PARTIE 3 – Du Polymorphisme à l'Héritage

I. Polymorphisme > Overloading

Leçon

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.

Regardez dans l'API pour les collections les méthodes surchargées proposées.

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 uniquement ?
   c) plusieurs étudiants à partir d'un nom et d'une année de naissance ?

Question individuelle & Codage

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

II. Equals

Leçon

Dans la classe **Course** nous avons défini une méthode *isEnrolled*. Nous aimerions mieux pouvoir vérifier si c'est vraiment le même étudiant, même age, même email, etc. Nous surchargeons donc la méthode et utilisons le *contains* qui cherche s'il existe un objet ***égal(equals)***.

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

Pour comparer deux étudiants nous ajoutons donc une méthode *equals* dans **Student** en ne tenant compte que des nom, prénom et email.

```
@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) &&
Objects.equals(email, student.email);
}
```

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 ?

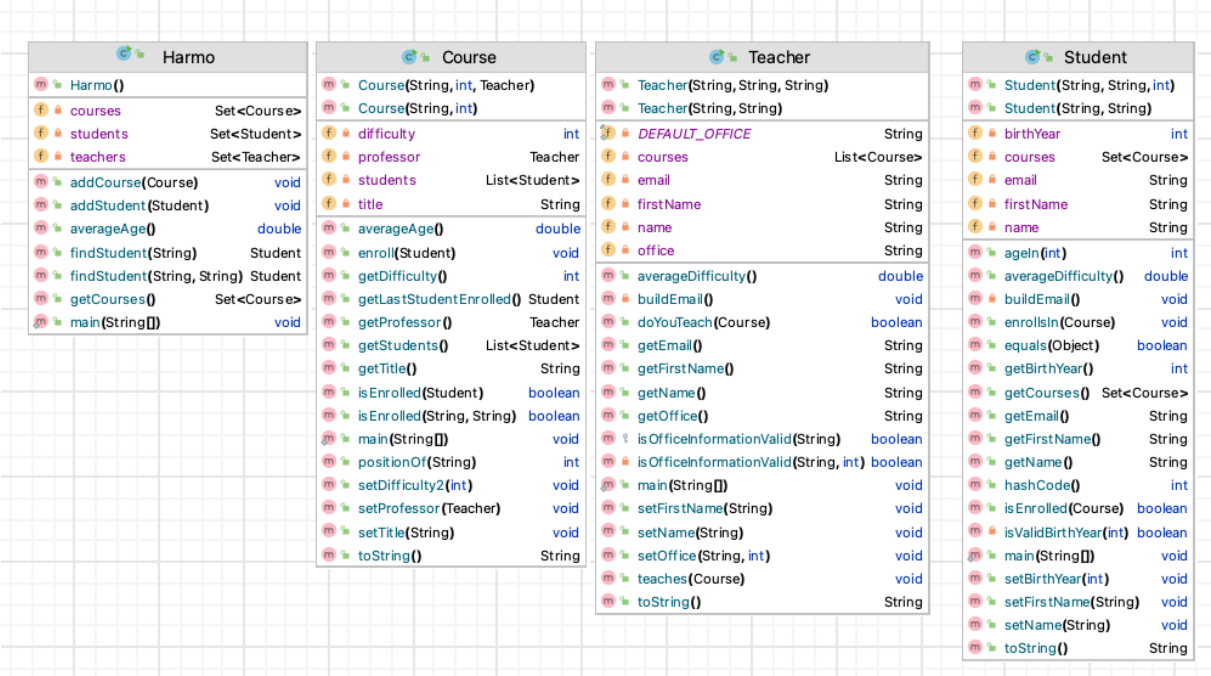
III. Héritage : définition et usages

Leçon

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). 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 majeur 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 à loisir (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.



Leçon

L'héritage entre classes en programmation orientée objet permet à une classe enfant d'hériter des caractéristiques (variables d'instances et méthodes) d'une classe parent.

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***[1] qui regroupe les propriétés communes :

```
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() {
```

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

```
        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.**

```
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 !