TD 1 Correction

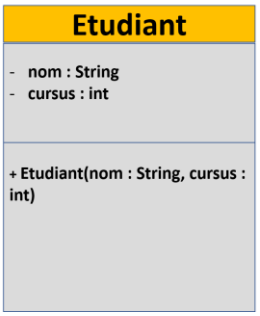
Exercice I

On commence par une classe `Etudiant`. Dans celle-ci nous allons avoir les prochains attributs :

- Un attribut nom de type `String` ;
- Un attribut `cursus` de type entier ;

De plus, nous aurons mettre en place un constructeur avec la signature `Etudiant(String nom, int cursus)`

Cette classe peut également être représentée par un *diagramme de classe* comme ci-dessous :



Nous allons voir les diagrammes de classe plus tard dans ce cours. Pour l'instant, prenons-les comme une illustration sommaire de la classe. Il faut savoir qu'un petit « - » devant un attribut ou une méthode veut dire que celui/celle-ci est privé(e), tandis qu'un petit « + » veut dire que l'attribut ou la méthode est publique.

Nous allons premièrement détailler cette classe.

1. Petit rappel :
    - Qu'est-ce qu'un attribut ?
    - Qu'est-ce qu'un constructeur ?
    - Qu'est-ce qu'une variable ou méthode publique ? Et une variable privée ?

SOLUTION :

Un attribut : une variable qui caractérise une classe (par exemple le nom d'un étudiant)
Un constructeur : une méthode spéciale, qui porte le nom de la classe, et qui sert à créer (instancier) des objets de cette classe
Une variable privée : une variable qui ne peut pas être directement utilisée en dehors de la classe
Une méthode/variable publique : elles peuvent être utilisées en dehors de la classe

2. Le premier constructeur initialisera les attributs de la classe par les valeurs données en entrée. Ecrivez ce constructeur.

SOLUTION :

```
public Etudiant(String nom, int cursus) {
   this.nom = nom;
   this.cursus = cursus;
}
```

3. Etudiez le code suivant :

```
public Etudiant(String nom, int cursus){
   nom = this.nom ;
   cursus = this.annee ;
}
```

Qu'est-ce que fait ce code ?

SOLUTION :

Premièrement on regarde le code. En l'observant on note les éléments suivants :

- La méthode porte le nom `Etudiant` et elle est dans la classe `Etudiant`. Il s'agit donc d'un constructeur.
- Le code `this.nom`, `this.annee` fait référence aux attributs de la classe (c'est pourquoi on voit la référence `this`).
- Les valeurs `nom` et `cursus` (sans `this`) se réfèrent aux paramètres en entrée.

L'analyse du code : D'un côté, ce code est censé d'être utilisé comme un constructeur -- ce qui veut dire que l'`etudiant` en question n'a pas été créé avant que cette méthode soit appelée. Ce qui veut dire que les attributs `this.nom` et `this.annee` n'ont pas été instanciés (ils ont été déclarés, mais n'ont pas encore de valeurs) -- par défaut ils sont mis à `null` en Java. Cette méthode met les paramètres en entrée à null, donc par exemple un appel à ce constructeur du type :

```
Etudiant marineDelafontaine = new Etudiant("Marine Delafontaine", 2);
```

créera un objet `marineDelafontaine` qui aura ses deux attributs mis à `null`.

Exercice II

Dans cet exercice nous allons utiliser notre constructeur pour créer des objets de type Etudiant.

1. Petit rappel :
   - Qu'est-ce qu'une méthode principale (`main`) ?

- Comment on définit une telle méthode dans une classe ?
- Comment peut-on initialiser des objets de type Etudiant dans une autre classe ?
- Nomenclature : comment écrit-on les noms des variables ?

SOLUTION :

Une méthode main : le point d'entrée à un programme, ce qui s'exécute.

Définir une méthode main :

```
public static void main (String[] args) {
        // code de la methode main
}
```

Comment initialiser des objets de type Etudiant : on a trois façons de le faire :

1. Déclaration, puis initialisation :

```
Etudiant marineDelafontaine;
marineDelafontaine = new Etudiant("Marine Delafontaine", 2);
```

2. Déclaration ET initialisation :

```
Etudiant marineDelafontaine = new Etudiant("Marine Delafontaine", 2);
```

3. Déclaration ET association à un objet déjà existent :

```
Etudiant marineDelafontaine = new Etudiant("Marine Delafontaine", 2);
Etudiant copieDeMarineDelafontaine = marineDelafontaine;
```

Nomenclature : comment nommer les variables

Les variables avec un nom simple (`nom`, `prenom`, `age`, `cursus`) doivent porter un nom qui commence par une minuscule (non-majuscule).

Les variables avec un nom composé (`premier index`, `carburant restant`) doivent porter un nom qui commence par une minuscule ET la première lettre de chaque nouveau mot commence par une majuscule : `premierIndex`, `carburantRestant`.

Attention : les noms de variables doivent être descriptifs ! Je pénalise des noms de variable du genre `Etudiant v`, `Cours j`, etc.

De plus, même les variables qui représentent des personnes (e.g. Jean Dupont) doivent porter des noms de variable qui commencent avec une minuscule, c.a.d. `jeanDupont`.

2. Dans une nouvelle classe MonTD1, dans la méthode principale, créez ces objets :

- Un étudiant de première année avec le nom « `Jean Dupont` ».
- Une étudiante de deuxième année avec le nom « `Marine Delafontaine` »

SOLUTION :

```
final Etudiant jeanDupont = new Etudiant("Jean Dupont", 1);
final Etudiant marineDelafontaine = new Etudiant("Marine Delafontaine",
2);
```

3. La plupart de nos étudiants vont être de première année. Pour simplifier leur initialisation, on écrit un deuxième constructeur dans la classe `Etudiant`, avec la signature `Etudiant(String nom)`. Ce constructeur initialise l'attribut `nom` à la valeur en entrée, et initialise l'attribut `cursus` à la valeur par défaut 1.
   En général, c'est une bonne pratique d'appeler le constructeur avec plus de paramètres dans celui avec moins de paramètres. On appelle le premier constructeur à partir du deuxième en utilisant le mot dédié `this(…)` -- en paranthèse les paramètres demandés par le constructeur appelé.

   - Pourquoi est-ce que c'est une bonne pratique de faire cela ?
   - Ecrivez le code du deuxième constructeur.
   - Comment peut-on remplacer le code qui crée l'étudiant au nom de Jean Dupont en utilisant ce nouveau constructeur ?
   - Peut-on utiliser le deuxième constructeur pour créer l'étudiant au nom Marine Delafontaine ?

SOLUTION :

- C'est une bonne pratique d'appeler le constructeur avec plus de paramètres dans le constructeur avec moins de paramètres pour éviter la déduplication de code (voire aussi le CM 3 pour se rappeler pourquoi ce n'est pas bien de copier/coller du code).

- Le code du deuxième constructeur :

```
public Etudiant(String nom) {
      this(nom, 1);
}
```

Pourquoi ce code ?
Un petit rappel : `this` est un mot dédié en Java, qui se réfère à l'objet lui-même (cela remplace le mot Etudiant dans la classe Etudiant).
Ce qu'on veut écrire pour ce deuxième constructeur serait :

```
public Etudiant(String nom) {
      this.nom = nom;
      this.cursus = 1;
}
```

Mais on ne veut pas dédupliquer le code du premier constructeur, donc ce qu'il faut faire c'est d'appeler le premier constructeur avec, en entrée le nom mis à `nom`, et le `cursus` mis à 1. Voici donc pourquoi `this(nom, 1);`

- Remplacer le code pour `jeanDupont` :

```
final Etudiant jeanDupont = new Etudiant("Jean Dupont");
```

- Remplacer le code pour `marineDelafontaine` ?

On ne peut pas remplacer le code pour l'initialisation de `marineDelafontaine`, car cet objet à la valeur de cursus mise à 2, tandis que le deuxième constructeur met la valeur de cursus par défaut à 1.

Exercice III

Dans cet exercice nous allons ajouter des méthodes pour qu'on puisse avoir accès aux attributs des objets de type `Etudiant`.

1. Disons que dans notre méthode principale nous avons le code suivant :

```
final thibaultLagrange = new Etudiant("Thibault Lagrange");
System.out.println(thibaultLagrange.cursus);
```

- Est-ce que ce code compile ?
- Qu'est-ce qui se passe lors de l'exécution ?

SOLUTION :

- La première ligne de code ne compile pas, puisqu'il nous manque le type de la variable `thibaultLagrange`.

  Même si on y rajoutait le type, c.a.d. si on remplaçait la première ligne par `final Etudiant thibaultLagrange = new Etudiant("Thibault Lagrange");`

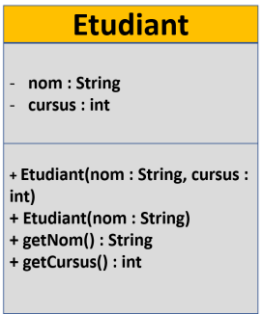
  on aurait encore un souci. Notamment la deuxième ligne de code ne compile pas non plus. Le code `thibaultLagrange.cursus` se réfère à l'attribut `cursus` de l'objet `thibaultLagrange`, un attribut défini comme étant privé. Un attribut privé ne peut

être utilisé que dans la classe dans laquelle il est défini. Dans notre cas, `thibaultLagrange` est un objet de type `Etudiant`, tandis que le code qu'on voit ici est dans la méthode main de la classe MonTD1. En conséquence ce code donnera une erreur de compilation.

- Ce code ne s'exécute pas, car le code ne compile pas

2. Pour avoir accès aux attributs de la classe `Etudiant` en dehors de cette classe on pourrait : a) faire les attributs publiques ; b) ajouter des méthodes de type `getAttribut()` qui rendront les valeurs stockées par les attributs en question.
    - Discutez les avantages et les inconvénients de ces deux méthodes
    - Petit rappel : quelle est la syntaxe d'une méthode qui rend une valeur à la sortie ? Qu'en des méthodes qui ne rendent aucune valeur ?
    - Ecrivez, dans la classe `Etudiant`, deux méthodes qui rendent les valeurs des attributs de la classe Etudiant, avec les signatures :
        - `String getNom()`
        - `int getCursus()`

    Ceci changera le diagramme de classe de la classe `Etudiant` à :



SOLUTION :

- Il est tentant de déclarer tout attribut en tant que valeur publique. Ceci nous donnerait la possibilité d'accéder aux variables de la classe Etudiant plus facilement dans d'autres classes. Mais d'un autre côté ceci peut causer des problèmes, puisque cela permet non seulement d'accéder à ces attributs, mais aussi de leur changer la valeur. Or, ceci fait le code particulièrement difficile à gérer, par rapport aux erreurs possibles qu'il peut générer. On va préférer donc de définir des méthodes de type getAttribut().
- Méthode qui rend une sortie :
  `<mentions> <type de sortie> <nom de la Méthode>(Paramètres, type et nom, séparés par une virgule)`

  Méthode qui ne rend aucune sortie (procédure) :

```
<mentions> void <nom de la Méthode>(Paramètres, type et nom, séparés par une virgule)
```

- Le code des méthodes getNom() et getCursus() :

```
public String getNom() {
	return this.nom;
}

public int getCursus() {
	return this.cursus;
}
```

3. Modifiez le code au point 1. tel qu'il finit par afficher le cursus de l'étudiant `thibaultLagrange`.

SOLUTION :

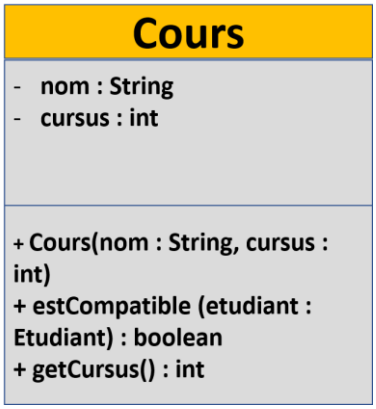
```
final Etudiant thibaultLagrange = new Etudiant("Thibault Lagrange");
System.out.println(thibaultLagrange.getCursus());
```

Exercice IV

Dans cet exercice nous allons créer une nouvelle classe, `Cours`, dont les objets seront des cours que les étudiants pourront choisir pour leur programme d'étude, à condition que le cours soit compatible avec leur année.

1. Nous allons mettre en place une classe `Cours` avec deux attributs et trois méthodes :
   - Un attribut `nom` de type `String`
   - Un attribut `cursus` de type entier
   - Un constructeur avec la signature `Cours(String nom, int cursus)`, qui mettra les attributs de l'instance de `Cours` aux valeurs en entrée
   - Une méthode `boolean estCompatible(Etudiant etudiant)` qui retourne une valeur de type `boolean` : `true` si l'étudiant a la même `cursus` que le cours, et `false` autrement
   - Une méthode `int getCursus()` qui retourne la valeur actuellement stockée par l'attribut `cursus` de l'instance de `Cours`.

Un résumé de cette classe est capturé par le diagramme de classe ci-dessous :



- Ecrivez la méthode `estCompatible(Etudiant etudiant)`

SOLUTION :

La méthode `estCompatible(Etudiant etudiant)` fait partie de la classe `Cours`. Dedans, il faut vérifier si l'attribut `cursus` de l'`etudiant` en question a la même valeur que l'attribut `cursus` de l'objet de type `Cours`. Ces deux attributs ont le type entier (`int`). Pour accéder à l'attribut `cursus` de la classe `Etudiant` dans la classe `Cours`, il faut utiliser la méthode `getCursus()`. (Pourquoi ?) Pour accéder à l'attribut cursus de l'objet de la classe `Cours`, on peut directement utiliser le nom de l'attribut. Alors notre code pourrait être :

```
public boolean estCompatible(Etudiant etudiant) {
      return(this.cursus == etudiant.getCursus());
}
```

2. Nous allons ensuite modifier la classe Etudiant : on y rajoute un attribut qui s'appelle mesCours et qui est de type Cours[]. Un étudiant pourra choisir jusqu'à un maximum de 20 cours.
    - Petit rappel : comment initialise-t-on un tableau dans un constructeur ?
    - Réécrivez les constructeurs de la classe Etudiant pour prendre en compte ce nouveau tableau.

SOLUTION :

Dans un constructeur, typiquement on initialise un tableau en indiquant sa taille. On peut supposer que le tableau a déjà été déclaré avant, notamment lorsqu'on a déclaré tous les attributs. Etant donc donné un tableau <nom du tableau> d'un type <type du tableau>, on initialise sa valeur comme-ça :

<nom du tableau> = new <type du tableau>[<taille du tableau>]

```java
public class Etudiant {
	private String nom;
	private int cursus;
	private Cours[] mesCours;

	// le premier constructeur
	public Etudiant(String nom, int cursus) {
		this.nom = nom;
		this.cursus = cursus;
		this.mesCours = new Cours[20];
	}

	// le deuxieme constructeur
	public Etudiant(String nom) {
		this(nom, 1);
	}

	// le reste du code

}
```

3. Nous voulons pouvoir ajouter un nouveau cours aux cours d'un étudiant. À ce fin nous voulons écrire une méthode `void ajouterCours(Cours cours)` dans la classe `Etudiant` qui :
   a. Cherche si le cours est compatible avec l'étudiant en question (`this`) et, le cas échéant,
   b. Cherche la première position non-occupée du tableau, pour
   c. Y ajouter le cours.

   - Voici une partie du code de cette méthode. Expliquez ce qu'elle fait.

```java
public void ajouterCours(Cours cours) {
	if (cours.estCompatible(this)) {

		boolean ajoute = false;
		int positionActuelle = 0;

		while (!ajoute && positionActuelle < mesCours.length) {
			if (null == this.mesCours[positionActuelle]) {
				//A COMPLETER
			}
			// A COMPLETER
		}
		if (!ajoute) {
			// A COMPLETER
		}
	}
}
```

   - Complétez ce code.

SOLUTION :

Prenons le code ligne par ligne :

`if (cours.estCompatible(this))` : on cherche si le cours est compatible avec l'étudiant.

Question pour vous : est-ce qu'on aurait pu faire l'inverse et écrire : `if (this.estCompatible(cours))` ? Si vous ne trouvez pas la réponse ou vous avez des doutes là-dessus, demandez-le à moi !

Selon la description de la méthode, le prochain pas serait de parcourir le tableau de cours `mesCours`, trouver la première position vide, et y mettre le cours donné en paramètre. Comment peut-on faire cela ?

Parcourir un tableau de 20 éléments peut être réalisé avec une boucle. Si on utilise une boucle `for`, on risque d'être moins efficaces. Ce qu'on veut c'est d'avoir la possibilité de s'arrêter dès qu'on puisse trouver une place vide dans le tableau où on peut mettre le cours en entrée. Le `boolean ajoute` est initialisé à `false`, et il deviendra `true` lorsqu'on trouve de la place en `mesCours`.

La boucle `while` cherchera donc dans le tableau `mesCours` jusqu'à ce que le `boolean ajoute` devient `true`. C'est ce que fait le code ci-dessous :

```
boolean ajoute = false;
int positionActuelle = 0;

while (!ajoute && positionActuelle < mesCours.length)
```

Dans le constructeur on a instancié le tableau `mesCours` à un tableau de type `Cours[]` de 20 éléments. Par défaut cela assigne à chaque position du tableau la valeur `null`. Il faut donc chercher la première position du tableau qui est égale a `null`.

Attention : `null` est un cas spécial. Normalement si on cherche à comparer deux objets on utilise le mot `equals`. Mais `null` est un cas spécial, on cherche notamment si l'objet existe ou non. Alors là on utilise == , comme pour les valeurs primitives.

```
if (null == this.mesCours[positionActuelle])
```

Qu'est-ce qu'on fait lorsqu'on trouve une position vide ? On assigne la valeur `cours` à la position qu'on a trouvé du tableau mesCours, on arrête les itérations de la boucle (en mettant la valeur ajoute à true). Finalement, on peut optionnellement afficher un message de confirmation d'avoir ajouté le cours. Le code deviendra alors :

```
if (null == this.mesCours[positionActuelle]) {
```

```
                    mesCours[positionActuelle] = cours;
                    ajoute = true;
                    System.out.println("J'ai ajoute le cours aux cours de " +
this.getNom());
                }
```

A chaque itération de la boucle on cherche si à la position actuelle de l'itérateur (donnée par la valeur de `positionActuelle`) le tableau `mesCours` a une disponibilité. Sinon, il faut passer à la prochaine itération. Il faut donc ajouter une mise à jour de l'itérateur. Ce que donne donc comme résultat la prochaine boucle `while` :

```
                while (!ajoute && positionActuelle < mesCours.length) {
                    if (null == this.mesCours[positionActuelle]) {
                        mesCours[positionActuelle] = cours;
                        ajoute = true;
                        System.out.println("J'ai ajoute le cours aux cours de " +
this.getNom());
                    }
                    positionActuelle++;
                }
```

Si on a trouvé déjà une position vide, alors `ajoute` a été mis à `true` et la boucle s'arrête. Mais il y a une autre façon de sortir de la boucle : notamment si l'itérateur arrive à la fin du tableau `mesCours` sans trouver une position disponible. Dans ce cas-ci, `ajoute` reste `false` et on devrait informer l'étudiant que sa collection de cours est complète et donc qu'on n'a pas réussi ajouter le cours.

Ce qui donne la méthode complétée :

```
                if (cours.estCompatible(this)) {

                    boolean ajoute = false;
                    int positionActuelle = 0;

                    while (!ajoute && positionActuelle < mesCours.length) {
                        if (null == this.mesCours[positionActuelle]) {
                            mesCours[positionActuelle] = cours;
                            ajoute = true;
                            System.out.println("J'ai ajoute le cours aux cours
de " + this.getNom());
                        }
                        positionActuelle++;
                    }
                    if (!ajoute) {
                        System.out.println("Cet etudiant a deja choisi le maximum
de cours possibles !");
                    }
                }
```

4. Ecrivez du code (à mettre dans la méthode principale) qui :
    - Crée un cours qui s'appelle `"Java"` de première année, et puis
    - Permet à l'Etudiant appelé `"Jean  Dupont"` d'ajouter ce cours.
    - Qu'est-ce qui se passe si l'étudiant Marine Delafontaine essaie d'ajouter ce cours ?

SOLUTION :

Pour créer le cours appelé Java, on appelle le constructeur de la classe Cours :

```
final Cours java = new Cours("Java", 1);
```

Puis, en utilisant la méthode `ajouterCours` de la classe `Etudiant` on ajoute le cours. Attention : on ajoute l'objet `java` et non pas juste le nom du cours `"Java"`

```
jeanDupont.ajouterCours(java);
```

Si `marineDelafontaine` essaie d'ajouter le cours, lorsqu'on essaie d'utiliser la méthode `ajouterCours`, la méthode `estCompatible` sera appelée implicitement -- et comme `marineDelafontaine` n'est pas compatible avec ce cours, cette méthode retournera la valeur `false`. Pour l'instant on n'a pas mis un message d'erreur au cas où l'étudiant n'est pas compatible avec le cours. Alors l'exécution de la méthode n'aura aucun effet visible, mais le cours ne s'ajoutera pas au tableau de cours de `marineDelafontaine`.