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

TD - Séance 5 - Correction

Héritage

Dans ce TD nous allons continuer à manipuler le concept d'héritage.

Exercice 1 On considère comme lors du TD3 une classe `Personne`, avec des attributs légèrement modifiés :

```
1  public class Personne {
2
3    protected String nom;
4    protected int argent;
5    protected int pdv; // les points de vie de la personne
6
7    public Personne(String nom, int argent, int pdv) {
8      this.nom = nom;
9      this.argent = argent;
10     this.pdv = pdv;
11   }
12
13   public int getArgent() {
14     return argent;
15   }
16
17   public void gain(int n) {
18     this.argent += n;
19   }
20
21   public boolean perte(int n) {
22     if (n <= this.argent) {
23       this.argent -= n;
24       return true;
25     } else {
26       return false;
27     }
28   }
29
30   public void blessure(int n) {
31     this.pdv -= n;
32   }
33
34
35   public void attaque(Personne p) {
36     p.blessure(0);
37   }
38
39   public String toString() {
```

```
40      return "Je m'appelle " + this.nom + ". J'ai "
41        + this.argent + " unités monétaires, et "
42        + this.pdv + " points de vie.";
43    }
44  }
```

Comme dans le TD3, on a des classes `Noble`, `Pretre` et `Roturier` qui héritent de `Personne`.

Dans cet exercice, nous allons modéliser l'une des activités favorites de la noblesse au Moyen-âge : la guerre.

On créera pour cela une classe `Chevalier` (i.e les combattants de la noblesse) héritant de `Noble` ainsi que des classes `Archer` et `Fantassin` héritant de `Roturier`. La classe `Fantassin` possède un attribut `int degats` qui indique le nombre des points de vie à retirer lors d'une attaque. Les comportements militaires seront modélisés par une méthode `void attaque(Personne p)` et devront être implémentés par `Chevalier`, `Archer` et `Fantassin`.

1. Proposez des modifications à la méthode `blessure` pour éviter les valeurs négatives et afficher un message en cas de mort de la personne (lorsque ses points de vie tombent à zéro).
2. Redéfinissez la méthode `attaque` dans les classes `Chevalier`, `Archer` et `Fantassin` (l'argument de la méthode `attaque` représente la personne attaquée), sachant que :
   (a) un `Archer` tue la personne qu'il attaque (i.e il lui enlève tous ses points de vie).
   (b) Un `Chevalier` n'attaque une personne que si elle est elle-même une instance de `Chevalier` et dans ce cas le `Chevalier` attaqué est capturé (i. e. il perd sa liberté et il ne peut plus attaquer). On rajoute pour cela un attribut `Personne geolier` dans la classe `Chevalier` indiquant qui a capturé ce `Chevalier`, ou valant `null` s'il est libre. Pourquoi écrire une méthode `void attaque(Chevalier p)` ne répondrait pas à la question ?
   Correction : La classe `Chevalier` hérite de `Noble` qui hérite de `Personne`, et donc `Chevalier` hérite la méthode `void attaque(Personne p)` de la classe `Personne`. En definissant `void attaque(Chevalier p)`, on *surchargerait* seulement cette méthode, on ne la remplacerait (override) pas. La méthode `void attaque(Personne p)` serait donc toujours disponible, et conturne donc les modifications qu'on a voulu faire pour cette méthode dans la classe `Chevalier`.
   (c) un `Fantassin` capture la personne qu'il attaque si celle-ci est une instance de `Chevalier` et enlève `degats` points de vie à la personne si celle-ci n'est pas une instance de `Chevalier`.
3. Lorsqu'un chevalier est capturé, il a la possibilité de payer une rançon pour racheter sa liberté. Rajouter dans `Chevalier` une méthode `boolean acheteLiberte()` qui fonctionne de la façon suivante : si le chevalier capturé a les moyens de payer la rançon (i. e. son attribut `argent` est

supérieur au montant de la rançon) alors il paye la rançon à son geôlier et regagne sa liberté. La méthode renvoie `true` si, et seulement si la rançon a été payée. Le montant de la rançon étant le même pour tous les chevaliers, vous le définirez dans un champ
`private static final int prixLiberté = 50;`

Exercice 2 La guerre au Moyen-âge donnait rarement lieu à des batailles rangées mais consistait principalement en escarmouches et en pillages. Le but de cet exercice est de modéliser les pillages.

1. Créer une classe `Village` possédant un attribut `LinkedList<Roturier>`.
2. Créer une classe `Condottiere` héritant de `Personne` et possédant des attributs `LinkedList<Archer>` et `LinkedList<Fantassin>`.
3. Créer une méthode `void attaque(Village v)` dans la classe `Condottiere`. Lorsqu'un `Condottiere` attaque un village, ce dernier est mis à sac (i. e. chaque villageois se fait voler la moitié de son argent). Les gains récupérés sont répartis pour moitié entre le condottiere et pour l'autre moitié entre les membres de sa compagnie.

Exercice 3 Pour des raisons techniques (sauvegarde d'un état par exemple), on souhaite avoir la possibilité de cloner les villages. Pour cela nous allons redéfinir la méthode `public Object clone()` que `Village` hérite de la classe `Object`. En effet cette méthode de clonage par défaut se contente d'une copie superficielle, souvent insuffisante.

1. Commencez par redéfinir la méthode `clone` dans la classe `Roturier`. Pensez à signaler à Java que la classe implémente l'interface `Cloneable` et que la méthode `clone` peut lever l'exception `CloneNotSupportedException`. N'hésitez pas à relire le cours correspondant.
   Correction : Si on essaie le suivant pour cloner une personne

```
1    Personne foo = new Personne("Moi", 10000, 99);
2    Personne bar = foo.clone();
```

   la compilation échoue avec le message d'erreur suivant :

```
1    correction/Test.java:21: error: clone() has protected access
         in Object
2            Personne bar = foo.clone();
3                           ^
```

   C'est parce que `clone()` dans la classe `Objet` est definie comme suit :

```
1    protected Object clone() throws CloneNotSupportedException
```

   ce qui veut dire que seulement des classes qui héritent de `Objet` peuvent accéder à cette méthode de clonage par défaut. Et c'est ce que nous faisons dans `Roturier` :

```
1    @Override
2    public Roturier clone() throws CloneNotSupportedException {
3        return (Roturier) super.clone();
4    }
```

Avec le mot clé `public` nous rendons accessible `clone()` à tout le monde.

2. Redéfinissez ensuite la méthode `clone` dans la classe `Village`. Soyez attentifs au clonage des habitants.

   Correction : L'utilisation de `super.clone()` est une convention qu'on respecte idéalement : Si une classe et toutes ces classes parents (sauf `Objet`) respectent cette convention, on a la garantie que `x.clone().getClass() == x.getClass()`.

   Par exemple, la bonne solution pour `clone()` dans la classe `Roturier` est :

```
1 |     @Override
2 |     public Roturier clone() throws CloneNotSupportedException {
3 |         return (Roturier) super.clone();
4 |     }
```

   Maintenant, si on fait

```
1 |     Roturier fanta = new Fantassin("Fanta4", 4, 100);
2 |     Roturier fanta2 = fanta.clone();
3 |     System.out.println(fanta.getClass());
4 |     System.out.println(fanta2.getClass());
```

   cela affiche bien

```
1 |     class Fantassin
2 |     class Fantassin
```

   Par contre, si on utilisait l'implementation suivante, qui crée un nouveau objet de type `Roturier` manuellement :

```
1 |     @Override
2 |     public Roturier clone() throws CloneNotSupportedException {
3 |         return new Roturier(this.nom, this.argent, this.pdv);
4 |     }
```

   on a

```
1 |     class Fantassin
2 |     class Roturier
```

   Donc cette version de `clone()` ignore le fait que `fanta` était crée comme `Fantassin`.

Exercice 4 On souhaite également pouvoir comparer des villages entre eux. Faites en sorte que la méthode `public boolean equals(Object o)`, initialement définie dans la classe `Object`, implémente les notions d'égalité suivantes.

1. Deux roturiers sont identiques lorsqu'ils ont le même nom, la même quantité d'argent et le même nombre de points de vie, ainsi que la même classe (i.e. `Archer` ou `Fantassin`) s'ils en ont une.
2. Deux villages sont identiques s'ils ont les mêmes habitants (pour simplifier on pourra dans un premier temps considérer qu'ils doivent avoir leurs habitants listés dans le même ordre pour être identiques).