe Percepteur

   PercepteurInegalitaire.java :

```java
public class PercepteurInegalitaire extends Percepteur {
  protected double proportion_paysan;
  protected double proportion_artisan;
  protected double proportion_bourgeois;
  public PercepteurInegalitaire(int n,
      double proportion_paysan,
      double proportion_artisan,
      double proportion_bourgeois) {
    super(n);
    // assumption: 0.0 <= proportion <= 1.0
    this.proportion_paysan = proportion_paysan;
    this.proportion_artisan = proportion_artisan;
    this.proportion_bourgeois = proportion_bourgeois;
  }

  public void impot() {
    Personne[] personnes = societe.getPersonnes();
    for (int i = 0; i < personnes.length; i++) {
      if (personnes[i] instanceof Roturier) {
        double impots;
        if (personnes[i] instanceof Paysan) {
          impots = personnes[i].getArgent() *
              proportion_paysan;
        }
        else if (personnes[i] instanceof Artisan) {
          impots = personnes[i].getArgent() *
              proportion_artisan;
        }
        else {
          impots = personnes[i].getArgent() *
              proportion_bourgeois;
        }
        if (personnes[i].donnerArgent((int)impots)) {
          super.argent += impots;
        }
      }
    }
  }
}
```

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>`.

Correction : Changer le contructeur de la classe Societe et tout autre part affecté.

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 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 }
```

Correction : Dans Societe.java, il faut utiliser `add` pour ajouter à la liste de personnes, `size` pour obtenir la taille, et `get` pour accéder à un élément.

```
1  import java.util.Random;
2  import java.util.LinkedList;
3
4  public class Societe {
5    private LinkedList<Personne> personnes;
6
7    public Societe(int n) {
8      double intervalle[] = {.1, .2, .5, .8};
9      personnes = new LinkedList<Personne>();
10     for (int i = 0; i < n; i++) {
11       double role = Math.random();
12       Random r = new Random();
13       if (role <= intervalle[0]) {
14         personnes.add(new Noble("Noble " + i));
15       }
16       else if (role <= intervalle[1]) {
17         personnes.add(new Pretre("Pretre " + i));
18       }
19       else if (role <= intervalle[2]) {
20         personnes.add(new Paysan("Paysan " + i));
```

```
      }
      else if (role <= intervalle[3]) {
        personnes.add(new Artisan("Artisan " + i));
      }
      else {
        personnes.add(new Bourgeois("Bourgeois " + i));
      }
    }
  }

  public LinkedList<Personne> getPersonnes() {
    LinkedList<Personne> cloned = new LinkedList<Personne>();
    for (int i = 0; i < personnes.size(); i++) {
      cloned.add(personnes.get(i));
    }
    return cloned;
  }

  public void anniversaire() {
    // we take the population size before the loop, such
    // that new children will not be taken into account
    int population_size_last_year = personnes.size();
    for (int i = 0; i < population_size_last_year; i++) {
      if (!personnes.get(i).anniversaire()) {
        personnes.remove(i);
        // the list will be shifted after the remove,
        // so we have to take this into account
        i--;
        population_size_last_year--;
      } else {
        boolean nouveau_enfant = (Math.random() <= .05);
        if (nouveau_enfant) {
          Personne enfant = personnes.get(i).enfanter();
          if (enfant != null) {
            personnes.add(enfant);
          }
        }
      }
    }
  }
}
```

Personne.java :

```
public class Personne {
  private String name;
  protected int age;
  private final int esperance_de_vie;
  public static final int age_majeur = 18;

  public Personne(String name, int esperance) {
    this.name = name;
    this.esperance_de_vie = esperance;
  }

  public boolean anniversaire() {
    if (this.age + 1 > esperance_de_vie) {
      return false;
    }
```

```
16      this.age += 1;
17      return true;
18    }
19
20    public Personne enfanter() {
21      return null;
22    }
23  }
```

Noble.java :

```
1  public class Noble extends Personne {
2    public static final int esperance_de_vie = 90;
3
4    public Noble(String name) {
5      super(name, esperance_de_vie);
6    }
7
8    public Personne enfanter() {
9      if (this.age < Personne.age_majeur) {
10       return null;
11     }
12     double role = Math.random();
13     if (role < .5) {
14       return new Noble("");
15     } else {
16       return new Pretre("");
17     }
18   }
19 }
```

Pretre.java ;

```
1  public class Pretre extends Personne {
2    public static final int esperance_de_vie = 90;
3
4    public Pretre(String name) {
5      super(name, esperance_de_vie);
6    }
7
8    // un prêtre sera sans enfants donc on ne remplace
9    // pas la fonction enfanter().
10 }
```

Roturier.java :

```
1  public class Roturier extends Personne {
2    public Roturier(String name, int esperance) {
3      super(name, esperance);
4    }
5
6    public Personne enfanter() {
7      if (this.age < Personne.age_majeur) {
8        return null;
9      }
10     double role = Math.random();
11     if (role < .4) {
12       return new Pretre("");
13     } else if (role < .6) {
14       return new Paysan("");
```

```
15      } else if (role < .8) {
16        return new Artisan("");
17      } else {
18        return new Bourgeois("");
19      }
20    }
21  }
```

Paysan.java :

```
1  public class Paysan extends Roturier {
2    public static final int esperance_de_vie = 60;
3
4    public Paysan(String name) {
5      super(name, esperance_de_vie);
6    }
7  }
```

Artisan.java :

```
1  public class Artisan extends Roturier {
2    public static final int esperance_de_vie = 66;
3
4    public Artisan(String name) {
5      super(name, esperance_de_vie);
6    }
7  }
```

Bourgeois.java :

```
1  public class Bourgeois extends Roturier {
2    public static final int esperance_de_vie = 75;
3
4    public Bourgeois(String name) {
5      super(name, esperance_de_vie);
6    }
7  }
```

Test.java :

```
1  public class Test {
2    public static void main(String[] args) {
3      Societe france = new Societe(100);
4      for (int i = 0; i < 50; i++) {
5        france.anniversaire();
6        System.out.println("nombre de personnes : " + france.
             getPersonnes().size());
7      }
8    }
9  }
```