

# 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__()` redéfinie, 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`.

## >\_ Solution

```
1 import math
2
3
4 # 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)
11        self.simplification()
12
13    # Question 4: redéfinition de la méthode __str__()
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
32        self.__den = d
33
34    # Question 6
35    def simplification(self):
36        if self.__num == 0:
37            self.__den = 1
38        if self.__den < 0:
39            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.__numérateur == f.__numérateur 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
52        # dénominateur
53        elif isinstance(f, int):
54            return self.__eq__(Fraction(f))
55        else:
56            return False
57
58    # Question 8
59    def add(self, f):
60        self.__num = self.__num * f.__den + f.__num * self.__den
61        self.__den = self.__den * f.__den
62        self.simplification()
63
64    def plus(self, f):
65        q = Fraction(self.__num, self.__den)
66        q.add(f)
67        return q
```

## >\_ Solution

```
1 # 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(
10             "Unsupported operand types for +: " + self.__class__.__name__ + " and " + other.__class__.__name__ +
11             "")
12
13 def __mul__(self, other):
14     if isinstance(other, Fraction):
15         return Fraction(self.__num * other.__num, self.__den * other.__den)
16     elif isinstance(other, int):
17         return Fraction(self.__num * other, self.__den)
18     # On affiche un message d'erreur lorsque other n'est pas une Fraction
19     else:
20         raise TypeError(
21             "Unsupported operand types for *: " + self.__class__.__name__ + " and " + other.__class__.__name__ +
22             "")
23
24 if __name__ == '__main__':
25     f1 = Fraction()
26     print(f1)
27     f1 = Fraction(4)
28     print(f1)
29     f1 = Fraction(denominateur=5)
30     print(f1)
31     f1 = Fraction(4, -6)
32     print(f1)
33     f2 = Fraction(2, -8)
34     print(f2)
35     print(f1+f2)
36     print(f1 == Fraction(22, -24))
37     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 :

```
1 public class Livre {
2
3 }
4
5 public class Livre.Audio extends Livre {
6
7 }
8
9 public 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'exemple 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 chaîne de caractères pour effectuer un retour à la ligne lors de l'affichage.

## >\_ Solution

```
1 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
19    public void setNote(int note) {
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+
26                "\nAnnée : "+annee+ "\nNote : non attribuée";
27        }
28        else{
29            return "A propos du livre \n----- \nTitre : " +titre+ "\nAuteur : "+auteur+
30                "\nAnnée : "+annee+ "\nNote : "+note;
31        }
32    }
33 }
34
35 class Livre_Audio extends Livre {
36     private String narrateur;
37
38     public Livre_Audio(String titre, String auteur, int annee, String narrateur){
39         super(titre, auteur, annee);
40         System.out.println("Création d'un livre audio");
41         this.narrateur = narrateur;
42     }
43 }
44
45 class Livre_Illustre extends Livre {
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 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.

#### >\_ Solution

```
1  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() {
12         return super.toString() + "\nNarrateur: " + narrateur + "\n"; //Ajoute narrateur à la chaîne de caractère
13         // créée par la classe mère (super)
14     }
15 }
16 class Livre.Illustre extends Livre {
17     private String illustrateur;
18
19     public Livre.Illustre(String titre, String auteur, int annee, String illustrateur) {
20         super(titre, auteur, annee);
21         System.out.println("Création d'un livre illustré");
22         this.illustrateur = illustrateur;
23     }
24     public String toString() {
25         return super.toString() + "\nIllustrateur: " + illustrateur + "\n"; //Ajoute illustrateur à la chaîne de
26         // caractère créée par la classe mère (super)
27     }
28 }
29
30 public class Main {
31     public static void main(String[] args) {
32         Livre Livre1 = new Livre.Audio("Hamlet", "Shakespeare", 1609, "William");
33         Livre1.setNote(5);
34         System.out.println(Livre1);
35     }
36 }
```

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 ressusciter 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 resurrection(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**

#### >\_ Solution

```

1 class Soigneur extends Fighter {
2
3     private int résurrection;
4
5     public Soigneur(String name, int health, int attack, int defense, int soin)
6     {
7         super(name,health,attack,defense);
8         résurrection = 1;
9     }
10
11     public int getRésurrection(){
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 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.

#### >\_ Solution

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

```
1 public void attack(Fighter other) {
2     if(!this.isAlive()) {
3         System.out.println(this.getName() + " est mort et ne peut plus rien faire");
4     }
5     else{
6         if (other.getHealth() >= 10) {
7             System.out.println(other.getName() + " a déjà le maximum de points de vie");
8         }
9         if (!other.isAlive()) {
10            System.out.println(other.getName() + " est déjà mort, ressuscitez le pour pouvoir le soigner");
11        } else {
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

```
1 class Attaquant extends Fighter{
2
3     private int multiplicateur;
4
5     public Attaquant(String name, int health, int attack, int defense, int multiplicateur){
6         super(name,health,attack,defense);
7         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 ?

### >\_ Solution

```
1 public void attack(String type, Fighter other) {
2     for (int i = 0; i < this.getMultiplicateur(); i++) {
3         System.out.println("Attaque n " + (i+1));
4         super.attack(type, other);
5     }
6 }
```

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);
8         P1.attack("tete",P2);
9         P1.attack("tete",P2);
10        P3.résurrection(P2);
11        P1.attack("pied",P2);
12        P1.attack("poing",P2);
13        P1.attack("tete",P2);
14        P3.attack(P2);
15        P2.attack("tete",P1);
16    }
17 }
```

Vous devriez obtenir :

```
1 P1 a encore 10 points de vie
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
14 P1 a encore 10 points de vie
15 P3 a encore 10 points de vie
16 -----
17 P2 vient de revenir à la vie
18 P1 a encore 10 points de vie
19 P3 a encore 10 points de vie
20 P2 a encore 10 points de vie
21 -----
22 P1 a encore 10 points de vie
23 P3 a encore 10 points de vie
24 P2 a encore 8 points de vie
25 -----
26 P1 a encore 10 points de vie
27 P3 a encore 10 points de vie
28 P2 a encore 6 points de vie
```

```
29 -----
30 P1 a encore 10 points de vie
31 P3 a encore 10 points de vie
32 P2 a encore 2 points de vie
33 -----
34 P1 a encore 10 points de vie
35 P3 a encore 10 points de vie
36 P2 a encore 6 points de vie
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
49 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**) :

```
1 import math
2
3 class Point:
4     def __init__(self, x, y):
5         self._x = x
6         self._y = y
7
8     def get_x(self):
9         return self._x
10
11    def get_y(self):
12        return self._y
13
14    def set_x(self, x):
15        self._x = x
16
17    def set_y(self, y):
18        self._y = y
19
20    def distance_euclidean(self, p2):
21        return math.sqrt((self._x - p2.get_x())**2 + (self._y - p2.get_y())**2)
22
23    def milieu(self, p2):
24        x_M = (self._x + p2.get_x()) / 2
25        y_M = (self._y + p2.get_y()) / 2
26        M = Point(x_M, y_M)
27        return M
28
29    def __str__(self):
30        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](#).

```
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  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ .

## >\_ Solution

```
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         return self._z
8
9     def set_z(self, z):
10        self._z = z
11
12    def vector_representation(self): # représentée sous forme de liste
13        return [self._x, self._y, self._z]
14
15    def distance_euclidean(self, p2): # i.e norme
16        other_x = p2.get_x()
17        other_y = p2.get_y()
18        other_z = p2.get_z()
19        return math.sqrt((self._x - other_x)**2 + (self._y - other_y)**2 + (self._z - other_z)**2)
20
21    def distance_manhattan(self, p2):
22        other_x = p2.get_x()
23        other_y = p2.get_y()
24        other_z = p2.get_z()
25        return sum((abs(self._x - other_x), abs(self._y - other_y), abs(self._z - other_z)))
26
27    def distance_minkowski(self, p2, order=3):
28        other_x = p2.get_x()
29        other_y = p2.get_y()
30        other_z = p2.get_z()
31        return sum((abs(self._x - other_x)**order, abs(self._y - other_y)**order, \
32                    abs(self._z - other_z)**order)**(1/order))
33
34    def milieu(self, p2):
35        other_x = p2.get_x()
36        other_y = p2.get_y()
37        other_z = p2.get_z()
38
39        x_M = (self._x + other_x)/2
40        y_M = (self._y + other_y)/2
41        z_M = (self._z + other_z)/2
42        return Point3D(x_M, y_M, z_M) # renvoie un point!
43
44
45    point1 = Point3D(1, 2, 3)
46    point2 = Point3D(3, 4, 5)
47
48    # exemple
49    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 ‘Professeur’ (ordinaire, titulaire, associé ou assitant) 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 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 ‘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  $\text{taux horaire} \times \text{nombre}$

d'heures de travail (par mois).

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



#### Conseil

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

## >\_ Solution

```
1 class Enseignant:
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     def salaire_annuel_moyen(self):
8         if self.full_time:
9             if self.years_experience < 10:
10                return 60000
11
12            else:
13                return 100000
14
15        else:
16            return "Le salaire annuel ne s'applique pas aux collaborateurs"
17
18
19 class Professeur(Enseignant):
20     def __init__(self, name, years_experience, monthly_salary, commission, num_committees):
21         super().__init__(name, years_experience, True)
22         self.monthly_salary = monthly_salary
23         self.commission = commission
24         self.num_committees = num_committees
25
26     def paye_mensuelle(self):
27         return self.monthly_salary + self.commission*self.num_committees
28
29 class Collaborateur(Enseignant):
30     def __init__(self, name, years_experience, hours_per_month, rate):
31         super().__init__(name, years_experience, False)
32         self.hours_per_month = hours_per_month
33         self.rate = rate
34
35     def paye_mensuelle(self):
36         return self.hours_per_month*self.rate
37
38 prof1 = Professeur("Alexandra", 8, 3000, 200, 4)
39 prof2 = Collaborateur("David", 10, 40, 30)
40
41 # exemples
42 print(prof1.salaire_annuel_moyen())
43 print(prof2.paye_mensuelle())
```