Algorithmes et Pensée Computationnelle

Programmation orientée objet - Exercices avancés

Le but de cette séance est de se familiariser avec un paradigme de programmation couramment utilisé : la Programmation Orientée Objet (POO). Ce paradigme consiste en la définition et en l'interaction avec des briques logicielles appelées **Objets**. Dans les exercices suivants, nous manipulerons des objets, aborderons les notions de classe, méthodes, attributs et encapsulation. Au terme de cette séance, vous serez en mesure d'écrire des programmes mieux structurés.
Les languages de programmation qui seront utilisés pour cette série d'exercices sont Java et Python.
Le temps indiqué (🕒) est à titre indicatif.

1 Interaction entre plusieurs instances d'une même classe (30 minutes)

Dans cette section, nous allons simuler un jeu de combat entre deux protagonistes représentant des instances d'une classe **Fighter** que nous allons créer. Chaque **Fighter** aura des attributs qui le définissent. Ces attributs sont :

- **nom:***(String)* chaque combattant sera identifié par un nom unique.
- **health:***(int)* représentant le nombre de points de vie d'un combattant. Il contient des valeurs comprises entre 0 et 10. À l'instanciation de l'objet, le combattant a 10 points de vie par défaut-
- **attaque:***(int)* représentant une valeur qui sera utilisée pour calculer le nombre de points de dégâts infligés à l'adversaire.
- **défense:***(int)* représentant une valeur qui sera utilisée pour calculer le nombre de points de dégâts reçus.

Deux attributs de classe seront également utilisés :

- **instances :** Liste comprenant les combattants qui ont été instanciés et qui sont toujours en vie.
- **attackModifier :** Dictionnaire comportant 3 types d'attaques, chacune modifiant les dégâts qui vont être infligés. Les trois types d'attaques sont **poing, pied et tête** modifiant respectivement l'attaque par 1, 2, 3.

Le but de cette partie est d'étudier les interactions entre deux instances d'une même classe. Cette classe se présentera sous la forme d'un **Fighter**. Chaque instance de la classe **Fighter** pourra attaquer les autres instances.
Vous devrez compléter les 4 méthodes suivantes :

1. **isAlive()**
2. **checkDead()**
3. **checkHealth()**
4. **attack(String type, Fighter other)**

Voici le squelette du code (à télécharger sur Moodle) :

```
1  import java.util.HashMap;
2  import java.util.List;
3  import java.util.ArrayList;
4  import java.util.Map;
5
6  public class Fighter {
7      private String name;
8      private int health;
9      private int attack;
10     private int defense;
11     private static List<Fighter> instances = new ArrayList<Fighter>();
12     private static HashMap<String, Integer> attackModifier = new HashMap(Map.of("poing",1,"pied",2,"tete",3));
13
14     public Fighter(String name, int health, int attack, int defense) {
15         this.name = name;
16         this.health = health;
17         this.attack = attack;
18         this.defense = defense;
19         instances.add(this);
```

```
20      }
21
22      public int getAttack() {
23          return attack;
24      }
25
26      public int getHealth() {
27          return health;
28      }
29
30      public int getDefense() {
31          return defense;
32      }
33
34      public String getName() {
35          return name;
36      }
37
38      public void setAttack(int attack) {
39          this.attack = attack;
40      }
41
42      public void setDefense(int defense) {
43          this.defense = defense;
44      }
45
46      public void setHealth(int health) {
47          this.health = health;
48      }
49
50      public void setName(String name) {
51          this.name = name;
52      }
53
54      public Boolean isAlive() {
55          // à compléter
56      }
57
58      public static void checkDead() {
59          // à compléter
60      }
61
62      public static void checkHealth() {
63          // à compléter
64      }
65
66      public void attack (String type, Fighter other){
67          // à compléter
68      }
69  }
```

Question 1: (*5 minutes*) isAlive()

Définir une méthode **isAlive()** de type **boolean** qui retournera **true** si l'instance a plus que 0 points de vie et **false** si l'instance en a moins.

Solution

```
1       public boolean isAlive() {
2           if (this.health > 0) {
3               return true;
4           } else {
5               return false;
6           }
7       }
```

Question 2: (*10 minutes*) checkDead()

Définir une méthode **checkDead()** qui parcourt la liste des instances, et contrôle que chacune d'entre elle est encore en vie. Si ce n'est pas le cas, l'instance en question est supprimée de la liste des instances et le message "**nomInstance** est mort" sera affiché.

Conseil

Prenez le problème dans l'autre sens, créez une liste temporaire. Si l'instance est vivante, ajoutez la à cette nouvelle liste. Pour finir, mettez à jour votre liste d'instances à l'aide de votre liste temporaire.
L'attribut **instances** étant une liste, vous pouvez parcourir cette liste d'instances en utilisant une boucle **for**.

Solution

```
1  public static void checkDead() {
2    // Initialisation de la liste de Combattants en vie
3      List<Combattant> temp = new ArrayList<Combattant>();
4      //Ici, on parcourt les instances de Combattant
5      for (Combattant f : Combattant.instances) {
6        // Et on fait appel à la méthode isAlive() pour vérifier que le Combattant est en vie
7        if (f.isAlive()) {
8          temp.add(f);
9        } else {
10         System.out.println(f.getName() + " est mort");
11       }
12     }
13     Combattant.instances = temp;
14   }
```

Question 3: (*5 minutes*) checkHealth()
Définir une méthode **checkHealth()** qui parcourt la liste des instances et affiche le nombre de points de vie qui reste au combattant sous le format "**nomInstance** a encore **healthInstance** points de vie".

Solution

```
1  public static void checkHealth() {
2      for (Fighter f : Fighter.instances) {
3        System.out.println(f.getName() + " a encore " + f.getHealth() + " points de vie");
4      }
5    }
```

Question 4: (*10 minutes*) attack(String type Fighter other)
Définir une méthode **attack(String type, Fighter other)** qui permettra de retirer des points de vie au combattant **other** en fonction de l'attaque de l'instance appelée, du type d'attaque sélectionné et de la défense de **other**.

Commencez par contrôler si **other** est encore en vie.Si tel n'est pas le cas, indiquez qu'il est déjà mort : "**other_name** est déjà mort".
Si **other** est encore en vie, retirez des points de vie à **other**. Le nombre de points de vie devant être retiré se calcule en utilisant la formule suivante : **attack_modifier(type)** * **attack_instance** - **defense_other**. Appelez ensuite les fonctions **checkDead()** et **checkHealth()** afin d'avoir un aperçu des combattants restants et de leur santé.

Solution

```
1  public void attack (String type,Combattant other){
2    if(other.isAlive()) {
3      int damage = (Integer)Combattant.attack_modifier.get(type) * this.attack – other.getDefense();
4      other.setHealth(other.getHealth() – damage);
5      Combattant.checkDead();
6      Combattant.checkHealth();
7    }
8    else{
9      System.out.println(other.getName() + " est déjà mort");
10   }
11 }
```

Pour terminer, vous pouvez exécuter le code ci-dessous (disponible dans le dossier Code sur Moodle) pour vérifier que votre programme fonctionne correctement :

```
public class Main {
    public static void main(String[] args) {
        Fighter P1 = new Fighter("P1", 10, 2, 2);
        Fighter P2 = new Fighter("P2", 10, 2, 2);
        Fighter P3 =  new Fighter("P3", 10, 2, 2);
        P1.attack("pied",P2);
        P1.attack("poing",P2);
        P1.attack("tete",P2);
        P1.attack("tete",P2);
    }
}
```

Vous devriez obtenir ce résultat :

```
P1 a encore 10 points de vie
P2 a encore 8 points de vie
P3 a encore 10 points de vie
P1 a encore 10 points de vie
P2 a encore 8 points de vie
P3 a encore 10 points de vie
P1 a encore 10 points de vie
P2 a encore 4 points de vie
P3 a encore 10 points de vie
P2 est mort
P1 a encore 10 points de vie
P3 a encore 10 points de vie

Process finished with exit code 0
```

2 Héritage en Python

Question 5: (🕓 *15 minutes*) **Classe Point (Suite)**

Lors de la séance précédente, vous avez défini une classe **Point** représentant un point de 2 dimensions, avec des coordonnées **x** et **y**, ainsi que des opérations basiques sur des points 2D. Pour cet exercice, vous allez implémenter une classe de points à 3 dimensions qui héritera de la classe **Point** créée précédemment.
Voici la classe **Point** qui a été légèrement modifiée (Vous trouverez le fichier sur Moodle, dans le dossier **Code**) :

```
import math

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def get_y(self):
        return self._y

    def set_x(self, x):
        self._x = x

    def set_y(self, y):
        self._y = y

    def distance_euclidean(self, p2):
        return math.sqrt((self._x - p2.get_x()) ** 2 + (self._y - p2.get_y()) ** 2)

    def milieu(self, p2):
        x_M = (self._x + p2.get_x()) / 2
        y_M = (self._y + p2.get_y()) / 2
        M = Point(x_M, y_M)
        return M

    def __str__(self):
```

```
30        return "Les coordonnées du point sont: x=" + str(self.get_x()) + ", y=" + str(self.get_y())
```

Écrivez une classe qui hérite de **Point**. Nommez la **Point3D**. Après avoir rajouté la 3ème dimension comme attribut, effectuez les opérations ci-dessous :

- Rajoutez une méthode qui renvoie une représentation vectorielle du point. Vous pouvez utiliser une liste.
- Recalculez la distance euclidienne et le milieu pour le point 3D.
- Pour aller plus loin Si vous voulez vous familiariser encore plus avec les méthodes de classe en Python, implémentez deux autres algorithmes de calculs de distance : les distances de Manhattan et Minkowski.

```
1   class Point3D(Point):
2     def __init__(self, x, y, z):
3       super().__init__(x, y)
4       self.__z = z
5
6     def get_z(self):
7       ...
8
9     def set_z(self, z):
10      ...
11
12    def vector_representation(self): # représentée sous forme de liste
13      ...
14
15    def distance_euclidean(self, p2): # i.e norme
16      ...
17
18    def distance_manhattan(self, p2):
19      ...
20
21    def distance_minkowski(self, p2, order=3):
22      ...
23
24    def milieu(self, p2):
25      ...
```

Conseil

Que fait `super().__init__()` ?
Dans un espace à 3 dimensions, la formule pour calculer la distance entre un $p_1 = (x_1, y_1, z_1)$ et $p_2 = (x_2, y_2, z_2)$ est $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (y_1 - y_2)^2}$.

Solution

```
class Point3D(Point):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.__z = z

    def get_z(self):
        return self.__z

    def set_z(self, z):
        self.__z = z

    def vector_representation(self): # représentée sous forme de liste
        return [self.__x, self.__y, self.__z]

    def distance_euclidean(self, p2): # i.e norme
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()
        return math.sqrt((self.__x - other_x)**2 + (self.__y - other_y)**2 + (self.__z - other_z)**2)

    def distance_manhattan(self, p2):
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()
        return sum((abs(self.__x - other_x), abs()), abs(self.__y - other_y), abs(self.__z - other_z))

    def distance_minkowski(self, p2, order=3):
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()
        return sum((abs(self.__x - other_x)**order), abs(self.__y - other_y)**order,\
                abs(self.__z - other_z)**order)**(1/order)

    def milieu(self, p2):
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()

        x_M = (self.__x + other_x)/2
        y_M = (self.__y + other_y)/2
        z_M = (self.__z + other_z)/2
        return Point3D(x_M, y_M, z_M) # renvoie un point!


point1 = Point3D(1, 2, 3)
point2 = Point3D(3, 4, 5)

# exemple
point1.vector_representation()
```

Question 6: (*15 minutes*) **Un exemple appliqué**

Dans les établissements universitaires, on rencontre souvent des problèmes lors du calcul de salaires du personnel. Sans penser aux recherches effectuées par certains professeurs, on va essayer de calculer les salaires de ceux qui sont reconnus comme 'Professeur' (ordinaire, titulaire, associé ou assistant) à l'université et ceux qui y donnent des cours à temps partiel (on va les considérer comme 'Collaborateurs' dans cet exercice).

La classe mère dans ce cas est nommée **Enseignant**, qui possède une propriété - le salaire annuel moyen. On voudrait que la méthode qui calcule cette quantité renvoie 60 000 (dollars américains) si l'enseignant a moins de 10 ans d'expérience, et 100 000 sinon. Si l'enseignant travaille à temps partiel, la méthode devrait renvoyer une chaîne qui dit 'Le salaire annuel ne s'applique pas aux collaborateurs'.

Ensuite, on veut calculer la paye mensuelle pour chaque type d'employé. Pour les **Professeurs**, la paye devrait être calculée sur la base de deux sources de revenu : un salaire mensuel et une commission pour chaque comité où ils participent.

D'autre part, pour les **Collaborateurs**, la paye est calculée sur une base horaire i.e *taux horaire* $\times$ *nombres*

d'heures de travail (par mois).
Complétez le code ci-dessous :

```
class Enseignant:
    def __init__(self, name, years_experience, full_time):
        ...
    def salaire_annuel_moyen(self):
        ...

class Professeur(Enseignant):
    def __init__(self, name, years_experience, monthly_salary, commission, num_committees):
        ...
    def paye_mensuelle(self):
        ...

class Collaborateur(Lecturer):
    def __init__(self, name, years_experience, hours_per_month, rate):
        ...
    def paye_mensuelle(self):
        ...

prof1 = Professeur("Alexandra", 8, 3000, 200, 4)
prof2 = Collaborateur("David", 10, 40, 30)
# exemples
print(prof1.salaire_annuel_moyen())
print(prof2.paye_mensuelle())
```

Conseil

Pensez à redéfinir les attributs de la classe mère en utilisant `super().__init__()`.

Solution

```
class Enseignant:
    def __init__(self, name, years_experience, full_time):
        self.name = name
        self.years_experience = years_experience
        self.full_time = full_time

    def salaire_annuel_moyen(self):
        if self.full_time:
            if self.years_experience < 10:
                return 60000

            else:
                return 100000

        else:
            return "Le salaire annuel ne s'applique pas aux collaborateurs"


class Professeur(Enseignant):
    def __init__(self, name, years_experience, monthly_salary, commission, num_committees):
        super().__init__(name, years_experience, True)
        self.monthly_salary = monthly_salary
        self.commission = commission
        self.num_committees = num_committees

    def paye_mensuelle(self):
        return self.monthly_salary + self.commission*self.num_committees

class Collaborateur(Enseignant):
    def __init__(self, name, years_experience, hours_per_month, rate):
        super().__init__(name, years_experience, False)
        self.hours_per_month = hours_per_month
        self.rate = rate

    def paye_mensuelle(self):
        return self.hours_per_month*self.rate

prof1 = Professeur("Alexandra", 8, 3000, 200, 4)
prof2 = Collaborateur("David", 10, 40, 30)

# exemples
print(prof1.salaire_annuel_moyen())
print(prof2.paye_mensuelle())
```