Algorithmes et Pensée Computationnelle

Programmation orientée objet : Héritage et Polymorphisme

Le but de cette séance est d'approfondir les notions de programmation orientée objet vues précédemment. Les exercices sont construits autour des concepts d'héritage, de surcharge d'opérateurs/méthodes et de polymorphisme. Au terme de cette séance, vous devez être en mesure de factoriser votre code afin de le rendre mieux structuré et plus lisible. Cette série d'exercices est divisée en 4 sections dont une section facultative. À chaque exercice, le langage de programmation à utiliser sera spécifié.

Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier Ressources.

1 Rappel: Surcharge des opérateurs - Python

Dans cette section, vous manipulerez des fractions sous forme d'objets. Vous ferez des opérations de base sur ce nouveau type d'objets.

Question 1: (5 minutes) Création de la classe

Dans un projet que vous aurez au préalable préparé, créez un fichier appelé surcharge.py. À l'intérieur de ce fichier, créer une classe Fraction qui aura comme attributs privés un numérateur et un dénominateur.

Question 2: (5 minutes) Constructeur par défaut

Définir un constructeur à votre classe. Assignez des valeurs par défaut à vos attributs.

Si un seul argument est passé à votre constructeur, la fraction devra être égale à l'entier correspondant. Empêchez l'utilisateur d'assigner la valeur zéro au dénominateur.

Conseil

Les valeurs par défaut seront assignées à votre objet au cas où il est instancié sans valeurs. Ainsi en faisant $\mathbf{f} = \mathbf{Fraction}()$, on obtiendra un objet $\mathbf{Fraction}$ ayant pour valeurs un numérateur à 0 et un dénominateur à 1 soit $\frac{0}{1}$.

En Python, vous pouvez donner des valeurs par défaut aux arguments de vos méthodes lors de leur définition. Par exemple, vous pouvez faire : def add(self, x=10, y=5):(...).

Question 3: (5 minutes) **Type casting**

Convertir les attributs en entier.

Conseil

Pensez à utiliser la fonction int.

Question 4: (O *5 minutes*) **Redéfinition de méthodes**

Rédéfinir la méthode __str__() pour produire une représentation textuelle de vos objets Fraction.

Conseil

Une fois la méthode _str()_ redefinie, lorsqu'on fera un print() sur une instance de votre classe Fraction, il affichera le message suivant : *Votre fraction a pour valeur numérateur/dénominateur*. numérateur et dénominateur étant les valeurs que vous passerez à votre objet Fraction.

Question 5: (5 *minutes*) **Accesseurs et mutateurs**

Créer des getters et setters pour chacun des attributs de votre classe Fraction.

Question 6: (**1** *10 minutes*) **Simplification de fractions**

Définir une méthode simplification qui réduit la Fraction. Cette méthode ne renverra rien, elle modifiera simplement l'instance. Pour la suite des exercices, assurez-vous de toujours manipuler des fractions simplifiées.

Pour ce faire, vous pouvez faire appel à votre méthode simplification après chaque opération sur un objet de type Fraction.



© Conseil

Afin de simplifier une fraction, vous devez diviser le numérateur et le dénominateur par leur plus grand diviseur commun. Pensez à utiliser la méthode math.gcd pour trouver le plus grand diviseur commun entre deux nombres.

Question 7: (15 minutes) Redéfinition de méthodes - __eq__

Redéfinissez la méthode d'instance _eq_ qui prend en entrée un objet Fraction que vous nommerez other (en plus de self) et qui renvoie True si self et l'objet passé en argument ont la même valeur.



© Conseil

Pour vérifier l'égalité entre a/b et c/d, tester que a*d est égal à b*c.

Utilisez la méthode isinstance() afin de vérifier que other est bien de type Fraction. Dans le cas contraire, affichez un message d'erreur.

La fonction isinstance prend en entrée une valeur et un type. Elle vérifie que cette valeur est du type défini. Par exemple : isinstance(nombre, int) renverra True si la variable nombre est de type int et False dans le cas contraire.

Question 8: (O 15 minutes) **Arithmétique**

Redéfinissez les méthodes ...add... et ...mul... afin d'effectuer des opérations d'addition et de multiplication sur vos objets de type Fraction. Attention, ces méthodes devront renvoyer des objets de type Fraction. Dans vos méthodes _add_ et _mul_, n'oubliez pas de simplifier ces fractions avant de les retourner. Gérer le cas où l'élément passé en argument n'est ni une Fraction, ni un int.

>_ Solution import math 2 3 # Question 1: Création de la classe Fraction 5 class Fraction: 6 # Question 2: Déclaration du constructeur et initialisation des valeurs 7 def __init__(self, numerateur=0, denominateur=1): 8 # Question 3: type casting 9 self.__num = int(numerateur) 10 self.__den = int(denominateur) self.simplification() 11 12 # Question 4: redéfinition de la méthode __str__() 13 14 def __str__(self): 15 return "Votre fraction a pour valeur {}/{}".format(self._num, self._den) 16 17 # Question 5: Getters et Setters 18 def get_num(self): 19 return self.__num 20 21 def get_den(self): 22 return self._den 23 24 def set_num(self, n): 25 $self._num = n$ 26 27 def set_den(self, d): 28 d = int(d)29 # Si on passe 0 au dénominateur, on lève une exception ce qui arrêtera le programme 30 if d == 0: 31 raise ZeroDivisionError $self._den = d$ 32 33 34 # Question 6 35 def simplification(self): 36 **if self.**__num == 0: self.__den = 1 37 38 if self._den < 0: self.__num = -self.__num 40 self.__den = -self.__den 41 pgcd = math.gcd(self.__num, self.__den) 42 self.__num = int(self.__num / pgcd) 43 self.__den = int(self.__den / pgcd) 44 45 # Question 7 46 def __eq__(self, f): 47 if isinstance(f, Fraction): 48 # vu que les fractions sont toujours en représentation simplifiée, on pourrait se contenter de 49 # self._numerateur == f._numerateur and self._denominateur = f.denominateur 50 return self._num * f._den == f._num * self._den 51 # Au cas où on re coit un seul argument, on créé une fraction ayant pour numérateur l'argument et 1 comme dénominateur 52 elif isinstance(f, int): return self.__eq__(Fraction(f)) 53 55 return False 56 57 # Question 8 58 def add(self, f): 59 $self._num = self._num * f._den + f._num * self._den$ 60 self._den = self._den * f._den $\textcolor{red}{\textbf{self.simplification}}()$ 61 62 def plus(self, f): 63 64 q = Fraction(self._num, self._den) q.add(f)65 66 return q

>_ Solution # Surcharge - Suite (2/2) 2 def __add__(self, other): 3 if isinstance(other, Fraction): 4 return self.plus(other) 5 elif isinstance(other, int): 6 $self.add_{--} = self._add_{--}(Fraction(other))$ 7 return self.add__ 8 else: 9 raise TypeError("Unsupported operand types for +: "" + self._class_.._name__ + "" and "" + other._class_.._name__ + 10 11 def __mul__(self, other): 12 13 if isinstance(other, Fraction): 14 return Fraction(self._num * other._num, self._den * other._den) 15 elif isinstance(other, int): 16 return Fraction(self._num * other, self._den) 17 # On affiche un message d'erreur lorsque other n'est pas une Fraction 18 else: 19 raise TypeError(20 "Unsupported operand types for *: "" + self._class_._name__ + "" and "" + other._class_._name__ + 22 if __name__ == '__main__': 23 f1 = Fraction() 24 print(f1) 25 f1 = Fraction(4)26 print(f1) 27 f1 = Fraction(denominateur=5) 28 print(f1) 29 f1 = Fraction(4, -6)30 print(f1) 31 f2 = Fraction(2, -8)32 print(f2) 33 print(f1+f2) print(f1 == Fraction(22, -24))34 35 print(f1 == Fraction(1, 2))

2 Notions d'héritage - Java

Le but de cette partie est de mettre en pratique les notions liées à l'héritage. Pour cela, nous allons nous inspirer de l'exemple présenté dans le cours.

Nous allons créer une classe Livre() qui représentera notre classe mère. Nous allons également créer deux classes filles, Livre_Audio() et Livre_Illustre(). Les classes filles hériteront des attributs et méthodes de la classe mère.

Question 9: (10 minutes) Création des différentes classes

Créez la classe mère Livre() avec les caractéristiques suivantes :

- un attribut privé String nommé titre,
- un attribut privé String nommé auteur,
- un attribut privé int nommé annee,
- un attribut privé int nommé note (initialisé à -1),
- le constructeur de la classe qui prendra les trois premiers arguments cités ci-dessus,
- une méthode setNote() qui permet de définir l'attribut note,
- une méthode getNote() qui permet de retourner l'attribut note,
- une méthode toString() qui retournera le titre, l'auteur, l'année et la note d'un ouvrage note (réécrire cette méthode permettra d'afficher un objet Livre en utilisant System.out.println()

Attention, si la **note** n'a pas été modifiée et qu'elle vaut toujours **-1**, affichez "Note : pas encore attribuée" au lieu de "Note : **note**" via la méthode **toString**().

Créez les classes filles avec les caractéristiques suivantes :

class Livre_Audio extends Livre

- un attribut privé String nommé narrateur

class Livre_Illustre extends Livre

— un attribut privé String nommé illustrateur

Voici le squelette du code à remplir :

```
public class Livre {

public class Livre_Audio extends Livre {

public class Livre_Illustre extends Livre {

public class L
```

Conseil

En java, lors de la déclaration d'une classe, le mot clé extends permet d'indiquer qu'il s'agit d'une classe fille de la classe indiquée.

Le mot clé super permet à la sous classe d'hériter d'éléments de la classe mère. super peut être utilisé dans le constructeur de la sous-classe selon l'example suivant : super(attribut_mère_1, attribut_mère_3, etc.);. Ainsi, il n'est pas nécessaire de redéfinir tous les attributs d'une classe fille!

L'instruction super doit toujours être la première instruction dans le constructeur d'une sousclasse.

Vous pouvez vous servir de n dans une chaine de caractères pour effectuer un retour à la ligne lors de l'affichage.

```
>_ Solution
     public class Livre {
 2
 3
       private String titre;
 4
       private String auteur;
 5
       private int annee;
 6
       private int note = -1;
 7
 8
       public Livre(String titre, String auteur, int annee){
 9
          System.out.println("Création d'un livre");
10
          this.titre = titre:
11
         this.auteur = auteur;
12
         this.annee = annee;
13
14
15
       public int getNote(){
16
         return this.note;
17
18
       public void setNote(int note) {
19
20
         this.note = note;
21
22
23
       public String toString() {
24
          if (note == -1)
25
            return "A propos du livre \n-
                                                            ---- \nTitre : " +titre+ "\nAuteur : "+auteur+
           "\nAnnée: "+annee+ "\nNote: non attribuée";
26
27
         else{
            return "A propos du livre \n--
                                                         -----\nTitre:"+titre+"\nAuteur:"+auteur+
28
           "\nAnnée: "+annee+ "\nNote: "+note;
29
30
31
     }
32
33
     class Livre_Audio extends Livre {
34
       private String narrateur;
35
36
       public Livre_Audio(String titre, String auteur, int annee, String narrateur){
37
          super(titre, auteur, annee);
38
          System.out.println("Création d'un livre audio");
39
          this.narrateur = narrateur;
40
41
42
     }
43
44
     class Livre_Illustre extends Livre {
45
46
       private String illustrateur;
47
48
       public Livre_Illustre(String titre, String auteur, int annee, String illustrateur) {
49
          super(titre, auteur, annee);
50
          System.out.println("Création d'un livre illustré");
51
         this.illustrateur = illustrateur;
52
53
54
    }
```

Question 10: (5 minutes) Méthode et héritage

Maintenant que vous avez créé la classe et les classes filles correspondantes, vous pouvez créer un objet Livre à l'aide du constructeur de la classe Livre_Audio (et des arguments donnés lors de la création de l'objet). Si vous manquez d'inspiration vous pouvez indiquer les valeurs suivantes : titre : "Hamlet", auteur : "Shakespeare", année : "1609" et le narrateur "William.

Une fois l'objet créé, attribuez lui une note à l'aide de la méthode setNote() définie précédemment.

Finalement, utilisez la méthode System.out.println() pour afficher les informations du livre.

La méthode étant définie dans la classe mère, elle n'a pas connaissance de la variable narrateur définie dans la sous-classe. Redéfinissez la méthode dans la classe fille pour y inclure l'information sur le narrateur.

Faites pareil avec la classe Livre_Illustre et son attribut Illustrateur

Conseil

Attention, on vous demande de créer un objet Livre et non pas Livre_Audio.

Le mot clef super peut être utilisé dans la redéfinition d'une méthode selon l'exemple suivant : super.nom_de_la_methode();. Le mot clé super représente la classe parent, tout comme le mot clé this représentait l'instance avec laquelle la méthode été appelée. Appeler la méthode via le mot clé super effectuera et renverra ce que la méthode de la classe mère effectue et renvoie.

L'instruction super doit toujours être la première instruction dans le redéfinition d'une méthode dans une classe fille.

```
>_ Solution
```

```
class Livre_Audio extends Livre {
 2
       private String narrateur;
 3
 4
       public Livre_Audio(String titre, String auteur, int annee, String narrateur){
 5
          super(titre, auteur, annee);
 6
          System.out.println("Création d'un livre audio");
 7
          this.narrateur = narrateur;
 8
 9
10
       // redéfinition de la fonction toString dans la classe fille Livre_Audio
11
       public String toString() {
          return super.toString() + "\nNarrateur: "+ narrateur+"\n"; //Ajoute narrateur à la chaine de caractère
12
           crée par la classe mère (super)
13
       }
14
     }
15
     class Livre_Illustre extends Livre {
16
17
18
       private String illustrateur;
19
20
       public Livre_Illustre(String titre, String auteur, int annee, String illustrateur) {
21
          super(titre, auteur, annee);
22
          System.out.println("Création d'un livre illustré");
23
          this.illustrateur = illustrateur;
24
25
       public String toString() {
          return super.toString() + "\nIllustrateur: "+ illustrateur + "\n"; //Ajoute illustrateur à la chaine de
26
           caractère crée par la classe mère (super)
27
       }
28
     }
     public class Main {
 2
 3
       public static void main(String[] args) {
 4
          Livre Livre1 = new Livre_Audio("Hamlet", "Shakespeare", 1609,"William");
 5
          Livre1.setNote(5):
 6
          System.out.println(Livre1);
 7
       }
 8
 9
     }
```

Lorsque toutes les étapes auront été effectuées, effectuez ce main :

```
public class Main {

public static void main(String[] args) {
```

```
Livre Livre1 = new Livre_Audio("Hamlet", "Shakespeare", 1609, "William");
5
         Livre1.setNote(5);
6
         System.out.println(Livre1);
 7
         Livre Livre2 = new Livre("Les Misérables", "Hugo", 1862);
8
         System.out.println(Livre2);
9
10
11
12
    }
     Vous devriez obtenir:
     Création d'un livre
     Création d'un livre audio
2
 3
     Création d'un livre
4
    A propos du livre
    Titre: Hamlet
6
7
     Auteur : Shakespeare
     Année : 1609
8
    Note: 5
    Narrateur: William
10
11
12
    A propos du livre
13
14
    Titre: Les Misérables
    Auteur : Hugo
15
16
     Année: 1862
17
    Note: non attribuée
18
    Process finished with exit code 0
```

3 Polymorphisme - Java

Dans cette partie, vous serez amenés à créer 2 nouvelles sous-classes de la classe mère Fighter. La première classe représentera un Soigneur, qui, lorsqu'il attaquera quelqu'un, le soignera au lieu de le blesser. La deuxième classe représentera un combattant spécialisé dans l'attaque Attaquant, qui aura la capacité d'attaquer un certain nombre de fois (ce nombre sera défini au moment où vous l'instancierez). Pensez à télécharger la dernière version de la classe Fighter dans le dossier ressources.

Voici le squelette du code que vous trouverez également dans le dossier ressources du moodle :

```
import java.util.HashMap;
     import java.util.List;
     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> attack_modifier = new HashMap(Map.of("poing", 2, "pied", 2, "tete", 3));
13
       public Fighter(String name, int health, int attack, int defense) {
14
15
          this.name = name:
16
         this.health = health;
17
          this.attack = attack;
18
         this.defense = defense;
19
         instances.add(this);
20
21
22
       public static void addInstances(Fighter other){
23
         instances.add(other);
24
25
       public int getAttack() {
```

```
27
          return attack;
28
        }
29
        public int getHealth() {
30
          return health;
31
32
33
34
        public \ \underline{int} \ getDefense() \ \big\{
35
          return defense;
36
37
38
        public String getName() {
39
          return name:
40
41
        public\ void\ setAttack(int\ attack)\ \big\{
42
43
          this.attack = attack;
44
45
46
        public void setDefense(int defense) {
47
          this.defense = defense;
48
49
50
        public void setHealth(int health) {
51
          this.health = health;
52
53
54
        public void setName(String name) {
55
          this.name = name;
56
57
        public Boolean isAlive() {
58
59
          if (this.health > 0) {
60
            return true;
          } else {
61
62
            return false;
63
64
65
        public\ static\ void\ checkDead()\ \big\{
66
67
          // Initialisation de la liste de Fighters en vie
68
          List<Fighter> temp = new ArrayList<Fighter>();
          //Ici, on parcourt les instances de Fighter
69
70
          for (Fighter f : Fighter.instances) {
71
            // Et on fait appel à la méthode isAlive() pour vérifier que le Fighter est en vie
72
            if (f.isAlive()) {
73
               temp.add(f);
74
             } else {
75
               System.out.println(f.getName() + " est mort");
76
77
78
          Fighter.instances = temp;
79
        }
80
81
        public static void checkHealth() {
82
83
          for (Fighter f : Fighter.instances) {
            System.out.println(f.getName() + "a encore" + f.getHealth() + "points de vie");
84
85
86
          System.out.println("-
87
        }
88
89
90
        public\ void\ attack(String\ type,\ Fighter\ other)\ \{
91
          if (!this.isAlive()) {
92
            System.out.println(this.getName() + " est mort et ne peut plus rien faire");
93
94
          else{
95
            if (!other.isAlive()) {
               System.out.println(other.getName() + " est déjà mort");
96
97
98
             else{
               int damage = (int) Fighter.attack_modifier.get(type) * this.attack - other.getDefense();
99
```

```
100
               other.set Health (other.get Health ()-damage);\\
101
               Fighter.checkDead();
102
               Fighter.checkHealth();
103
104
          }
105
106
        }
     }
107
108
109
      class Soigneur extends Fighter { // a la capacité de soigner et réssuciter quelqu un
110
111
112
        public Soigneur(String name, int health, int attack, int defense, int soin)
113
114
          //TODO
115
116
117
        //TODO
118
119
120
        public void résurrection(Fighter other){
121
122
          //TODO
123
124
125
        public\ void\ attack(Fighter\ other)\ \big\{
126
          //TODO
127
128
129
130
      class Attaquant extends Fighter { // a la capacité d attaquer deux fois
131
        //TODO
132
133
        public Attaquant(String name, int health, int attack, int defense, int multiplicateur){
134
135
          //TODO
136
137
138
        //TODO
139
        public void attack(String type, Fighter other) {
140
141
          //TODO
142
143
```

Question 11: (5 *minutes*) Sous-classe Soigneur

Commencez par déclarer une nouvelle sous-classe Soigneur. Cette sous-classe prendra un nouvel attribut private, int, nommé résurrection, qui vaudra 1 lors de l'instanciation.

Déclarez le constructeur de cette classe ainsi que les getter et setter permettant d'interagir avec ce nouvel attribut (résurrection).



Pensez à utiliser le constructeur de votre classe mère Fighter

>_ Solution class Soigneur extends Fighter { 2 3 4 5 6 private int résurrection; public Soigneur(String name, int health, int attack, int defense, int soin) 7 super(name,health,attack,defense); 8 résurrection = 1; 9 10 public int getRésurrection(){ 11 12 return this.résurrection; 13 14 15 public void setRésurrection(int etat){ 16 this.résurrection = etat; 17

Question 12: (10 minutes) Méthode résurrection(Fighter other) de la sous-classe Soigneur

Commencez par déclarer une nouvelle méthode nommée résurrection(Fighter other).

Cette méthode permettra de faire revenir un Fighter à la vie, mais le Soigneur ne pourra le faire qu'une seule fois.

Commencez par contrôler que l'instance depuis laquelle la méthode est appelée soit toujours en vie. Si ce n'est pas le cas, indiquez : nom_instance est mort et ne peut plus rien faire.

Contrôlez ensuite que l'instance other soit vraiment morte. Si ce n'est pas le cas, indiquez le via : nom_other est toujours en vie.

Pour finir, contrôlez que l'attribut résurrection de l'instance depuis laquelle la méthode est appelée est égale à 1. Si ce n'est pas le cas, indiquez : nom.instance ne peut plus ressusciter personne.

Si tous ces éléments sont réunis, faites revenir le Fighter other à la vie en lui remettant 10 points de vie et en l'ajoutant à la liste instances de la classe Fighter. Pensez également à mettre l'attribut résurrection de l'instance appelée à 0 afin de l'empêcher de réutiliser ce pouvoir, à appeler la méthode checkHealth(), et à indiquer : nom.other est revenu à la vie!

Conseil

Utilisez un branchement conditionnel pour les contrôles.

Une nouvelle méthode nommée addInstances(Fighter other) a été créée dans la classe Fighter. Regardez à quoi elle sert et utilisez la.

Pour les indications en fonction des différentes conditions, imprimmez simplement la phrase en question.

```
>_ Solution
     public void résurrection(Fighter other){
2
          if(!this.isAlive()) {
            System.out.println(this.getName() + " est mort et ne peut plus rien faire");
4
5
6
7
          else{
            if (other.isAlive()) {
              System.out.println(other.getName() + " est toujours en vie !");
8
9
               if (this.getRésurrection() == 0) {
10
                 System.out.println(this.getName() + " ne peut plus ressuciter personne");
11
12
                 other.setHealth(10);
                 Fighter.addInstances(other);
13
14
                 this.setRésurrection(0);
                 System.out.println(other.getName() + " vient de revenir à la vie");
15
16
                 Fighter.checkHealth();
17
18
19
20
       }
```

Question 13: (10 minutes) Méthode attack de la sous-classe Soigneur

Réécrivez la méthode attack de la sous-classe Soigneur afin d'ajouter des points de vie à other au lieu de lui en retirer.

Le seul argument nécessaire pour cette méthode sera le Fighter other.

Commencez par contrôler que le Soigneur depuis lequel la méthode est appelée est encore en vie. Si ce n'est pas le cas, indiquez : nom_instance est mort et ne peut plus rien faire.

Contrôlez ensuite si other est toujours en vie. Si ce n'est pas le cas indiquez : nom_other est déjà mort, ressuscitez le afin de pouvoir le soigner. Contrôlez également qu'il ait moins de 10 points de vie. Si ce n'est pas le cas, indiquez le via : nom_other a déjà le maximum de points de vie.

Si toutes ces conditions sont réunies, ajoutez la valeur de l'attaque de l'instance qui appelle la méthode aux points de vie de other, puis appelez la méthode de classe checkHealth().

Conseil

Pensez à utiliser du branchement conditionnel pour les contrôles.

Le nombre de points de vie à ajouter est simplement égal à l'attaque de l'instance depuis laquelle la méthode est appelée. Ajoutez la valeur de cet attribut attack au Fighter other

```
>_ Solution
     public void attack(Fighter other) {
2
          if(!this.isAlive()) {
            System.out.println(this.getName() + " est mort et ne peut plus rien faire");
4
5
6
          else{
            if (other.getHealth() >= 10) {
7
              System.out.println(other.getName() + " a déjà le maximum de points de vie");
8
9
10
              System.out.println(other.getName() + " est déjà mort, ressucitez le pour pouvoir le soigner");
11
12
              other.setHealth(other.getHealth() + this.getAttack());\\
13
              Fighter.checkHealth();
14
15
         }
       }
16
```

Question 14: (5 minutes) Sous-classe Attaquant

Commencez par déclarer une nouvelle sous-classe Attaquant. Cette sous-classe prendra un nouvel attribut private, int, nommé multiplicateur, qui sera passé en argument du constructeur de la sous-classe.

Déclarez le constructeur de cette classe ainsi que les getter et setter permettant d'interagir avec ce nouvel attribut multiplicateur.

Conseil

Pensez à utiliser le constructeur de votre classe mère Combattant.

>_ Solution class Attaquant extends Fighter{ 2 3 private int multiplicateur; 5 6 7 public Attaquant(String name, int health, int attack, int defense, int multiplicateur){ super(name,health,attack,defense); this.multiplicateur = multiplicateur; 8 9 10 public int getMultiplicateur() { 11 return multiplicateur; 12 13 14 public void setMultiplicateur(int multiplicateur){ 15 this.multiplicateur = multiplicateur; } 16

Question 15: (10 minutes) Méthode attack de la sous-classe Attaquant

Réécrivez la méthode attack de la sous-classe Attaquant afin d'effectuer plusieurs attaques sur other en fonction de l'attribut multiplicateur.

Y'a t-il besoin de contrôler si l'instance depuis laquelle la méthode est appelée est encore en vie?

Indiquez systématiquement le numéro de l'attaque, puis effectuez l'attaque. Répétez le procédé jusqu'à ce que le numéro de l'attaque soit égal à celui de multiplicateur_instance.

Conseil

Aidez vous de la méthode attack de la classe mère Combattant.

Comment peut-on effectuer plusieurs fois une même séquence d'action en programmation?

```
public void attack(String type, Fighter other) {

for (int i = 0; i < this.getMultiplicateur(); i++) {

System.out.println("Attaque n " + (i+1));

super.attack(type, other);

}

}
```

Si tout est correct, en utilisant ce main:

```
public class Main {
       public static void main(String[] args) {
 2
 3
          Fighter P1 = new Fighter("P1", 10, 2, 2);
 4
          Attaquant P2 = new Attaquant("P2", 10, 2, 2,2);
         Soigneur P3 = new Soigneur("P3",10,4,2,4);
 5
 6
         P1.attack("pied",P2);
         P1.attack("poing",P2);
 7
 8
         P1.attack("tete",P2);
 9
         P1.attack("tete",P2);
         P3.résurrection(P2);
10
11
         P1.attack("pied",P2);
         P1.attack("poing",P2);
12
         P1.attack("tete",P2);
13
14
          P3.attack(P2);
         P2.attack("tete",P1);
15
16
17
     }
     Vous devriez obtenir:
     P1 a encore 10 points de vie
 1
 2
     P2 a encore 8 points de vie
 3
     P3 a encore 10 points de vie
 4
 5
     P1 a encore 10 points de vie
 6
     P2 a encore 6 points de vie
 7
     P3 a encore 10 points de vie
 8
 9
     P1 a encore 10 points de vie
10
     P2 a encore 2 points de vie
11
     P3 a encore 10 points de vie
12
13
     P2 est mort
     P1 a encore 10 points de vie
14
15
     P3 a encore 10 points de vie
16
17
     P2 vient de revenir à la vie
     P1 a encore 10 points de vie
18
19
     P3 a encore 10 points de vie
     P2 a encore 10 points de vie
20
21
     P1 a encore 10 points de vie
22
23
     P3 a encore 10 points de vie
24
     P2 a encore 8 points de vie
25
     P1 a encore 10 points de vie
26
27
     P3 a encore 10 points de vie
```

P2 a encore 6 points de vie

```
29
30
     P1 a encore 10 points de vie
31
     P3 a encore 10 points de vie
     P2 a encore 2 points de vie
32
33
34
     P1 a encore 10 points de vie
35
     P3 a encore 10 points de vie
     P2 a encore 6 points de vie
36
37
38
     Attaque n 1
39
     P1 a encore 6 points de vie
40
     P3 a encore 10 points de vie
41
     P2 a encore 6 points de vie
42
43
     Attaque n 2
44
     P1 a encore 2 points de vie
45
     P3 a encore 10 points de vie
46
     P2 a encore 6 points de vie
47
48
     Process finished with exit code 0
```

4 Héritage en Python Optionnel

Question 16: (**1** *10 minutes*) **Classe Point (Suite)**

Dans la série dernière, vous avez rencontré un exemple de classe en Python. Celui-là représente un point de 2 dimensions, x et y, ainsi que des calculs basiques des points 2D. Pour cet exercice, vous allez implémenter une classe des points de 3 dimensions en utilisant de l'héritage sur la classe **Point** qu'on a implémentée! À travers cet exercice, nous voudrions également vous présenter une syntaxe en Python qui concerne l'utilisation des 'décorateurs'.

Avant de commencer, nous voudrions attirer votre attentions sur les points suivants de la classe mère Point :

- Tout d'abord, nous avons récrit la classe de la semaine dernière pour utiliser les 'decorators' en Python. Par exemple, vous trouverez une explication ici pour le décorateur setter. Lisez bien les documentations et le code suivant pour comprendre l'usage de ces décorateurs.
- Nous avons changé le nom de la méthode distance() pour euclidean_distance() pour la distinguer des autres types de distance.
- Traiter la classe Point comme mère et faire l'hériter depuis la classe Point3D n'est pas la meilleure structure d'un programme Python, mais on la garde pour le moment afin de vous montrer comment les méthodes de classe mère peuvent être manipulées dans une classe fille.

Voici la classe Point qui était légèrement modifiée :

```
import math
 2
 3
        class Point:
 4
           def __init__(self, x, y):
 5
              self._x = x
 6
              self._y = y
 7
 8
           @property
 9
           def x(self):
10
              return self._x
11
12
           @property
13
           def v(self):
14
              return self.__y
15
16
           @x.setter
17
           def x(self, x):
18
              self._x = x
19
20
           @y.setter
21
           def v(self, v):
22
              self._y = y
23
24
           def euclidean_distance(self, p2):
25
              \textcolor{return}{\textbf{return}}\ \textbf{math.sqrt}((\textcolor{ret}{\textbf{self.}}\_\textbf{x} - \textbf{p2.get}\_\textbf{x}()) **2 + (\textcolor{ret}{\textbf{self.}}\_\textbf{y} - \textbf{p2.get}\_\textbf{y}()) **2)
```

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

- Rajoutez une méthode qui renvoie une représentation vectorielle du point. Vous pouvez utiliser la liste en Python.
- Recalculez la distance euclidean et le milieu pour le point 3D.
- (Optionnel) Si vous voulez vous familiariser encore plus avec les méthodes de classe en Python, implémentez deux autres calculs de distance: Manhattan et Minkowski (détails: https://www. analyticsvidhya.com/blog/2020/02/4-types-of-distance-metrics-in-machine-learning/-)

```
class Point3D(Point):
       def __init__(self, x, y, z):
 2
 3
          super().__init__(x, y)
 4
          self._z = z
 5
 6
       @property # decorator, nouvelle dimension
 7
       def ...:
 8
 9
10
        @z.setter
11
       def ...:
12
13
14
        @property
15
       def vector_representation(self): # represented in the list format
16
17
18
       def euclidean_distance(self, p2): # i.e norme
19
20
21
       def manhattan_distance(self, p2):
22
23
24
       def minkowski_distance(self, p2, order=3):
25
26
27
       def milieu(self, p2):
```

Conseil

Recherchez sur Internet des documentations sur les décorateurs Python, notamment sur 'property', 'setter' et 'getter'.

Que fait **super.__init__()**?

Vous voudrez aussi vous demander si la représentation vectorielle d'un point pourrait être une propriété au lieu d'une méthode.

```
>_ Solution
     class Point3D(Point):
        def __init__(self, x, y, z):
 2
 3
           super().\_init\_(x, y)
 4
           self._z = z
 5
 6
        @property # decorator
 7
        def z(self):
 8
           return self.__z
 9
10
        @z.setter
11
        def z(self, z):
12
           self._z = z
13
14
15
        def vector_representation(self): # represented in the list format
16
           return [self.x, self.y, self.z]
17
18
        def euclidean_distance(self, p2): # i.e norme
19
           other_x = p2.x
20
           other_y = p2.y
21
           other_z = p2.z
22
           \textcolor{return}{\textbf{return}}\ math.sqrt((\textcolor{ret}{\textbf{self.x}} - \textbf{other}\_\textbf{x})**2 + (\textcolor{ret}{\textbf{self.y}} - \textbf{other}\_\textbf{y})**2 + (\textcolor{ret}{\textbf{self.z}} - \textbf{other}\_\textbf{z})**2)
23
24
        def manhattan_distance(self, p2):
25
           other_x = p2.x
26
           other_y = p2.y
27
           other_z = p2.z
28
           return sum((abs(self.x - other_x), abs()), abs(self.y - other_y), abs(self.z - other_z))
29
30
        def minkowski_distance(self, p2, order=3):
31
           other_x = p2.x
32
           other_y = p2.y
33
           other_z = p2.z
34
           return sum((abs(self.x - other_x)**order), abs(self.y - other_y)**order,\
35
                  abs(self.z - other_z)**order)**(1/order)
36
37
        def milieu(self, p2):
38
           other_x = p2.x
39
           other_y = p2.y
40
           other_z = p2.z
41
42
           x_M = (self.x + other_x)/2
43
           y_M = (self.y + other_y)/2
44
           z_M = (self.z + other_z)/2
           45
46
47
48
     point1 = Point3D(1, 2, 3)
49
     point2 = Point3D(3, 4, 5)
50
51
     # exemple
     point1.vector_representation
```

Question 17: (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 'Professor' à l'université et ceux qui y donnent des cours à temps partiel ('Part-time Lecturer').

La classe mère dans ce cas est nommée **Lecturer**, qui possède une propriété - le salaire annuel moyen. On voudrais que la méthode qui calcule cette quantité renvoie 60 000 (dollars américains) si l'enseignant a moins de 10 ans d'expériences, et 100 000 sinon. Si l'enseignant travaille à temps partiel, la méthode devrait renvoyer une chaîne qui dit 'Salary for part-time lecturers unknown'.

Ensuite, on veut calculer la paye mensuelle pour chaque type d'employé. Pour les 'Professors', 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 'Part-time Lecturers', la paye est calculée sur une base horaire i.e taux horaire \times nombres d'heures de travail (par mois).

```
class Lecturer:
 2
       def __init__(self, name, years_experience, full_time):
 3
 4
 5
       def avg_annual_salary(self):
 6
 7
 8
 9
     class Professor(Lecturer):
10
       def __init__(self, name, years_experience, monthly_salary, commission, num_committees_served):
11
12
13
       def monthly_payroll(self):
14
15
16
     class ParttimeLecturer(Lecturer):
       def __init__(self, name, years_experience, hours_per_month, rate):
17
18
19
20
       def monthly_payroll(self):
21
22
     prof1 = Professor("Alexandra", 8, 3000, 200, 4)
23
24
     prof2 = ParttimeLecturer("David", 10, 40, 30)
25
26
     # exemples
27
     print(prof1.avg_annual_salary)
     print(prof2.monthly_payroll)
```

•

Conseil

Où devrait-on mettre **super.__init__()** dans cet exemple?

De nouveau, réfléchissez bien si quelques méthodes peuvent être traitées comme 'propriétés'.

>_ Solution **class** Lecturer: 2 def __init__(self, name, years_experience, full_time): 3 self.name = name 4 self.years_experience = years_experience 5 self.full_time = full_time 6 7 @property 8 ${\color{red} \textbf{def} \ avg_annual_salary(self):}$ 9 if self.full_time: 10 if self.years_experience < 10: return 60000 11 12 13 else: 14 return 100000 15 16 else: 17 return "Salary for part-time lecturers unknown" 18 19 20 class Professor(Lecturer): 21 ${\color{red} \textbf{def _init__(self, name, years_experience, monthly_salary, commission, num_committees_served):} \\$ 22 super().__init__(name, years_experience, True) 23 self.monthly_salary = monthly_salary 24 **self.**commission = commission 25 self.num_committees_served = num_committees_served 26 27 @property 28 def monthly_payroll(self): ${\bf return\ self.monthly_salary+self.commission*self.num_committees_served}$ 29 30 31 class ParttimeLecturer(Lecturer): $\textcolor{red}{\textbf{def}} \ _\texttt{init}_(\textbf{self}, name, years_experience, hours_per_month, rate):$ 32 33 super().__init__(name, years_experience, False) 34 self.hours_per_month = hours_per_month 35 self.rate = rate 36 37 @property 38 def monthly_payroll(self): 39 return self.hours_per_month*self.rate 40 41 prof1 = Professor("Alexandra", 8, 3000, 200, 4) prof2 = ParttimeLecturer("David", 10, 40, 30) 42 43 44 # exemples 45 print(prof1.avg_annual_salary) 46 print(prof2.monthly_payroll)