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 (avec le label Optionnel). 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 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 **f = Fraction()**, on obtiendra un objet **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: (*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: (*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: (*15 minutes*) **Addition et multiplication**
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**.

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 :

```
1  class Livre {
2
3  }
4
5  class Livre_Audio extends Livre {
6
7  }
8
9  class Livre_Illustre extends Livre {
10
11 }
```

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_2, 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 sous-classe.
Vous pouvez vous servir de **n** dans une chaine de caractères pour effectuer un retour à la ligne lors de l'affichage.

Question 10: (*5 minutes*) **Méthode et héritage**

Maintenant que vous avez créé la classe mère 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).

En instanciant l'objet, vous pourriez utiliser 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-clé **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 était appelée.
L'instruction **super** doit toujours être la première instruction dans le redéfinition d'une méthode dans une classe fille.

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

```
1  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          Livre Livre2 = new Livre("Les Misérables","Hugo",1862);
8          System.out.println(Livre2);
9
10     }
11
12 }
```

Vous devriez obtenir :

```
1  Création d'un livre
2  Création d'un livre audio
3  Création d'un livre
4  A propos du livre
5  ------------------
6  Titre : Hamlet
7  Auteur : Shakespeare
8  Année : 1609
9  Note : 5
10 Narrateur: William
11
12 A propos du livre
13 ------------------
14 Titre : Les Misérables
15 Auteur : Hugo
16 Année : 1862
17 Note : non attribuée
18
19 Process finished with exit code 0
```

3 Polymorphisme - Java

Les exercices de cette section sont une suite des exercices de la section 2 de la semaine passée (TP12). 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" un **Fighter**, 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 :

```
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> attack_modifier = new HashMap(Map.of("poing", 2, "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 static void addInstances(Fighter other){
23         instances.add(other);
24     }
25
26     public int getAttack() {
27         return attack;
28     }
29
30     public int getHealth() {
31         return health;
32     }
33
34     public int getDefense() {
35         return defense;
36     }
37
38     public String getName() {
39         return name;
40     }
41
42     public void setAttack(int attack) {
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
58     public Boolean isAlive() {
59         if (this.health > 0) {
60             return true;
```

```
61          } else {
62              return false;
63          }
64      }
65
66      public static void checkDead() {
67          // Initialisation de la liste de Fighters en vie
68          List<Fighter> temp = new ArrayList<Fighter>();
69          //Ici, on parcourt les instances de Fighter
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
82      public static void checkHealth() {
83          for (Fighter f : Fighter.instances) {
84              System.out.println(f.getName() + " a encore " + f.getHealth() + " points de vie");
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()) {
96                  System.out.println(other.getName() + " est déjà mort");
97              }
98              else{
99                  int damage = (int) Fighter.attack_modifier.get(type) * this.attack - other.getDefense();
100                 other.setHealth(other.getHealth() - 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     //TODO
112
113     public Soigneur(String name, int health, int attack, int defense, int soin)
114     {
115         //TODO
116     }
117
118     //TODO
119
120
121     public void résurrection(Fighter other){
122         //TODO
123     }
124
125     public void attack(Fighter other) {
126         //TODO
127     }
128 }
129
130 class Attaquant extends Fighter{ // a la capacité d attaquer deux fois
131
132     //TODO
133
```

```
134     public Attaquant(String name, int health, int attack, int defense, int multiplicateur){
135         //TODO
136     }
137
138     //TODO
139
140     public void attack(String type, Fighter other) {
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**).

> **Conseil**
>
> Pensez à utiliser le constructeur de votre classe mère **Fighter**

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 aussi à :

- Mettre l'attribut **résurrection** de l'instance appelée à 0 afin de l'empêcher de réutiliser ce pouvoir,
- 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, imprimez simplement la phrase en question.

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**

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**.

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 ?

Si tout est correct, en utilisant ce **main** :

```
1  public class Main {
2      public static void main(String[] args) {
3          Fighter P1 = new Fighter("P1", 10, 2, 2);
4          Attaquant P2 = new Attaquant("P2", 10, 2, 2,2);
5          Soigneur P3 = new Soigneur("P3",10,4,2,4);
6          P1.attack("pied",P2);
7          P1.attack("poing",P2);
```

```
        P1.attack("tete",P2);
        P1.attack("tete",P2);
        P3.résurrection(P2);
        P1.attack("pied",P2);
        P1.attack("poing",P2);
        P1.attack("tete",P2);
        P3.attack(P2);
        P2.attack("tete",P1);
    }
}
```

Vous devriez obtenir :

```
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 6 points de vie
P3 a encore 10 points de vie
--------------------------------
P1 a encore 10 points de vie
P2 a encore 2 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
--------------------------------
P2 vient de revenir à la vie
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 10 points de vie
--------------------------------
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 8 points de vie
--------------------------------
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 2 points de vie
--------------------------------
P1 a encore 10 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------
Attaque n 1
P1 a encore 6 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------
Attaque n 2
P1 a encore 2 points de vie
P3 a encore 10 points de vie
P2 a encore 6 points de vie
--------------------------------

Process finished with exit code 0
```

4 Héritage en Python Optionnel

Question 16: (*15 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 opérations basiques sur 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 !

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

- Nous avons changé le nom de la méthode **distance()** en **distance_euclidean()** pour la distinguer des autres types de distance.
- Traiter la classe **Point** comme une classe mère et la faire hériter de 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 a été légèrement modifiée (Vous trouverez le fichier sur Moodle, dans le dossier **Ressources**) :

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

Ecrivez une classe qui hérite de **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 **list** en Python.
- 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 calculs de distance : Manhattan et Minkowski.

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

    def get_z(self):
        ...

    def set_z(self, z):
        ...

    def vector_representation(self): # représentée sous forme de liste
        ...

```

```
    def distance_euclidean(self, p2): # i.e norme
        ...

    def distance_manhattan(self, p2):
        ...

    def distance_minkowski(self, p2, order=3):
        ...

    def milieu(self, p2):
        ...
```

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 $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (y_1 - y_2)^2}$.

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 '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__()`.