Algorithmes et Pensée Computationnelle

Programmation orientée objet

Le but de cette séance est de se familiariser avec un paradigme de programmation couramment utilisé : la Programmation Orientée Objet (POO). Ce paradigme consiste en la définition et en l'intéraction avec des briques logicielles appelées **Objets**. Dans les exercices suivants, nous manipulerons des objets, aborderons les notions de classe, méthodes, attributs et encapsulation. Au terme de cette séance, vous serez en mesure d'écrire des programmes mieux structurés. Afin d'atteindre ces objectifs, nous utiliserons principalement le langage **Java** qui offre une panoplie d'outils pour mieux comprendre ce paradigme de programmation.

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

1 Création de votre première classe en Java

Le but de cette première partie est de créer votre propre classe en Java. Cette classe sera une classe nommée **Dog()** représentant un chien. Elle aura plusieurs attributs et méthodes que vous implémenterez au fur et à mesure.

Question 1: (*10 minutes*) Création de classe et encapsulation

Commencez par créer une nouvelle classe **Java** dans votre projet, et initialisez les attributs suivants :

1. Un attribut **public** String nommé **name**
2. Un attribut **private** List nommé **tricks**
3. Un attribut **private** String nommé **race**
4. Un attribut **private** int nommé **age**
5. Un attribut **private** int nommé **mood** initialisé à 5 (correspondant à l'humeur du chien)
6. Un attribut de classe (**static**) **private** int nommé **nb_chiens**

Ensuite, créez une méthode publique du même nom que la classe (**Dog**). Cette méthode est appelée le **constructeur**, elle va servir à initialiser les différentes instances de notre classe. Un **constructeur** en **Java** aura le même nom que la classe, et un **constructeur** en **python** sera la méthode **__init__**. Cette méthode prendra en argument les éléments suivants et initialisera les attributs de notre instance :

1. Une chaîne de caractère **name**
2. Une liste **tricks**
3. Une chaîne de caractère **race**
4. Un entier **age**

Pour finir, cette méthode doit incrémenter l'attribut de classe **nb_chiens** qui va garder en mémoire le nombre d'instances crées.

> **Conseil**
>
> N'oubliez pas de préciser si vos attributs sont **public** ou **private**.
> Le mot **static** correspond à un élément de classe (attribut ou méthode), cet élément pourra ensuite être appelé via la classe directement.
> Pour attribuer des valeurs à vos attributs d'instance, utilisez le mot-clé **this.attribut**.
> Pour accéder aux attributs de classe, utilisez **nom_classe.nom_attribut**

Question 2: (*10 minutes*) Getters et setters

Il faut maintenant créer des méthodes nommées **setter** et **getter** pour les attributs privés. Ce sont ces méthodes qui vous permettront d'interagir avec les attributs **private** des instances de la classe. Pour les attributs publics, il vous suffit d'utiliser **nom_instance.attribut** pour l'obtenir.

Les méthodes **getter** d'accéder à la valeur d'un attribut. Les **setter** permettent de modifier la valeur de l'attribut. Les **setter** sont souvent utilisés pour modifier la valeur d'attributs privés.

Créez les méthodes suivantes :

- **getTricks()**
- **getRace()**
- **getAge()**
- **getMood()**
- **setTricks()**
- **setRace()**
- **setAge()**
- **setMood()**

Créez également une méthode de classe permettant de retourner le nombre de **Dog** instanciés (une méthode **getter**).

> **Conseil**
>
> Intellij vous permet de générer automatiquement certaines méthodes telles que les getters et setters. Vous pouvez consulter le lien suivant pour plus d'informations : `https://www.jetbrains.com/help/idea/generating-code.html#generate-delegation-methods`.
> Toutefois, pour cet exercice, nous vous encourageons à le faire manuellement.

Question 3: (*5 minutes*) Manipulation d'attributs - Listes
Créez une méthode **public** nommée **add_trick(String trick)** qui prend en entrée une chaîne de caractère et l'ajoute à la liste **tricks**.

> **Conseil**
>
> La liste **tricks** est une liste comme les autres. Si vous voulez la modifier, vous aurez besoin de passer par une **LinkedList** temporaire.

Question 4: (*5 minutes*) Manipulation d'attributs - setter
Créez deux méthodes qui vont avoir un impact sur l'attribut **mood** du **Dog**. La méthode **leash()** décrémentera mood de 1 et **eat()** l'incrémentera de 3.

Question 5: (*5 minutes*) Manipulation d'attributs d'une autre instance
Créez une méthode nommée **get_oldest(Dog other)** qui prend comme argument un élément de type **Dog**, puis retourne le nom et l'age du **Dog** le plus agé sous le format suivant : "**nom_chien** est le chien le plus agé avec **age_chien** ans".

> **Conseil**
>
> L'élément **Dog** que vous passez en argument est un objet de type **Dog**, vous pouvez donc lui appliquer les méthodes que vous avez créé tout à l'heure. Faites attention à la façon d'accéder aux différents attributs de votre deuxième chien (**public** vs **private**).

Question 6: (*5 minutes*) Redéfinition de méthodes
Créez une méthode **toString()** qui retourne une chaîne de caractères contenant toutes les informations d'une instance de **Dog**. Ainsi, dans votre **main**, en faisant **System.out.println(...)** sur une instance de **Dog**, vous obtiendrez une texte sous le format suivant : **nom_chien** a **age_chien** ans, est un **race_chien** et a une humeur de **mood_chien**. Il sait faire les tours suivants : **tricks_chien**.

Informations utiles

La méthode **toString()** hérite de la super classe Object. La notion d'héritage sera présentée la semaine prochaine. Retenez juste qu'il est possible de choisir ce que vaudra le texte descriptif de nos objets de type **Dog**. Il est également possible de redéfinir d'autres méthodes comme par exemple l'addition ou la soustraction, ce qui permettrait de choisir comment 2 objets de type **Dog** seraient additionnés ou soustraits.

Pour contrôler que vos méthodes et attributs ont été implémentés correctement, vous pouvez essayer le code suivant dans votre classe Main :

```
public class Main {
    public static void main(String[] args) {
        Dog Lola = new Dog("Loola",List.of("rollover"),"Bouviier",10);
        Dog Tobi = new Dog("Tobi",List.of("rollover","do a barrel"),"Doggo",17);
        System.out.println(Lola.getAge());
        System.out.println(Lola.getMood());
        System.out.println(Lola.getRace());
        System.out.println(Lola.name);
        System.out.println(Lola.getTricks());
        Lola.setAge(13);
        Lola.setMood(8);
        Lola.setRace("Bouvier");
        Lola.name = "Lola";
        Lola.setTricks(List.of("rollover","do a barrel"));
        Lola.eat();
        Lola.leash();
        Lola.add_trick("sit");
        System.out.println(Dog.getNb_chiens());
        System.out.println(Lola.get_oldest(Tobi));
        System.out.println(Lola);
    }
}
```

Vous devriez obtenir ce résultat :

```
10
5
Bouviier
Loola
[rollover]
2
Tobi est le chien le plus agé avec 17 ans
Lola a 13 ans, est un Bouvier et a une humeur de 10. Il sait faire les tours suivants : [rollover, do a barrel, sit]
```

2 Interaction entre plusieurs instances d'une même classe

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

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

2 attributs de classe seront également utilisés :

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

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

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

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

```
1  import java.util.HashMap;
2  import java.util.List;
3  import java.util.ArrayList;
4  import java.util.Map;
5
6  public class Fighter {
7      private String name;
8      private int health;
9      private int attack;
10     private int defense;
11     private static List<Fighter> instances = new ArrayList<Fighter>();
12     private static HashMap<String, Integer> attack_modifier = new HashMap(Map.of("poing",1,"pied",2,"tete",3));
13
14     public Fighter(String name, int health, int attack, int defense) {
15         this.name = name;
16         this.health = health;
17         this.attack = attack;
18         this.defense = defense;
19         instances.add(this);
20     }
21
22     public int getAttack() {
23         return attack;
24     }
25
26     public int getHealth() {
27         return health;
28     }
29
30     public int getDefense() {
31         return defense;
32     }
33
34     public String getName() {
35         return name;
36     }
37
38     public void setAttack(int attack) {
39         this.attack = attack;
```

```
40      }
41
42      public void setDefense(int defense) {
43          this.defense = defense;
44      }
45
46      public void setHealth(int health) {
47          this.health = health;
48      }
49
50      public void setName(String name) {
51          this.name = name;
52      }
53
54      public Boolean isAlive() {
55          // à compléter
56      }
57
58      public static void checkDead() {
59          // à compléter
60      }
61
62      public static void checkHealth() {
63          // à compléter
64      }
65
66      public void attack (String type, Fighter other){
67          // à compléter
68      }
69  }
```

Question 7: (*5 minutes*) isAlive()
Définir une méthode **isAlive()** qui retournera **true** si l'instance a plus que 0 points de vie et **false** si l'instance en a moins.

Question 8: (*10 minutes*) checkDead()
Définir une méthode **checkDead()** qui consiste à parcourir la liste des instances, et à contrôler que chacune d'elle est encore en vie. Si ce n'est pas le cas, l'instance en question est supprimée de la liste des instances et le message "**nom_instance** est mort" sera affiché.

> **Conseil**
>
> Prenez le problème dans l'autre sens, créez une liste temporaire, si l'instance est vivante ajoutez la à cette nouvelle liste, et pour finir, mettez à jour votre liste d'instances à l'aide de votre liste temporaire.

Question 9: (*5 minutes*) checkHealth()
Définir une méthode **checkHealth()** qui parcourt la liste des instances et imprime le nombre de points de vie qui reste au combattant sous le format "**nom_instance** a encore **health_instance** points de vie".

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

Commencez par contrôler si **other** est encore en vie, si ce n'est pas le cas, indiquez qu'il est déjà mort : "**other_name** est déjà mort".

Si **other** est encore en vie, retirez des points de vie à **other**. Le nombre de points de vie devant être retiré se calcule en utilisant la formule suivante : **attack_modifier(type)** * **attack_instance** - **defense_other**. Appelez ensuite les fonctions **checkDead()** et **checkHealth()** afin d'avoir un aperçu des combattants restants et de leur santé.

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

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

Vous devriez obtenir ce résultat :

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

Process finished with exit code 0
```

3 Manipulation de graphes en POO (Java)

Cette partie constitue un rappel sur la notion de graphe orienté. Avant de traiter les exercices de la partie 2, veuillez répondre aux questions suivantes :

1. Quels sont les composantes d'un **graphe orienté** ?
2. Combien de classes sont nécessaires pour représenter un graphe orienté ainsi que toutes ses composantes ?
3. Sachant que vous avez une classe qui représente les **arêtes** et que les **sommets** sont représentés par des chaînes de caractères. Quels pourraient être les attributs de la classe **graph**.

Maintenant que vous avez une idée des attributs de la classe **graph** à définir, déterminer quelles méthodes sont nécessaires au fonctionnement de cet objet. Pour ce faire, vous pouvez répondre aux questions suivantes :

1. Est-ce que l'utilisateur aura besoin de modifier l'objet une fois celui-ci initialisé ?
2. Est-ce que l'utilisateur aura besoin de vérifier l'état de l'objet (existence d'attributs, vérification de valeurs...) ou de certaines parties de l'objet ?

Conseil

1. Revenir sur le cours de la semaine 7 sur Moodle.

Voici les réponses aux questions posées ci-dessus.
Certaines questions peuvent avoir plus d'une réponse étant donné qu'il existe plusieurs manières de représenter un graphe en programmation.

1. (a) Les **sommets**.
   (b) Les **arêtes**.
   (c) Le **poids** de chaque arête.
2. Il faut 2 classes (ou 3 si on cherche à avoir plus d'informations dans les sommets) :
   (a) Une classe **Edge** qui va représenter les arêtes.
   (b) Une classe **Graph**.
3. La classe **Graph** va avoir 2 attributs : un ensemble contenant les **sommets** et un autre ensemble contenant les **arêtes** de celui-ci.

Le code de la classe **Edge** est fourni dans le dossier ressources sur Moodle. Utiliser le fichier **Main.java** dans le dossier **Ressources** sur Moodle pour effectuer des tests. Une fois exécuté, il devrait afficher :

```
The vertex number 1 has a value of: Lausanne
The vertex number 2 has a value of: Geneve
The vertex number 3 has a value of: Berne
{from_vertex=Geneve, weight=35.0, to_vertex=Lausanne}
{from_vertex=Lausanne, weight=100.0, to_vertex=Berne}
{from_vertex=Geneve, weight=120.0, to_vertex=Berne}
Edge between Geneve and Berne has been deleted.
The vertex number 1 has a value of: Lausanne
The vertex number 2 has a value of: Geneve
The vertex number 3 has a value of: Berne
{from_vertex=Geneve, weight=35.0, to_vertex=Lausanne}
{from_vertex=Lausanne, weight=100.0, to_vertex=Berne}

Process finished with exit code 0
```

Question 11: (*20 minutes*)

Voici une partie de la classe **graph** codée. Implémentez les méthodes **update_weight()**, **new_edge** et **edge_exist**.

```
1  import java.util.List;
2  import java.util.HashMap;
3  import java.util.Vector;
4
5  public class graph_empty {
6
7     // Les attributs de la classe graphe
8     public List<Edges> edges = new Vector();   // Utilisation de vector car il faut que l on puisse rajouter ou supprimer des é
          léments de la liste
9     public List<String> vertices = new Vector();
10
11    // Méthode qui permet l ajout d un sommet au graphe.
12    public void add_vertex(String name){
13       this.vertices.add(name); // Méthode qui permet d ajouter un sommet au graphe
14    }
15
16    // Cette méthode va tester si le sommet demandé existe dans le graphe. Si oui retourne le poids, sinon retourne 0.
17
18    public double edge_exist(String from_vertex, String to_vertex){
19
20       // Ecrire votre code ici
21
22    }
23
24    // La méthode ci-dessous vous permet de générer une arête lorsque vous cherchez à en ajouter une à votre graphe. Elle
          fait aussi le test si jamais les sommets utilisés font partis du graphe ou non. Si non, elle va les ajouter au graphe. Cette
          méthode peut être utile dans la méthode new_edge
25
26    private void generate_edge(String from_vertex, String to_vertex, double weight){
27       if (this.vertices.contains(from_vertex) & this.vertices.contains(to_vertex)){
28          Edges new_edge = new Edges(from_vertex,to_vertex,weight);
29          this.edges.add(new_edge);
30       }
31       else {
32          if (!this.vertices.contains(from_vertex)){
33             this.vertices.add(from_vertex);
34          }
35          if (!this.vertices.contains(to_vertex)){
36             this.vertices.add(to_vertex);
37          }
38          Edges new_edge = new Edges(from_vertex,to_vertex,weight);
39          this.edges.add(new_edge);
40       }
41
42    }
43
44    public void update_weight(String from_vertex, String to_vertex, double weight){
45       // Ecrire votre code ici
46    }
47    // Méthode qui va ajouter l arête dans le graphe.
48    public void new_edge(String from_vertex, String to_vertex, double weight){
49       // Ecrire votre code ici
50    }
51
52    // Méthode nous permettant de supprimer une arête du graphe.
53    public void del_edge(String from_vertex, String to_vertex){
54       for(Edges edge : this.edges ){
55          if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
56             this.edges.remove(edge);
57             System.out.println("Edge between " + from_vertex + " and " + to_vertex + " has been deleted.");
58             break;
59          }
60       }
61    }
62
63    // Fonction qui permet d imprimer les composants d un graphe
64    public void print(){
65       for(int i=0; i< this.vertices.size(); ++i){
66          System.out.println("The vertex number " + (i+1) + " has a value of: " + this.vertices.get(i));
```

```
67          }
68          for (Edges edge : this.edges){
69              edge.print();
70          }
71      }
72  }
```

1. La méthode **update_weight()** doit prendre en paramètres : le **sommet** d'origine, le **sommet** d'arrivée ainsi que le poids d'une **arête**. Si cette **arête** existe alors elle change son poids. Sinon, elle imprimera une phrase indiquant que l'**arête** n'existe pas.
2. La méthode **edge_exist()** prendra en paramètre le **sommet** d'origine et le **sommet** d'arrivée. Si cette **arête** est dans le **graph** alors la méthode renvoie son poids, sinon, elle renvoie 0.
3. La méthode **new_edge** doit créer une instance de **Edge** et l'ajouter à l'ensemble edges si la connexion n'existe pas déjà. Si elle existe avec un autre poids mettre à jour le poids. Si elle existe de façon identique alors retournez la dans la console avec print. Enfin, si on est dans aucun des deux cas précédents utiliser la méthode **generate_edge** qui vous est donnée pour créer et ajouter cette arête au **graph**. La méthode **new_edge** aura pour paramètres : le **sommet** d'origine, le **sommet** d'arrivée, le poids.

Conseil

1. Utiliser une boucle **for** pour parcourir toutes les **arêtes** dans le **graph**. Faire un test sur les attributs de **Edge** pour changer le poids.
2. Il faut tester pour chaque **arête** (itération) si elle est égale à celle donnée en paramètres.
3. Il faut utiliser les méthodes **edge_exist()**, **update_edge()** et **generate_edge()** pour écrire cette méthode. Il y a 4 tests à effectuer :
   (a) Si **l'arête** existe.
   (b) Si **l'arête** existante a le même poids que celle indiquée en paramètre de la méthode.
   (c) Si **l'arête** existante n'a pas le même poids que celle indiquée en paramètre de la méthode.
   (d) Utilisez le résultat de **edge_exist** pour simplifier ces tests.

4 Notions de POO en Python

Dans cette section, nous créerons pas-à-pas une classe **Point** contenant des attributs et des méthodes utiles. Dans votre IDE, créez un nouveau projet Python (Fichier Nouveau Projet). Dans un dossier de votre choix, créez un fichier **question12.py**.

Question 12: (🕒 *15 minutes*) Classe Point

— Créez une classe **Point** et un constructeur par défaut contenant deux paramètres (x et y).

> **Conseil**
>
> Pour rappel, un constructeur est une fonction **__init()__** que vous redéfinirez dans votre classe.

— Définissez deux attributs privés pour votre classe **Point**. Ces attributs seront les coordonnées x et y de vos points. Par défaut, assignez leur les valeurs données dans le constructeur.

> **Conseil**
>
> À l'intérieur d'une classe, utilisez le mot-clé **self** pour accéder aux méthodes et attributs de l'instance que vous manipulez.
> En Python, pour spécifier qu'un attribut est privé, rajouter un double underscore au nom de l'attribut (Exemple : **__score=0**)

— Définir des getters et setters.

> **Conseil**
>
> En Python, le mot-clé **self** est l'équivalent de **this** utilisé en Java.

— Définissez une méthode **distance** qui prend en entrée l'instance du Point (**self**) et une autre point **p2**. Cette méthode **distance** retournera la distance euclidienne entre le point **self** et **p2**.

> **Conseil**
>
> Pour rappel, la distance euclidienne entre deux points est définie par la formule $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.
> Utilisez la fonction **sqrt()** de la librairie **math** pour calculer la racine carrée. Pensez à importer la libraire **math**

— Définissez une méthode **milieu** qui prendra en entrée **self** et **p2** et qui retournera un objet **Point** situé entre **self** et **p2**.

> **Conseil**
>
> Pour trouver les coordonnées d'un point M(x_M, y_M) situé au milieu du segment défini par des points A(x_A, y_A) et B(x_B, y_B), utilisez les formules suivantes : $x_M = \frac{x_1+x_2}{2}$ et $y_M = \frac{y_1+y_2}{2}$

— Redéfinissez une méthode **__str__()** dans la classe **Point** qui retournera une chaîne de caractères contenant les coordonnées (x, y) d'un point. Ainsi, lorsqu'on fera un **print** d'une instance de la classe **Point**, le message qui s'affichera sera le suivant : *Les coordonnées du Point sont : x = "remplacez par la valeur de x" et y = "remplacez par la valeur de y"*