Algorithmes et Pensée Computationnelle

Classes abstraites et interfaces

Les exercices sont construits autour des concepts d'héritage, de classes abstraites et d'interfaces. Au terme de cette séance, vous devez être en mesure de différencier une classe abstraite d'une interface, savoir à quel moment utiliser l'un ou l'autre, utiliser le concept d'héritage multiple, factoriser votre code afin de le rendre mieux structuré et plus lisible.
Cette feuille d'exercices est divisée en 2 sections. L'une portant sur les classes abstraites et l'autre sur les interfaces.
Une autre feuille comportant des exercices de consolidation sur les notions abordées précédemment est disponible sur Moodle.
Les exercices de cette feuille d'exercices doivent être faits uniquement en **Java**.
Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier **Ressources**.

1 Consolidation - Programmation Orientée Objet

Question 1: (*10 minutes*) Encapsulation - Java
L'encapsulation sert à cacher les détails d'implémentation. L'encapsulation sert uniquement à montrer que les informations essentielles aux utilisateurs.

En Java, il est recommandé de déclarer les attributs des classes comme étant **private** et de mettre à disposition des utilisateurs des méthodes publiques d'accès afin qu'ils puissent accéder ou modifier la valeur des attributs privés.

Les méthodes publiques d'accès comme **getName**et **setName** doivent être nommées avec soit **get** soit **set** suivi du nom de l'attribut avec la 1ère lettre en majuscule (Java Naming convention)[https ://www.oracle.com/java/technologies/javase/codeconve namingconventions.html]).
Le mot clé **this** fait référence à l'objet en question.

1. Dans votre IDE, créez une classe **Person**,
2. Ajoutez-y un attribut privé **name**,
3. Créez un getter et un setter pour l'attribut **name** en suivant la convention de nommage des méthodes.
4. Dans votre **main**, créez une instance de **Person**,
5. En utilisant le setter défini précédemment, donnez un nom (**name**) à votre instance.
6. Affichez le nom de l'instance en utilisant le getter de l'attribut **name**.

Question 2: (*10 minutes*) Héritage - Java
Comme vous le savez, il est possible que des classes-filles héritent des attributs ou méthodes de classes-mères. En Java, il faut utiliser le mot-clé **extends** lorsqu'on défini une classe-fille. Ainsi, l'héritage permet la réutilisation des attributs et méthodes d'une classe existante.

1. Créez une classe-mère **Vehicle** ayant un attribut protégé appelé **brand**.
2. Créez un constructeur pour la classe **Vehicle** et assignez une valeur à l'attribut **brand**.
3. Définissez une méthode **honk** qui affiche **"Tuut, tuut!"**
4. Créez une classe-fille **Car** qui hérite de **Vehicle** et ayant pour attribut **modelName** avec pour valeur par défaut **"Mustang"**.

Conseil

Dans le constructeur de la classe-fille, n'oubliez pas de faire appel au constructeur de la classe-mère en utilisant le mot-clé **super**

2 Classes abstraites

Question 3: (*15 minutes*) Création d'une classe abstraite

Une classe abstraite est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable. Elle est déclarée en utilisant le mot-clé **abstract**. Elle peut inclure des méthodes abstraites ou non. Bien que ne pouvant être instanciées, les classes abstraites servent de base à des sous-classes qui en sont dérivées. Lorsqu'une sous-classe est dérivée d'une classe abstraite, elle complète généralement l'implémentation de toutes les méthodes abstraites de la classe-mère. Si ce n'est pas le cas, la sous-classe doit également être déclarée comme abstraite.

```
// Exemple de classe abstraite
public abstract class Animal {
   private int speed;
   // Déclaration d une méthode abstraite
   abstract void run();
}

public class Cat extends Animal {
      // Implémentation d une méthode abstraite
      void run() {
         speed += 10;
      }
```

— Implémentez une classe abstraite appelée **Item**. 1. Elle doit avoir 4 attributs d'instance et un attribut de classe, qui sont les suivants :

```
private int id;
private static int count = 0;
private String name;
private double price;
private ArrayList<String> ingredients;
```

> **Conseil**
>
> Les attributs d'instance sont créés lors de l'instanciation d'un objet (à l'aide du mot clé **new**) et détruits lors de la destruction de l'objet. Les attributs de classes (attributs statiques), quant à eux, sont créés lors de l'exécution du programme et détruits lors de l'arrêt du programme. En Java, les attributs de classes sont accessibles en utilisant le nom de la classe soit : **ClassName.AttributeName**.

— Créer un constructeur pour initialiser les attributs **name, price, ingredients et** id. L'attribut id incrémentera à chaque instanciation de la classe.

> **Conseil**
>
> Pensez à utilisez **count** pour initialiser la valeur d'id. Ainsi, dans le constructeur, id sera égal à **++count**.

— Implémentez les **getters** des attributs id**, name, price et ingredients.**

— Implémentez les méthodes **equals**(Object o) et **toString**().

> **Conseil**
>
> La méthode **equals** permet de comparer deux objets. Elle prend en entrée un objet de type **Object** et doit retourner True si l'objet instancié est égal à l'objet passé en paramètre.

Question 4: (*15 minutes*) Classe abstraite et types d'attributs

— Implémentez une classe abstraite **Figure** contenant deux attributs protégés : **largeur** et **longueur** et deux méthodes abstraites : **getAire**()et **getPerimetre**().

— Créez deux classes **Carre** et **Rectangle** qui héritent de la classe **Figure**. À l'intérieur de ces classes, implémentez les méthodes **getAire**() et **getPerimetre**().

Conseil

Un attribut protégé est accessible aussi bien dans la classe-mère que dans la(les) classe(s)-fille(s).
On utilise le mot clé **protected** pour rendre des attributs protégés.
Pour rappel, pour créer une classe fille, on utilise le mot-clé **extends** :**public class Carre extends Figure**.

3 Interfaces

Question 5: (*15 minutes*) Interface et héritage (Liée à la question 1)

En Java, une interface se déclare comme suit :

```
public interface IMakeSound{
    final double MY_DECIBEL_VALUE = 75;
    void makeSound();
}
```

Les méthodes déclarées dans une interface doivent être implémentées dans des sous-classes :

```
public class Cat extends Animal implements IMakeSound {
    void makeSound(){
        System.out.println("I meow at" + MY_DECIBEL_VALUE + "decibel.");
    }
}
```

- Implémentez une interface **Edible** contenant une méthode **eatMe** qui ne retourne aucune valeur.
- Implémentez une interface **Drinkable** contenant une méthode **drinkMe** qui ne retourne aucune valeur.
- Implémentez une classe **Food** qui hérite la classe **Item** (définie dans la section 1) et qui implémente l'interface **Edible**. Implémentez le constructeur de **Food** et la méthode **eatMe** (dans la classe **Food**).

Conseil

Vous pouvez reprendre la classe **Item** du premier exercice.
Dans la méthode **eatMe**(), vous pouvez simplement afficher un message en utilisant un **println**.

Certains aliments ne sont pas seulement **Edible** (mangeable) mais aussi **Drinkable** (buvable) comme les soupes par exemple.

4. Implémentez une classe **Soup** qui hérite de **Food** et implémente l'interface **Drinkable**. Ensuite, implémentez à la fois un constructeur pour **Soup** ainsi que la méthode **drinkMe** (dans la classe **Soup**).

Vous pouvez ensuite créer des instances de **Soup** et **Food** à l'aide des lignes suivantes pour tester les méthodes **eatMe**() et **drinkMe**().

```
Soup s1 = new Soup("Kizili soup", 7.7, new ArrayList<String>(Arrays.asList("bulgur", "meat", "tomato")));

Food f = new Food("Stuffed peppers", 12,new ArrayList<String>(Arrays.asList("rice", "tomato", "onion")));
```