

Algorithmes et Pensée Computationnelle

Programmation orientée objet

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()` sensée représenter un chien.

Cette classe aura différents attributs et différentes méthodes que vous implémenterez au fur et à mesure de l'exercice.

Question 1: (🕒 10 minutes) Exercice 1

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

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

Ensuite, créez une méthode publique du même nom que la classe (`Dog`), qui va servir à initialiser nos différentes instances de cette classe. Cette méthode prendra en argument les éléments suivants et initialisera les attributs de notre instance avec :

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 créées.

💡 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.

>_ 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) Exercice 2

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 privés de la fonction. Pour les attributs publics, il vous suffit d'utiliser `nom.instance.attribut` pour l'obtenir.

Les méthodes getter consistent à retourner la valeur de l'attribut. Les setter consisteront à modifier la valeur de l'attribut. Les **setter** sont souvent utilisés pour modifier la valeur d'attributs privés.

Créez les méthodes suivantes :

1. `getTricks()`
2. `getRace()`
3. `getAge()`
4. `getMood()`
5. `setTricks()`
6. `setRace()`
7. `setAge()`
8. `setMood()`

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

💡 Conseil

Les setter et getter vous seront proposés via l'autocomplétion d'IntelliJ, nous vous recommandons tout de même d'en faire quelques unes à la main.

>_ 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) Exercice 3

Créez une méthode nommée `add_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) Exercice 4

Créez deux méthodes qui vont avoir un impact sur l'attribut `mood` du chien. 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) Exercice 5

Créez une méthode nommée `get_oldest` qui prend en argument un élément de type `Dog`, puis retourne le nom et l'âge du chien le plus âgé sous le format suivant : "`nom_chien est le chien le plus âgé 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 âgé avec " + this.age + " ans";
4     }
5     else{
6         return other.name + " est le chien le plus âgé avec " + other.getAge() + " ans";
7     }
8 }
```

Question 6: (🕒 5 minutes) Exercice 6

Créez une méthode de surcharge de la fonction `System.out.println` pour les objets de type `Dog`. Pour ce

faire, le nom de votre méthode doit être le suivant : toString().

Veillez à retourner une chaîne de caractère 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.

>_ 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 :

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Dog Lola = new Dog("Loola",List.of("rollover"),"Bouvier",10);  
4         Dog Tobl = new Dog("Tobi",List.of("rollover","do a barrel"),"Doggo",17);  
5         System.out.println(Lola.getAge());  
6         System.out.println(Lola.getMood());  
7         System.out.println(Lola.getRace());  
8         System.out.println(Lola.name);  
9         System.out.println(Lola.getTricks());  
10        Lola.setAge(13);  
11        Lola.setMood(8);  
12        Lola.setRace("Bouvier");  
13        Lola.name = "Lola";  
14        Lola.setTricks(List.of("rollover","do a barrel"));  
15        Lola.eat();  
16        Lola.leash();  
17        Lola.add_trick("sit");  
18        System.out.println(Dog.getNb_chiens());  
19        System.out.println(Lola.get_oldest(Tobl));  
20        System.out.println(Lola);  
21    }  
22 }
```

Vous devriez obtenir ce résultat :

```
1 10  
2 5  
3 Bouvier  
4 Loola  
5 [rollover]  
6 2  
7 Tobl est le chien le plus âgé avec 17 ans  
8 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 combattant (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)
- **défense:**(int)

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.

Les instances comprendront 4 attributs : l'attaque, la défense, les points de vie et un nom. Il y a également un attribut de classe nommé instances. Cet attribut est une liste comportant toutes les instances de la classe. Vous devrez compléter les 4 méthodes suivantes :

1. isAlive()
2. checkDead()
3. checkHealth()
4. attack(Fighter other)

Voici le squelette du code :

```

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
11     public Fighter(String name, int health, int attack, int defense) {
12         this.name = name;
13         this.health = health;
14         this.attack = attack;
15         this.defense = defense;
16         instances.add(this);
17     }
18
19     public int getAttack() {
20         return attack;
21     }
22
23     public int getHealth() {
24         return health;
25     }
26
27     public int getDefense() {
28         return defense;
29     }
30
31     public String getName() {
32         return name;
33     }
34
35     public void setAttack(int attack) {
36         this.attack = attack;
37     }
38
39     public void setDefense(int defense) {
40         this.defense = defense;
41     }
42
43     public void setHealth(int health) {
44         this.health = health;
45     }
46
47     public void setName(String name) {
48         this.name = name;
49     }
50
51     public Boolean isAlive() {
52         // à compléter
53     }
54
55     public static void checkDead() {
56         // à compléter
57     }
58
59     public static void checkHealth() {
60         // à compléter
61     }
62
63     public void attack (Fighter other){
64         // à compléter

```

```

65     }
66 }

```

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 soit 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 à l'envers, 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

```

1  public static void checkDead() {
2      List<Fighter> temp = new ArrayList<Fighter>();
3      for (Fighter f : Fighter.instances) {
4          if (f.isAlive()) {
5              temp.add(f);
6          } else {
7              System.out.println(f.getName() + " est mort");
8          }
9      }
10     Fighter.instances = temp;
11 }

```

Question 9: (🕒 5 minutes) `checkHealth()`

Définir une méthode `checkHealth()` qui parcourt la liste des instances et imprime le nombre de points de vie qu'il lui reste sous le format "`nom_instance` a encore `health_instance` points de vie".

>_ Solution

```

1  public static void checkHealth() {
2      for (Fighter f : Fighter.instances) {
3          System.out.println(f.getName() + " a encore " + f.getHealth() + " points de vie");
4      }
5  }

```

Question 10: (🕒 10 minutes) `attack(Fighter other)`

Définir une méthode `attack(Fighter other)` qui consiste à retirer des points de vie au combattant `other` en fonction de l'attaque de l'instance appelée 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_instance - defense_other`. Appelez ensuite les fonctions `checkDead()` et `checkHealth()` afin d'avoir un aperçu des combattants restants et de leur santé.

>_ Solution

```
1 public void attack (Fighter other){
2     if(other.isAlive()) {
3         int damage = this.attack - other.getDefense();
4         other.setHealth(other.getHealth() - damage);
5         Fighter.checkDead();
6         Fighter.checkHealth();
7         System.out.println("-----");
8     }
9     else{
10        System.out.println(other.getName() + " est déjà mort");
11        System.out.println("-----");
12    }
13 }
```

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

```
1 public class Main {
2
3     public static void main(String[] args) {
4         Fighter P1 = new Fighter("P1", 50, 30, 10);
5         Fighter P2 = new Fighter("P2", 50, 25, 10);
6         Fighter P3 = new Fighter("P3", 50, 25, 10);
7         P1.attack(P2);
8         P1.attack(P2);
9         P1.attack(P2);
10        P1.attack(P2);
11    }
12 }
```

Vous devriez obtenir ce résultat :

```
1 P1 a encore 50 points de vie
2 P2 a encore 30 points de vie
3 P3 a encore 50 points de vie
4 P1 a encore 50 points de vie
5 P2 a encore 10 points de vie
6 P3 a encore 50 points de vie
7 P2 est mort
8 P1 a encore 50 points de vie
9 P3 a encore 50 points de vie
10 P2 est déjà mort
11
12 Process finished with exit code 0
```

3 Conception de classes pour coder des weighted graphs

3.1 Partie 1

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 de l'exercice a pour but de poser des questions théoriques qui sont une aide pour comprendre la logique de conception.

Question 11: (🕒 15 minutes) Exercice 1

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 avez 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 constituent un graph. Réfléchir 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 nécessaires et leur type.
4. Penser de la même manière que lorsque vous utilisez une application et critiquez (en bien ou en mal) les fonctionnalités de celle-ci. Ici, il faut réfléchir à ce qu'une personne qui va utiliser votre "application" (la classe) aurait besoin.

>_ Solution

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 segmentarise 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 12: (🕒 20 minutes) Exercice 2

Voici une partie de la classe `graph` codée. Implémentez les méthodes `"update_weight"`, `"new_edge"` et `"edge_exist"`.

Java :

```
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 class graph
8     public List<Edges> edges = new Vector(); // Utilisation de vector car il faut que l'on puisse rajouter ou supprimer des é
        éléments de la liste
9     public List<String> vertices = new Vector();
10
11
12
13     // Methode qui permet l'ajout d'un sommet au graph.
14     public void add_vertex(String name){
15         this.vertices.add(name); // Méthode qui permet d'ajouter un sommet au graph
16     }
17
18     // Cette méthode va tester si le sommet demandé existe dans le graph. Si oui retourne le poids, sinon retourne 0.
19     public double edge_exist(String from_vertex, String to_vertex){
20         // Ecrire votre code ci-dessous
21
22
23
24
25         // Fin de la zone d'écriture
26     }
27     // L'implémentation de la méthode ci dessous n'est pas importante pour vous à comprendre. Elle vous est utile pour
28     // générer une arête lorsque vous cherchez à en ajouter une à votre graph. Elle fait aussi le test si jamais
29     // les sommets utilisés font partis du graph ou non. Si non, elle va les ajouter au graph. Cette méthode peut
30     // être utile dans la méthode new_edge
31     private void generate_edge(String from_vertex, String to_vertex, double weight){
32         if (this.vertices.contains(from_vertex) & this.vertices.contains(to_vertex)){
33             Edges new_edge = new Edges(from_vertex,to_vertex,weight);
34             this.edges.add(new_edge);
35         }
36         else {
37             if (!this.vertices.contains(from_vertex)){
38                 this.vertices.add(from_vertex);
39             }
40             if (!this.vertices.contains(to_vertex)){
41                 this.vertices.add(to_vertex);
```

```

42     }
43     Edges new_edge = new Edges(from_vertex,to_vertex,weight);
44     this.edges.add(new_edge);
45 }
46
47 }
48
49 public void update_weight(String from_vertex, String to_vertex, double weight){
50     // Ecrire votre code ci-dessous
51
52
53
54
55
56
57     // Fin de la zone d'écriture
58 }
59 // Méthode qui va ajouter l'arête dans le graph.
60 public void new_edge(String from_vertex, String to_vertex, double weight){
61     // Ecrire votre code ci-dessous
62
63
64
65
66
67
68
69
70
71
72
73
74     // Fin de la zone d'écriture
75 }
76
77 // Méthode nous permettant de supprimer une arête du graph.
78 public void del_edge(String from_vertex, String to_vertex){
79     for( Edges edge : this.edges ){
80         if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
81             this.edges.remove(edge);
82             System.out.println("Edge between " + from_vertex + " and " + to_vertex + " has been deleted.");
83             break;
84         }
85     }
86 }
87
88 }
89
90 // Fonction qui permet d'imprimer les composants d'un graph
91 public void print(){
92     for(int i=0; i< this.vertices.size(); ++i){
93         System.out.println("The vertex number " + (i+1) + " has a value of: " + this.vertices.get(i));
94     }
95     for (Edges edge : this.edges){
96         edge.print();
97     }
98 }
99 }

```

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 :

```
1 import java.util.List;
2 import java.util.HashMap;
3 import java.util.Map;
4
5 public class Edges {
6     public String from_vertex; // node de départ
7     public String to_vertex; // node d'arrivée ( chaîne de caractère)
8     public double weight; // poids de l'arête
9
10    public Edges(String from_vertex, String to_vertex, double weight) {
11        this.from_vertex = from_vertex;
12        this.to_vertex = to_vertex;
13        this.weight = weight;
14    }
15
16    public void print(){
17        Map<String, String> edge_rep = new HashMap <String,String>(); // Création d'un dictionnaire pour
18        pouvoir afficher une arête
19        edge_rep.put("from_vertex",this.from_vertex);
20        edge_rep.put("to_vertex",this.to_vertex);
21        edge_rep.put("weight", String.valueOf(this.weight));
22        System.out.println(edge_rep);
23    }
24 }
```

>_ Solution

Java :

```
1 import java.util.List;
2 import java.util.HashMap;
3 import java.util.Vector;
4
5 public class graph {
6
7     // Les attributs de la class graph
8     public List<Edges> edges = new Vector(); // Utilisation de vector car il faut que l'on puisse rajouter ou
        supprimer des éléments de la liste
9     public List<String> vertices = new Vector();
10
11
12
13     // Methode qui permet l'ajout d'un sommet au graph.
14     public void add_vertex(String name){
15         this.vertices.add(name); // Méthode qui permet d'ajouter un sommet au graph
16     }
17
18     // Cette méthode va tester si le sommet demandé existe dans le graph. Si oui retourne le poids, sinon retourne 0.
19     public double edge_exist(String from_vertex, String to_vertex){
20         for (Edges edge : this.edges) {
21             if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex) {
22                 return edge.weight;
23             }
24         }
25         return 0;
26     }
27     // L'implémentation de la méthode ci dessous n'est pas importante pour vous à comprendre. Elle vous est utile
        pour
28     // générer une arête lorsque vous cherchez à en ajouter une à votre graph. Elle fait aussi le test si jamais
29     // les sommets utilisés font partis du graph ou non. Si non, elle va les ajouter au graph. Cette méthode peut
30     // être utile dans la méthode new_edge
31     private void generate_edge(String from_vertex, String to_vertex, double weight){
32         if (this.vertices.contains(from_vertex) & this.vertices.contains(to_vertex)){
33             Edges new_edge = new Edges(from_vertex,to_vertex,weight);
34             this.edges.add(new_edge);
35         }
36         else {
37             if (!this.vertices.contains(from_vertex)){
38                 this.vertices.add(from_vertex);
39             }
40             if (!this.vertices.contains(to_vertex)){
41                 this.vertices.add(to_vertex);
42             }
43             Edges new_edge = new Edges(from_vertex,to_vertex,weight);
44             this.edges.add(new_edge);
45         }
46     }
47
48
49     public void update_weight(String from_vertex, String to_vertex, double weight){
50         for (Edges edge : this.edges){
51             if(edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
52                 edge.weight = weight;
53                 System.out.println("Weight of" + edge + "has been updated");
54             }
55             else {
56                 System.out.println("The vertex between the two nodes given does not exist, it will be created.");
57             }
58         }
59     }
60 }
61 // Méthode qui va ajouter l'arête dans le graph.
62 public void new_edge(String from_vertex, String to_vertex, double weight){
63     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.
64     if ( test_existence == weight){
65         System.out.println("Edge between" + from_vertex + " and " + to_vertex + " with the same weight already
            exists");
66     }
67     else{
68         if ( test_existence != 0) {
69             System.out.println("Edge between" + from_v13tex + " and " + to_vertex + "exists but with a different
            weight and will be overwritten");
70             this.update_weight(from_vertex, to_vertex, weight);
71         }
72     }
```

>_ Solution

Java :

```
1 public void update_weight(String from_vertex, String to_vertex, double weight){
2     for (Edges 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
16     elle n'existe pas.
17     if ( test_existence == weight){
18         System.out.println("Edge between" + from_vertex + " and " + to_vertex + " with the same weight already
19         exists");
20     }
21     else{
22         if ( test_existence != 0) {
23             System.out.println("Edge between" + from_vertex + " and " + to_vertex + "exists but with a different weight
24             and will be overwritten");
25             this.update_weight(from_vertex, to_vertex, weight);
26         }
27         else{
28             this.generate_edge(from_vertex, to_vertex, weight);
29         }
30     }
31 }
32 // Méthode nous permettant de supprimer une arête du graph.
33 public void del_edge(String from_vertex, String to_vertex){
34     for( Edges edge : this.edges ){
35         if (edge.from_vertex == from_vertex & edge.to_vertex == to_vertex){
36             this.edges.remove(edge);
37             System.out.println("Edge between " + from_vertex + " and " + to_vertex + " has been deleted.");
38             break;
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 (Edges edge : this.edges){
47         edge.print();
48     }
49 }
50 }
```