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**

Solution

```
1  public class Dog {
2
3      public String name;
4      private List tricks;
5      private String race;
6      private int age;
7      private int mood = 5;
8      private static int nb_chiens = 0;
9
10     public Dog(String name, List tricks, String race, int age) {
11         this.name = name;
12         this.race = race;
13         this.tricks = tricks;
14         this.age = age;
15         nb_chiens++;
16     }
17
18 }
```

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.

Solution

```
1   public List getTricks() {
2       return tricks;
3   }
4
5   public int getAge() {
6       return age;
7   }
8
9   public int getMood() {
10      return mood;
11  }
12
13  public String getRace() {
14      return race;
15  }
16
17  public static int getNb_chiens() {
18      return nb_chiens;
19  }
20
21  public void setTricks(){
22      this.tricks=tricks;
23  }
24
25  public void setAge(int age) {
26      this.age = age;
27  }
28
29  public void setMood(int mood) {
30      this.mood = mood;
31  }
32
33  public void setRace(String race) {
34      this.race = race;
35  }
```

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.

Solution

```
1   public void add_trick(String trick) {
2       LinkedList temp = new LinkedList(this.tricks);
3       temp.add(trick);
4       this.tricks = temp;
5   }
```

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 3 et **eat()** l'incrémentera de 2.

Solution

```
1    public void eat() {
2        this.mood = mood + 3;
3    }
4
5    public void leash() {
6        this.mood --;
7    }
```

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**).

Solution

```
1    public String get_oldest(Dog other) {
2        if(other.getAge() < this.getAge()){
3            return this.name + " est le chien le plus agé avec " + this.age + " ans";
4        }
5        else{
6            return other.name + " est le chien le plus agé avec " + other.getAge() + " ans";
7        }
8    }
```

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.

Solution

```
1    public String toString(){return this.name + " a " + this.age + " ans, est un " + this.race +
2            " et a une humeur de " + this.mood + ". Il sait faire les tours suivants : " + this.tricks;}
```

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 **Combattant** que nous allons créer. Chaque **Combattant** 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 **combattant**. 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, Combattant other)**

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

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

```
39
40      public void setDefense(int defense) {
41          this.defense = defense;
42      }
43
44      public void setHealth(int health) {
45          this.health = health;
46      }
47
48      public void setName(String name) {
49          this.name = name;
50      }
51
52      public Boolean isAlive() {
53          // à compléter
54      }
55
56      public static void checkDead() {
57          // à compléter
58      }
59
60      public static void checkHealth() {
61          // à compléter
62      }
63
64          public void attack (String type, Combattant other){
65              // à compléter
66          }
67      }
```

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.

Solution

```
1       public Boolean isAlive() {
2           if (this.health > 0) {
3               return true;
4           } else {
5               return false;
6           }
7       }
```

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.

Solution

```
public static void checkDead() {
   // Initialisation de la liste de Combattants en vie
      List<Combattant> temp = new ArrayList<Combattant>();
      //Ici, on parcourt les instances de Combattant
      for (Combattant f : Combattant.instances) {
         // Et on fait appel à la méthode isAlive() pour vérifier que le Combattant est en vie
         if (f.isAlive()) {
            temp.add(f);
         } else {
            System.out.println(f.getName() + " est mort");
         }
      }
      Combattant.instances = temp;
   }
```

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".

Solution

```
public static void checkHealth() {
      for (Combattant f : Combattant.instances) {
         System.out.println(f.getName() + " a encore " + f.getHealth() + " points de vie");
      }
   }
```

Question 10: (*10 minutes*) attack(String type Combattant other)

Définir une méthode **attack(String type, Combattant 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.

Solution

```
public void attack (String type,Combattant other){
      if(other.isAlive()) {
         int damage = (Integer)Combattant.attack_modifier.get(type) * this.attack - other.getDefense();
         other.setHealth(other.getHealth() - damage);
         Combattant.checkDead();
         Combattant.checkHealth();
      }
      else{
         System.out.println(other.getName() + " est déjà mort");
      }
```

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) {
```

```
3         Combattant P1 = new Combattant("P1", 10, 2, 2);
4         Combattant P2 = new Combattant("P2", 10, 2, 2);
5         Combattant P3 =  new Combattant("P3", 10, 2, 2);
6         P1.attack("pied",P2);
7         P1.attack("poing",P2);
8         P1.attack("tete",P2);
9         P1.attack("tete",P2);
10      }
11  }
```

Vous devriez obtenir ce résultat :

```
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   P1 a encore 10 points de vie
5   P2 a encore 8 points de vie
6   P3 a encore 10 points de vie
7   P1 a encore 10 points de vie
8   P2 a encore 4 points de vie
9   P3 a encore 10 points de vie
10  P2 est mort
11  P1 a encore 10 points de vie
12  P3 a encore 10 points de vie
13
14  Process finished with exit code 0
```

3 Manipulation de graphes en POO

3.1 Introduction

Ici, on cherche à pousser votre réflexion sur le processus de conception de classes pour implémenter un concept existant en dehors de la programmation. Pour cela, il faut se placer du point de vue de l'utilisateur pour savoir ce dont il aurait besoin. Il faut aussi savoir de quoi est constitué le concept que l'on cherche à implémenter. C'est-à-dire, pour utiliser une analogie, les différentes briques qui sont utilisées pour construire le mur qui est votre objet final.
Cette partie est une introduction qui a pour but de poser des questions théoriques et les résoudre qui permettront de mieux comprendre la logique de conception.
Voici les questions, essayez d'y répondre avant de vous référer aux réponses juste en dessous :

1. Quels sont les 3 composants d'un **weighted graph** ?
2. Combien de classes sont nécessaires pour représenter tous les composants du **graph** et le **graph** lui même ?
3. Sachant que vous avez une classe qui représente les **arêtes** et que les **sommets** sont représentés par des chaînes de caractères. Déterminer quels sont les attributs de la class **graph**.
4. Maintenant que vous avec le schéma général de votre classe **graph**, il faut déterminer quelles méthodes sont nécessaires au fonctionnement de cet objet.
   Il faut donc réfléchir sur les questions suivantes :
   (a) Est-ce que l'utilisateur aura besoin de modifier l'objet une fois celui-ci initialisé ?
   (b) 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 ?
   (c) Est-il nécessaire de définir des méthodes qui ne seront pas utiles pour l'utilisateur mais utiles pour éviter la redondance du code ?

Une fois ces questions répondues, le schéma des méthodes nécessaires devient plus claire.
Il reste enfin à traiter l'implémentation de tout cela. C'est à dire, comment coder la classe pour qu'elle soit fonctionnelle.

Conseil

1. Il faut décomposer ce qui constitue un **graph**. Revenir sur le cours de la semaine 7 sur moodle.
2. Réfléchir aux différents objets du type les **arêtes** qui consituent un **graph**. Réflechir si ils ont besoin d'être représenté avec une classe ou est-ce que les types de "base" sont suffisants.
3. Les attributs sont les briques qui constituent votre classe. Il faut donc poser les variables nécéssaires et leur type.
4. Penser de la même manière que lorsque vous utilisez une application et critiquez (en bien ou en mal) les fonctionnalitées de celle-ci. Ici, il faut réfléchir à ce qu'une personne qui va utiliser votre "application" (la classe) aurait besoin.

Voici les réponses aux questions posées ci-dessus. Essayez de comprendre la logique derrière :
Certaines questions n'ont pas qu'une réponse car il y a toujours plus d'une manière d'implémenter une classe. Voici une façon de répondre :

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 **Edges** 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.
4. Les réponses à cette partie sont courtes mais assez essentielles pour savoir quelles méthodes coder :

(a) Oui, l'utilisateur voudra sûrement rajouter des **arêtes**, des **sommets** ou potentiellement changer le poids d'une **arête**. Il faut donc lui laisser cette possibilité

(b) Les graphs sont souvent accompagnés par des algorithmes de recherche, par exemple l'algorithme de Kruskal pour les **weighted graphs**. Il est donc utile de pouvoir identifier si un **sommet** ou une **arête** sont des composants de l'instance.

(c) Certaines méthodes pourraient avoir un code très long si on ne ségmentarise pas leur contenu. Cela peut entraver la relecture de votre code par vous ou par un tiers.

De plus dans un cadre d'optimisation, il faut éviter un maximum la redondance du code. Il est donc de bonne pratique de définir des méthodes qui vont éviter cela dans une classe.

3.2 Partie 2

Le code pour la classe **Edges** est fourni dans le dossier ressources sur Moodle. Utiliser le fichier **Main.java** dans le dossier **Ressources** sur Moodle pour effectuer des tests. Il devrait retourner :

```
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**.

Java :

```java
import java.util.List;
import java.util.HashMap;
import java.util.Vector;

public class graph_empty {

    // Les attributs de la classe graphe
    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
    public List<String> vertices = new Vector();

    // Methode qui permet l ajout d un sommet au graphe.
    public void add_vertex(String name){
        this.vertices.add(name); // Méthode qui permet d ajouter un sommet au graphe
    }

    // Cette méthode va tester si le sommet demandé existe dans le graphe. Si oui retourne le poids, sinon retourne 0.
    public double edge_exist(String from_vertex, String to_vertex){
        // Ecrire votre code ici
    }
    // L implémentation de la méthode ci dessous n est pas importante pour vous à comprendre. Elle vous est utile pour
    // 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
    private void generate_edge(String from_vertex, String to_vertex, double weight){
        if (this.vertices.contains(from_vertex) & this.vertices.contains(to_vertex)){
            Edges new_edge = new Edges(from_vertex,to_vertex,weight);
            this.edges.add(new_edge);
        }
        else {
            if (!this.vertices.contains(from_vertex)){
                this.vertices.add(from_vertex);
            }
            if (!this.vertices.contains(to_vertex)){
                this.vertices.add(to_vertex);
```

```
35          }
36          Edges new_edge = new Edges(from_vertex,to_vertex,weight);
37          this.edges.add(new_edge);
38        }
39
40      }
41
42      public void update_weight(String from_vertex, String to_vertex, double weight){
43        // Ecrire votre code ici
44      }
45      // Méthode qui va ajouter l arête dans le graphe.
46      public void new_edge(String from_vertex, String to_vertex, double weight){
47        // Ecrire votre code ici
48      }
49
50      // Méthode nous permettant de supprimer une arête du graphe.
51      public void del_edge(String from_vertex, String to_vertex){
52        for(Edges edge : this.edges ){
53          if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
54            this.edges.remove(edge);
55            System.out.println("Edge between " + from_vertex + " and " + to_vertex + " has been deleted.");
56            break;
57          }
58        }
59      }
60
61      // Fonction qui permet d imprimer les composants d un graphe
62      public void print(){
63        for(int i=0; i< this.vertices.size(); ++i){
64          System.out.println("The vertex number " + (i+1) + " has a value of: " + this.vertices.get(i));
65        }
66        for (Edges edge : this.edges){
67          edge.print();
68        }
69      }
70  }
```

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 cette **arête** n'existe pas.
2. La méthode **edge_exist()** qui va prendre en paramètre le **sommet** d'origine et le **sommet** d'arrivée. Si cette **arête** est dans le **graph** alors la méthode ressort son poids, 0 sinon.
3. La méthode **new_edge** doit créer une instance de **Edges** 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 comme 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 **Edges** pour changer le poids.
2. Il faut tester pour chaque **arête** (itération) si elle est égale à celle rentrée 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 celui indiqué en paramètre de la méthode.
   (c) Si l'arête existante n'a pas le même poids que celui indiqué en paramètre de la méthode.
   (d) Utilisez le résultat de **edge_exist** pour simplifier ces tests.

Solution

Java :

```java
import java.util.List;
import java.util.HashMap;
import java.util.Map;

public class Edge {
    public String from_vertex; // node de départ
    public String to_vertex;  // node d'arrivée ( chaîne de caractère)
    public double weight;  // poids de l'arête

    public Edge(String from_vertex, String to_vertex, double weight) {
        this.from_vertex = from_vertex;
        this.to_vertex = to_vertex;
        this.weight = weight;
    }

    public void print(){
        Map<String, String> edge_rep = new HashMap <String,String>(); // Création d'un dictionnaire pour
          pouvoir afficher une arête
        edge_rep.put("from_vertex",this.from_vertex);
        edge_rep.put("to_vertex",this.to_vertex);
        edge_rep.put("weight", String.valueOf(this.weight));
        System.out.println(edge_rep);
    }

}
```

Solution

Java :

```java
public class graph {
    // Les attributs de la class graphe
    public List<Edge> edges = new Vector();   // "Utilisation de vector car il faut que l'on puisse rajouter ou supprimer des éléments de la liste"
    public List<String> vertices = new Vector();



    // "Methode qui permet l'ajout d'un sommet au graphe."
    public void add_vertex(String name){
        this.vertices.add(name); // "Méthode qui permet d'ajouter un sommet au graphe"
    }

    // "Cette méthode va tester si le sommet demandé existe dans le graphe. Si oui retourne le poids, sinon retourne 0."
    public double edge_exist(String from_vertex, String to_vertex){
        for (Edge edge : this.edges) {
            if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex) {
                return edge.weight;
            }
        }
        return 0;
    }
    // "L'implémentation de la méthode ci dessous n'est pas importante pour vous à comprendre. Elle vous est utile pour"
    // "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"
    private void generate_edge(String from_vertex, String to_vertex, double weight){
        if (this.vertices.contains(from_vertex) & this.vertices.contains(to_vertex)){
            Edge new_edge = new Edges(from_vertex,to_vertex,weight);
            this.edges.add(new_edge);
        }
        else {
            if (!this.vertices.contains(from_vertex)){
                this.vertices.add(from_vertex);
            }
            if (!this.vertices.contains(to_vertex)){
                this.vertices.add(to_vertex);
            }
            Edge new_edge = new Edge(from_vertex,to_vertex,weight);
            this.edges.add(new_edge);
        }

    }

}
```

Solution

Java :

```
1  public void update_weight(String from_vertex, String to_vertex, double weight){
2      for (Edge edge : this.edges){
3      if(edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
4      edge.weight = weight;
5      System.out.println("Weight of" + edge + "has been updated");
6          }
7      else {
8      System.out.println("The vertex between the two nodes given does not exist, it will be created.");
9          }
10     }
11
12 }
13 // "Méthode qui va ajouter l'arête dans le graph."
14 public void new_edge(String from_vertex, String to_vertex, double weight){
15     double test_existence = this.edge_exist(from_vertex,to_vertex); // Peut valoir soit le point de l'arête soit 0 si
         elle n'existe pas.
16     if ( test_existence == weight){
17     System.out.println("Edge between" + from_vertex + " and " + to_vertex + " with the same weight already
         exists");
18     }
19     else{
20     if ( test_existence != 0) {
21     System.out.println("Edge between" + from_vertex + " and " + to_vertex + "exists but with a different weight
         and will be overwritten");
22     this.update_weight(from_vertex, to_vertex, weight);
23     }
24     else{
25     this.generate_edge(from_vertex, to_vertex, weight);
26         }
27     }
28
29 }
30
31 // "Méthode nous permettant de supprimer une arête du graph."
32 public void del_edge(String from_vertex, String to_vertex){
33     for( Edge edge : this.edges ){
34     if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
35     this.edges.remove(edge);
36     System.out.println("Edge between " + from_vertex + " and " + to_vertex + " has been deleted.");
37     break;
38         }
39     }
40 }
41
42 public void print(){
43     for(int i=0; i< this.vertices.size(); ++i){
44     System.out.println("The vertex number " + (i+1) + " has a value of: " + this.vertices.get(i));
45         }
46     for (Edge edge : this.edges){
47     edge.print();
48         }
49     }
50 }
```

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"*

Solution

```
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(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())
31
32 if __name__ == '__main__':
33     p = Point(3, 2)
34     p2 = Point(5,4)
35     print(str(p.distance(p2)))
36     print(str(p.milieu(p2)))
```