Algorithmes et Pensée Computationnelle

Consolidation 2 - Programmation Orientée Objet

Le but de cette séance est d'approfondir les notions de programmation orientée objet vues précédemment. Les exercices sont construits autour des concepts d'héritage, de 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 série d'exercices est divisée en 3 sections dont les premières portant sur les classes abstraites et les interfaces en général. La dernière section comporte des exercices pratiques sur les notions abordées précédemment.
Les exercices doivent être faits uniquement en **Java**.
Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier **Ressources**.

1 Programmation Orientée Objet - Classes abstraites

Question 1: (*10 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.

```
1  // Exemple de classe abstraite
2  public abstract class Animal {
3     private int speed;
4     // Déclaration d une méthode abstraite
5     abstract void run();
6  }
7
8  public class Cat extends Animal {
9        // Implémentation d une méthode abstraite
10       void run() {
11          speed += 10;
12       }
```

— Implémentez une classe abstraite appelée **Item**. 1. Elle doit avoir 4 variables (attributs) d'instance et une variable de classe, qui sont les suivantes :

```
1  private int id;
2  private static int count = 0;
3  private String name;
4  private double price;
5  private ArrayList<String> ingredients;
```

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

— Créer un constructeur pour initialiser les variables **name, price, ingredients et id**. La variable **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 variables **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.

Solution

```
1  import java.util.*;
2
3
4  public abstract class Item {
5
6     private int id;
7     private static int count = 0;
8     private String name;
9     private double price;
10    private ArrayList<String> ingredients;
11
12    public Item (String name, double price, ArrayList<String> ingredients) {
13       this.id = ++count;
14       this.name = name;
15       this.price = price;
16       this.ingredients = ingredients;
17    }
18
19    public int getID() {
20       return this.id;
21    }
22
23    public String getName() {
24       return this.name;
25    }
26
27    public double getPrice() {
28       return this.price;
29    }
30
31    public ArrayList<String> getIngredients() {
32       return this.ingredients;
33    }
34
35    public boolean equals(Object o) {
36       if (o instanceof Item) {
37          Item i = (Item) o;
38          return i.getID() == this.getID();
39       }
40       return false;
41    }
42
43    public String toString() {
44       return "*-*-*-*-*-*" +
45             "\nID: " + this.getID() +
46             "\nName: " + this.getName() +
47             "\nPrice: " + this.getPrice() + " CHF" +
48             "\nList of ingredients: " + this.getIngredients().toString() +
49             "\n*-*-*-*-*-*";
50    }
51 }
```

Question 2: (*10 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 les attributs protégés.
Pour rappel, pour créer une classe fille, utiliser le mot-clé **extends** :**public class Carre extends Figure**.

Solution

```
1  abstract class Figure {
2
3     protected float largeur;
4     protected float longueur;
5
6     public Figure(float largeur, float longueur){
7        this.largeur = largeur;
8        this.longueur = longueur;
9     }
10
11    public abstract float getperimetre();
12    public abstract float getaire();
13 }
14
15 class Carre extends Figure {
16
17    public Carre(float largeur) {
18       super(largeur, largeur);
19    }
20
21    @Override
22    public float getperimetre() {
23       return this.largeur * 4;
24    }
25
26    @Override
27    public float getaire() {
28       return this.largeur * this.largeur;
29    }
30
31 }
32
33 class Rectangle extends Figure {
34
35    public Rectangle (float largeur, float longueur){
36       super(largeur, longueur);
37    }
38
39    @Override
40    public float getperimetre(){
41       return (this.largeur + this.longueur)*2;
42
43    }
44    @Override
45    public float getaire(){
46       return this.largeur * this.longueur;
47    }
48 }
49
50 public class Main {
51    public static void main(String[] args) {
52       Carre c = new Carre(5.0f);
53       Rectangle r = new Rectangle(4.0f, 3.0f);
54
55       System.out.println(c.getperimetre());
56       System.out.println(r.getaire());
57    }
58 }
```

2 Interfaces

Question 3: (*10 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")));
```

Solution

```
1  import java.util*;
2
3
4  public interface Edible{
5     void eatMe();
6  }
7  public interface Drinkable{
8     void drinkMe();
9  }
10 public class Food extends Item implements Edible{
11    public Food (String name, double price, ArrayList<String> ingredients){
12       super(name, price, ingredients);
13    }
14    public void eatMe(){
15       System.out.println("Eat me!" + toString());
16    }
17 }
18
19 public class Soup extends Food implements Drinkable{
20    public Soup(String name, double price, ArrayList<String> ingredients){
21       super(name, price, ingredients);
22    }
23    public void drinkMe(){
24       System.out.println("Drink the soup !" + toString());
25    }
26 }
```

3 Exercices complémentaires

Question 4: (*10 minutes*) **Programmation de base**
Ecrivez un programme Python qui imprime tous les nombres impairs à partir de 1 jusqu'à un nombre **n** défini par l'utilisateur. Ce nombre **n** doit être supérieur à 1. Exemple : si n = 6, résultat attendu : 1, 3, 5

Solution

```
def nombresImpairs(limite):
    for nb in range(limite+1):
        if nb % 2 == 1:
            print(nb)

limit = int(input("Entrez une valeur maximale: "))

print("Nombres impairs compris entre 1 et " + str(limit) + " : ")
nombresImpairs(limit)
```