# Introduction à la POO
# Premiers diagrammes de classes

**Bertrand LIAUDET**



Edition juin 2019

**Présentation du cours**

- Le cours est découpé en 4 parties :
  - 1 : Les types de langage et la notion d'objet
  - 2 : Les notions de classe, de constructeur et d'encapsulation.
  - 3 : Relations entre classes : agréation, composition, énumération.
  - 4 : Relations entre classes : héritage et polymorphisme. Dernières notions : objet et référence, attributs et méthodes de classe, classe abstraite et interface.
- Les 2 premières parties posent les bases en se concentrant sur les concepts centraux : l'objet et la classe.
- Les 2 dernières parties sont plus difficiles et plus technique et traite des relations entre les classes.
- Chaque partie consiste en une présentation théorique et des exercices.
- Les exercices peuvent être fait en mode « algorithmique - UML » ou en mode « programmation ». Dans ce cas, on peut les faire en Python ou en Java.
- Après cette présentation, il faut mettre en pratique et coder tous les exercices, en Python ou en Java, au choix !
- Ou passer à de la conception plus avancée/
- La présentation est associée à un ensemble d'exemples de code téléchargeables en Java et en Python.

**Sources Java :**

https://docs.oracle.com/javase/9/docs/api/index.html?overview-summary.html : en mode frame

http://maurise-software.e-monsite.com/medias/files/polyjava.pdf

**Sources Python :**

https://docs.python.org/fr/3/

https://docs.python.org/fr/3/tutorial/classes.html

https://www.courspython.com/

https://www.courspython.com/classes-et-objets.html

# 1 - TYPES DE LANGAGE - NOTION D'OBJET

## Introduction

### Programmation objet et langage objet

- On dit « **programmation objet** » ou « **programmation orientée objet** » ou « **P.O.O.** »
- La POO est **un type de programmation parmi d'autres**.
- La programmation objet s'est **industrialisée dans les années 90** (langages C++ et Ada) et **généralisée dans les années 2000**.
- La programmation objet aujourd'hui, c'est **essentiellement** : **le Java, le C#, le C++.**
- **Mais aussi** : le **PHP**, le **JavaScript**, le **Python** et bien d'autres langages.
- Dans cette journée, on verra des **exemples en Java et en Python**. Mais le but est de comprendre les principes.
- Les exemples **Java** correspondent à de la **programmation objet « standard ».** C'est proche du **C#**, du **C++** ou du **PHP**.
- Les exemples en **Python** correspondent à de la **programmation objet non typée**. C'est proche du **JavaScript**.
- Les exercices seront essentiellement des **exercices « papier »**, codables accessoirement en Java ou en Python
- **L'évaluation** consistera en **un petit QCM et un exercice simple**, « papier » ou codable, mais sur papier de toute façon.

## Un type parmi les 4 grands types de programmation

- On peut distinguer 4 types de programmation et 4 types de langages associés :

### 1 : La programmation WEB : Le HTML

- C'est le langage pour fabriquer les pages WEB.
- On peut le mettre à part car ses principes et sa syntaxe sont très spéciaux.
- Un langage unique : le HTML-CSS

### 2 : La programmation procédurale et les langages procéduraux.

- C'est la programmation qu'on peut appeler « de base ».
- Ce qu'on apprend en **algorithmique de base** : Variable, Type, Test, Boucle, **Fonction**, Tableau, Structure, Tableau de structures.
- Fonction = Procédure. « fonction » et « procédure » veulent dire quasiment la même chose.
- On dit aussi « **langage impératif** » à la place de « langage procédural ».
- **Principaux langages procéduraux :** C, Fortran, Pascal, Cobol, Perl, PL-SQL

- A noter que la **programmation fonctionnelle**, c'est autre chose. La programmation fonctionnelle utilise des **lambda-expressions**. On ne s'y intéresse pas. C'est une activité de niche, même si les lambda-expressions tendent à coloniser tous les langages.

## 3 : La programmation événementielle et les langages événementiels.

- C'est une programmation qui gère la prise en compte d'événements pour réagir en conséquence.
- Ces événements sont essentiellement des **événement** du type « **clavier** », « **souris** », « **écran tactile** » en provenance d'un utilisateur. Mais il peut y en avoir d'autres : de **événements système**, des **événements http** (réseau).
- Principal langage événementiel : le JavaScript
- La plupart des langages procéduraux et objet fournissent des bibliothèques leur permettant de faire de la programmation événementielle (**Tkinter en Python**, **Swing en Java**, etc.)

## 4 : La programmation objet et les langages objet.

- C'est une **programmation qui fabrique des « objets » informatiques** qui servent à simuler les objets de la réalité de la façon la plus intuitive possible.
- Une fois ces objets fabriqués, on va chercher les **fonctions qu'on peut leur appliquer**.
- Par exemple, j'ai un **objet voiture**. Je peux lui appliquer la **fonction démarrer( )**, ou la **fonction changerVitesse( ).** Ces fonctions changent l'état de la voiture : le moteur tourne, la vitesse n'est plus la même.
- L'intérêt de la programmation objet est qu'on peut réutiliser les objets créés facilement dans d'autres programmes.
- Principaux langages objets : le Java, le C++, le C#

## Les types de programmation se mélangent

- Aujourd'hui, la plupart des langages sont des **langages objets** qui permettent de faire de la **programmation événementielle** et aussi de faire de la **programmation procédurale.**

- Le **Python** est un **langage plutôt procédural** qui permet de faire :
  - de la **programmation procédurale**,
  - de la **programmation événementielle** grâce à des bibliothèques (Tkinter),
  - de la **programmation objet**.

- Idem pour le Java qui est un langage intrinsèquement objet.
- Idem pour le JavaScript qui est un langage plutôt événementiel.

- A noter toutefois que faire de la programmation procédurale en Python est une activité « normale ». On peut se limiter à un usage non-objet du Python.
- Faire de la programmation procédurale en Java n'est pas une activité normale ! Le Java est un langage intrinsèquement objet qui n'est pas du tout fait pour faire du procédural.

## Paradigme objet versus paradigme procédural

**Programmation procédurale : analyser le « main » par procédure (ou fonction)**

**Principes**

- On analyse le problème complet en le découpant en fonctions successives.

**Exemple général**

1) : On fait telle chose, par exemple l'initialisation des données. Une fonction gère ce travail.

2) : On fait telle chose suivante : par exemple la saisie de d'information par l'utilisateur.

3) : On fait telle autre chose suivante : par exemple un calcul sur les données à partir des informations de l'utilisateur. Une fonction gère ce travail.

4) : En fonction des résultats du calcul, on fait appel à telle ou telle autre fonction.

**Exemple du jeu des allumettes (jeu de Marienbad)**

http://therese.eveilleau.pagesperso-orange.fr/pages/jeux_mat/textes/marienbad.htm

```
1:  initialisationDuJeu() ; afficherLeJeu()
2:  choixDunJoueurDeDepart()
3:  SI joueur de depart = ordi ALORS ordinateurJoue()
4:  BOUCLE répéter
5:          leJoueurJoue() ; afficherLeJeu()
6:          SI  jeuGagnant() ALORS afficher("le joueur a gagné") et quitter boucle
7:          lOrdinateurJoue() ; afficherLeJeu()
8:          SI jeuGagnant() alors afficher("l'ordi a gagné") et quitter la boucle
9:  FIN de BOUCLE répéter
```

On se retrouve avec au moins 6 fonctions :

- initialisationDuJeu()
- afficherLeJeu()
- choixDunJoueurDeDepart()
- leJoueurJoue()
- lOrdinateurJoue()
- jeuGagnant()

## Programmation objet : analyser par objet

### Principes

- **On analyse les « objets »** qui entrent en jeu dans le problème.
- Les objets doivent simuler des objets de la réalité.
- Ensuite, **on définit les fonctions associées aux objets** (les méthodes).
- Enfin, **on écrit le programme** qui utilise ces objets pour produire les résultats attendus.

### Exemples

- Un objet **utilisateur** et des **méthodes associées** :
  - changer le mot de passe d'un utilisateur.
  - afficher les informations d'un utilisateur.
- Un objet **voiture** et des **méthodes associées :**
  - démarrer une voiture.
  - changer la vitesse d'une voiture.
- Un objet **agenda** et des **méthodes associées** :
  - ajouter une personne dans un agenda.
  - rechercher une personne dans un agenda.
- etc.

**Bilan**

**Paradigmes**

- On parle de « **paradigme objet** » *versus* le « **paradigme procédural** ».
- Un paradigme, c'est une **façon de voir les choses, de réfléchir, de faire**, qui est complètement différente, pour arriver au même résultat à la fin : un programme (le « main ») qui fait ce qu'on veut.

**Paradigme objet**

- La façon de faire « objet » est complètement différente de la façon de faire « procédurale ».
- **On ne s'intéresse pas d'abord au programme principal, au but ultime qu'on découpe en fonctions**.
- **On s'intéresse d'abord aux objets** qui vont participer à la réalisation de ce but et aux fonctions (méthodes) qu'on peut attacher aux objets pour réaliser ce but.

**Le « main »**

- Dans les deux cas, il faudra un « main », qui relève globalement de la programmation procédurale, et qui va organiser les traitements.

**Paradigme du débutant : tout dans le main !**

- A noter que le programmeur débutant n'applique ni le paradigme procédural, ni le paradigme objet, mais le « **paradigme du débutant** » (ou paradigme du main) : il met tout son code dans le main sans fonction, ou presque !

## Exemple d'objet : le feu de circulation

### Présentation

- Un feu de circulation (ou feu tricolore ou feu rouge) est un dispositif d'éclairage : la boite avec ses différentes ampoules. On veut simuler son fonctionnement.
- On considère que le feu marche tout le temps. Il est forcément allumé et dans un certain état : vert, orange ou rouge.
- On simplifie donc les choses : il n'y a pas de orange clignotant et pas de feu éteint. Le feu passe forcément du vert au orange, du orange au rouge et du rouge au vert.

### Comment caractériser cet objet : quelles propriétés (attributs) lui donner ?

- **Une propriété**, c'est une **information qui caractérise un objet** et qui **peut changer de valeur ou pas** au fur et à mesure de la vie de l'objet. Par exemple : la marque d'une voiture est une propriété qui ne change pas. Elle est définie à sa création. Par contre l'immatriculation d'une voiture est une propriété qui peut changer. Comme sa couleur.
- Ici, on en a une seule propriété : la couleur du feu.
- La couleur peut être : vert, orange ou rouge.
- En programmation objet, les propriétés des objets sont appelées « **attribut** ».
- « **couleur** » est un **attribut** de l'objet feu de circulation.

**Que peut-il arriver à notre feu ? Qu'est-ce qui peut arriver à ses attributs ?**

- **La couleur peut changer de valeur**. Ce changement est fonction de la couleur précédente : du vert on passe au orange, du orange au rouge et du rouge au vert.
- On va définir une fonction **changerCouleur()** qui permet de gérer ce changement. Cette fonction n'a pas de paramètre (on ne passe pas d'informations entre les parenthèses).
- Les fonctions s'appliquent forcément à un objet : quand on applique **changerCouleur**() à un objet feu, la fonction change la couleur de l'objet, la couleur actuelle de l'objet.
- On veut aussi pouvoir **accéder à l'état du feu même sans le voir** ! Pour ça on définit une fonction **getCouleur()** : elle nous renvoie la couleur du feu.
- En programmation objet, les fonctions associées aux objets sont appelés « **méthode**».
- **changerCouleur**() et **getCouleur**() sont deux méthodes de l'objet feu de circulation.

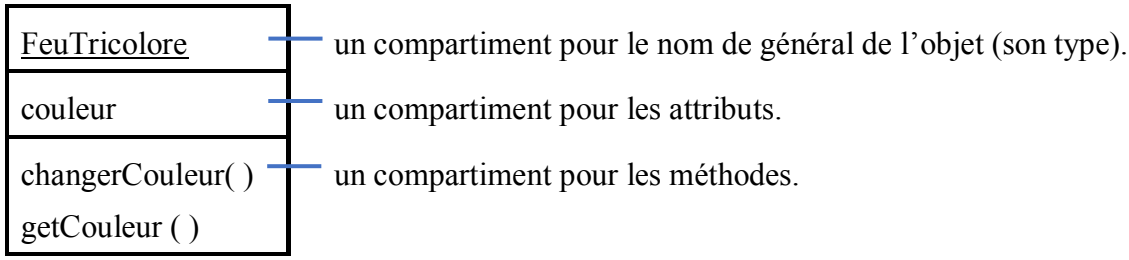
**A noter :**

- On aurait pu, mais on évite, définir une fonction **setCouleur(nouvelleCouleur).**
- Cette fonction permet de donner une nouvelleCouleur à la place de la couleur actuelle. On passe la nouvelle couleur en paramètre. On dit « setCouleur » qui veut dire « donner une valeur à la l'attribut « couleur ». Cette valeur c'est le paramètre « nouvelleCouleur ».
- **On ne va pas faire ça !!!** Pourquoi ? Si définit cette fonction, alors on pourra donner n'importe quelle couleur à n'importe quel moment, donc passer du rouge au orange. On veut éviter ça !

## Représentation graphique et convention de nommage : diagramme de classes UML

### Diagramme de classes

- On représente la synthèse de notre analyse graphiquement dans un rectangle à 3 compartiments :




- Cette façon d'écrire est standard : c'est un **diagramme de classes** UML.
- C'est très pratique : ça permet de synthétiser efficacement notre analyse.

### Convention de nommage

- Le nom général (ou le type) pour tous les objets qui seront des feux tricolores c'est ce qu'on appelle sa « **classe** ». Sa **première lettre** d'un nom de classe est toujours une **majuscule**.
- La **première lettre** du nom des **attributs** et des **méthodes** est en **minuscule**.
- On utilise ensuite le « **camel case** » : chaque mot suivant du nom complet commence par une majuscule. Pas de tiret, ni d'espace ou d'autres caractères spéciaux.

- Cette convention de nommage est standard : on la retrouve dans tous les langages objets.
- C'est très pratique : avec l'habitude, ça permet de mieux s'y retrouver dans le code.

## Exercices : analyse d'objets, diagramme de classes – Séquence 1 – Slide 12

### Principes

- L'objectif des exercices consiste, à partir de la description d'une situation, à trouver le ou les objets manipulés, leurs attributs et leurs méthodes.
- Le résultat de l'analyse est présenté dans un diagramme de classes.
- Ces exercices seront repris dans les séquences suivantes pour aborder d'autres notions.

### 1 - Télévision

- Une télévision a une marque et une diagonale. Elle peut être allumée ou éteinte. On peut changer de chaine en donnant le numéro de la chaine ou en montant ou descendant le numéro en cours. On peut monter ou baisser le son. On peut aussi couper le son ou le remettre, avec le même bouton. Quand on remet le son, il revient à son niveau avant d'avoir coupé le son. On peut connaître le niveau du son ainsi que la chaine actuellement sélectionnée.
- A l'allumage, le son est à 3 par défaut et la chaine à 0 par défaut.

### 2 - Voiture

- Une voiture a une marque, un modèle, une immatriculation. Elle peut avoir 2 portes, 4 portes ou 5 portes.
- On peut mettre le moteur de la voiture en route et passer des vitesses. Le passage de vitesse se fait forcément de 1 en 1, en montant ou en descendant. On peut aussi passer au point mort à tout moment. On peut aussi freiner ou accélérer.

### 3 - Afficheur de score

- Afficheur de score est un objet qui affiche des scores de matchs de football.
- Par exemple :
  - Pour les matchs à venir, on affiche : Paris – Marseille : à venir
  - Pour les matchs en cours, on affiche : Paris – Marseille : 1-0 - en cours
  - Pour les maths finis, on affiche : Paris – Marseille : 3-2 - terminé
- L'affichage commence toujours par le nom de l'équipe qui reçoit, suivi de l'équipe qui se déplace.
- Il s'agit d'un match sans prolongation ni tirs au but pour nous simplifier la tâche.
- L'afficheur permet de faire évoluer le score au fur et à mesure de la partie.
- Le score ne peut changer que pendant le match, ni avant, ni après.
- On initialise l'objet en lui donnant le nom des deux équipes. Automatiquement, le match est « à venir ».

### 4 - Distributeur de boissons

- Un distributeur automatique permet de choisir une boisson au choix.
- On peut préciser le niveau de sucre qu'on souhaite avoir.
- Un fois la boisson versée, le distributeur rend la monnaie.

- Le client peut demander la restitution de l'argent qu'il a déjà mis.

**5 - Personnage de jeu vidéo**

- Un personnage de jeu vidéo a un nom, une force, une localisation, une expérience et des dégâts.
- Le personnage peut se déplacer et combattre un autre personnage.

**Idées au choix**

- Avez-vous des idées de systèmes simples à modéliser sous la forme d'objet ?

# 2 - CLASSE, MOULE, CONSTRUCTEUR, ENCAPSULATION – EXEMPLES DE CODE.

## La notion de classe : un moule pour créer des objets

### Principes

- La programmation objet consiste à manipuler des objets qui contiennent des attributs et des méthodes.
- **La classe est un « type ».** Un objet est d'une certaine **sorte** : les feux de circulation, les télévisions, les voitures, etc. La « sorte » de l'objet, c'est ce qu'on appelle le « **type** » en programmation procédural et la classe en programmation objet.
- **La classe est un « moule »** : la classe, c'est ce qui permet de décrire ce que seront les objets concrets qu'on fabriquera à partir de cette classe. C'est un peu comme un moule pour fabriquer des objets concrets. Avec un moule, comme avec une classe, on pourra fabriquer plein d'objets, qui pourront être particularisé en fonction de ce qu'on met dans le moule et encore après les avoir fabriqué.
- **Exemple** : le **moule à gaufres**. On peut fabriquer des gaufres différentes selon la pate qu'on utilise (standard, sans lactose, bio, allégé, etc.). On peut ensuite mettre de la garniture : du sucre ou de la purée de marron.

moule à gaufres = classe Gaufre

- **La classe est une « abstraction ».** Les objets sont concrets. La classe, c'est le modèle : est une abstraction (ou une généralisation).

## Notion de constructeur

- La classe est un moule qui permet de fabriquer des objets.
- Le **constructeur** est une **méthode particulière** d'une classe. Cette méthode est utilisée **pour fabriquer un objet**.
- Le constructeur permet de créer un objet d'une certaine classe, comme on crée une variable d'un certain type.
- L'objet est finalement comme une variable dont le type correspond à la classe de l'objet :

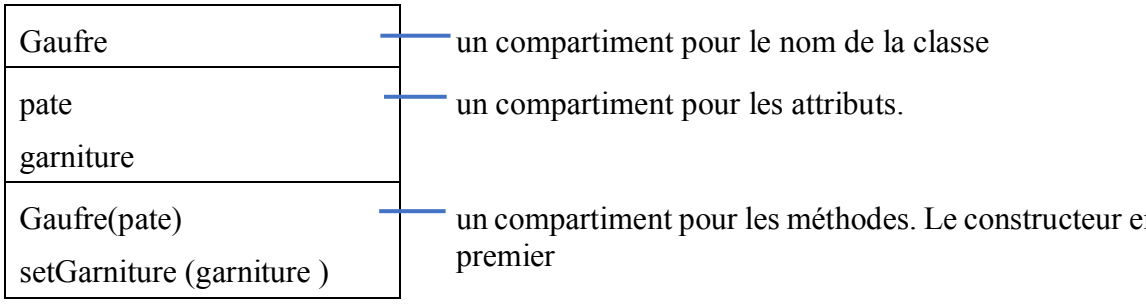
- objet = variable - classe = type

- Le constructeur a toujours le nom de la classe à laquelle il appartient.
- **Le constructeur peut avoir des paramètres** : ainsi, on pourra fabriquer des objets différents les uns des autres si on le souhaite. Par exemple, une gaufre standard ou une gaufre bio. Ou un voiture de telle marque et tel modèle ou d'une autre marque et d'un autre modèle.

## Représentation graphique

### La classe Gaufre

- ***représentation UML***




- ***constructeur : Gaufre (pate)***

Le constructeur permet de fabriquer un nouvel objet gaufre. On lui passe en paramètre la pate utilisée.

- ***setGarniture (garniture)***

setGarniture est une méthode qui permet de donner une garniture à une gaufre. On lui passe en paramètre la garniture utilisée.

### Convention de nommage

- Le constructeur est une méthode a le nom de la classe et qui commence par une majuscule.
- On le met en premier dans le compartiment des des méthodes

**Convention de nommage**

- On donne **des noms** de classe, d'attribut et de méthode **les plus explicites que possibles**.
- Le nom de la **classe** commence par une **majuscule** continue en minuscules
- Le nom du **constructeur** reprend celui de la **classe**.
- Le nom des **attributs** et des méthodes commence par une **minuscule** et continue en minuscules.
- Quand un nom de classe, d'attribut ou de méthode est composé de **plusieurs parties**, chaque partie commence par une majuscule : c'est le style **camel-case**.
- Quand un paramètre de méthode correspond à un attribut, on lui donne le nom de l'attribut.
- Pour accéder à un attribut dans une méthode, on le préfixe toujours par « **this.** ».
- Les méthodes qui servent à donner une valeur à un attribut s'appellent des « **setters** ». La signature (= l'en-tête) de ces méthodes est toujours : « **setAttribut (attribut)** ».

## Code Java de la classe Gaufre (syntaxe simplifiée)

```
// Déclaration de la classe Gaufre en Java

class Gaufre {
    String pate;        // standard, bio, sansGluten, allégée, etc.
    String garniture;  // sucre, marron, etc.

    Gaufre(String pate) {
        this.pate = pate;
        this.garniture = null;
    }

    void setGarniture(String garniture) {
        if (this.garniture == null) {
            this.garniture = garniture;
        }
    }

    void afficher() {
        System.out.println("gaufre : "+this.pate+" "+this.garniture);
    }
}
```

**Explications du code Java**

- On retrouve la **structure du schéma UML** : 1) le nom de la **classe** ; 2) les **attributs** ; 3) les **méthodes**.
- Le **Java** est un **langage typé**. Le type String est utilisé pour les chaines de caractères. La « pate » et la « garniture » sont des chaines de caractères.
- **this**: c'est un paramètre qui permet d'accéder aux attributs de la classe. Le « pate » de « this.pate » c'est attribut « pate » de la classe. Le « pate » de « = pate », c'est le paramètre « pate » du constructeur « Gaufre ( ).
- **null :** le permet d'initialiser une chaine de caractère à rien.
- constructeur : ici, « **pate** » et « **garniture** ». Seule l'attribut « pate » est initialisé avec une valeur par le constructeur. L'attribut garniture est initialisé à « vide » : "". On pourra ensuite lui donner la valeur qu'on veut, de n'importe quel type.
- **La méthode afficher ( )** : souvent, on met une méthode afficher( ) dans les classes pour pouvoir afficher la valeur des attributs d'un objet.
- **System.out.println** ( ) : c'est la fonction Java qui permet d'afficher du texte à l'écran.
- **void** : signifie en Java que la fonction ne retourne rien.

## Code Python de la classe Gaufre

```
# Déclaration de la classe Gaufre en Python
class Gaufre:
    def __init__(self, pate):
        self.pate = pate
        self.garniture = ""

    def setGarniture(self, garniture):
        if self.garniture == "" : # on met la garniture s'il n'y en a pas déjà
            self.garniture = garniture

    def afficher(self):
        print("gaufre : "+self.pate+" "+self.garniture)
```

## Explications du code Python

- **__init__** : c'est le constructeur. En Python, il s'appelle toujours comme ça.
- **self** : c'est un paramètre qui permet d'accéder aux attributs de la classe.
- **Les attributs** sont définis dans le constructeur : ici, « **pate** » et « **garniture** ». Seul l'attribut « pate » est initialisé avec une valeur par le constructeur. L'attribut garniture est initialisé à « vide » : "". On pourra ensuite lui donner la valeur qu'on veut, de n'importe quel type.
- **La méthode afficher(self)** : souvent, on met une méthode afficher( ) dans les classes pour pouvoir afficher la valeur des attributs d'un objet.

## Utilisation d'une classe

### Principes

- **La classe est un moule.** Le moule va nous permettre de créer des objets (avec le constructeur) puis d'utiliser les objets avec leurs méthodes.
- Il y donc **deux actions principales** que permet une classe :
  - **La fabrication des objets** : on parle **d'instanciation.**
  - **L'utilisation de méthode à partir d'un objet** : on parle d'**envoi de message** à un objet.
- La fabrication et l'utilisation peuvent se faire dans le **programme principal** (le « **main** » en référence au langage C) ou dans le code de n'importe quelle méthode d'une autre classe le plus souvent.
- Une fois la ou les classes écrites, on peut écrire un main pour tester les méthodes de la classe : c'est ce qu'on appelle un **test unitaire** (un main par classe à tester).
- On peut aussi écrire un programme complet !

**Notion d'instanciation : new**

- Quand on fabrique un objet à partir d'une classe, on dit qu'on **instancie** un nouvel objet. C'est une **instanciation** d'objet.
- On utilise en général le mot clé « **new** ». On écrit :

```
objet=new Constructeur( … );
```

- par exemple :

```
gaufre=new Gaufre("standard");
```

**Accès au méthode : envoi de message**

- Quand on a un objet, on peut accéder aux méthodes de sa classe. Quand **on fait appel à une méthode d'un objet pour l'objet**, on dit qu'**on envoie un message à l'objet.**
- par exemple :

```
gaufre.setGarniture("marron");
```

## Exemple du moule à gaufre

### Objectifs du « main »

- On veut fabriquer une gaufre standard sucrée et afficher les résultats.
- On veut ensuite mettre du marron sur cette gaufre et vérifier que ça ne fait rien.
- On veut ensuite fabriquer une deuxième gaufre, bio cette fois et aux marrons, puis afficher les résultats.

**Code algorithmique**

```
Main :
    // fabrication d'une gaufre standard et affichage
    gaufre1=new Gaufre("standard");
    gaufre1.setGarniture("sucre");
    gaufre1.afficher();

    // ajout d'une autre garniture et affichage
    gaufre1.setGarniture("marron");
    gaufre1.afficher();      // gaufre1 n'a pas changé

    // fabrication d'une gaufre bio aux marrons et affichage
    gaufre2=new Gaufre("bio");
    gaufre2.setGarniture("marron");
    gaufre2.afficher();
```

- **new** : **pour créer un objet**, on utilise le mot clé « new » devant le constructeur. On considère que le constructeur retourne un objet qui est mis ici dans « gaufre1 ». Ensuite un deuxième objet est créé et mis dans gaufre2.
- Accès aux méthodes : pour accéder aux méthodes, on écrit : objet.méthode(…). Ici par exemple : gaufre1.afficher() et gaufre1.setGarniture(« sucre »).

## Code Java du « main » utilisant la classe Gaufre – syntaxe simplifiée

```
class GaufreMain {
    public static void main (String args[]) {
        // fabrication d'une gaufre standard sucré et affichage
        Gaufre gaufre1=new Gaufre("standard");
        gaufre1.setGarniture("sucre");
        gaufre1.afficher();

        // ajout d'une autre garniture et affichage
        gaufre1.setGarniture("marron");
        gaufre1.afficher();     // gaufre1 n'a pas changé

        // fabrication d'une gaufre bio aux marrons et affichage
        Gaufre gaufre2=new Gaufre("bio");
        gaufre2.setGarniture("marron");
        gaufre2.afficher();
    }
}
```

- Le code du main est placé dans la fonction « public static void **main(** String args[] **)** »
- « **public static void** » sont à écrire obligatoirement devant le « main » avec **String args[]** en paramètre.
- La fonction est mise dans une classe dédiée au main : ici la classe GaufreMain
- Le reste, c'est comme l'algorithme, sauf que les objets gaufre1 et gaufre2 sont typés. On écrit Gaufre gaufre1 = …

## Code Python du « main » utilisant la classe Gaufre

```
# "main" python : utilisation de la classe gaufre
#
# fabrication d'une gaufre standard et affichage
gaufre1=Gaufre("standard");
gaufre1.afficher();

# ajout d'une garniture affichage
gaufre1.setGarniture("sucre");
gaufre1.afficher();

# ajout d'une autre garniture et affichage
gaufre1.setGarniture("marron");
gaufre1.afficher();  # gaufre1 n'a pas changé

# fabrication d'une gaufre bio aux marrons et affichage
gaufre2=Gaufre("bio");
gaufre2.setGarniture("marron");
gaufre2.afficher();
```

- C'est comme l'algorithme :
  - Le code du main est placé en dehors de toute fonction.
  - On n'utilise pas le mot clé « new »
  - On n'utilise pas de type.

## Principe d'encapsulation – attributs « private » et méthodes « public »

**Objet = état + comportement**

**Objet** = **données** (état) + **méthodes** (comportement, rôles, responsabilités)

- Un objet informatique est une **variable** avec une ou plusieurs valeurs (les attributs) qui seront manipulées (en lecture ou en écriture) par **les fonctions associées** à l'objet (les méthodes).

**Etat d'un objet : ses attributs**

- **L'état** d'un objet à un moment c'est **la valeur** de ses attributs à ce moment.
- Certaines valeurs peuvent **changer au cours du temps**. D'autres peuvent **être constantes**.

➢ ***Comportement d'un objet : ses méthodes (rôle, responsabilités, fonctionnalités)***

- Le **comportement** d'un objet c'est l'ensemble de ses **méthodes**. On parle aussi du **rôle** d'un objet ou des **responsabilités** d'un objet. On peut aussi dire : les **fonctionnalités** d'un objet.

**Utilisation d'un objet**

- Pour utiliser un objet, il faut **commencer par le créer avec le constructeur** (l'instancier). Cette première étape définira un **premier état de l'objet**.
- Ensuite, on peut **lui appliquer toutes ses méthodes** qui permettront, entre autres, de **consulter** ou de **modifier** l'état de l'objet.

## Encapsulation : principes et syntaxe

### Principes

- L'encapsulation consiste à **cacher l'état de l'objet** (ses attributs et leurs valeurs).
- Les cacher ? C'est-à-dire ?
- On ne pourra accéder à l'état de l'objet, que ce soit pour le consulter ou pour le modifier, qu'en utilisant les méthodes de l'objet. On dit que les données sont encapsulées par les méthodes.
- Techniquement, pour encapsuler les attributs, on les définira en précisant le mot clé « **private** ». Ce mot clé fait qu'on ne pourra plus accéder à cet attribut en dehors de sa classe.
- Puis on définira les méthodes avec le mot clé « **public** ». Ce mot clé fait qu'on pourra utiliser les méthodes en dehors de la classe (dans un « main » par exemple). Les méthodes se chargeront de consulter ou de modifier les attributs.

### Syntaxe : 2 niveaux de visibilité principaux : private et public

| Symbole | Mot-clé | Signification |
|---|---|---|
| - | private | Visible uniquement **dans la classe** |
| + | public | Visible **partout** |

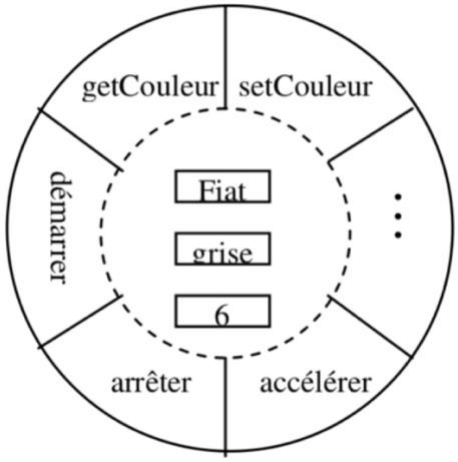
## Encapsulation : exemples – Voiture et FeuTricolore

### Exemple 1 : la classe Voiture

- La classe Voiture :

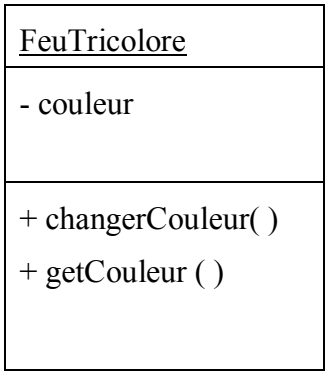
| Voiture |
|---|
| - marque<br>- couleur<br>- chevaux |
| + démarrer ( )<br>+ arrêter ( )<br>+ accélérer<br>+ getCouleur ( )<br>+ setCouleur (couleur)<br>+ getMarque()<br>… |

- un objet voiture :




- Ici, le marque de la voiture (Fiat) et le nombre de chevaux (6) sont des attributs qui ne seront pas modifiés. Il n'y aura donc **pas de** méthodes **setMarque**(marque) ni **setChevaux**(chevaux). Par contre, il y a **une méthode setCouleur**(couleur) pour le cas où on repeint la voiture !

**Exemple 2 : le feu de circulation**




- On peut appliquer la méthode **changerCouleur()** qui modifie la couleur en fonction de son état (passe du vert au orange, du orange au rouge, du rouge au vert).
- **On évite d'avoir une méthode setCouleur(couleur)** qui permettrait de donner n'importe quelle couleur quel que soit l'état initial et donc permettrait de faire passer du vert au rouge directement.

**L'art du programmeur**

- Il consiste à :
  - **Eviter tout ce qui peut rendre l'état de l'objet incohérent** et tout ce qui peut correspondre à une évolution incohérente de l'objet.
  - **L'encapsulation au sens restreint** consiste à éviter un accès direct aux attributs pour éviter que le programmeur puisse leur donner des valeurs sans faire de vérification de cohérence.
  - **L'encapsulation au sens large** consiste ce que les actions réalisées par les méthodes le soient de façon cohérente, c'est-à-dire en prenant en compte l'état de l'objet et les règles de fonctionnement de l'objet. Le but est d'éviter toute possibilité d'action incohérente (par exemple : change la pate d'une gaufre, ou changer la garniture d'une gaufre).
  - L'art du programmeur, c'est de gérer proprement l'encapsulation en offrant un jeu de méthodes judicieux qui protège contre une utilisation non conforme.

## Codage de l'encapsulation

**code Java de la classe Gaufre - syntaxe standard**

```
public class Gaufre {
    private String pate;
    private String garniture;

    public Gaufre(String pate){
        this.pate = pate;
        this.garniture = null;
    }

    public void setGarniture(String garniture){
        if (this.garniture == null){
            this.garniture = garniture;
        }
    }

    public void afficher(){
        System.out.println("gaufre : "+this.pate+" "+this.garniture);
    }
}
```

Le mot clé « **public** » s'applique aussi aux classes. Cela veut dire qu'on peut les utiliser.

**code Java de la classe GaufreMain – syntaxe standard**

```
public class GaufreMain {
    public static void main (String args[]) {
        // fabrication d'une gaufre standard sucré et affichage
        Gaufre gaufre1=new Gaufre("standard");
        gaufre1.setGarniture("sucre");
        gaufre1.afficher();

        // ajout d'une autre garniture et affichage
        gaufre1.setGarniture("marron");
        gaufre1.afficher();     // gaufre1 n'a pas changé

        // fabrication d'une gaufre bio aux marrons et affichage
        Gaufre gaufre2=new Gaufre("bio");
        gaufre2.setGarniture("marron");
        gaufre2.afficher();
    }
}
```

Le mot clé « public » est obligatoire pour le « main ». On le met aussi sur la classe.

## codage Python de l'encapsulation

- Le python de gère pas l'encapsulation. Il n'y a pas de mot clé « private » et « public » ou équivalent.
- On retrouve donc le même code que celui déjà présenté mais on crée un fichier avec la classe et un fichier avec le main : le main importe le fichier avec la classe :

```
from Gaufre import *   # on importe le fichier de la classe Gaufre
```

## L'art du programmeur objet en Python

- Il consiste à « faire comme si » : faire comme si les attributs étaient encapsulés par des méthodes !
  - On évite de manipuler directement les attributs.
  - On évite les méthodes setAttribut (attribut) inutiles.
- Il consiste aussi à profiter judicieusement des facilités offertes par le Python ! Autrement dit, à ne pas appliquer à la lettre le principe d'encapsulation.

## Catégorie de méthodes

- Les méthodes peuvent être classées en différents types :

  - **Les constructeurs** : pour créer les objets (création).
  - **Les getter**: (ou accesseurs) pour renvoyer l'état d'un objet (consultation).
  - **Les setter** : (ou mutateurs) pour modifier l'état d'un objet (modification).
  - **Les responsabilités ou rôles** : ces méthodes correspondent aux fonction de haut niveau permettant la réalisation des fonctionnalités de l'objet.

- En phase de **conception**, on s'intéresse surtout aux **responsabilités** de la classe. C'est ce qui caractérise le mieux la classe : la voiture peut démarrer, s'arrêter, accélérer, etc. On peut aussi changer la couleur et on aura un setCouleur( couleur ). A noter qu'il vaudrait mieux avoir un changerCouleur( couleur ) qui serait une responsabilité comme une autre.

- On voit donc qu'on retrouve la programmation procédurale. Il s'agit trouver les bonnes fonctions-méthodes. Mais on commence par définir les objets pour ensuite trouver les bonnes fonctions-méthodes qu'on va leur appliquer.

## Exercices – Classe – Constructeur – Main - Séquence 2 – Slide 22

### Principes

- Pour tous les exercices, il d'agit de créer des classes avec un constructeur et d'écrire le main en utilisant ces classes. On précise bien les paramètres des méthodes.
- On fait le diagramme de classes avec le constructeur et les paramètres des méthodes.
- On écrit le code du main.
- On écrit les codes des méthodes.
- L'écriture du code peut se faire en pseudo-code, en Java ou en Python !

- Encapsulation : précisez si les attributs et les méthodes sont private (-) ou public (+).
- Bien réfléchir à la logique d'encapsulation « **au sens large** » : avoir les bonnes méthodes pour réaliser les fonctionnalités de la classe.

### 1 - Feu tricolore

- On reprend la description de la séquence 1 :
- Un feu de circulation (ou feu tricolore ou feu rouge) est un dispositif d'éclairage : la boite avec ses différentes ampoules. On veut simuler son fonctionnement.
- On considère que le feu marche tout le temps. Il est forcément allumé et dans un certain état : vert, orange ou rouge.
- On simplifie donc les choses : il n'y a pas de orange clignotant et pas de feu éteint. Le feu passe forcément du vert au orange, du orange au rouge et du rouge au vert.
- On ajoute la précision suivante :
- On considère qu'à la création, le feu est rouge, ce qui est plus prudent !
- On veut créer un main qui permet de :
  - Créer un feu.
  - Afficher la couleur du feu.
  - Changer la couleur et l'afficher.
  - Répéter l'opération jusqu'au feu rouge.
  - Faire ensuite une boucle qui répète le changement et l'affichage du feu rouge au feu rouge.
- Pour cet exercice, on ne rajoute pas de nouvelle méthode. Par contre, on peut utiliser une fonction d'affichage directement dans le main (print pour le Python, System.out.println pour le Java).

## 2 - Télévision

- On reprend la description de la séquence 1 :
- Une télévision a une marque et une diagonale. Elle peut être allumée ou éteinte. On peut changer de chaine en donnant le numéro de la chaine ou en montant ou descendant le numéro en cours. On peut monter ou baisser le son. On peut aussi couper le son ou le remettre, avec le même bouton. Quand on remet le son, il revient à son niveau avant d'avoir coupé le son. On peut connaître le niveau du son ainsi que la chaine actuellement sélectionnée.
- On ajoute la précision suivante :
- A l'allumage, le son est à 3 par défaut et la chaine à 0 par défaut.
- On veut créer un main qui permet de :
  - Créer une télévision, l'allumer et afficher la chaine au démarrage et le niveau du son au démarrage. Afficher aussi toutes les informations.
  - Mettre la 9ème chaine et monter le son 3 fois. Afficher le numéro de la chaine et le niveau du son. Afficher aussi toutes les informations.
  - Couper le son. Afficher le numéro de la chaine et le fait que le son est coupé. Afficher aussi toutes les informations.
  - Remettre le son. Descendre de 2 chaines. Baisser le son 2 fois. Afficher le numéro de la chaine et le niveau du son. Afficher aussi toutes les informations.

## 3 – Voiture

- On reprend la description de la séquence 1 :
- Une voiture a une marque, un modèle, une immatriculation. Elle peut avoir 2 portes, 4 portes ou 5 portes.
- On peut mettre le moteur de la voiture en route et passer des vitesses. Le passage de vitesse se fait forcément de 1 en 1, en montant ou en descendant. On peut aussi passer au point mort à tout moment. On peut aussi freiner ou accélérer.
- On veut créer un main qui permet de :
  - Créer une voiture. Afficher ses caractéristiques.
  - Démarrer. Passer en 1ère. Accélérer. Passer en 2ème. Accélérer. Passer en 3ème. Afficher la vitesse. Afficher toutes les caractéristiques de la voiture.
  - Passer en 1ere. Afficher la vitesse. Afficher toutes les caractéristiques de la voiture.
  - Freiner. Passer au point mort. Afficher la vitesse. Afficher toutes les caractéristiques de la voiture.

## 4 - Afficheur de score

- On reprend la description de la séquence 1 :
- Afficheur de score est un objet qui affiche des scores de matchs de football.
- Par exemple :
  - Pour les matchs à venir, on affiche : Paris – Marseille : à venir
  - Pour les matchs en cours, on affiche : Paris – Marseille : 1-0 - en cours
  - Pour les maths finis, on affiche : Paris – Marseille : 3-2 - terminé
- L'affichage commence toujours par le nom de l'équipe qui reçoit, suivi de l'équipe qui se déplace.
- Il s'agit d'un match sans prolongation ni tirs au but pour nous simplifier la tâche.
- L'afficheur permet de faire évoluer le score au fur et à mesure de la partie.
- Le score ne peut changer que pendant le match, ni avant, ni après.
- On initialise l'objet en lui donnant le nom des deux équipes. Automatiquement, le match est « à venir ».
- On veut créer un main qui permet de :
- S'occuper d'un match Paris – Marseille.
- Créez l'afficheur correspondant et affichez-le.
- Faites démarrez le match et affichez la situation.
- Paris marque. Marseille égalise. Paris marque à nouveau. Codez et affichez la situation au fur et à mesure.
- Le math est fini. Codez la situation et affichez la situation.

## 5 - Distributeur de boissons

- On reprend la description de la séquence 1 :
- Un distributeur automatique permet de choisir une boisson au choix. On peut préciser le niveau de sucre qu'on souhaite avoir (entre 0 et 4, 2 par défaut). Un fois la boisson versée, le distributeur rend la monnaie. Le client peut demander la restitution de l'argent qu'il a déjà mis.
- Le fonctionnement est le suivant : quand on clique sur un bouton de boisson, la boisson se préparer si montant fourni est suffisant.
- Quand on a choisi sa boisson et qu'elle se prépare, le distributeur affiche les informations de boisson (son nom et le niveau du sucre) ainsi que le prix, le montant payé et la monnaie prévue.
- On simulera uniquement deux boissons : la boisson numéro 1 : un café à 40 centimes et la boisson numéro 2 : un chocolat à 50 centimes.
- On veut créer un main qui permet de :
- Commander un café. C'est la boisson n°1. Elle coûte 40 centimes. On veut un café sans sucre.
- On met 20 centimes, on choisit le niveau de sucre puis on commande le café.
- On rajoute 50 centimes puis on commande le café.

- Vous voulez un autre café normalement sucré. Après avoir entré 30 centimes, vous réalisez qu'il vous en manque dix. Vous récupérer votre argent. Affichez la situation.
- Le but est d'afficher :

```
Le distributeur encaisse 20 centimes.
niveau de sucre baissé : 1
niveau de sucre baissé : 0
Le distributeur prépare la boisson n°1
Le montant encaissé est trop faible
Le distributeur encaisse 50 centimes.
Le distributeur prépare la boisson n°1
café sucre=0 en cours de préparation
Prix:40 - payé:70 - rendu:30

Le distributeur encaisse 10 centimes.
Le distributeur encaisse 20 centimes.
Le distributeur rend 30 centimes.
```

# 3 : AGREGATION – COMPOSITION – - ENUMERATION - TESTS UNITAIRES

## Relations entre les classes : 1 - Agrégation

### Présentation des relations (1ère fois)

- Les objets, et les classes qui leur correspondent, peuvent avoir des relations entre elles.
- Il y a principalement **4 types de relations** :
  - **L'agrégation**
  - **La composition**
  - **L'énumération**
  - **L'héritage**
- Agrégation et composition sont 2 relations assez semblables.
- L'énumération est une sorte de composition.
- L'héritage est une relation à part.
- On va expliquer ce que sont ces trois relations avec des exemples. Il faut déjà retenir le vocabulaire.

## Exemple d'agrégation : un répertoire de personnes

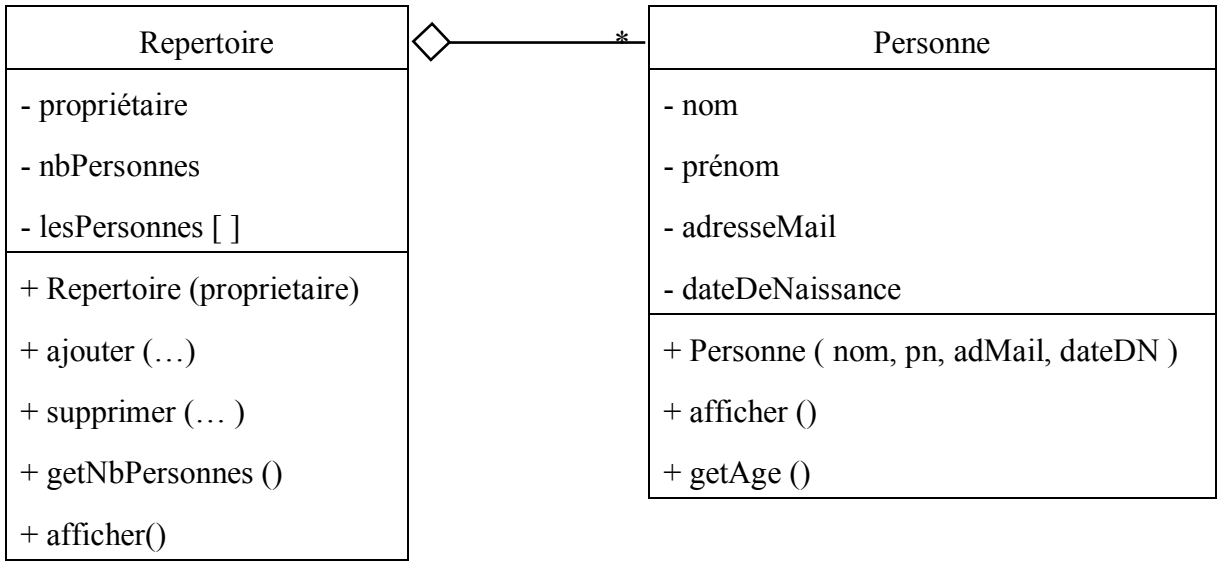
### Présentation de l'exemple

- On veut manipuler dans notre programme un répertoire de personnes.
- Le **répertoire** a un **propriétaire** identifié par son nom. Le propriétaire est une information fixée à la création et non modifiable. Il contient des personnes.
  - On peut **ajouter** et **supprimer** des personnes dans le répertoire.
  - On peut aussi **modifier** les informations d'une personne.
  - On peut récupérer le **nombre de personnes**.
  - On peut **afficher** la liste de toutes les personnes du répertoire avec le nom du propriétaire et le nombre de personnes. On affiche une ligne par personne avec toutes les informations de la personne et son âge plutôt que sa date de naissance.
- Une personne a un nom, un prénom, une adresse mail et une date de naissance.
  - On peut **afficher** les informations de chaque personne (toutes les informations sur une seule ligne, avec son **âge** plutôt que sa date de naissance).
- On va **écrire un programme** qui permet de :
  - Créer et afficher un répertoire dont vous êtes le propriétaire.
  - Mettre 3 personnes dedans et afficher le répertoire.
  - Supprimer la deuxième personne créée puis afficher le répertoire.

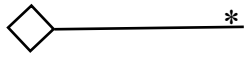
**Exemple d'affichage des résultats attendus :**

```
Affichage du répertoire après la création du répertoire :
propriétaire : Bertrand : 0 Personnes

Ajout de 3 personnes et affichage du répertoire :
propriétaire : Bertrand : 3 Personnes
toto - lolo - toto@gmail.com - 19
titi - lili - titi@gmail.com - 19
tutu - lulu - tutu@gmail.com - 18

Affichage du répertoire après suppression :
propriétaire : Bertrand : 2 Personnes
toto - lolo - toto@gmail.com - 19
tutu - lulu - tutu@gmail.com - 18
```

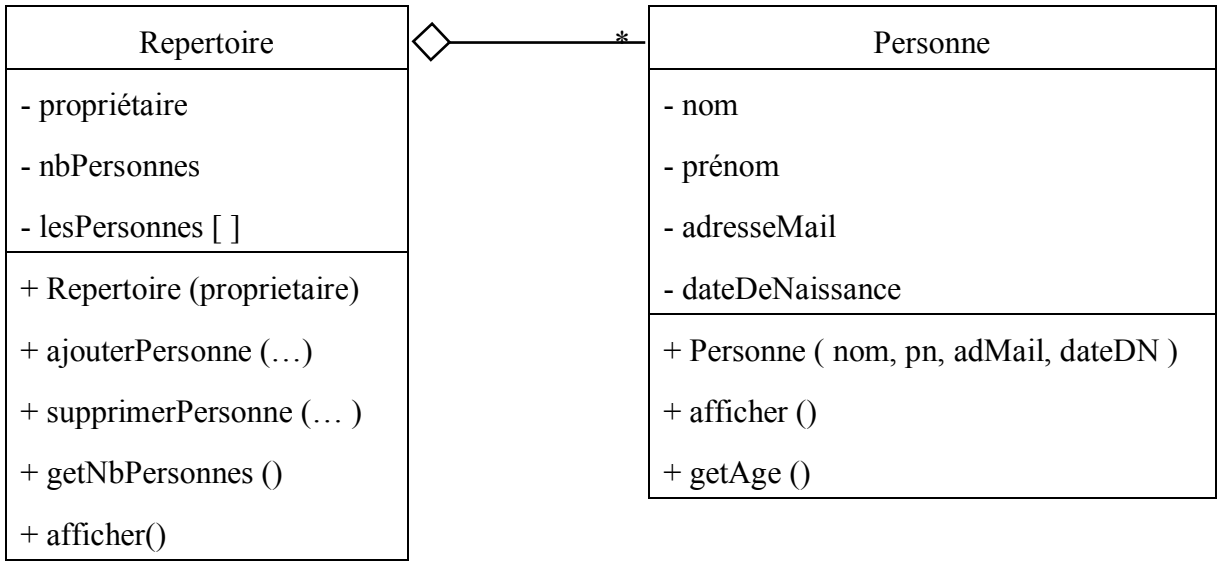
**Diagramme de classes**




- La classe répertoire contient une collection de personnes. Collection = plusieurs, un tableau ou une liste par exemple.
- On appelle cette collection : « lesPersonnes ». C'est une convention simple : « les » pour la collections, suivi du nom de la classe. Des crochets « [ ] » pour montrer que c'est une collection.
- On met les paramètres des méthodes si c'est facile, rien s'il n'y en a pas, « … » si on ne sait pas encore.
- La relation entre Repertoire et Personne matérialisée graphiquement par le lien :



## Principes de l'agrégation




- Le répertoire **agrège** des personnes. Cela veut dire qu'il **contient une liste** de personnes.
- C'est une **agrégation** et pas une composition. Cela veut dire que **si on supprime le répertoire**, on ne doit **pas forcément supprimer les personnes** : elles sont peut-être utilisées dans un autre répertoire.
- Graphiquement :
  - on met un losange du côté de **l'agrégat**, le **conteneur** : ◇
  - on met une étoile du côté des **éléments agrégés**, les **contenus** : « * ». « * » veut dire plusieurs : le répertoire agrège (contient) plusieurs personnes.

## Codage de l'agrégation

**La classe Personne : Rien à signaler, ni en pseudo-code, ni en Java, ni en python**

**La classe Repertoire : pseudo-code**

- ***Surcharge de la méthode ajouter()***

On va avoir **2 méthodes ajouterPersonne() avec des paramètres différents**. C'est ce qu'on appelle une « **surcharge** ». C'est très pratique.

**Première version :** on passe les informations d'une personne et on instancie la personne dans la méthode :

```
ajouterPersonne( nom, prenom, adMail, annee ) :
    lesPersonnes.add( new Personne(nom, prenom, adMail, annee) )
```

**Deuxième version :** on passe une personne en paramètre :

```
ajouterPersonne( Personne p ) :
    lesPersonnes.add(p)
```

A noter qu'on se dote d'une méthode « add » qui permet d'ajouter un élément dans la collection.

- ***méthode supprimerPersonne( Personne p)***

Pour la méthode supprimer(), on passe en paramètre l'objet personne qu'on veut supprimer.

```
supprimerPersonne(Personne p) :
    lesPersonnes.remove(p)
```

A noter qu'on se dote d'une méthode « remove» qui permet de supprimer un élément dans la collection.

## La classe Repertoire : code Java

On a une classe Java pour le type de l'attribut lesPersonnes : classe ArrayList. Notez la syntaxe avec < > : private ArrayList<Personne>  **lesPersonnes**;

On initialise l'attribut lesPersonnes à l'instanciation avec un new. Notez la syntaxe.

On a deux méthodes ajouterPersonnes() avec des paramètres différents.

Les méthode add et remove sont du Java.

```
import java.util.ArrayList;
public class Repertoire {
   private String proprietaire;
   private ArrayList<Personne>   lesPersonnes;

   public Repertoire(String proprietaire) {
      this.proprietaire = proprietaire;
      lesPersonnes = new ArrayList<Personne>();
   }
   public void ajouterPersonne(String nom, String prenom, String adMail, int
     annee) {
      lesPersonnes.add(new Personne(nom, prenom, adMail, annee));
   }
   public void ajouterPersonne(Personne p) {
      lesPersonnes.add(p);
   }
…
}
```

## La classe Repertoire : code Python

Il n'y a **pas de surcharge basique** en python.

Dans un premier temps, on ajoute une méthode « **ajouterPersonne2()** » : ce n'est pas pratique. Il y a des solutions techniques plus complexes pour améliorer cette situation.

Le reste est « basique ». On utilise **les [ ] pour avoir la collection** de base du Python qui donne accès aux méthodes **append** (pour add) et remove.

Le fichier doit commencer par importer le fichier de la classe Personne

```
from Personne import *      # on importe le fichier de la classe Personne
class Repertoire:
   def __init__(self, proprietaire):
      self.proprietaire = proprietaire
      self.lesPersonnes = []

def ajouterPersonne(self, nom, prenom, adMail, anneeNaissance):
    self.lesPersonnes.append(Personne(nom, prenom, adMail, anneeNaissance))

def ajouterPersonne2(self, personne): # pas de surcharge simple en Python
    self.lesPersonnes.append(personne)
…
```

## Notion de tests unitaires

### Présentation

- Avec l'agrégation on voit un premier **exemple de code avec 2 classes**. Le programme va utiliser ces deux classes et leurs méthodes.
- Comment vérifier que chaque classe fonctionne correctement ?
- C'est la notion de **tests unitaires**.
- Pour chaque classe on va créer une classe en plus qui sera une classe de « tests unitaires »
- On appelle la classe de tests unitaires « **ClasseTU** ». Cette classe contient un « main » et s'occupe de tester toutes les méthodes de la classe pour vérifier que tout fonctionne.

**Exemple**

- une classe « **PersonneTU** » pour les tests unitaires de la Personne.
- une classe « **RepertoireTU** » pour les tests unitaires du Repertoire. RepertoireTU correspond finalement au programme demandé qui est très simple et ne fait que des petits tests.

## Tests unitaires de PersonneTU

- ***pseudo-code de PersonneTU***

Il faut tester **le constructeur**, **afficher**() et **getAge**(). On peut donc simplement écrire :

```
# Tests unitaires de Personne
personne=Personne("toto", "lolo", "toto@gmail.com", 2000);
personne.afficher();
print("age : "+personne.getAge())
```

- ***Java***

```
public class PersonneTU{
   public static void main (String args[]){
      Personne personne=new Personne("toto", "lolo", "toto@gmail.com", 2000);
      personne.afficher();
      System.out.println(("age : "+personne.getAge());
   }
}
```

- ***Python***

```
from Personne import *      # on importe le fichier de la classe Personne
personne=Personne("toto", "lolo", "toto@gmail.com", 2000);
personne.afficher();
print("age : "+personne.getAge())
```

- ***Conclusion***

C'est presque pareil en pseudo-code, Java ou Python. Il n'y a que la structure qui change.

## Tests unitaires de RepertoireTU

- ***pseudo-code de RepertoireTU***

Il faut tester **le constructeur**, et **toutes les méthodes**. Cela correspond au programme demandé.

```
# Tests unitaires de ReperoireTU
print("Affichage du répertoire après l'instanciation : ")
rep=new Repertoire("Bertrand")
rep.afficher()

print("Ajout de 3 personnes et affichage du répertoire : ")
rep.ajouterPersonne("toto", "lolo", "toto@gmail.com", 2000)
p = new Personne("titi", "lili", "titi@gmail.com", 2000)
rep.ajouterPersonne(p)
rep.ajouterPersonne(new Personne("tutu","lulu","tutu@gmail.com",2001) )
rep.afficher()

print("Affichage du répertoire après suppression : ")
rep.supprimerPersonne(p)
rep.afficher()
```

- ***code Java et code Python***

Il n'y a rien de particulier en plus dans ces codes par rapport au pseudo-code.

## Surcharge

### Principes

- En POO, on peut définir des méthodes dans une même classe avec le même nom mais des paramètres différents : c'est ce qu'on appelle **la surcharge.**
- La surcharge peut s'appliquer au constructeur.
- C'est très pratique et rend l'écriture et l'usage des codes simplifiés.

### Exemples

**Le constructeur de la classe propriétaire**

- Le premier constructeur envisagé crée un répertoire vide :

```
Repertoire (proprietaire)
```

- On pourrait se doter d'un 2ème constructeur permettant de créer un répertoire en le remplissant avec une liste de personnes qui existe déjà :

```
Repertoire (proprietaire, lesPersonnes[] )
```

**La méthode « ajouterPersonne() »**

- Une version avec un objet personne passée en paramètre :

```
ajouterPersonne(personne)
```

- Une version avec les paramètres d'une personne passés en paramètres de la méthode :

```
ajouterPersonne(nom, prenom, adresseMail, anneeNaissance)
```

## Code Java et code Python

### Code Java

- L'écriture se fait « tout naturellement » : il suffit d'écrire un deuxième constructeur ou une deuxième méthode avec les nouveaux paramètres.

### Code Python

- Le python de permet pas de surcharge à proprement parler (2 méthodes avec le même nom et des paramètres différents). Mais il permet d'avoir une méthode avec une liste variable de paramètres :

```
def ajouterPersonne(self, nom=None, prenom=None, adMail=None,
                                    anneeNaissance=None,
        personne=None):
    # usage 1 : ajouterPersonne(nom, prenom, adMail, anneeNaissance)
    # usage 2 : ajouterPersonne(personne=personne)
    if personne:
        self.lesPersonnes.append(personne)
    else:
        self.lesPersonnes.append(Personne(nom, prenom, adMail,
    anneeNaissance))
```

➢ ***usage :***

```
repertoire.ajouterPersonne("toto", "lolo", "toto@gmail.com", 2000);
p = Personne("titi", "lili", "titi@gmail.com", 2000);
repertoire.ajouterPersonne(personne=p);
```

- Notez le (personne = p)
- Le Python offre d'autres façon de gérer des listes de paramètres variables (kwargs).

## Relations entre les classes : 2 - Composition

**Présentation des relations (2ème fois)**

- Les objets, et les classes qui leur correspondent, peuvent avoir des relations entre elles.
- Il y a principalement **4 types de relations** :
  - **L'agrégation**
  - **La composition**
  - **L'énumération**
  - **L'héritage**
- Agrégation et composition sont 2 relations assez semblables.
- L'énumération est une sorte de composition.
- L'héritage est une relation à part.
- On va expliquer ce que sont ces trois relations avec des exemples. Il faut déjà retenir le vocabulaire.

**Exemple de composition : la télévision et ses composants (tuner et ampli)**

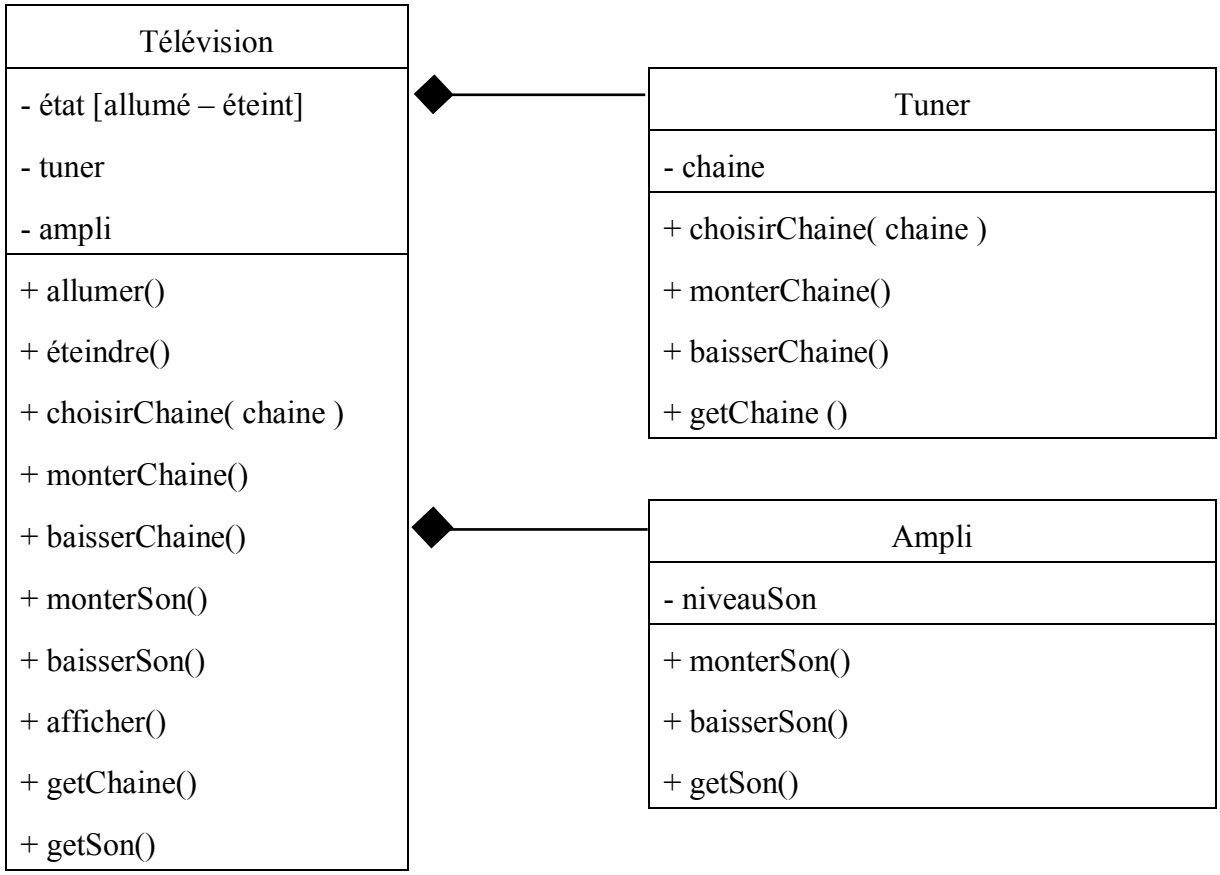
## Présentation de l'exemple

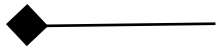
- On veut **manipuler une télévision** (exemple déjà abordé en exercice).
- On peut allumer et éteindre la télé, monter et baisser le son, changer de chaine en donnant le numéro de la chaine, monter ou baisser le numéro de la chaine.
- **La télévision est composée d'un tuner** qui gère les chaines **et d'un ampli** qui gère le son.
- **On veut gérer séparément les objet « tuner » et « ampli »**.
- On veut écrire un programme qui permet de tester la télévision :
  - Afficher l'état général après l'instanciation.
  - Allumer et afficher l'état général.
  - Choisir une chaine (la 5), monter le son 3 fois et afficher l'état général.
  - Monter la chaine 3 fois, baisser la chaine 1 fois, baisser le son et afficher l'état général.
  - Eteindre et afficher l'état général.

## Résultats attendus

```
off : chaine =  1 - son : 3
on : chaine =  1 - son : 3
on : chaine =  5 - son : 6
on : chaine =  7 - son : 5
off : chaine =  7 - son : 5
```

**Diagramme de classes**




- La classe Télévision est composée d'un objet « tuner » et d'un objet « ampli ».
- Cette relation est matérialisée dans le diagramme par le lien : 

**Principes de la composition**

- La télévision est composée d'un ampli et d'un tuner. Cela veut dire qu'elle contient un ampli et un tuner.
- C'est une **composition** et pas une agrégation. Cela veut dire que **si on supprime la télévision**, **on supprime aussi l'ampli et le tuner**. L'ampli et le tuner ne sont utilisés que par la télévision.
- Graphiquement :
  - on met un losange du côté de **l'agrégat**, le **conteneur** :
  - on met une étoile du côté des **éléments agrégés**, les **contenus** : « * ». « * » veut dire plusieurs : le répertoire agrège (contient) plusieurs personnes.
- Graphiquement,
  - on met un losange en noir **du côté du composé** : ◆
  - on ne met **rien du côté du composant**. Le « **rien** » veut dire « **1 seul** » : la télévision est composé d'un seul ampli et d'un seul tuner.

## Codage de la composition

**Les classes Ampli et Tuner : Rien à signaler, ni en pseudo-code, ni en Java, ni en python**

**La classe Television : pseudo-code**

- ***Constructeur : on instancie les composants***

Le constructeur instancie ses composants :

```
Television() :
    etat = eteint
    ampli = new Ampli()
    tuner = new Tuner()
```

- ***Délégation***

Les méthodes correspondant aux composants « délèguent » le travail aux composants :

```
monterSon() :
    ampli.monterSon()

choisirChaine( chaine )
    tuner.choisirChaine (chaine)

getSon( ):
     return ampli.getSon()
etc.
```

**Code Java et code Python**

- Rien de particulier à signaler. Regarder directement l'exemple.

## Code des tests unitaires

### 3 classes à tester

- Ici, on peut faire des tests unitaires sur les 3 classes : Télévision, Ampli et Tuner.
- On peut donc avoir 3 classes de tests unitaires : TelevisionTU, AmpliTU, TunerTU, avec leurs « main » pour faire les tests.
- La classes TelevisionTU correspond au programme de tests qu'on veut écrire dans l'exemple.

### La classe TelevisionTU : pseudo-code

Le pseudo code est très simple et très simple à traduire en Java ou en Python

```
television = Television();
television.afficher();

television.allumer();
television.afficher();

television.choisirChaine(5);
television.monterSon();
etc.
```

## Relations entre les classes : 3 - Enumération

**Présentation des relations (3è fois)**

- Les objets, et les classes qui leur correspondent, peuvent avoir des relations entre elles.
- Il y a principalement **4 types de relations** :
  - **L'agrégation**
  - **La composition**
  - **L'énumération**
  - **L'héritage**
- Agrégation et composition sont 2 relations assez semblables.
- L'énumération est une sorte de composition.
- L'héritage est une relation à part.
- On va expliquer ce que sont ces trois relations avec des exemples. Il faut déjà retenir le vocabulaire.

## Classe énumération

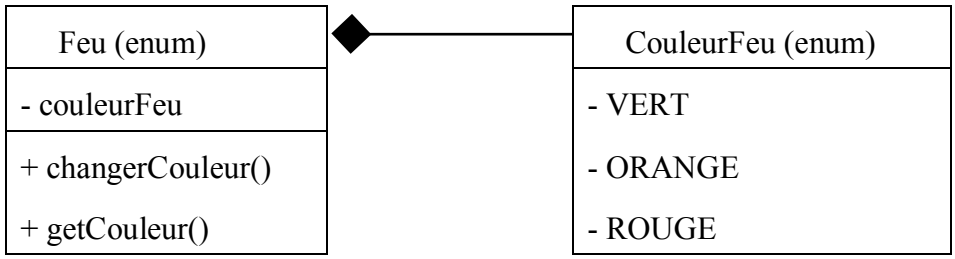
### Présentation

- En POO, on utilise souvent des classes d'énumération qui nous permettent de définir des petites listes de valeurs constantes.
- Ces valeurs sont utilisées pour fixer les valeurs possibles pour un attribut et ainsi éviter les erreurs.
- L'énumération devient alors un type pour un attribut.

### Technique

- On définit l'énumération comme une classe particulière : une classe d'énumération.
- Les attributs de l'énumération sont les valeurs possibles.
- Il n'y a pas de méthodes.

### Exemple



**Code Python**

```
from enum import Enum
class CouleurFeu(Enum):
    VERT = 0;
    ORANGE = 1;
    ROUGE = 2;
```

```
class FeuTricolore:
    def __init__(self):
        self.couleur = CouleurFeu.ROUGE;
    def changerCouleur(self):
        if self.couleur == CouleurFeu.VERT :
            self.couleur = CouleurFeu.ORANGE;
        elif …
```

```
# main
feu = FeuTricolore();
print("Couleur : ", feu.getCouleur());
if(feu.getCouleur()==CouleurFeu.ROUGE):
    print("Le feu est rouge, il faut s'arrêter")
```

**Code Java**

```
public enum CouleurFeu{
  VERT,
  ORANGE,
  ROUGE
}
```

```
public FeuTricolore(){
  this.couleur = CouleurFeu.ROUGE;
}

public void changerCouleur(){
  if (this.couleur == CouleurFeu.VERT){
    this.couleur = CouleurFeu.ORANGE;
  }else if …
```

```
# main
feu = new FeuTricolore();
System.out.println("Couleur : ", feu.getCouleur());
if(feu.getCouleur()==CouleurFeu.ROUGE)
      System.out.println ("Le feu est rouge, il faut s'arrêter")
```

## Exercices - Séquence 3 – Slide 23

### Agrégation et composition

**La voiture**

- On reprend un exercice précédent.
  - Une voiture a une marque, un modèle, une immatriculation. Elle peut avoir 2 portes, 4 portes ou 5 portes.
  - On peut mettre le moteur de la voiture en route et passer des vitesses. Le passage de vitesse se fait forcément de 1 en 1, en montant ou en descendant. On peut aussi passer au point mort à tout moment. On peut aussi freiner ou accélérer.
- On ajoute les éléments suivants :
  - La voiture a un moteur qui a un nom et une puissance.
  - La voiture a des roues (2 roues avant, 2 à l'arrière et 1 de secours). Chaque roue a un type de pneu, une marque de pneu, et le pneu peut être neuf ou usagé ou crevé. La position des roues sera gérée avec une énumération.
  - On veut pouvoir changer une roue en cas de crevaison.
  - On veut pouvoir récupérer l'état de sa roue de secours.
  - La voiture peut « crever ».
- On testera la voiture ainsi :
  - On crée une voiture (une Renault Espace 5 portes moteur v6 245 chevaux avec des pneus Dunlop de type mixte, tous neufs). On affiche ses caractéristiques.
  - On démarre. On passe les vitesses jusqu'en 3ème. On affiche la vitesse en cours à chaque changement de vitesse.
  - On crève la roue avant droite. On freine et on passe au point mort. On affiche les caractéristiques de la voiture.
  - On change la roue crevée. On affiche les caractéristiques de la voiture.

**Personnage de jeu vidéo**

- On reprend un exercice précédent.
  - Un personnage de jeu vidéo a un nom, une force, une localisation, une expérience et des dégâts.
  - Le personnage peut se déplacer et combattre un autre personnage.
- On y ajoute le fait qu'un personnage peut trouver des armes et les accumuler dans un stock. Une arme a un nom, une puissance et une valeur.
- On écrit un programme qui permet de :
  - Créer un joueur et afficher ses caractéristiques avec l'arme qu'il porte (ou pas).
  - Créer 3 armes (on considère qu'elles sont sur le terrain et que le joueur les trouve).
  - Le joueur ajoute ces 3 armes à son stock. Affichez le stock.
  - Le joueur sélectionne la dernière arme dans son stock et porte cette arme (quand il porte une arme, elle sort de son stock). Affichez les caractéristiques du joueur avec l'arme qu'il porte et affichez le stock de ses armes (l'arme qu'il porte n'est plus dans le stock).
  - Le joueur veut changer d'arme. Il sélectionne à nouveau la dernière arme dans son stock (ce n'est pas la même que précédemment) et remplace l'arme qu'il porte par ce nouveau choix.
  - Le joueur combat puis il range son arme. Affichez les caractéristiques du joueur et son stock d'armes.
  - Le joueur se sépare de la première arme de son stock. Affichez le stock.

**Distributeur**

- On reprend un exercice précédent.
- On va gérer une classe boisson séparément du distributeur. On gère 2 boissons : un café à 40 centimes et un chocolat à 50 centimes.
- Le but est d'obtenir les résultats suivants :

```
Le distributeur encaisse 20 centimes.
niveau de sucre baissé : 1
niveau de sucre baissé : 0
Le distributeur prépare la boisson n°1
Le montant encaissé est trop faible
Le distributeur encaisse 50 centimes.
Le distributeur prépare la boisson n°1
café sucre=0 en cours de préparation
Prix:40 - payé:70 - rendu:30

Le distributeur encaisse 10 centimes.
Le distributeur encaisse 20 centimes.
Le distributeur rend 30 centimes.

Le distributeur encaisse 50 centimes.
Le distributeur prépare la boisson n°3
La boisson 3 n'est pas disponible.
Le distributeur prépare la boisson n°2
chocolat sucre=2 en cours de préparation
Prix:50 - payé:50 - rendu:0
```

# 4 : HERITAGE – POLYMORPHISME – REFERENCE STATIC – CLASSE ABSTRAITE - INTERFACE

## Relation entre classes : 4 - L'héritage

### Présentation des relations (4ème fois !)

- Les objets, et les classes qui leur correspondent, peuvent avoir des relations entre elles.
- Il y a principalement **4 types de relations** :
  - **L'agrégation**
  - **La composition**
  - **L'énumération**
  - **L'héritage**
- Agrégation et composition sont 2 relations assez semblables.
- L'énumération est une sorte de composition.
- L'héritage est une relation à part.
- On va expliquer ce que sont ces trois relations avec des exemples. Il faut déjà retenir le vocabulaire.

**Exemple conduisant à l'héritage**

- On a une classe Chat et une classe Chien :

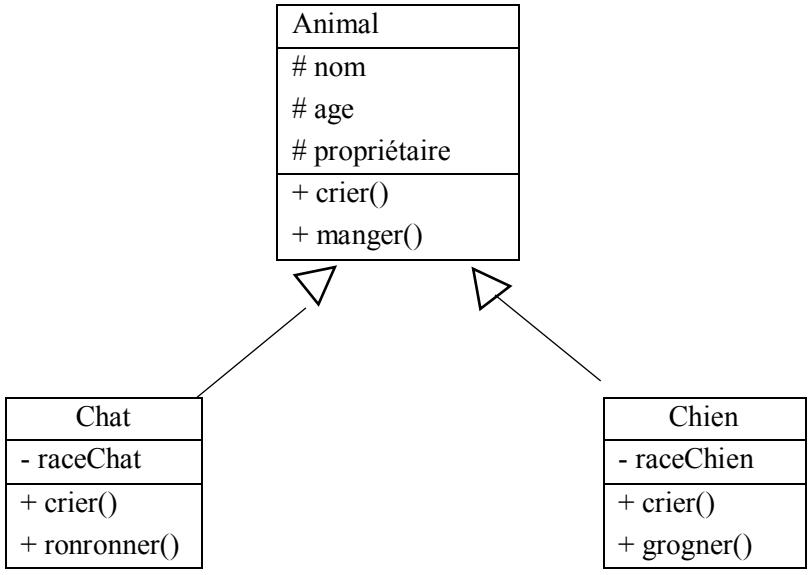
| Chat |
|---|
| - nom<br>- age<br>- propriétaire<br>- raceChat |
| + crier()<br>+ manger()<br>+ ronronner() |

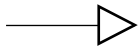
| Chien |
|---|
| - nom<br>- age<br>- propriétaire<br>- raceChien |
| + crier()<br>+ manger()<br>+ grogner() |

- On voit que les 2 classes ont des attributs et des méthodes en commun.
- Concernant les méthodes, on peut considérer que manger() est la même méthode pour le chat et pour le chien.
- Par contre, crier() est différente : le chat miaule et le chien aboie !

**Principe de l'héritage : mettre le commun dans une classe de base**

- **Dès que 2 classes ont des attributs ou des méthodes en commun, on retire ces attributs et méthodes de leurs classes de départ pour créer une nouvelle classe avec les attributs et méthodes communes.** On fait ensuite « hériter » les classes de départ de la nouvelle classe.
- Les classes Chat et Chien héritent de la classe Animal :



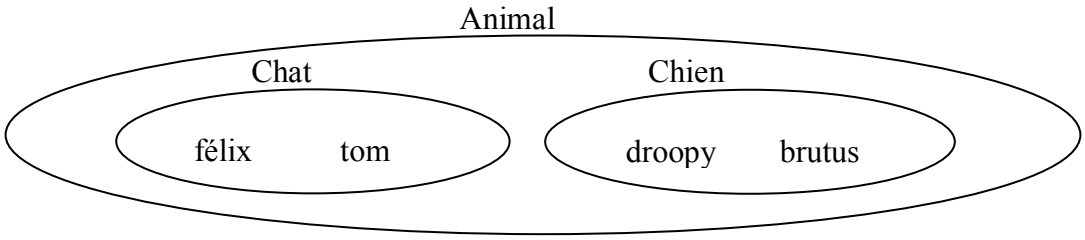

- La nouvelle classe est appelée : **classe de base** (ou **classe-mère** ou **classe-parent**).
- Les classes de départ sont appelées : **classe dérivée** (ou **classe-fille** ou **classe-enfant**).
- On dit que **les classes-enfant « héritent » de la classe parent.** En effet, tous les attributs et les méthodes de la classe-parent seront accessibles par la classe enfant.
- La flèche avec un triangle blanc au bout est une flèche d'héritage : 

**Inclusion ensembliste, relation « est-un »**

## Inclusion ensembliste

L'inclusion ensembliste est ce qui traduit le mieux la notion d'héritage.

Un ensemble correspond à une classe. Un élément d'ensemble correspond à un objet. Un ensemble inclut dans un autre correspond à un héritage.




## Relation « est un »

L'héritage, la flèche à triangle, l'inclusion sont des **relations « est un »**.

Un Chat « est un » animal. Félix « est un » Chat. félix est un Animal. Etc.

## Utilité

L'héritage est utilisé pour :

- **Factoriser le code** : ça évite d'avoir à écrire deux fois la même chose !
- **Utiliser un code existant déjà** : c'est une forme de factorisation.
- **Le polymorphisme** (possibilité qu'une même méthode se réalise différemment).

**Mise en œuvre technique**

**Encapsuation : 1 nouveau niveau de visibilité : protected**

| Symbole | Mot-clé | Signification |
|---|---|---|
| - | private | Visible uniquement **dans la classe** |
| # | protected | Visible dans la classe et **dans les classes dérivées** |
| + | public | Visible **partout** |

**Redéfinition de méthode**

- Dans la classe dérivée, **on peut redéfinir une méthode déjà présente** dans la classe de base**.**
- **Il vaut mieux n'hériter que d'une classe** pour éviter les incohérences. Une classe a un seul parent direct et pas plusieurs. **On évite donc l'héritage multiple** (interdit en Java, possible en Python).

**Constructeur**

- Constructeur de Animal : **Animal** ( nom, age, propriétaire )
- Constructeur de Chat : **Chat** ( nom, age, propriétaire, raceChat )
- Constructeur de Chien : **Chien** ( nom, age, propriétaire, raceChien)

Le constructeur de l'enfant (le chat, le chien) reprend les attributs du parent (l'animal).

## Principes du codage de l'héritage

### Dire le lien d'héritage : faire le lien entre la classe enfant et la classe mère

- Java :

```
public class Chat extends Animal {
}
```

- Python :

```
class Chat(Animal) :
```

### Encapsulation

- En Java, les attributs sont souvent « protected ».
- En Python, il n'y a pas d'encapsulation.

### Accéder à une méthode du parent : mot clé « super »

- On utilise le **mot clé « super »** pour accéder à une méthode ou au constructeur de la classe parent.
- Pour utiliser la méthode afficher() de la classe Animal dans une méthode de la classe Chat, on écrit : **super.afficher()**
- Il y a des variantes entre le Java et le Python.

**Code des classes**

**La classe Animal : rien à signaler, ni en pseudo-code, ni en Java, ni en python**

**Les classes Chat et Chien : pseudo-code**

- ***Constructeur : on instancie le parent***

```
Chien (self, nom, age, proprietaire, raceChien):
    super(nom, age, proprietaire)  // on instancie le parent, l'Animal
    this.raceChien = raceChien
```

- ***Appel à une méthode du parent : afficher()***

```
afficher() : // afficher du Chien
    super.afficher()          // on appelle la méthode afficher du parent, l'Animal
    print("Je suis un chien de race : "+this.raceChien)
```

**Code Java**

- Rien de particulier par rapport au principe général. Regarder le code complet.

**Code Python**

- En Python, l'instanciation du parent se fait par sa méthode __init__
- En Python, le mot clé super est écrit avec des parenthèses.
  - Exemple de la classe Chat :

```
class Chat(Animal):
    def __init__(self, nom, age, proprietaire, raceChat):
        super().__init__(nom, age, proprietaire)
        self.raceChat = raceChat
    def afficher(self):
        super().afficher()
        print("Je suis un chat de race : "+self.raceChat)
    def crier(self):
        print("miaou")
    def ronronner(self):
        print("ronron")
```

**Tests Unitaires**

- Pour tester les classes, on va instancier un objet de chaque classe et faire quelques appels de méthodes.
- On applique les méthodes possibles selon la classe instanciée.

```
animal=Animal("Marsupilami", 67, "André Franquin")
animal.afficher()

felix=Chat("Felix",100, "Otto Mesmer", "de gouttière")
felix.afficher()
felix.crier();
felix.ronronner();

droopy=Chien("Droopy", 82, "Tex Avery", "Basset Hound")
droopy.afficher()
droopy.crier();
droopy.grogner();
```

## Polymorphisme

### Principe de substitution

- Dans le code, **si on remplace un objet d'une classe parent par un objet d'une classe enfant, ça va continuer à marcher** (compilation et exécution).
- En effet, **toutes les méthodes accessibles par un objet d'une classe parent seront accessibles par un objet d'une classe enfant**, et donc le code continuera à marcher.
- Cette possibilité de substitution (remplacement), **c'est ce qu'on appelle le principe de substitution**.

**Principe de substitution :**

On peut substituer :

un **objet d'une classe mère** (ou classe de base)

par

un **objet d'une classe enfant** (ou classe dérivée ou sous-classe).

C'est ce qui permettra le polymorphisme.

## Polymorphisme

### Principes

- Il y a polymorphisme quand la méthode appelée est différente selon l'objet effectivement présent.
- Le fait que la méthode soit différente fait que le comportement du programme change bien que le code écrit reste identique.
- Ce changement de comportement, c'est ce qu'on appelle du polymorphisme ou un comportement polymorphe
- C'est rendu possible par le principe de substitution.

### Par exemple,

- si on écrit : **animal.afficher( )**, on pense que c'est la **méthode afficher() de la classe animal** qui va s'exécuter.
- Mais si animal est en réalité un chat, ce sera la méthode afficher() de la classe Chat qui va s'exécuter.
- Et si animal est en réalité un chien, ce sera la méthode afficher() de la classe Chien.
- C'est ce qu'on appelle du polymorphisme ou un comportement polymorphe.

**Code polymorphe**

```
// on a trois objets : marsupilami qui est un animal, felix un chat, droopy un chien
tabAnimaux=[]
tabAnimaux.append(marsupilami)
tabAnimaux.append(felix)
tabAnimaux.append(droopy)
pour i de 1 à taille(tabAnimaux)
   print("----------------------")
   tabAnimaux[i].afficher()     // instruction polymorphe
   tabAnimaux[i].crier();       // instruction polymorphe
```

**Résultats d'un code polymorphe**

```
nom: Marsupilami - age: 67 - proprietaire: André Franquin
je crie !!!
----------
nom: Felix - age: 100 - proprietaire: Otto Mesmer
Je suis un chat de race : de gouttière
miaou
----------
nom: Droopy - age: 82 - proprietaire: Tex Avery
Je suis un chien de race : Basset Hound
ouah
```

## Objet et référence

### Identité de l'objet – référence de l'objet

- Chaque objet possède une **identité** attribuée de manière implicite à la création de l'objet et qui n'est jamais modifiée (c'est son **adresse en mémoire**).
- De ce fait, chaque objet a forcément une identité distincte d'un autre objet et est donc forcément différent d'un autre objet.
- On accède à l'identité de l'objet par une variable avec un nom qu'on appelle aussi référence.
- **Au sens strict, l'objet, c'est l'identité.** La référence c'est une variable qui amène à l'objet.

| REFRENCE | | IDENTITE |
|---|---|---|
| nom_1 | ⟶ | un objet en tant qu'identité |
| nom_2 | ⟶ | un autre objet en tant qu'identité |

- Techniquement, l'objet c'est la mémoire qui permet de stocker l'état et les comportements de l'objet. L'identité de l'objet c'est cette mémoire.
- Notons que deux objets différents ayant une identité différente peuvent avoir le même état.

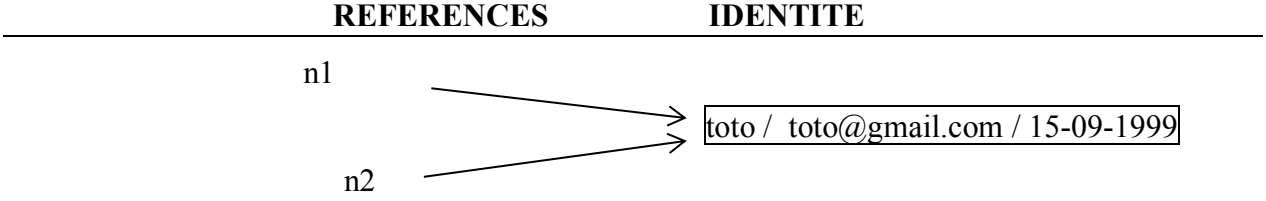
## Une identité unique – plusieurs références possibles

### Références possible d'un objet

- Un objet peut être référencé par une variable.
- Un objet peut être référencé par un éléments d'un tableau.
- Un objet peut être référencé par un attribut d'un autre objet

### Références multiples d'un même objet

- On peut avoir plusieurs références pour un même objet.
- Il peut y avoir plusieurs variables qui font référence au même objet.
- On peux avoir deux références n1 et n2 pour un même objet (une même identité) :




➢ Exemple de code aboutissant à ça :

```
n1 = new Personne(« toto », « toto@gmail.com », « 15-09-1999 »)
n2=n1
```

Ici, on n'a pas dupliqué l'objet en faisant n2=n1. On s'est juste doté d'une 2ème référence vers l'objet en tant qu'identité.

Pour dupliquer l'objet, il faut faire une nouvelle instanciation.

## Attributs et méthodes de classes

### Présentation

- Les attributs et les méthodes vues jusqu'à présent sont des attributs et des méthodes d'instance : ils s'appliquent à chaque objet instancié d'une classe.
- Les attributs et les méthodes de classes sont des attributs et des méthodes qui sont définis au niveau d'une classe et qui n'ont pas besoin d'un objet pour être utilisés.

### Les deux usages

- On peut définir 2 usages des attributs et des méthodes de classes :

**1 : propriété concernant toute la classe**

- Attributs et méthodes de classe permettent de **définir des caractéristiques générales à la classe** qui sert normalement à instancier des objets.
- Par exemple, on peut définir un attribut de classe qui donnerait le nombre d'objets instanciés pour une classe et sa méthode de classe associée qui « get » cet attribut.

**2 : bibliothèque dans une classe sans objet**

- Attributs et méthodes de classes permettent de **définir des fonctions qui seront utilisées indépendamment de tout objet de la classe.**
- Dans ce cas, la classe ne sert pas à instancier des objets mais uniquement à regrouper des méthodes utilitaires et des attributs qui seront des constantes : c'est une **bibliothèque de fonctions** sur le même thème qui sont regroupées dans une classe.

**Exemple usage 1 en Java : mot clé « static »**

```
public class Gaufre {
    // attributs et méthodes de classe
    private static int nbGaufres=0;
    public static int getNbGaufres(){
        return nbGaufres;
    }
    // attributs et méthodes d'objet (ou d'instance)
    private String pate;
    private String garniture;
    ...
    public static void main (String args[]) {
        Gaufre gaufre1=new Gaufre("standard");
        gaufre1.afficher();
        System.out.println(Gaufre.getNbGaufres()+" gaufre(s) instanciées");

        Gaufre gaufre2=new Gaufre("bio");
        gaufre2.afficher();
        System.out.println(Gaufre.getNbGaufres()+" gaufre(s) instanciées");
    }
}
```

## Exemple usage 2 en Python : décorateurs @classmethod et @staticmethod

```
import sys
class Comparaisons :
  seuil = 10;                  # attribut de classe
  @classmethod                 # methode de classe, technique 1
  def min(cls, a, b) :         # cls (classe) plutot que self
    print("min2:"+str(cls))  # pour voir le contenu de cls
    if (a < b):
      return a;
    else:
      return b;
  @staticmethod                # methode de classe, technique 2, pas de cls
  def grand(a) :               # pas de cls dans les paramètres
     return (int(a) > Comparaisons.seuil);
# main
n = int(sys.argv[1]);
p = int(sys.argv[2]);
print("Le plus petit des 2 premiers : " + str(Comparaisons.min(n, p)));
if (Comparaisons.grand(n)):
  print(str(n) + " est grand que le seuil qui vaut " + str(Comparaisons.seuil));
```

# Classe abstraite - Interface

## Classe abstraite

### Définition

- Une classe abstraite est une classe qui ne peut pas être instanciée : on ne peut pas créer d'objet correspondant à cette classe. Ce n'est donc pas un moule !

### A quoi sert-elle ?

- Elle sert en tant que classe mère. Si une autre classe hérite d'une classe abstraite, cette classe enfant pourra, elle, être instancié. **La classe abstraite devient une partie du moule** de la classe enfant.

### Pourquoi faire ?

- Si on reprend le modèle Animal – Chat – Chien, on peut considérer que créer un objet animal n'a pas de sens : c'est trop général. Pourtant, la classe Animal est utile pour factoriser ce qui est commun.
- La classe Animal pourrait être rendue abstraite pour éviter qu'on puisse créer des objets à partir de cette classe.

### Dans le code

- En Java, on ajoute le mot clé « abstract » devant le mot clé « class ».
- En Python… le Python ne gère pas les classes abstraites. Il faudra faire « comme si ».

## Méthodes abstraites

### Définition

Une méthode abstraite est une méthode qui n'a qu'une en-tête mais pas de corps.

Quand une classe contient une méthode abstraite, elle est forcément abstraite.

### A quoi sert-elle ?

Elle sert à obliger les classes enfants qui hériteront de la classe mère abstraite à définir la méthode abstraite. C'est donc une obligation qui est faite au programmeur qui utilise la classe abstraite.

### Pourquoi faire ?

Si on reprend le modèle Animal – Chat – Chien, on a la méthode crier () qui se trouve dans la classe animal et qui devrait être abstraite.

Il faut avoir cette méthode dans la classe crier () pour pouvoir utiliser le polymorphisme.

## Interface

Les interfaces sont des techniques utiles pour avoir un code le plus générique possible.

Elles ne contiennent que des méthodes abstraites : aucun attributs d'instance.

C'est donc un cas particulier de classe abstraite.

La différence est qu'une classe peut « implémenter » une ou plusieurs interfaces alors que l'héritage à tout intérêt à être unique pour éviter les incohérences.

Ca l'oblige donc à écrire le code des méthodes abstraites qui se trouvent dans l'interface.

C'est une notion complexe qui permet de mettre en œuvre au mieux le polymorphisme et d'avoir un code facile à faire évoluer.

## Conclusion : les 5 concepts fondateurs de la programmation objet

- La POO est basée sur **5 concepts fondateurs** :

1. **L'objet** (encapsulation : les attributs sont accessibles via les méthodes)

2. **Message** (l'échange entre deux objets : l'utilisation des méthodes)

3. **Classe** (la généralisation de l'objet)

4. **Héritage** (la factorisation de propriétés et de fonctions entre 2 classes)

5. **Polymorphisme**

## Exercices - Séquence 4 – Slide 22

### Jeu vidéo

**1)**

- Soit un jeu vidéos avec des joueurs. Les joueurs ont un nom, une santé, une force et une expérience. Ces trois dernières caractéristiques sont des entiers positifs ou nuls. L'expérience vaut 0 quand le personnage débute.
- On se dote d'un constructeur qui permet de donner un nom, une santé et une force au joueur.
- On peut afficher les caractéristiques d'un joueur (son nom et la valeur de tous ses attributs).
- Un joueur peut attaquer un autre joueur. Une attaque réduit la santé du personnage attaqué de 1. Quand la santé vaut 0, le joueur est mort. Une attaque augmente l'expérience de l'attaquant de 2.
- Ecrire un main qui déclare deux joueurs et affiche leur caractéristiques (le deuxième à une santé de 3). Ensuite le main fait attaquer le deuxième joueur par le premier. Afficher les nouvelles caractéristiques des joueurs. Ensuite le premier attaque 2 fois le second. La méthode attaquer affichera le fait que le joueur 2 est mort.

**2)**

- On crée maintenant des ennemis. Les ennemis ont les mêmes caractéristiques que les joueurs mais ils n'ont pas d'expérience et ils ont une race (troll, elfe, sorcier, etc.), et une valeur qui est un entier positif. Quand un joueur attaque un ennemi, il gagne en expérience la valeur de l'ennemi. Quand un ennemi attaque un personnage, il lui fait perdre 1 point de santé.
- Ecrire un *main* qui déclare deux joueurs et un ennemi et affiche leurs caractéristiques. Ensuite le main fait attaquer l'ennemi par le premier joueur. Puis le main fait attaquer le deuxième joueur par l'ennemi. Afficher les nouvelles caractéristiques de l'ennemi et des joueurs.

### Point en couleur

**1)**

- On définit un point dans un repère géométrique par ses coordonnées x et y.
- On veut pouvoir créer plusieurs points et afficher leurs coordonnées.

**2)**

- La classe permettant de faire ça existant, on veut pouvoir gérer des points en couleur, sans modifier la première partie Un programme permettra de créer 2 points en couleur et d'afficher leurs coordonnées et leurs couleurs.

**Figures géométriques**

- On veut gérer des figures géométriques : un rectangle, un carré et un cercle.
- Un cercle est défini par son rayon et son centre qui est un point avec 2 coordonnées x et y
- Le rectangle est défini par son centre et la taille d'un côté horizontal et celle d'un côté vertical. On considère que les rectangles sont forcément posés à l'horizontal.
- Le carré est défini par son centre et la taille de son côté.
- A noter qu'un carré est un rectangle particulier.
- On peut afficher les figures ou les effacer (les rendre invisible). On peut aussi afficher les points ou les effacer.
- On peut récupérer les sommets des carrés et des rectangles qui sont des points.
- On peut déplacer les figures : pour ça on fournit les coordonnées d'un nouveau centre
- On peut changer la taille des figures. Pour cela, une méthode reçoit un coefficient multiplicateur. S'il est compris entre 0 et 1, la figure diminue de taille. S'il est supérieur à 1, la figure augmente de taille.
- Ecrire un programme qui crée un cercle, un rectangle et un carré. Afficher ces figures.
- Afficher les informations de chaque figure.
- Déplacer le rectangle et afficher ses nouvelles informations.
- Affichez les informations des sommes du rectangle après déplacement.