# 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*) 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 un numérateur et un dénominateur.

**Question 2:** (🕓 *5 minutes*) Définir un constructeur à votre classe. Assignez des valeurs par défaut à vos attributs.

> **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 et un dénominateur à 1 soit $\frac{1}{1}$.

**Question 3:** (🕓 *5 minutes*)

## 2 Notions d'héritage - Java

Le but de cette partie est de pratiquer et d'assimiler 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 4:** (🕓 *10 minutes*) Création des différentes classes

Créez la classe mère **Livre()** avec les caractéristiques suivantes :
- un attribut **privé String** nomée **titre**
- un attribut **privé String** nommé**auteur**
- un attribut **privé** int nommé **annee**
- un attribut **privé** int nommé**note** (initialisée 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** avec **System.out.println()**

Attention, si la **note** n'a pas été modifiée et qu'elle vaut toujours -1, utilisez "Note : pas encore attribuée" au lieu de "Note : **note**" dans la chaine de caractères que vous allez retourner avec **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'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.

**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+
          "\nAnnée : "+annee+ "\nNote : non attribuée";
26       }
27       else{
28          return "A propos du livre \n---------- \nTitre : " +titre+ "\nAuteur : "+auteur+
          "\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 5:** (*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

```
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 chaine de caractère
          crée par la classe mère (super)
13    }
14 }
15
16 class Livre_Illustre extends Livre {
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() {
26       return super.toString() + "\nIllustrateur: "+ illustrateur + "\n"; //Ajoute illustrateur à la chaine de
          caractère crée par la classe mère (super)
27    }
28 }
```

```
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     }
8
9  }
```

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

```
1  public class Main {
2
3     public static void main(String[] args) {
```

```
        Livre Livre1 = new Livre_Audio("Hamlet", "Shakespeare", 1609,"William");
        Livre1.setNote(5);
        System.out.println(Livre1);
        Livre Livre2 = new Livre("Les Misérables","Hugo",1862);
        System.out.println(Livre2);

    }

}
```

Vous devriez obtenir :

```
Création d'un livre
Création d'un livre audio
Création d'un livre
A propos du livre
--------------------
Titre : Hamlet
Auteur : Shakespeare
Année : 1609
Note : 5
Narrateur: William

A propos du livre
--------------------
Titre : Les Misérables
Auteur : Hugo
Année : 1862
Note : non attribuée

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;
import java.util.Map;

public class Fighter {
    private String name;
    private int health;
    private int attack;
    private int defense;
    private static List<Fighter> instances = new ArrayList<Fighter>();
    private static HashMap<String, Integer> attack_modifier = new HashMap(Map.of("poing", 2, "pied", 2, "tete", 3));

    public Fighter(String name, int health, int attack, int defense) {
        this.name = name;
        this.health = health;
        this.attack = attack;
        this.defense = defense;
        instances.add(this);
    }

    public static void addInstances(Fighter other){
        instances.add(other);
    }

    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 6:** (*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 7:** (*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**

```
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 ressuciter 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 8:** (*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, ressucitez le pour pouvoir le soigner");
11         } else {
12             other.setHealth(other.getHealth() + this.getAttack());
13             Fighter.checkHealth();
14         }
15     }
16 }
```

**Question 9:** (*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 10:** (*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
```

```
--------------------------------
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 11:** (*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 :

```python
import math

class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    @x.setter
    def x(self, x):
        self.__x = x

    @y.setter
    def y(self, y):
        self.__y = y

    def euclidean_distance(self, p2):
        return math.sqrt((self.__x - p2.get_x())**2 + (self.__y - p2.get_y())**2)
```

```
26
27      def milieu(self, p2):
28          x_M = (self.__x + p2.get_x()) /2
29          y_M = (self.__y + p2.get_y()) / 2
30          M = Point(x_M, y_M)
31          return M
32
33      def __str__(self):
34          return "Les coordonnées du point sont: x="+str(self.get_x())+", y="+str(self.get_y())
```

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/` - )

```
1   class Point3D(Point):
2       def __init__(self, x, y, z):
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):
28          ...
```

**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):
        super().__init__(x, y)
        self.__z = z

    @property # decorator
    def z(self):
        return self.__z

    @z.setter
    def z(self, z):
        self.__z = z

    @property
    def vector_representation(self): # represented in the list format
        return [self.x, self.y, self.z]

    def euclidean_distance(self, p2): # i.e norme
        other_x = p2.x
        other_y = p2.y
        other_z = p2.z
        return math.sqrt((self.x - other_x)**2 + (self.y - other_y)**2 + (self.z - other_z)**2)

    def manhattan_distance(self, p2):
        other_x = p2.x
        other_y = p2.y
        other_z = p2.z
        return sum((abs(self.x - other_x), abs()), abs(self.y - other_y), abs(self.z - other_z))

    def minkowski_distance(self, p2, order=3):
        other_x = p2.x
        other_y = p2.y
        other_z = p2.z
        return sum((abs(self.x - other_x)**order), abs(self.y - other_y)**order,\
                abs(self.z - other_z)**order)**(1/order)

    def milieu(self, p2):
        other_x = p2.x
        other_y = p2.y
        other_z = p2.z

        x_M = (self.x + other_x)/2
        y_M = (self.y + other_y)/2
        z_M = (self.z + other_z)/2
        return Point3D(x_M, y_M, z_M) # renvoie un point!


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

# exemple
point1.vector_representation
```

**Question 12:** (*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:
    def __init__(self, name, years_experience, full_time):
        ...

    def avg_annual_salary(self):
        ...


class Professor(Lecturer):
    def __init__(self, name, years_experience, monthly_salary, commission, num_committees_served):
        ...

    def monthly_payroll(self):
        ...

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

    def monthly_payroll(self):
        ...

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

# exemples
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:
    def __init__(self, name, years_experience, full_time):
        self.name = name
        self.years_experience = years_experience
        self.full_time = full_time

    @property
    def avg_annual_salary(self):
        if self.full_time:
            if self.years_experience < 10:
                return 60000

            else:
                return 100000

        else:
            return "Salary for part-time lecturers unknown"


class Professor(Lecturer):
    def __init__(self, name, years_experience, monthly_salary, commission, num_committees_served):
        super().__init__(name, years_experience, True)
        self.monthly_salary = monthly_salary
        self.commission = commission
        self.num_committees_served = num_committees_served

    @property
    def monthly_payroll(self):
        return self.monthly_salary + self.commission*self.num_committees_served

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

    @property
    def monthly_payroll(self):
        return self.hours_per_month*self.rate

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

# exemples
print(prof1.avg_annual_salary)
print(prof2.monthly_payroll)
```