Septembre 24

Harmonisation/Propédeutique POO

Anne-Marie Pinna-Dery & Mireille Blay-Fornarino
Version originale par Anne-Marie Pinna-Dery

Certains éléments des leçons ont été rédigés avec l'aide de chatGPT.

Prérequis : Vous savez tous écrire des fonctions et/ou des procédures qui prennent des arguments et renvoient des valeurs.
Objectifs : Acquérir les premiers principes de la POO que vous approfondirez en cours et les illustrer avec le langage Java, support technique du cours de POO.
Planning :

- Les 2 premières séances ont pour objectif de vous présenter les bases de la POO avec la notion de classes et d'instances.
- Un test sera fait en début de séance 3 pour vérifier le niveau d'acquisition des concepts de base avant de poursuivre.
- Les 2 séances suivantes seront consacrées davantage à la modélisation objet en mettant l'accent sur la création d'une application avec plusieurs classes.

Table des Matières

Étude de cas : Modélisation du fonctionnement de l'harmonisation

Tout au long de votre formation, vous serez amené à résoudre des études de cas, ce qui implique d'aborder des problèmes concrets présentés par des clients. Vous devrez analyser ces cas pour comprendre les besoins, puis proposer et concevoir des solutions qui seront mises en œuvre en utilisant un langage de programmation. Au cours de cette harmonisation, nous suivrons la méthode suivante : nous aborderons ensemble une étude de cas spécifique, en la développant étape par étape.

Dans le cadre de la spécialité informatique de Polytech Nice, les premières semaines sont consacrées à une période préliminaire d'introduction aux cours principaux de la première année nommée *Harmonisation*. Dans la suite, nous vous explicitons une mise en oeuvre possible decette période de cours.
Pendant cette période, des enseignants de l'école sont responsables de cours d'introduction qui sont délivrés aux étudiants inscrits en fonction de leurs connaissances préalables dans la matière.
Les principales données sont les suivantes :

1. Un cours est caractérisé par son intitulé (une chaine de caractères), les étudiants inscrits, un niveau de difficulté noté entre 1 (facile) et 5 (difficile) et un enseignant responsable.
2. Plusieurs cours sont proposés (POO, Réseaux, …) aux étudiants entrants en fonction de leur formation préalable mais l'accès reste ouvert aux étudiants qui le souhaitent sous réserve qu'ils s'inscrivent au cours.
3. Chaque étudiant est défini par son nom, prénom et son année de naissance.
4. Un enseignant est également décrit par son nom et prénom mais aussi par le numéro de son bureau par exemple *A 321*. Les enseignants qui n'ont pas de bureau ont un numéro par défaut qui est *A 0*
5. D'autres informations peuvent être ajoutées pour gérer au mieux les semaines d'introduction comme une note d'auto-évaluation correspondant à la progression de l'étudiant pendant les semaines.

Ces informations vont nous permettre par un programme de :

1. Obtenir la liste des étudiants inscrits à un cours ;
2. Obtenir la moyenne d'âge des étudiants d'un cours ;
3. Obtenir la liste des cours suivis par un étudiant ;
4. Obtenir le nombre et la moyenne des niveaux de difficulté des cours suivis par un étudiant.

Il est également important de proposer une solution fiable qui permet de travailler avec des informations correctes et cohérentes sur cette période en caractérisant la liste des enseignants impliqués, des cours et des étudiants afin de par exemple

5. Obtenir la liste des cours proposés en Harmonisation ;
6. Vérifier qu'un étudiant inscrit à un cours est bien référencé par ce cours ;
7. Vérifier que tout enseignant a au moins un cours dont il est responsable.

Nous pouvons facilement voir que ce type de problème relativement précis pourrait être repensé ou étendu pour proposer des outils de gestion d'années à Polytech.

PARTIE 1- Modéliser une classe simple et l'implémenter en Java

I. La notion de Classe et de variables d'instance

Leçon 1

Face à un nouveau problème, on réfléchit en identifiant les principaux concepts manipulés dans ce programme. Pour cela on va avoir une réflexion que l'on qualifie **d'orientée objet.** On ne manipule plus que des **objets** qui contiennent des données et qui savent répondre à des questions en exécutant un code très proche de ce que vous savez faire quand vous écrivez des fonctions ou des procédures. Chaque concept correspond à une **classe**.

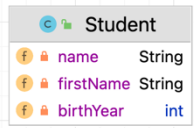
Illustration avec la notion d'*étudiant :*
« Chaque étudiant est défini par un nom, prénom et son année de naissance »

L'étudiant est un "concept important à modéliser" car nous allons devoir la manipuler dans le programme.
Chaque classe a plusieurs **instances** : il y a potentiellement autant d'instances différentes d'étudiants que d'étudiants inscrits en SI3. On parle pour les instances aussi **d'objets**.
Quand on décrit une classe, on peut préciser les données qui permettent de distinguer une instance d'une autre. On appelle ces données des **variables d'instances.**
Par exemple pour un étudiant, il y a au moins 3 variables d'instances :

- le nom (une chaîne de caractères, par exemple "Haddock"),
- un prénom (une chaîne de caractères, par exemple "Archibald"),
- son année de naissance (un entier par exemple 2002)

Le code suivant vous présente une première définition possible de cette classe, la classe **Student**, avec sa visualisation dite UML à gauche. Les variables d'instances *name* et *firstName* sont « **private** » c'est-à-dire qu'elles ne sont accessibles qu'à l'intérieur de la classe. La classe est définie dans **un fichier** qui porte son nom et se termine par .java, ici ***Student.java***

Notez que le **nom d'une classe commence par une majuscule** en java.
Notez que le **nom des variables d'instances commence par une minuscule** et si c'est un nom composé, on utilise une majuscule pour distinguer les 2 mots (firstName).




```
public class Student {
    private String name;
    private int birthYear;
}
```

Questions en groupe

1. Quelles sont les variables d'instance de la classe *Student* ?
2. Que représentent-elles ? Quel est leur type ?
3. Quelle variable a été oubliée ? Quel est son type ?

Le mot clé private permet de protéger les données qui ne seront visibles que dans la classe. On appelle cela le **principe d'encapsulation**.

Question individuelle & Codage

4. Un **enseignant** est décrit par un nom, un prénom et la référence à son bureau par exemple A 321.
   Créez la classe **Teacher** (donc bien dans un nouveau fichier qui s'appelle *Teacher.java*) et définir ses variables d'instances.

II. Constructeurs (V0)

Leçon 2

Une instance peut être vue comme une boite noire (une structure en mémoire) qui contient ses données.
Pour créer des instances à partir d'une classe (créer les espaces mémoire contenant les données), il faut définir des **constructeurs**.
En l'absence de constructeur explicite, Il existe un constructeur par défaut qui alloue l'espace mémoire, mais avec des valeurs initiales des données dépendantes du type, voire inexistante. Pour pouvoir initialiser une instance à sa création, il vaut mieux implémenter ses propres constructeurs, par exemple Student(String aName, String firstName, int aBirthYear) est un constructeur qui nous permet de créer un étudiant en précisant son nom, son prénom et son année de naissance. Dans le code ci-dessous «**this**» désigne le nouvel objet.

```
public Student(String aName, String firstName, int aBirthYear) {
    name = aName;
    this.firstName = firstName;
}
```

public Student(String aName, String firstName, int aBirthYear) est la **signature** du constructeur qui correspond au nom de la classe suivi d'autant de paramètres que nécessaire pour initialiser les variables d'instances. La suite entre { et } est le **corps de la méthode**, ici un constructeur, ensemble des instructions qui vont s'exécuter à l'appel du constructeur.

Questions en groupe & Codage individuel

5. Que fait ce constructeur ? Nous avons volontairement défini deux affectations des variables, pourquoi utilise-t-on *this* à la 2e et pas à la 1e ?
6. Quelle instruction a été oubliée ?

Question individuelle & Codage

7. On souhaite pouvoir créer un Enseignant à partir de son nom et son prénom, uniquement.
   Ajouter le constructeur correspondant dans la classe **Teacher**

III. Méthodes d'instances, focus sur les accesseurs

Leçon 3

Quand on décrit une classe, on peut préciser les comportements des instances. On appelle l'expression de ces comportements, **méthodes d'instances.** On dit qu'on invoque une méthode sur une instance lorsqu'on demande à une instance d'exécuter le corps de la méthode à partir de ses propres données.
Une **instance** doit être vue comme une boite noire (une structure en mémoire) qui contient ses données. Pour pouvoir accéder aux données en lecture (resp. en écriture) de l'extérieur de la classe, il faut utiliser une méthode spécifique qu'on appelle **accesseur en lecture,** par exemple String **get**Name(), (resp **en écriture ou mutator,** par exemple void **set**Name(String name)).
On peut également définir une méthode d'instances qui va calculer l'âge d'un étudiant par rapport à une année donnée. Les codes suivants présentent les différentes implémentations associées.

Remarquez l'utilisation de ***this*** dans setName pour bien distinguer la variable d'instance *name* de l'objet (*this.name*) du paramétre *name*.

La convention de nommage veut que les accesseurs en lecture aient un nom préfixé par **get** et les accesseurs en écriture par **set.**

void est utilisé quand la méthode ne renvoie pas de valeur (c'est une procédure et non une fonction).
return désigne la valeur renvoyée par une fonction. Bien sûr, la valeur doit être du même type que le type dans la signature de la méthode. Par exemple, dans getName(), email doit être de type String conformément à **la signature de l'accesseur.**

Attributs et méthodes sont définis dans leur classe qui commence par exemple par { et se termine par }

public class Student {

//variables, méthodes et constructeurs sans ordre prédéfini et se termine par }

```
public class Student {
 //…. Variables d'instances
 //…. Constructeurs

  public String getName() {
    return name;
  }

  public int getBirthYear() {
    return birthYear;
  }

  public void setName(String name) {
    this.name = name;
  }
  public void setBirthYear(int birthYear) {
    this.birthYear = birthYear;
  }
}
```

On définit également la méthode suivante :

```
public int ageIn(int year) {
    return year - birthYear;
}
```

Questions en groupe & Codage individuel

8. Quelles sont les variables d'instances utilisées dans ce code ? Quels sont les paramètres des méthodes ?
9. Ajouter les accesseurs en lecture et écriture au prénom.
10. Que fait la méthode *ageIn* ? que prend-t-elle en paramètre ?

Question individuelle & Codage

11. On souhaite pouvoir accéder en lecture au *nom* de l'enseignant, à son *prénom* et au nom du bureau.

12. On souhaite pouvoir associer à un enseignant un nom de bureau sur la base suivante : *nom du bâtiment* et *numéro*

```
Par exemple, office = building + " " + number;
```

Le + permet de **concaténer des chaines de caractères**. Notons qu'ici number est un entier et que l'opérateur + fait une conversion automatique en chaine de caractères.

13. Modifier la classe Teacher pour correspondre à cette spécification.
    Pour vous guider, combien de paramètres sont nécessaires pour enregistrer le nouveau nom d'un bureau ?

IV. Instances et appels aux constructeurs

Leçon 4

En appliquant le premier constructeur, par un **new**, on peut **créer une instance** « etudiant » de la classe *Student* en précisant son nom, son prénom et son année de naissance :

```
Student etudiant = new Student("Doe", "Jane", 2000);
```

Le . permet **d'envoyer un message** à une instance (par exemple, etudiant instance de *Student*) pour demander d'exécuter une méthode publique définie dans sa classe (par exemple *getName*()).
Il est ainsi possible d'obtenir son nom : etudiant.getName(), de **l'affecter** (stocker sa valeur) dans une variable, par exemple la variable *sonNom* de type String

```
String sonNom = etudiant.getName(). //envoi du message getName() à
```

l'instance ***s*** de *Student* et affectation de la valeur de retour à la variable *sonNom* de type String.

Questions en groupe & Codage individuel

14. Étudiez le code suivant qui se trouve dans la classe *Student (cf. ci-dessous)*
    i. Que fait le code en ligne 46 ?

System.out.println vous permet d'afficher la chaîne en paramètre sur votre écran.

ii. Que fait le code en ligne 48 ?
iii. Qu'affichera le code de la ligne 49 ? Nous vérifierons après.
iv. Que fait le code en ligne 50 ? Qu'affichera-t-il ? Nous vérifierons après.

```
44  public static void main(String[] args) {
45      Student s = new Student( herName: "Doe",  firstName: "Jane",  dateOfBirth: 2000);
46      System.out.println(s.getName());
47      System.out.println(s.getFirstName());
48      int age = s.ageIn( year: 2021);
49      System.out.println(age);
50      System.out.println(s.ageIn( year: 2021));
51
```

La **méthode *main*** est une méthode principale d'une classe Java qui est un point d'entrée où l'exécution du programme commence.
La méthode main doit être définie avec **une signature spécifique** :
public static void main(String[] args)

- public : La méthode main est accessible de l'extérieur de la classe.
- static : La méthode main est une méthode de classe, ce qui signifie qu'elle peut être appelée sans instancier la classe.
- void : La méthode main ne retourne pas de valeur.
- main : C'est le nom de la méthode.
- String[] args : Les arguments passés au programme sont reçus sous forme de tableau de chaînes de caractères.

Sa signature spécifique la rend identifiable pour le système d'exécution Java, qui cherche cette méthode pour lancer le programme.

15. Ajoutez le code précédent à votre classe *Student*
    public static void main(String[] args) { …

16. Exécutez le code et vérifier la cohérence des résultats obtenus.
17. Définissez une nouvelle instance de Student qui a pour nom « Doe », pour prénom « John » et est née en 2001.
18. Vérifiez vos résultats. Plus tard, vous apprendrez à tester vos codes. Ici, nous "vérifions" simplement en regardant que les codes se comportent bien comme nous l'attendons.
19. Que se passe-t-il si on exécute les instructions suivantes :

```
Student s = new Student("Doe", "Jane", 2000);
System.out.println(s);
System.out.println(s.getName());
s.setName("Haddock");
System.out.println(s.getName());
```

20. Que se passe-t-il si on exécute les instructions suivantes :

```
Student s1 = new Student("Doe", "John", 2000);
Student s2 = new Student("Doe", "Jane", 2000);
s1 = s2;
System.out.println(s1.getFirstName());
s1.setName("Schmitt");
System.out.println(s2.getName());
```

21. Que se passe-t-il si on demande l'exécution des instructions suivantes :
    a) s.setName(Dupo) ;
    b) s.getName(12) ;
    c) s.getNom();

Question individuelle & Codage

22. Ajoutez un « main » dans votre classe **Teacher** et créez des instances de Teacher.
23. Vérifiez que vous pouvez bien accéder au nom de l'enseignant.
24. Faites appel à la méthode qui permet de modifier le nom de son bureau et vérifier le résultat obtenu.

V. Visibilités

Leçon 5

Vous avez dû remarquer la présence des mots clefs **public** et **private**.

Par exemple la déclaration de la classe *Student* est **publique** (e.g., `public class Student`) c'est-à-dire que n'importe quelle partie du programme peut y accéder. Nous avons fait de même pour toutes les méthodes que nous avons écrites (e.g., `public int ageIn(int year)`). Inversement toutes les variables d'instances sont déclarées comme **privées** dans nos codes (e.g., `private String name;`). On ne peut y accéder qu'au travers des méthodes publiques définies par la classe, y compris les accesseurs. Il est vivement recommandé de toujours protéger ainsi le contenu de vos instances, on appelle cela **le principe d'encapsulation.**

Questions en groupe & Codage individuel

25. Étudiez le code ci-après

a) Qu'est-ce ? Dans quelle classe est-il défini ?
b) En ligne 11, le code isValidBirthYear est équivalent à **this**.isValidBirthYear
    i. Dans quelle classe la méthode *isValidBirthYear* doit-elle être définie ?
    ii. Quelle est sa signature ? Est-elle conforme à sa définition en ligne 18 ?
c) La méthode en ligne 18 est privée.
    i. Pouvez-vous expliquer pourquoi elle est privée ?
    ii. Que fait-elle ?
    iii. Identifiez bien le return
d) Que fait le code en ligne 11 ?
e) Que fait le code des lignes 11 à 16 ?
f) Quelle valeur est affectée lorsque l'année de naissance est considérée comme erronée ?

26. Implémentez ce code, adaptez-le pour que les dates soient plus réalistes (avoir plus de 15 ans aujourd'hui semblerait une bonne limite 😊) et vérifiez qu'il fonctionne.

```java
public Student(String aName, String firstName, int dateOfBirth) {
    name = aName;
    this.firstName = firstName;
    if (isValidBirthYear(dateOfBirth)) {
        this.birthYear = dateOfBirth;
    }
    else {
        this.birthYear = 1901;
    }
}
1 usage  new *
private boolean isValidBirthYear(int dateOfBirth) {
    return (dateOfBirth >= 1900) && (dateOfBirth <= 2021);
}
```

Question individuelle & Codage

27. Modifiez le code de la méthode suivante pour ne modifier le nom du bureau que si le numéro est supérieur à 0 et est strictement inférieur à 500.

```java
public void buildOfficeName(String building, int number) {
        this.office = building + " " + number;
    }
}
```

28. On considère à présent que le nom du bâtiment doit être **une** lettre comprise entre A et F. Ces contraintes pourront évoluer et on ne souhaite pas qu'il soit possible d'y accéder en dehors de la classe. Proposez une implémentation d'une méthode privée qui vérifie que les informations données sont correctes avant de modifier le nom du bureau.

Le code suivant est là pour vous aider.

```java
if ((building.length() != 1) || (building.charAt(0) < 'A') ||
    (building.charAt(0) > 'F')) {
    return false;
}
```

Les String commencent à l'indice 0 comme tous les tableaux en java.
L'appel à la méthode **length()** retourne la taille de la chaine de caractères.
L'appel à la méthode **charAt()** retourne le caractère qui se trouve en position en paramètre (ici 0) dans la chaine de caractères.
length et *charAt* sont des méthodes publiques de la classe **String** prédéfinies dans Java.
Attention un **caractère** tel que 'A' est noté entre simples côtes alors qu'une **chaine de caractères** comme "Doe" est notée entre double côtes.
Par exemple la String "Doe" est de longueur 3 et le caractère en position 0 est 'D', tandis que celui en position 2 est 'e'.

!= signifie **différent**.
|| exprime un **ou**. L'opérande à droite n'est évalué que si celle de gauche est fausse, donc dans l'exemple ci-dessous que si la longueur est égale à 1.

VI. Visibilité en action

Leçon 6

Nous poursuivons sur la visibilité des variables d'instances « privée ».

Si nous ajoutons le code suivant à la classe *Student*, il compile parce que, dans la classe qui les définit, les propriétés, variables ou méthodes, même privées sont accessibles.

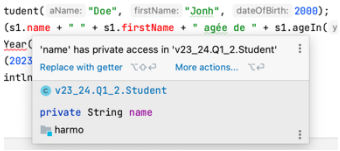
```
public static void main(String[] args) {
…
Student s1 = new Student("Doe", "John", 2000);
System.out.println(s1.name + " " + s1.firstName + " agée de " +
s1.ageIn(2023));
if (s.isValidBirthYear(2023))
    s.setBirthYear(2023);
else System.out.println("too young!!");

}
```

Nous définissons à présent une nouvelle classe ainsi :

```
3  public class TestStudents {
       new *
4      public static void main(String[] args) {
5
6          Student s = new Student( aName: "Doe", firstName: "Jane", dateOfBirth: 2000);
7          System.out.println(s);
8          System.out.println(s.getName());
9          s.setName("Haddock");
10         System.out.println(s.getName());
11
12         Student s1 = new Student( aName: "Doe", firstName: "Jonh", dateOfBirth: 2000);
13         System.out.println(s1.name + " " + s1.firstName + " agée de " + s1.ageIn( year: 2023));
14         if (s.isValidBirthYear( dateOfBirth: 2023))
15             s.setBirthYear(2023);
16         else System.out.println("too young!!");
17     }
```

Les lignes 13 et 14 sont en erreur. En effet, il n'est pas possible d'accéder aux variables d'instance privée de la classe *Student* depuis la classe *TestStudents* et de même pour la méthode *isValidBirthYear*.



Questions en groupe & Codage individuel

29. Implémentez ces codes, vérifiez et comprenez. Donc créez une classe *TestStudents*
    a) Bien entendu vous la créez dans le fichier adéquate et vous respectez les majuscules dans le nom de la classe (voir leçon 1, si besoin).
30. Comment pouvez-vous récupérer le nom d'un étudiant depuis la classe *TestStudents*?
31. Pensez-vous qu'il est logique de tester la valeur de l'année en dehors de la classe *Student* elle-même ? Modifier le code de la classe Student pour n'affecter une nouvelle date de naissance que si elle respecte les contraintes énoncées.

Question individuelle & Codage

32. Créez la classe *TeachersTests* et vérifiez que le *main (donc la méthode main)* fonctionne.

VII. Cascades de constructeurs

Leçon 7

Dans le contexte de la création de plusieurs constructeurs au sein d'une classe en Java, une pratique courante consiste à les faire s'appeler par **this(…).** Cette approche permet d'éviter la duplication de code, tout en simplifiant la création d'objets par le biais de diverses combinaisons d'arguments. De manière concrète, par exemple, un constructeur plus général définit l'initialisation des variables. Un autre constructeur, moins spécifique l'invoque avec des valeurs par défaut. Cette démarche « en cascade » favorise non seulement la clarté du code, mais aussi sa maintenabilité ; en modifiant le constructeur le plus général, tous les constructeurs plus spécifiques sont automatiquement mis à jour.

Nous souhaitons pouvoir créer des étudiants uniquement avec un nom et un prénom et mettre comme année de naissance dans ce cas 1901.
Nous définissons donc un nouveau constructeur ainsi :

```
public Student(String aName, String firstName) {
    this(aName, firstName, 1901);
}
```

Ce nouveau constructeur qui permet de créer une instance de *Student* sans connaitre son année de naissance et en faisant appel par **this(…) en première ligne du constructeur** au constructeur que vous avez défini précédemment, par exemple :

```
public Student(String aName, String firstName, int dateOfBirth) {
    name = aName;
    this.firstName = firstName;
    if (isValidBirthYear(dateOfBirth)) {
        this.birthYear = dateOfBirth;
    }
    else {
        this.birthYear = 1901;
    }
}
```

Questions en groupe & Codage individuel

33. Introduisez une nouvelle variable d'instance dans la classe *Student*
private String email;

34. On ne souhaite pas pouvoir modifier le contenu de l'email depuis l'extérieur. L'email est affecté automatiquement à la création de l'instance comme la conjonction du nom et du prénom et de l'adresse de l'université.
    a. Ajoutez uniquement un accesseur en lecture ;
    b. Modifiez la construction d'une instance de *Student* pour affecter la variable *email* par

```
this.email = name + "." + firstName + "@etu.univ-cotedazur.fr";
```

    c. Quel constructeur choisissez-vous de modifier et pourquoi ?

35. Actuellement la valeur de *email* est donc construite à la construction de l'instance et la valeur obtenue est stockée dans la variable d'instance *email*.
    a) Mais que se passe-t-il si vous modifiez le nom de l'étudiant ?
    b) Modifiez votre code, pour que toute modification du nom, respectivement du prénom, mette à jour la valeur de la variable d'instance représentant *l'email*. (Tips : comment est-il possible de modifier le nom ou le prénom d'un étudiant en dehors de la classe ? En quoi, l'encapsulation nous aide ici ?)
    c) Avez-vous du code dupliqué ? Quelles améliorations proposez-vous pour éviter la duplication de code ?
    d) Une autre solution aurait été de définir, comme nous l'avons fait pour l'âge une unique méthode qui construit *l'email* à la demande. Nous choisissons ici, pour des raisons pédagogiques, de garder la variable d'instance *email*.
    Quels sont d'après vous les avantages et inconvénients de chaque solution ?

Question individuelle & Codage

36. Ajoutez la variable d'instance *email* à Teacher dans les mêmes conditions que précédemment, sauf qu'elle se termine par : "@univ-cotedazur.fr"

37. Ajoutez un constructeur qui prend en paramètre le nom, le prénom et le nom du bureau, si ce n'est pas déjà fait et qu'il mette à jour la nouvelle variable d'instance *email*.
38. Mettez à jour le constructeur de la classe *Teacher* pour qu'il affecte par défaut le nom du bureau à « A 0 » et tienne compte la nouvelle variable d'instance *email*, si ce n'est pas déjà fait 😉.

VIII. toString

Leçon 8

Avez-vous remarqué ce qui est affiché si vous exécutez le code suivant ?

```
Student s = new Student("Doe", "Jane", 2000);
System.out.println(s);
```

Vous devez avoir un affichage ressemblant à :
`Student@1f32e575`, ce qui n'est pas très utile. Par défaut, lorsque l'on demande à afficher un objet, celui s'affiche sous la forme du nom de la classe et une référence mémoire.

System.out.println appelle **automatiquement** la méthode toString() ainsi System.out.println(etudiant) est équivalent à System.out.println(etudiant.toString())

Il est possible de définir la méthode **toString** dans chaque classe, pour former une chaine de caractères (string) contenant les informations spécifiques à l'objet. La méthode **toString** en Java permet de convertir une instance en une représentation sous forme de chaîne de caractères. Cela

facilite l'affichage et le débogage en fournissant une description significative de l'objet. Par défaut, la méthode renvoie le nom de la classe et une référence mémoire, mais elle peut être redéfinie pour afficher des informations spécifiques à l'objet.

Par exemple, soit la méthode suivante définie dans la classe Student

```
public String toString() {
    return "Student : \n\t" + name + "\t " + firstName + ", \n\t" + email +
", \n\t" + birthYear + "\n";
}
```

Nous obtiendrons à présent comme affichage :

Student :
Doe Jane,
Doe.Jane@etu.univ-cotedazur,
2000

Notez que

- **\n** permet d'aller à la ligne et ***\t*** d'ajouter une tabulation
- **+** avec une variable de type int fait automatiquement la conversion de type de int à String.
- **\'** permet d'ajouter un ' dans une String par exemple :
  "name='" + name + "\'" correspond à name='Doe'

Questions en groupe & Codage individuel

39. Ajoutez la méthode *toString* à la classe *Student*
40. Testez-la par des appels à System.out.println avec en paramètre des instances de Student.
41. Adaptez l'affichage à votre convenance.

Question individuelle & Codage

42. Définissez la méthode toString de la classe Teacher.
43. Voici une proposition d'affichage, mais vous pouvez choisir celle qui vous convient :

Teacher: name='Doe',firstName='John',email='Doe.John@univ-cotedazur.fr'., office='A 0'

IX. Constantes

Leçon 9

Avez-vous remarqué qu'à plusieurs reprises nous avons utilisé une même valeur, par exemple « A 0 » pour donner un nom par défaut à un bureau, 1901 comme date de naissance par défaut ?

Il est préférable lorsque l'on manipule des valeurs qui ne seront jamais modifiées de les définir comme des **constantes**.

```
static final String DEFAULT_OFFICE = "A 0";
```

En ajoutant **static** à la déclaration de la variable nous exprimons qu'elle sera partagée par toutes les instances de la classe Teacher.

En ajoutant **final** à la déclaration de la variable nous exprimons qu'elle ne pourra pas être modifiée dans le code.

Utilisation de la constante dans le constructeur de Teacher :

```
public Teacher(String name, String firstName) {
    this.name = name;
    this.firstName = firstName;
    this.office = DEFAULT_OFFICE;
    buildEmail();
}
```

Questions en groupe & Codage individuel

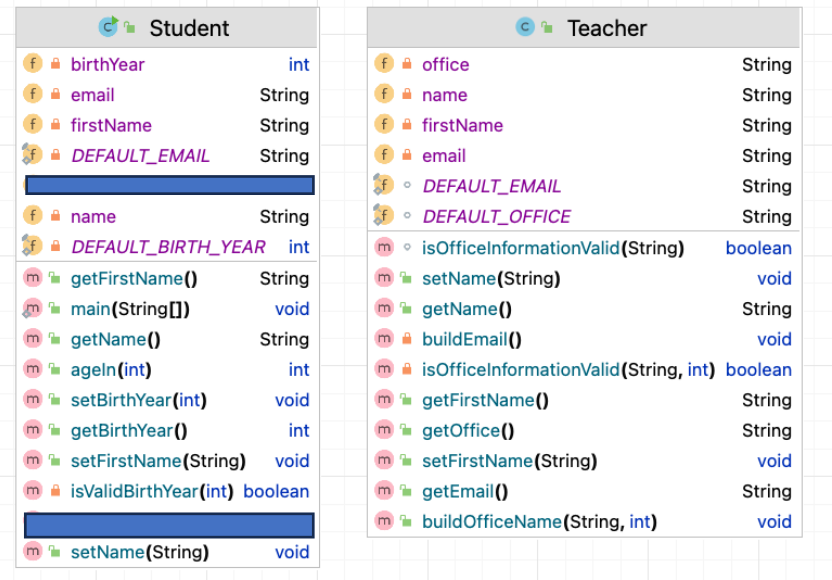
44. Mettez à jour la classe Teacher avec la déclaration de cette nouvelle constante et remplacer dans tout votre code les occurrences de « A 0 »
45. Ajoutez une constante dans la classe *Student* pour représenter l'année de naissance par défaut dont la valeur est 1901 et remplacer toutes les occurrences de 1901.
46. Quelle autre constante pourrait être définie ?

Question individuelle & Codage

47. Vérifiez que vos codes s'exécutent « bien ».

X. Bilan

Vous devriez avoir une classe *Student* et une classe Teacher définies par les éléments suivants.



Nous avons ajouté la méthode suivante pour vérifier dans le constructeur de Teacher que le nom du bureau est valide.

```
boolean isOfficeInformationValid(String officeName) {
    if (officeName.length() <3) {
        return false;
    }
    if ((officeName.charAt(0) < 'A') || (officeName.charAt(0) > 'F')) {
        return false;
    }
    if (officeName.charAt(1) != ' ') {
        return false;
    }

    int number = 0;
    String numberPart = officeName.substring(2,officeName.length() - 1);
    try {
        number = Integer.parseInt(numberPart);
    }
    catch (NumberFormatException e) {
        return false;
    }
    return (number >= 0) || (number <= 500);
}
```

Questions en groupe & Codage individuel

1. Comprenez le code de la méthode qui a été ajoutée et ajoutez-le à votre code en vérifiant dans le constructeur adéquat que le nom du bureau est bien formé.

PARTIE 2 – Relations entre classes

I. Des variables dont le type est une classe

Leçon 10

En POO quasiment tous les types sont des classes. Seuls les **types primitifs** ne sont pas des classes, par exemples : *int, float, double, char, boolean.*

Toute variable peut être typée soit par un type primitif, soit par une classe prédéfinie ou définie par vous.

Reprenons l'énoncé et la définition de Cours :

Un cours est caractérisé par son intitulé (une chaine de caractères), les étudiants inscrits, un niveau de difficulté noté entre 1 (facile) et 5 (difficile) et un enseignant responsable.

Dans cette définition vous pouvez remarquer que :

- *L'intitulé* correspond au type primitif String
- l*es étudiants* correspond à une collection de Student, donc la classe que vous aviez définie
- et *l'enseignant* correspond à une instance de la classe Teacher que vous avez déjà définie.

Voici un début d'implémentation en java possible de la classe ***Cours*** et sa représentation graphique[1].

```
import java.util.ArrayList;
import java.util.List;

public class Course {
    private String title;
    private List<Student> students;
    private int difficulty;
    private Teacher professor;

    public Course(String title, int difficulty, Teacher professor) {
        this.title = title;
        this.difficulty = difficulty;
        this.professor = professor;
        this.students = new ArrayList<>();
    }

    public void enroll(Student student) {
        students.add(student);
    }
    public List<Student> getStudents() {
        return students;
    }

    public static void main(String[] args){
        Course c1 = new Course("Java", 3, new Teacher("Lovelace", "Ada"));
        Course c2 = new Course("C++", 4, new Teacher("Turing", "Alan"));
        Student s1 = new Student("Dupont", "Paul", 2000);
        Student s2 = new Student("Durand", "Benedicte", 2001);
        Student s3 = new Student("Smith", "John");
        c1.enroll(s1);
        c1.enroll(s2);
        c2.enroll(s2);
        c2.enroll(s3);
        System.out.println(c1);
```

[1] La représentation graphique est un peu plus complète que le code qui vous est donné.

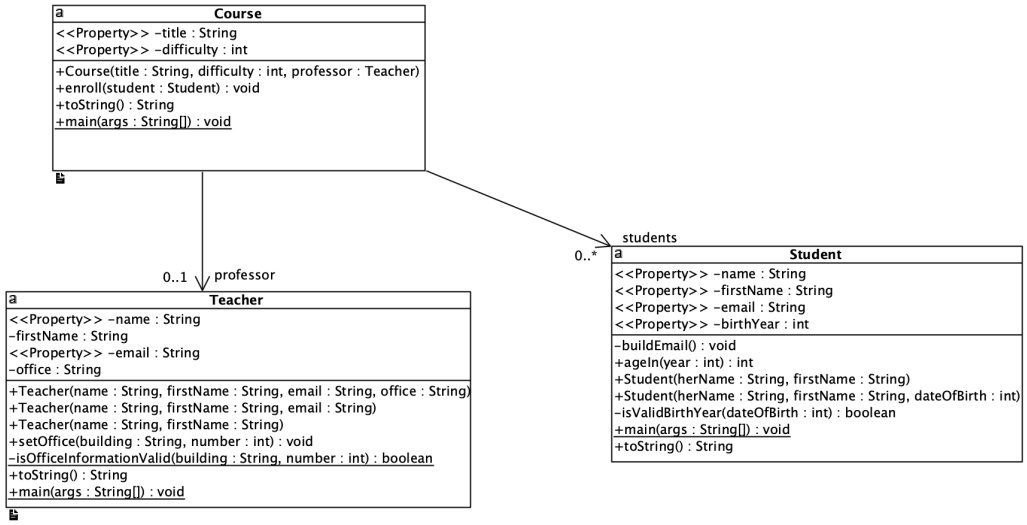
```
        System.out.println(c2);
        System.out.println("Students in java course : ------- \n" +
c1.getStudents());
        System.out.println("Students in C++ course : ------- \n" +
c2.getStudents());
    } }
```

Dans le code précédent, nous utilisons la notion de liste (List et **ArrayList**) des étudiants associés à un cours et nous utilisons la méthode **add** pour ajouter un élément à la liste. Regardez la spécification de ces classes et remarquez en particulier que nous ajoutons les éléments en fin de la liste.

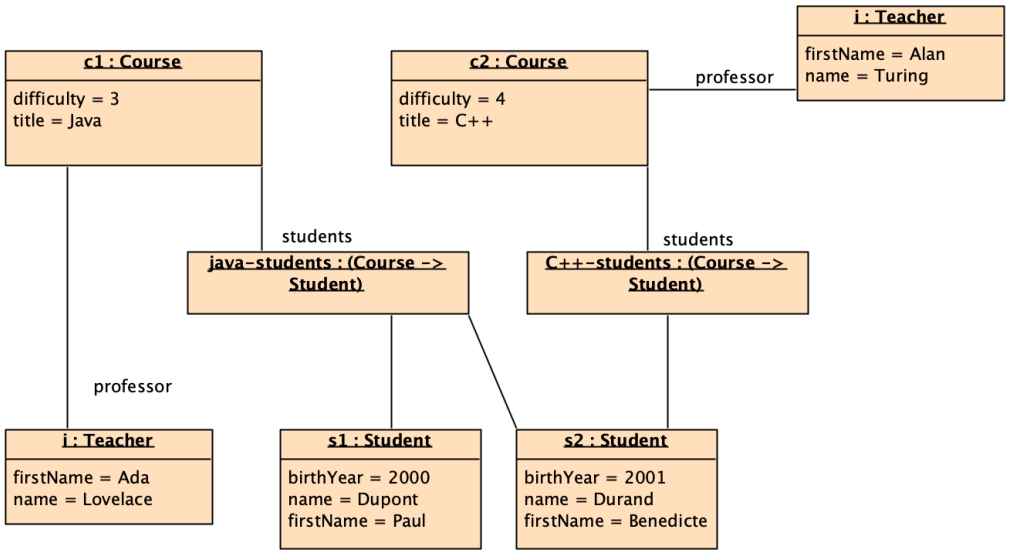
Si vous voulez en savoir plus sur les classes **Java existantes** allez consulter l'API Java : https://docs.oracle.com/en/java/javase/21/docs/api/
En particulier vous avez déjà vu la classe **String** dont vous avez utilisé différentes méthodes telles que : *charAt*.

Vous pouvez, vous aussi, définir l'API des classes que vous créez en écrivant la javadoc (cf. https://www.oracle.com/fr/technical-resources/articles/java/javadoc-tool.html).

Ci-dessous nous donnons une représentation graphique de ce code. Lorsque le type d'une variable d'instance est un type défini par votre application nous le représentons ci-dessous par une flèche entre les classes dont la fin vous donne le nom de la variable d'instance.



Le diagramme suivant visualise partiellement (c'est long à faire) les objets qui ont été créés dans le « main » et leurs relations.



Variable d'instance versus Attribut, Champs, fields : Une variable d'instance est une variable déclarée au niveau de la classe et qui conserve son état propre à chaque instance (objet) créée à partir de cette classe. Chaque objet a sa propre copie de cette variable, ce qui lui permet de stocker des données uniques. C'est ainsi qu'elles ont été introduites au début de cet énoncé. **Attribut** est un terme plus général qui fait référence aux propriétés ou aux caractéristiques d'une entité[2]. Dans le contexte de la programmation orientée objet, un attribut est souvent implémenté à l'aide de variables d'instance. C'est une manière de modéliser les données et les propriétés des objets. On utilise donc souvent le terme d'attribut à la place de variable d'instance mais dans un contexte objet, il s'agit de la même chose. Certains parlent aussi de **champs** et utilise le terme de **field**.

Questions en groupe & Codage individuel

1. Comprenez l'ensemble des codes donnés et en particulier
   i. la différence entre List et ArrayList
   ii. le code d'initialisation de la liste des étudiants
   iii. le code d'ajout un étudiant dans la liste
2. Implémentez en comprenant la classe *Course* et en ajoutant les accesseurs manquants.
3. Modifiez votre code pour vérifier que le coefficient de difficulté est compris en 1 et 5.
4. Est-ce que le *System.out.println* d'un cours affiche ce que vous souhaitez ou pas ? Corrigez votre code si besoin.

[2] Une **entité** fait référence à un objet particulier, une instance unique d'une classe. Dans la programmation orientée objet, une classe sert de modèle ou de plan pour créer des objets, mais chaque objet créé à partir de cette classe est une entité distincte. Cette entité possède ses propres attributs (ou variables d'instance), qui conservent des données spécifiques à cet objet.

Question individuelle & Codage

5. L'harmonisation fait appel à des enseignants et des étudiants. Elle est composée de cours.
Implémentez la classe *Harmo* pour tenir compte de cette spécification.

Vous pouvez choisir de représenter l'ensemble des cours délivrés en harmonisation, non par une liste comme nous l'avons fait précédemment mais par un ensemble (**Set** et **HashSet**). Analysez la spécification de ces deux classes et remarquez qu'elle ne préserve pas l'ordre d'ajouts des éléments dans l'ensemble, c'est juste un « ensemble » 😉

Voici des exemples de code :

```
private Set<Course> courses = new HashSet<>();//Déclaration d'un ensemble de cours

public void addCourse(Course c) {
    courses.add(c);
}
public Set<Course> getCourses() {
    return courses;
}
```

On résume :

En Java, **List** et **Set** sont des interfaces qui représentent des **collections**, c'est-à-dire des groupes d'objets. Ces interfaces ne définissent pas comment les éléments sont stockés ou gérés, mais elles spécifient les opérations qu'une collection peut effectuer, comme ajouter, supprimer ou accéder à des éléments.
List est une collection ordonnée où chaque élément a une position spécifique, et les éléments peuvent être dupliqués. Les classes qui implémentent **List** incluent par exemple **ArrayList** et **LinkedList**, chacune ayant ses propres caractéristiques en termes de performance et de structure de données.
Set, quant à lui, est une collection qui ne permet pas les doublons, c'est-à-dire que chaque élément est unique. **HashSet** et **TreeSet** sont des exemples de classes qui implémentent **Set**, avec des différences dans l'ordre de stockage et la vitesse d'accès.
Ainsi, en utilisant **List** ou **Set**, vous pouvez choisir l'implémentation qui convient le mieux à vos besoins tout en respectant les mêmes règles d'utilisation définies par l'interface.

II. Interactions entre classes

Leçon 11

Nous avons abordé pour l'instant la construction des relations entre classes dans la déclaration des variables d'instances et avons permis d'ajouter des instances dans un ensemble et une liste, par exemple, des étudiants à un cours, des cours à l'harmonisation, etc.

Nous nous intéressons ici à la complexité de ces relations en gérant les **relations inverses**, un étudiant est inscrit à une liste de cours, donc on ajoute à l'étudiant la liste des cours auxquels il est inscrit. Cela suppose donc que non seulement nous ajoutions la variable *courses* à l'étudiant mais qu'également nous décidions de quand l'affecter.
Faisons le choix suivant : un étudiant s1 choisit un cours c1 revient à appeler s1.enrollsIn(c1) qui déclenche un appel à c1.enroll(s1). *On peut donc considérer que l'étudiant choisit un cours et que le cours enregistre l'étudiant.*

Nous aurions pu faire un choix inverse ; si pour qu'un étudiant puisse s'inscrire à un cours il faut d'abord que sa demande soit acceptée nous aurions pu avoir : c1.enroll(s1) qui n'appelle s1.enrollsIn(c1) que si la demande est valide.

Questions en groupe & Codage individuel

6. Implémenter dans la classe *Student* la méthode *enrollsIn(Course c)* avec tout ce qui est nécessaire, et testez là.
7. Analysez le code donné ci-dessus qui retourne le dernier étudiant inscrit à un cours. Ce code se trouve dans la classe *Course*

```
public Student getLastStudentEnrolled() {
    return students.get(students.size() - 1);
}
```

8. Pourriez-vous définir une méthode dans la classe *Student* qui renvoie le dernier cours auquel un étudiant s'est inscrit ? Regardez la spécification associée aux classes utilisées. **Vous voyez qu'elles ne préservent pas l'ordre. Le choix de List ou de Set a donc un impact important.**

Question individuelle & Codage

Un enseignant connait la liste des cours dont il est responsable. Évidemment un cours dont il est responsable doit avoir pour responsable lui-même.

9. Implémenter cette spécification en tenant bien compte de la bi-directionnalité entre le responsable de cours et la liste des cours dont un enseignant est responsable : un enseignant choisit à un cours c1 dont il prend la responsabilité.
10. Attention que se passe-t-il si aucun étudiant n'est inscrit au cours ? Dans ce cas, renvoyez null.

III. Boucles

Leçon 12

Nous allons à présent parcourir les ensembles ou les listes que vous avez définis.

En Java, la boucle `for` est couramment utilisée pour itérer sur les éléments d'une liste ou d'une collection. Il existe plusieurs variantes de la boucle `for` en Java, mais deux formes courantes pour parcourir une liste sont :
Boucle `for` classique qui parcourt une liste avec des indices.
Boucle `for-each` qui parcourt directement les éléments d'une collection.

Boucles for-each

Dans la classe **Course**

```
public boolean isEnrolled(String  name, String firstName) {
    for (Student s : students) {
        if (s.getName().equals(name) && s.getFirstName().equals(firstName))
{
            return true;
        }
    }
    return false;
}
```

Dans cet exemple : **for** (Student s : students) signifie que pour chaque élément de type `Student` dans la liste `students`, l'élément actuel est assigné à la variable `s`.
Et dans la boucle on compare le nom et le prénom aux valeurs en paramètre et s'il y a égalité, on renvoie « vrai » puisqu'il n'est pas nécessaire d'aller au bout de la boucle pour savoir si un étudiant avec ce nom et prénom existe.

Dans la classe **Harmo**

```
/**
 * Permet de retrouver un étudiant inscrit dans nos listes à partir de son
email
 * @param email
 * @return l'étudiant dont l'émail correspond au paramétre, retourne null
sinon.
 */
public Student findStudent(String email) {
    for (Student student : students) {
        if (student.getEmail().equals(email)) {
            return student;
        }
    }
    return null;
}
```

&& exprime un **et**. L'opérande à droite n'est évalué que si celle de gauche est vraie, donc dans l'exemple ci-dessous que si les noms sont les mêmes.

Boucles for

- La boucle for classique en Java a la structure suivante :

```
for (initialisation; condition; incrémentation) {
    // bloc de code à exécuter
}
```

initialisation : C'est la partie où vous initialisez la variable de boucle. Cette partie est exécutée une seule fois au début de la boucle.
condition : C'est la condition qui est vérifiée avant chaque itération. Tant que cette condition est vraie, le bloc de code à l'intérieur de la boucle sera exécuté.
incrémentation : Cette partie est exécutée après chaque itération du bloc de code. Elle est souvent utilisée pour mettre à jour la variable de boucle.

```java
//Renvoyer une string représentant les emails des étudiants inscrits à ce
cours  précédé du numéro
public String getOrderedEmails() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < students.size(); i++) {
        sb.append(i + 1)
                .append(". ")
                .append(students.get(i).getEmail())
                .append("\n");
    }
    return sb.toString();
}
```

Questions en groupe & Codage individuel

11. Comprendre les codes
12. Écrire une méthode qui renvoie la position d'un étudiant de nom donné dans un cours. Si aucun étudiant avec ce nom n'est inscrit, elle renvoie -1.
    a) Dans quelle classe devez-vous écrire cette méthode ?
13. Écrire la méthode qui calcule la moyenne d'âge des étudiants inscrits à un cours.
    a) Dans quelle classe définissez-vous cette méthode ?
    b) Que se passe-t-il si aucun étudiant n'est inscrit au cours ?

Question individuelle & Codage

14. Écrire la méthode qui calcule la moyenne du niveau de difficulté des cours choisis par un étudiant. Dans quelle classe définissez-vous cette méthode ?
15. Écrire la méthode qui calcule la moyenne du niveau de difficulté des cours dont il est responsable. Dans quelle classe définissez-vous cette méthode ?
16. Écrire la méthode qui calcule la moyenne d'âge des étudiants de l'harmonisation. Dans quelle classe définissez-vous cette méthode ?
17. Supposons que nous voulions ajouter le fait que les étudiants évaluent leur progression à la fin du cours sur les notions abordées avec un pourcentage. Comment proposez- vous de définir cette donnée et de faire évoluer le code en conséquence ? N'hésitez pas à regarder les classes de l'API Java.

PARTIE 3 – Du Polymorphisme à l'Héritage

I. Polymorphisme > Overloading

Leçon 13

Nous vous proposons d'ajouter une nouvelle méthode dans la classe **Harmo.** Nous l'appellons comme précédemment *findStudent* mais changeons ses paramètres.

```
/**
 * Permet de retrouver un étudiant inscrit dans nos listes à partir de son
nom et son prenom
 * @param name
 * @param firstName
 * @return
 */
public Student findStudent(String name, String firstName) {
    for (Student student : students) {
        if (student.getName().equals(name) &&
student.getFirstName().equals(firstName)) {
            return student;
        }
    }
    return null;
}
```

Nous avons défini 2 méthodes " findStudent" qui varient sur leurs paramètres d'entrée.

Overloading (Surcharge de méthodes) :

Overloading se produit lorsque vous avez plusieurs méthodes dans une même classe qui portent le même nom mais ont des paramètres différents (type, ordre ou nombre de paramètres différents). Les méthodes **surchargées** ont donc le même nom mais des signatures de méthode différentes. La résolution de la méthode à appeler se fait au moment de la compilation en se basant sur les arguments passés.

Questions en groupe & Codage individuel

1. Comment définiriez-vous la méthode qui recherche
   a) un étudiant à partir de son nom uniquement ?
   b) plusieurs étudiants à partir d'un nom?
   c) plusieurs étudiants à partir d'une année de naissance ?

Question individuelle & Codage

2. Écrire les méthodes *findCourses* qui font sens pour vous dans Harmo

II. Equals

Leçon 14

Dans la classe **Course** nous avons défini une méthode *isEnrolled* qui parcourt la liste pour déterminer s'il existe un étudiant inscrit au cours qui a le nom et prenom attendu.
Nous aimerions mieux pouvoir vérifier si c'est vraiment le même étudiant, même âge, même nom et prénom, sans tenir compte des cours et de l'email.
Nous considérons donc que deux instances de *Student* sont égales si elles ont le même nom, le même prénom et la même date de naissance. Nous définissons donc la méthode **equals** dans Student comme suit.

```
Nous considérons donc que deux instances de Students sont égales si elles
ont le même nom, le même prénom et la même date de naissance.

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student student = (Student) o;
    return birthYear == student.birthYear && Objects.equals(name,
student.name) && Objects.equals(firstName, student.firstName);
}
```

La méthode equals étant utilisée par les méthodes sur les listes, nous implémentons très simplement notre nouvelle méthode comme suit.

```
public boolean isEnrolled(Student student) {
    return students.contains(student);
}
```

Questions en groupe & Codage individuel

3. Comprenez les codes

Question individuelle & Codage

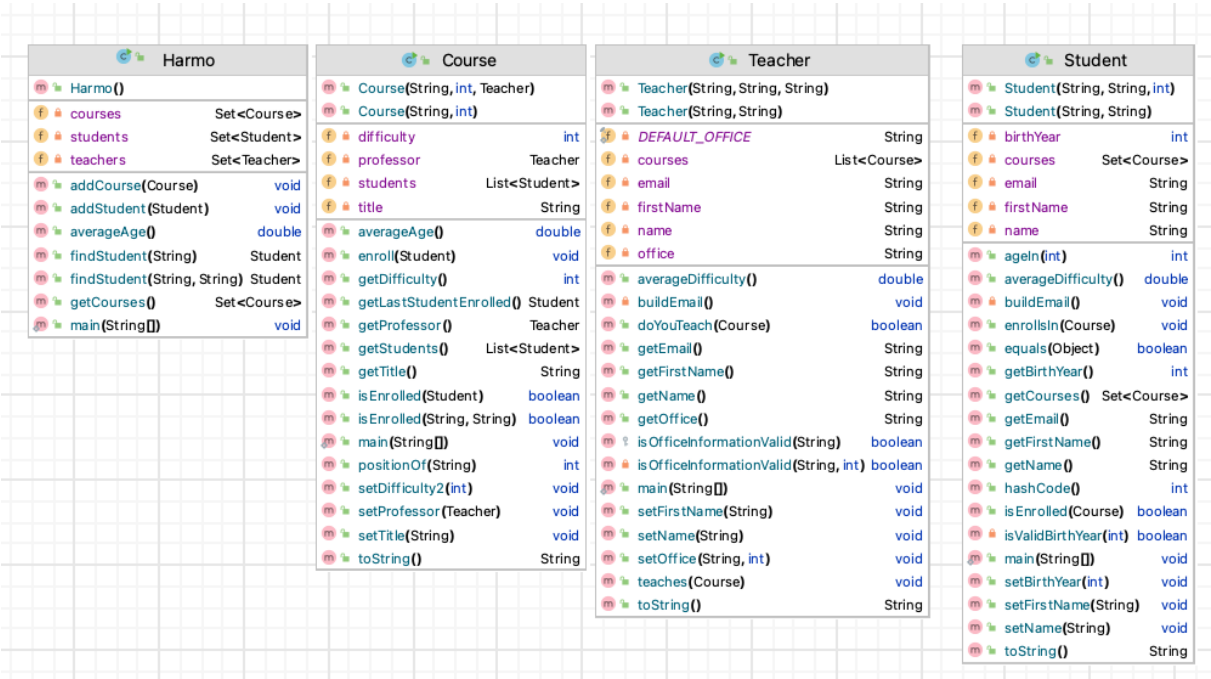
4. Ajouter à Enseignant une méthode qui vérifie s'il enseigne bien un cours donné.
a) Dans quelle classe faut-il définir la méthode equals ?

III. Héritage : définition et usages

Leçon 15

L'**héritage** (en anglais *inheritance*) est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'"*héritage*" (pouvant parfois être appelé *dérivation de classe*) provient du fait que la classe dérivée (la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive, dont elle hérite). L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.
Par ce moyen on crée une hiérarchie de classes de plus en plus spécialisées. Cela a comme avantage de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière il est possible d'acheter dans le commerce des librairies de classes, qui constituent une base, pouvant être spécialisées (on comprend encore un peu mieux l'intérêt pour l'entreprise qui vend les classes de protéger les données membres grâce à l'encapsulation...).[https://web.maths.unsw.edu.au/~lafaye/CCM/poo/heritage.htm]

Si nous reprenons notre exemple, avez-vous remarqué que beaucoup de codes se ressemblent ? Peut-être même l'avez-vous recopié ? Or, du code dupliqué, c'est aussi des bugs dupliqués.



Nous avons implémenté un enseignant par un nom, un prénom, une adresse mail et un bureau et un étudiant par un nom, prénom, une adresse mail etc.
Ils ont, de fait, des variables d'instances et méthodes identiques.
Nous utilisons l'héritage pour éviter cette duplication et exprimer que tous ces objets sont simplement Membres de Polytech et ont donc des comportements communs.

Nous définissons une classe ***PolytechMember***[3] qui regroupe les propriétés communes :

[3] IntelliJ vous permet de le faire simplement en procédant sous refactoring à une extraction de « superclass ».

```
public class PolytechMember {
    protected String name;
    protected String firstName;
    protected String email;


    public PolytechMember(String name, String firstName) {
        this.name = name;
        this.firstName = firstName;
        buildEmail();
    }

    protected void buildEmail() {
        this.email = name + "." + firstName + "@univ-cotedazur";
    }

    public String getName() {
        return name;
    }
…..
//A compléter en mettant en commun les codes qui peuvent être partagés.
}
```

Notez que les **variables d'instances sont protected** ce qui les rend visibles dans les sous-classes de la classe dans laquelle elles sont définies.

Puis nous définissons chacune des classes *Student* et *Teacher* **par extension de cette classe (extends).**

```
public class Student extends PolytechMember{
    private int birthYear;
…
}
```

Pour le constructeur qui permet de spécifier la date de naissance, il doit faire appel au constructeur défini dans la superclasse (donc *PolytechMember*). Il le fait par un **appel à super**. Puis nous donnons sa valeur à birthYear.

```
public Student(String aName, String firstName, int dateOfBirth) {
    super(aName, firstName);
    if (isValidBirthYear(dateOfBirth)) {
        this.birthYear = dateOfBirth;
    }
    else {
        this.birthYear = ConstantForHarmo.DEFAULT_BIRTH_YEAR;
    }
}
```

La construction de l'email qui était fixée dans nos classes par @etu.univ-cotedazur.fr pour les étudiants et par @univ-cotedazur.fr pour les enseignants nous pose problème parce que pour un étudiant et pour un enseignant la terminaison n'est pas la même.
Nous redéfinissons la méthode *buildEmail* dans la classe ***Student*** pour « **surcharger** » cette fois ci **au sens override** la méthode définie dans *PolytechMember*.

```
@Override
protected void buildEmail() {
```

```
    this.email = name + "." + firstName + "@etu-univ-cotedazur";
}
```

Pour *Student*, nous faisons le choix de réutiliser la méthode *toString* définie dans PolytechMember. Pour cela nous faisons **appel** à nouveau à **super**.

```
@Override
public String toString() {
   // return "Student : \n\t" + name + "\t " + firstName + ", \n\t" + email
+ ", \n\t" + birthYear + "\n";
    return "Student " + super.toString() + ", \n\t" + birthYear + "\n";
}
```

Questions en groupe & Codage individuel

5. Comprenez le code précédent et complétez la définition de PolytechMember avec toutes les informations qui sont partagées par Teacher et Student.
6. Relancez vos tests sur *Student* pour vérifier que votre code fonctionne toujours « extérieurement » comme précédemment.
7. Quelle autre variable d'instance aurions-nous pu mettre en commun ? Qu'en pensez-vous ?

Question individuelle & Codage

8. Modifiez votre classe Teacher pour qu'elle hérite aussi de *PolytechMember*.

AMUSEZ-VOUS avec vos codes !