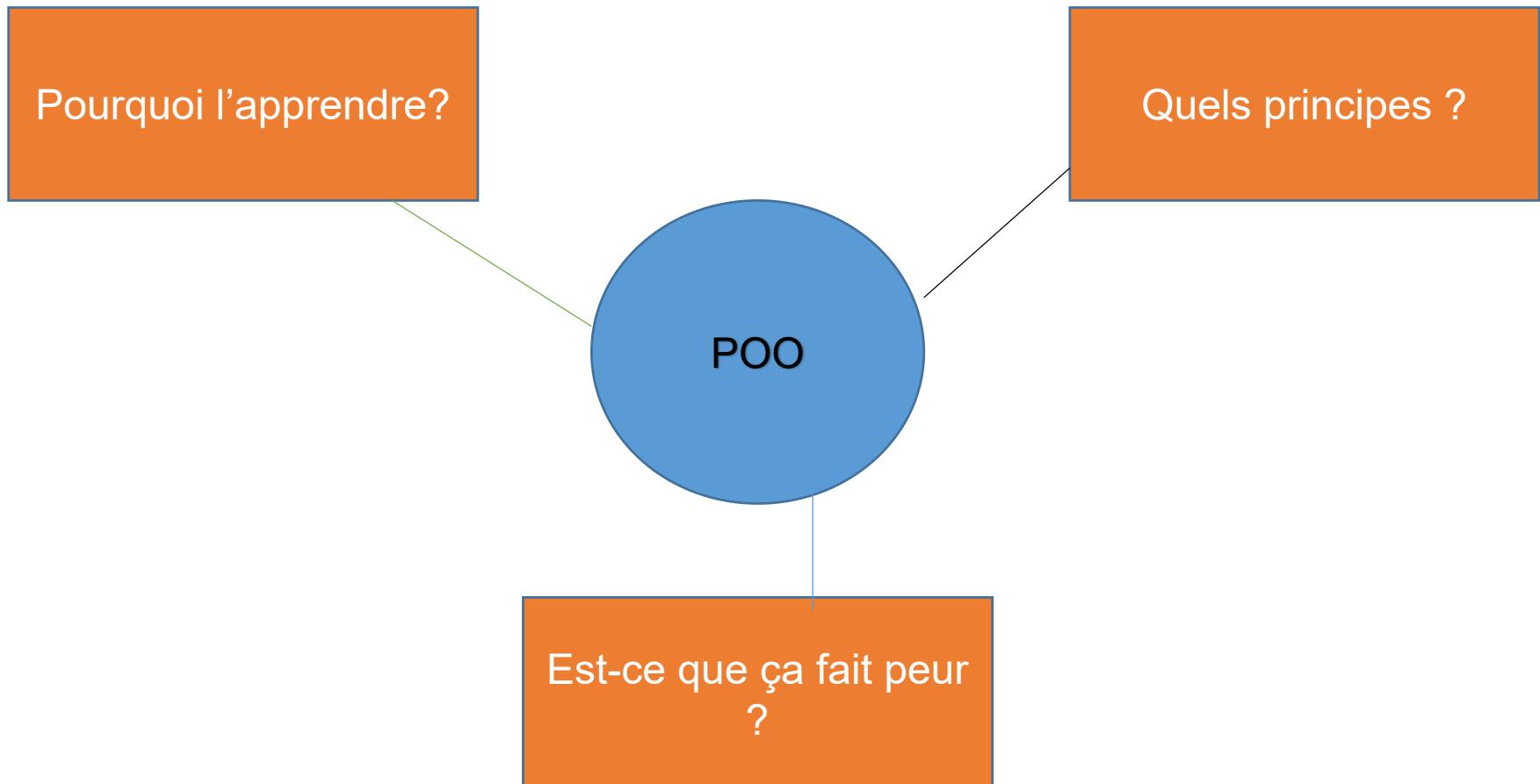


# Programmation – Concepts avancées

- Théorie

# Pourquoi ?



- Paradigme : représentation du monde dans un sens large



- Paradigme de programmation = façon de concevoir le code informatique
- Un moyen de formuler le problème dans un code informatique /Imaginer sa solution et l'implémenter
- Une façon de penser

- Quel est le sens du mot implémentation ?
- L'implémentation est la réalisation, l'exécution ou la mise en pratique d'un plan, d'une méthode ou bien d'un concept, d'une idée, d'un modèle, d'une spécification, d'une norme ou d'une règle dans un but précis. L'implémentation est donc l'action qui doit suivre une réflexion pour la concrétiser.

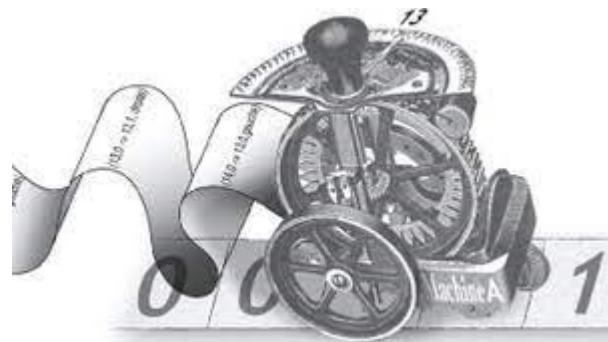


## Paradigmes courants

- Ex : procédural , Orienté objet, événementiel, fonctionnel
- Objet souvent en acte 2 après le procédurale

- La programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. Cela signifie que les programmes impératifs donnent des instructions à l'ordinateur sur comment effectuer une tâche, étape par étape.
- La programmation structurée est un sous-ensemble de la programmation impérative. Elle est basée sur des principes clés pour développer des programmes de manière claire et organisée :

- Procédurale : programmation impérative et structuré
  - Suite d'instructions avec structures de contrôle
  - Appel à des procédures réutilisable
  - S'inspire de l'architecture des ordinateurs (Vn)



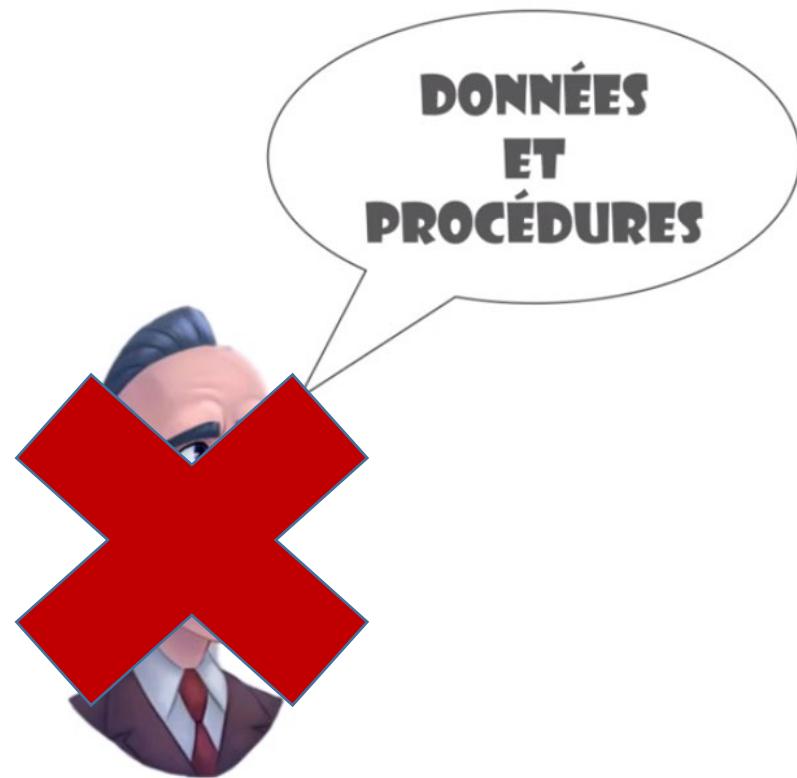
- Exemple simple :

A=(0,0)  
B=(1,1)

Données

distance(A,B)

procédure



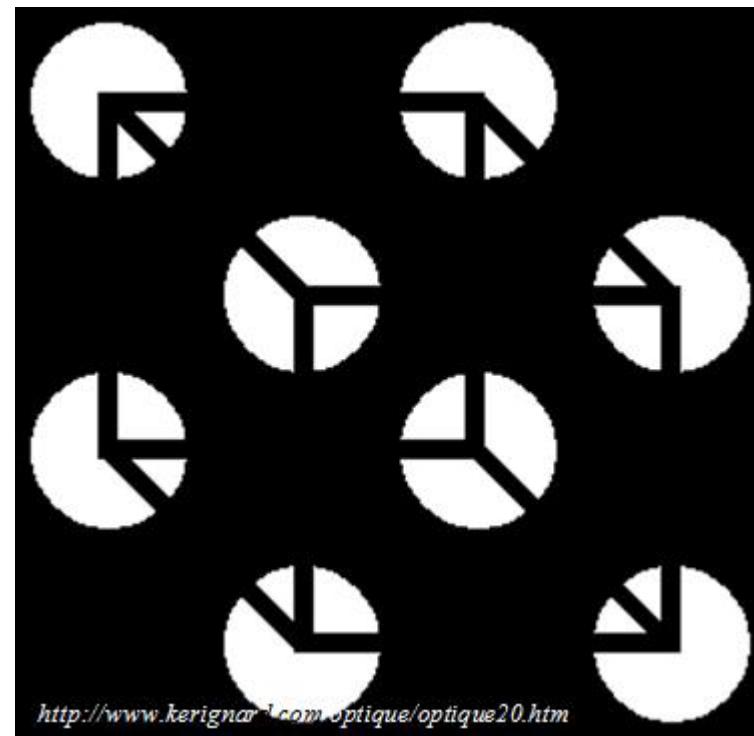
- L'humain voit la vie comme des objets pas comme des données



# Perception des objets



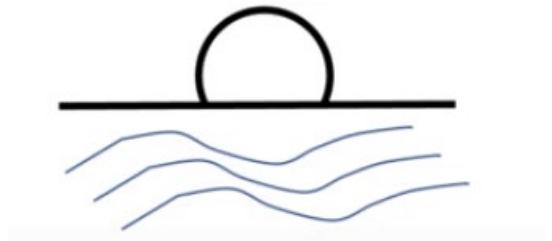
# Création visuel d'objet



<http://www.kerignar.com/optique/optique20.htm>

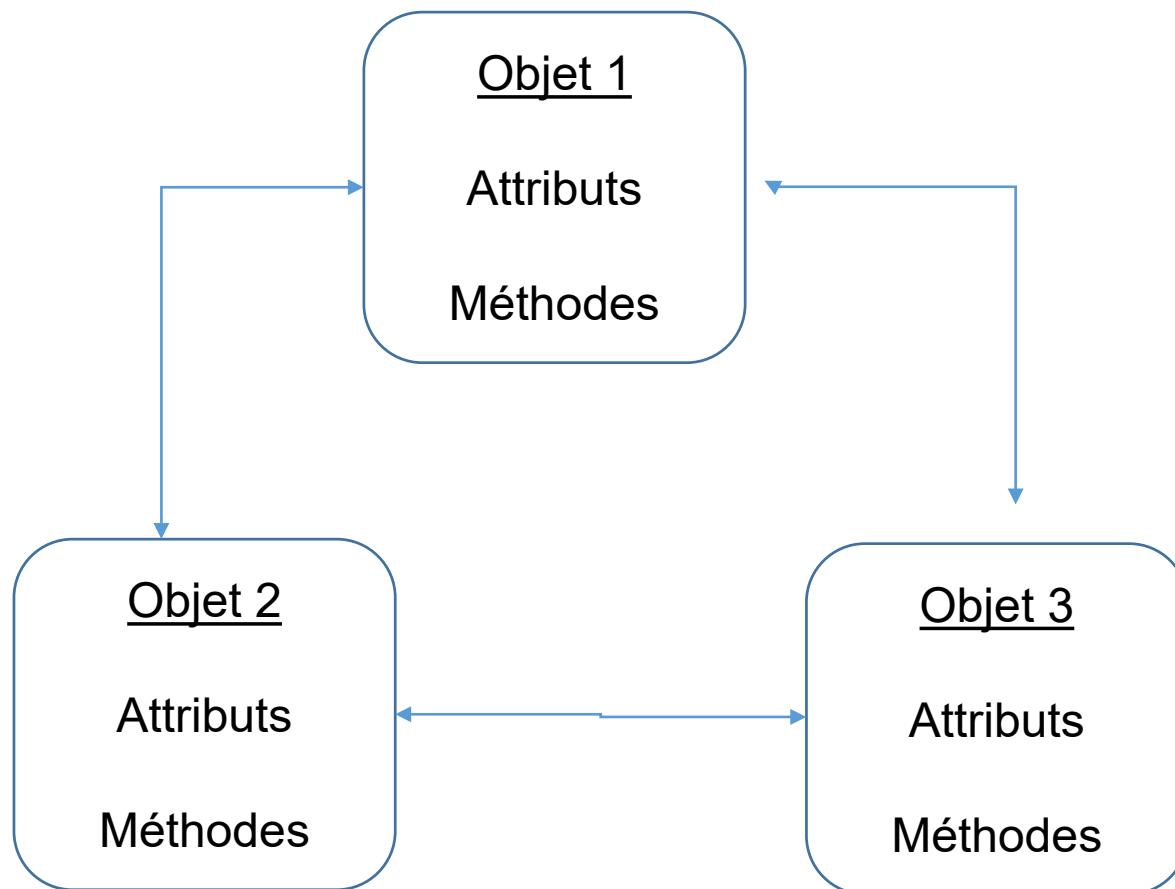
- On voit donc qu'un humain pense plus POO → Bien pour vous non ?
- En POO on conçoit le programme avec des briques logiciels appelées objets
- L'interaction entre les objets via leurs relations réalise les fonctionnalités voulues

- Attention objet du monde réel complexe !
- Ici non



- Données sont passées entre les procédures





A = (0, 0)

B = (1, 1)

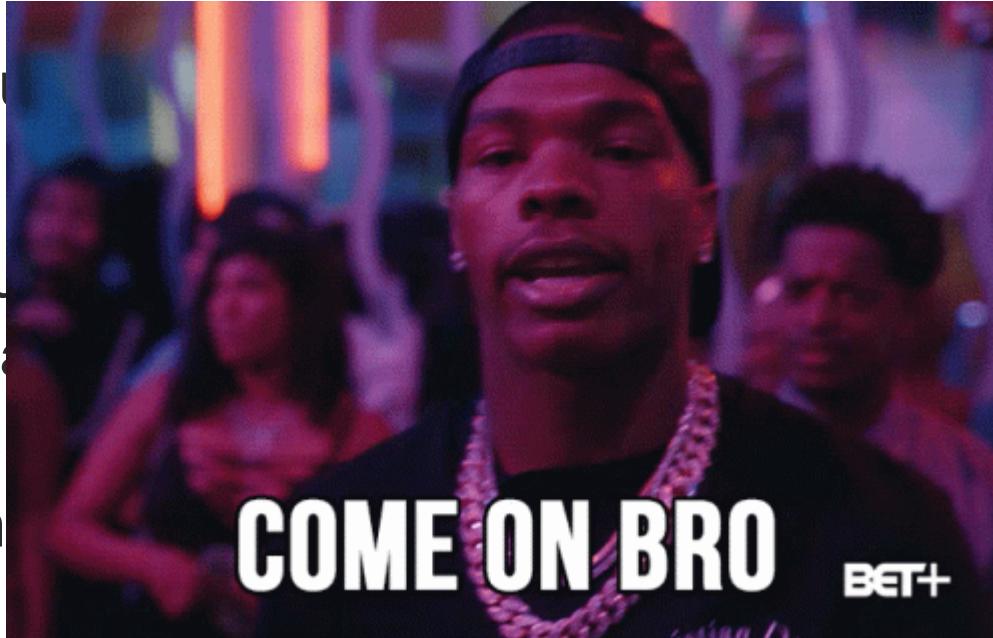
distance(A, B)

A = Point(0, 0)

B = Point(1, 1)

B.distance(A)

# Python

- Python est un langage de programmation
  - Il est déjà utilisé dans de nombreux domaines
    - ageUtilisateur
  - Tu sais donc que tu vas répondre à la question 2 !!!
- 
- Integer  
ct to Q 2 !!!

## On connaît kouwah ?

- Afficher du texte
- Saisir des informations
- Variables
- Conditions
- Boucles
- Fonctions-modularités
- Fichiers
- Gestion des erreurs
- Tests

- L'ordinateur est un objet :
- il possède ce qu'on appelle des **propriétés** caractérisées sous forme de données. Pour l'ordinateur, c'est une marque, un clavier, une souris, de la RAM, un processeur, un disque dur, etc...
- il peut réaliser des actions. L'ordinateur peut s'allumer ou s'éteindre, réaliser des traitements.
- Il peut interagir avec d'autres objets. L'ordinateur doit interagir avec un écran (qui est un objet) pour fonctionner, un humain (qui est un objet aussi) peut allumer ou éteindre l'ordinateur.
- Bien entendu, la logique veut que l'on peut avoir plusieurs objets (donc plusieurs ordinateurs si on reprend notre exemple) mais qui auront certainement des caractéristiques différentes (les propriétés seront les mêmes mais les données seront différentes). On parlera alors **d'instance**.

# Qu'est-ce qu'un objet informatique?

```
In [1]: a = 1
```

```
In [2]: type(a)  
out[2]: int
```

```
In [3]: b = 0.1
```

```
In [4]: type(b)  
out[4]: float
```

```
In [5]: c = 'text'
```

```
In [6]: type(c)  
out[6]: str
```

- Ex : nom = « Johan »

```
texte = « Bienvenue{}`».format(nom)  
print (texte)
```

Et oui c'est de l'objet !!! On utilise l'objet nom qui est un str  
Et on utilise la méthode format de la classe Str

Objet

Se trouve dans un état

[propriétés]

Répond à des messages

[comportements]

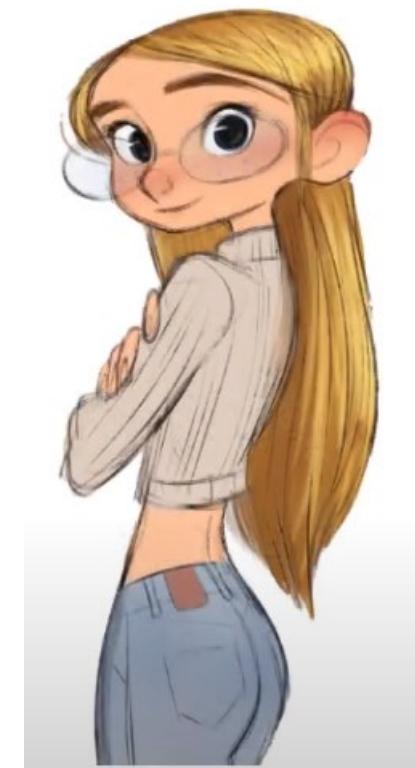
## Exemple

Matières	Points
Mathématiques	15
Programmation	2
Electricité	13

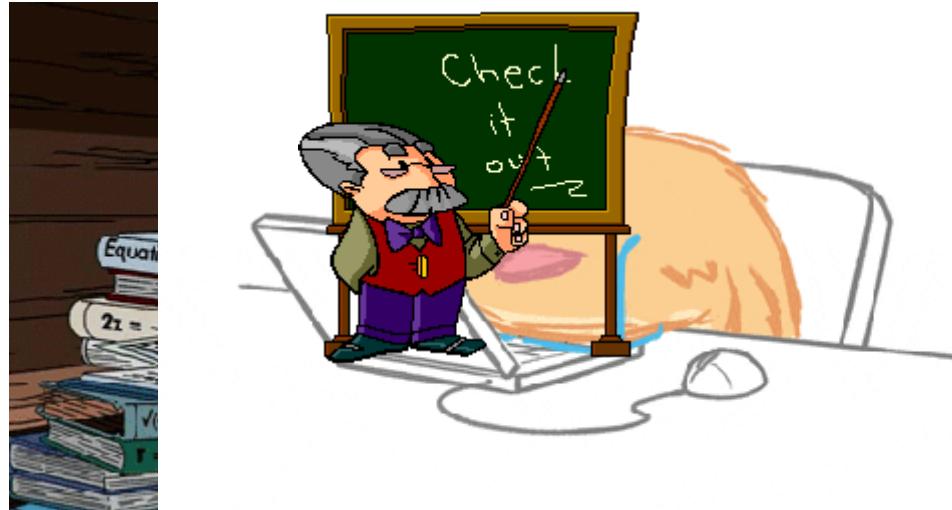
Mr Depreter



Travail insuffisant.  
Vous pouvez(devez) faire mieux

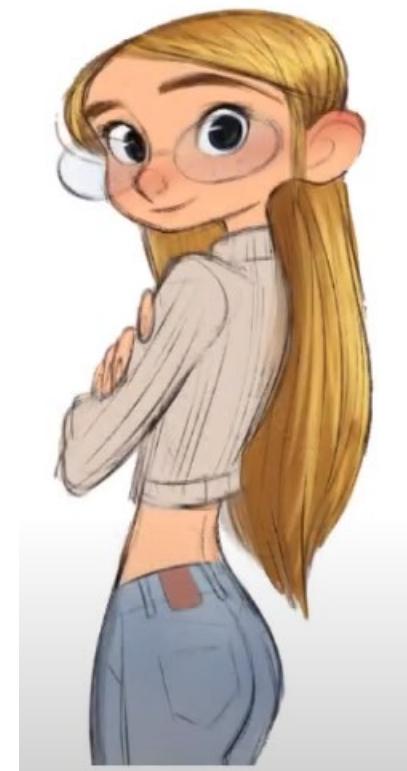


OVER IT.



SLOTHILDA.COM

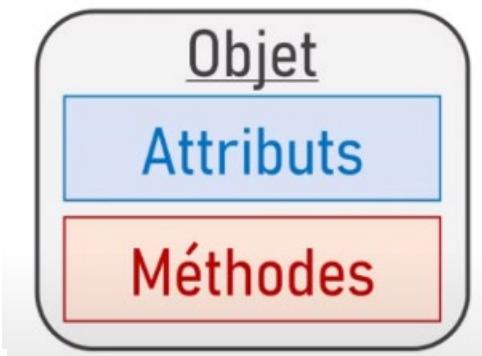
Matières	Points
Mathématiques	15
Programmation	12
Electricité	13



## Etudier

1 <sup>er</sup> Quadrimestre	2 <sup>ème</sup> Quadrimestre
15	15,5
2	12
13	13





Se trouve dans un état [propriétés]

Répond à des messages [comportements]

- Attribut : variable de classe
  - Nom, prénom, age, taille, ....
- Propriété : Attribut = propriété d'un Humain euh ???
  - Certains parlent de compétence/capacité pour la propriété , de naturel/catégorisation pour l'attribut.
  - TROOOOOOPPPP subtile !!!
- Pour python : Propriété = Manière de manipuler des attributs (lecture seule, accès non autorisée, ...).

On pense POO → nouvelle vision : Voiture ne modifie pas un Humain par exemple.

## Attention

- Attention dans beaucoup de langage propriété = attribut
- Un vidéo, un tuto, ... peut mélanger les deux noms



## Repreneons le code

A=(0,0)

B=(1,1)

distance(A,B)

Priorité à l'algo

A= Point(0,0)

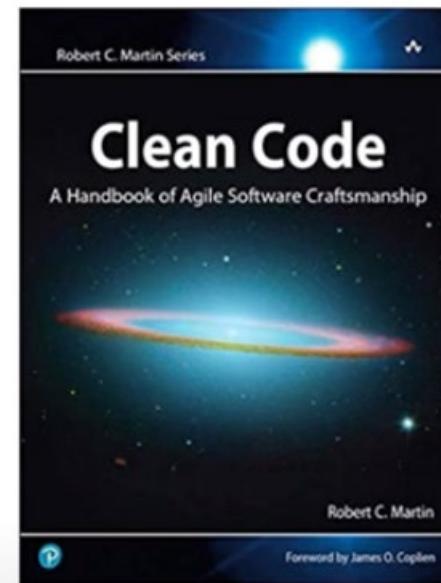
B=Point(1,1)

B.distance(A)

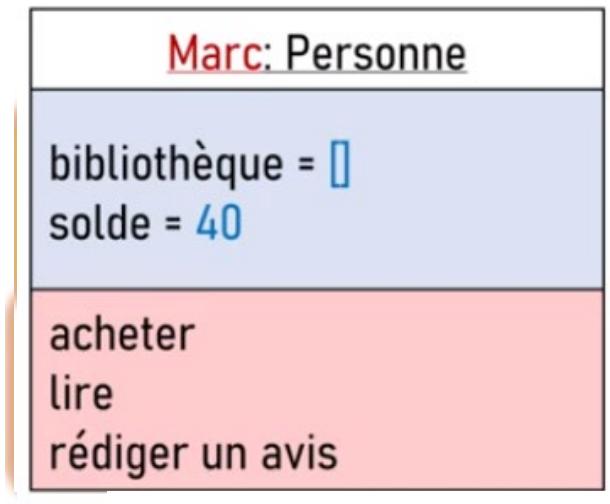
Priorité à l'objet



Marc



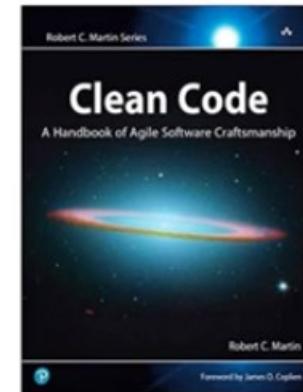
Auteur : Robert C Martin  
Prix : 38 euros



Attributs



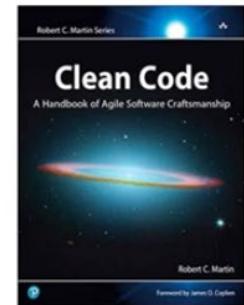
Méthodes





Marc: Personne
bibliothèque = <code>□</code> solde = <code>40</code>
acheter lire rédiger un avis

Clean Code: Livre
auteur = <code>Robert C Martin</code> titre = <code>Clean Code</code> prix = <code>38</code>
<code>afficher le prix</code>

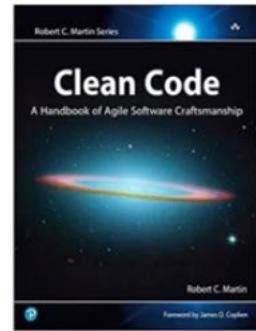


Marc consulte le prix du livre « Clean Code »



Marc: Personne
bibliothèque = [Clean Code] solde = 2
acheter lire rédiger un avis

Clean Code: Livre
auteur = Robert C Martin
titre = Clean Code
prix = 38
afficher le prix



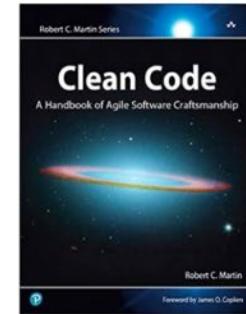
Marc consulte le prix du livre « Clean Code »  
Marc achète le livre





Marc: Personne
bibliothèque = [Clean Code]
solde = 2
acheter
lire
rédiger un avis

Clean Code: Livre
auteur = Robert C Martin
titre = Clean Code
prix = 38
afficher le prix



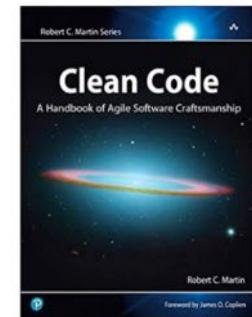
Marc consulte le prix du livre « Clean Code »  
Marc achète le livre  
Marc lit le livre





<b>Marc:</b> Personne
bibliothèque = [Clean Code] solde = 2
acheter lire
<b>rédiger un avis</b>

<b>Clean Code:</b> Livre
auteur = Robert C Martin titre = Clean Code prix = 38
afficher le prix



Marc consulte le prix du livre « Clean Code »  
Marc achète le livre  
Marc lit le livre  
Marc rédige un avis sur le livre

## Avantages

Permet d'exprimer le problème et sa solution en termes d'objet du monde réel.

→ Lisibilité et maintenance du code

Fournit des moyens spécifiques pour une meilleure factorisation de traitements (polymorphisme, généricité)

→ code source réduit, modulaire, évolutif

## Limites

- En général l'apprentissage est plus complexe que le procédural car beaucoup de concepts



## Précisions sur les classes

- La classe est une description d'un ensemble d'objet qui partagent des attributs et des méthodes.

# Objets étudiants



X  
15,0



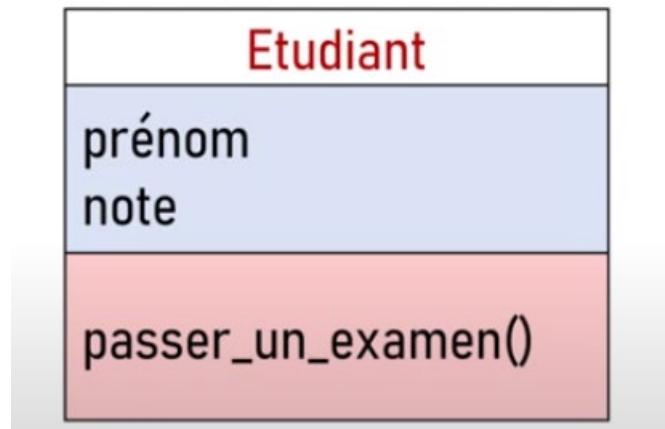
Y  
16,0



Z  
13,0

Réalisé par Erwin Desmet et Johan Depreter

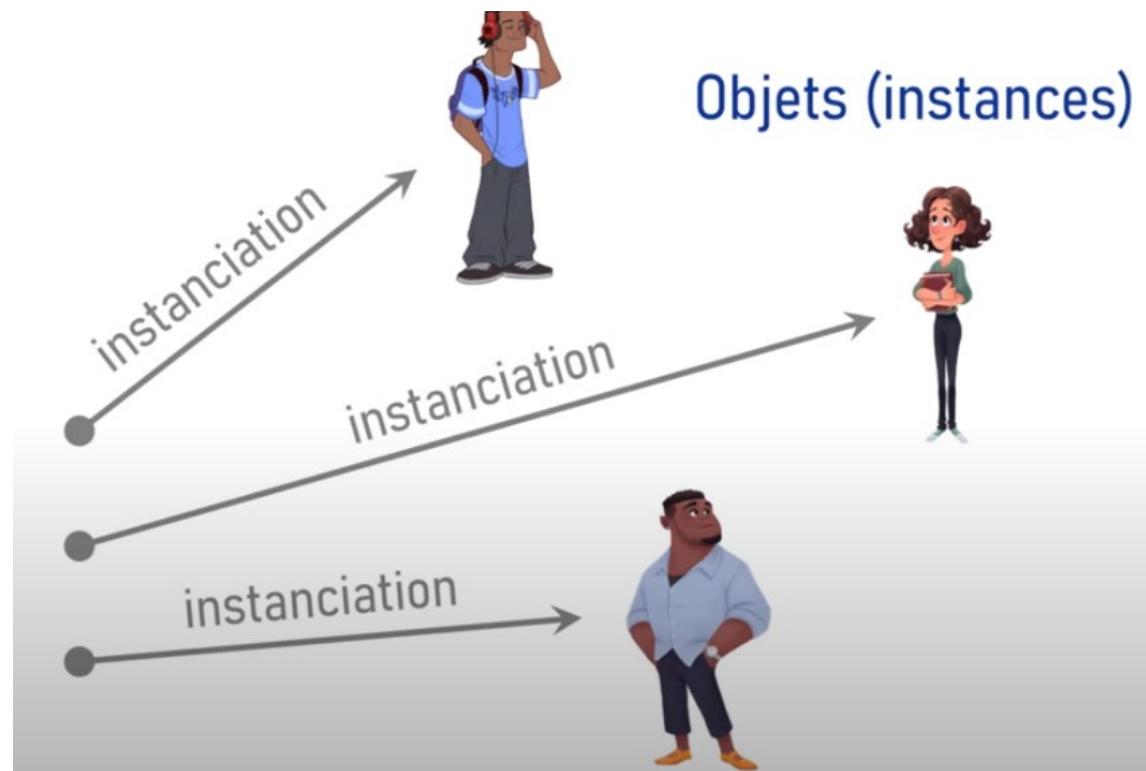
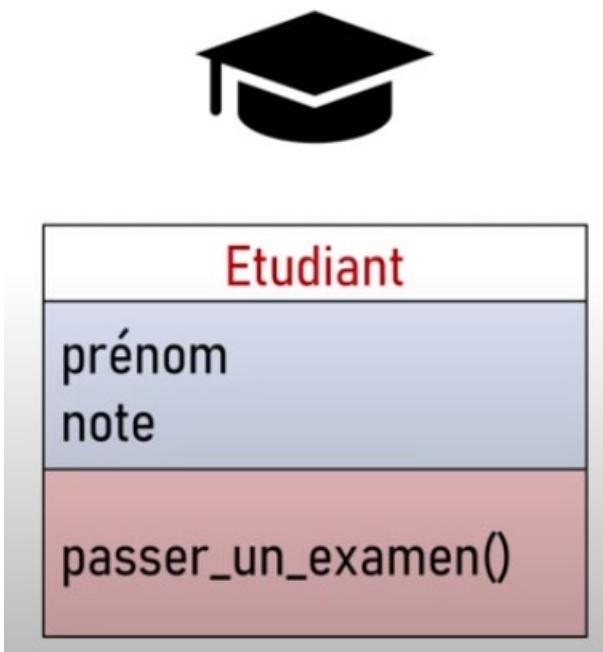
# Classe Etudiant



# Les classes et les objets

- Classe : Un peu comme un plan de conception , genre
  - Ex : Humain
- Objet = une instance de classe
  - Du genre de la classe → Ex : Johan, Erwin, Joakim,...

## Classe Etudiant



Camion
capacité
vitesse maximale
coût d'amortissement
procéder au paiement
calculer le coût de revient

La classe « Camion »,  
abstraction du **client**

Camion
fabriquer
stocker

La classe « Camion »,  
abstraction du **responsable de  
gestion de la production**



Camion
délai de disponibilité
mode de livraison
variantes
vendre
calculer une remise de prix

La classe « Camion »,  
abstraction du  
**responsable commercial**

## Quizz

a



b



Est-ce que ces objets sont des instances :

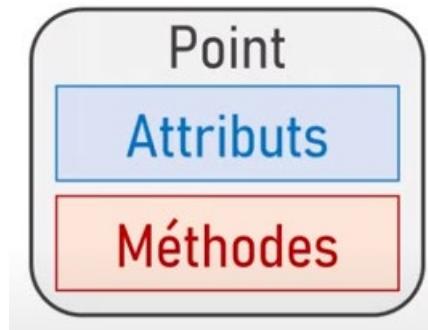
De classes différentes

De la même classe

## Points à retenir

- ① Un **objet** est une entité qui possède un état (défini par ses **attributs**) et des comportements (définis par ses **méthodes**).
- ② Une **classe** est une **description** d'un ensemble d'objets qui partagent les **mêmes attributs** et **méthodes**.
- ③ Un **objet** est une **instance** de sa classe.
- ⚡ Une **classe** peut admettre plusieurs **conceptions**.

# Identifions une classe point ?

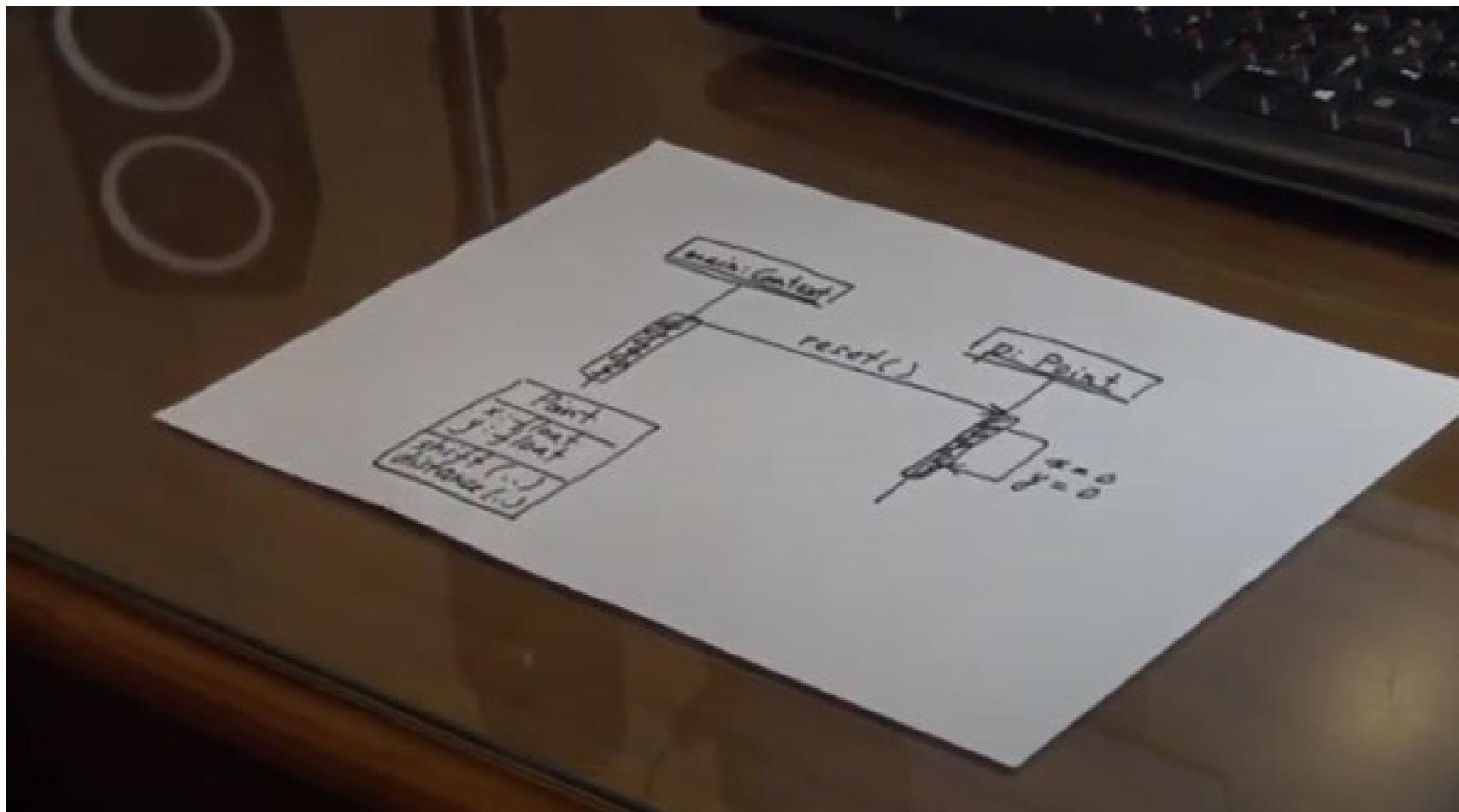


Quels sont ses attributs ?

Quels sont ses méthodes ?

Comment le savoir ?

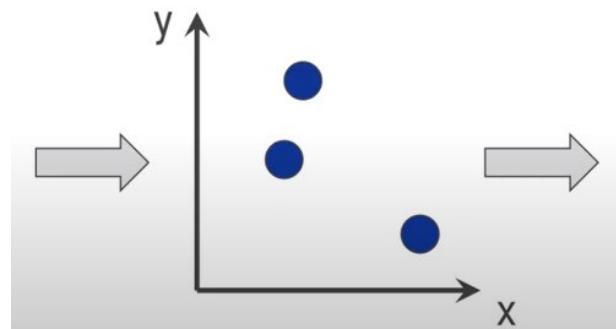
# Conception → UML voir plus loin



## Objets réels



Concept «point »  
(abstraction/math)



Classe concept  
(abstraction.code)



# Implémentation de la classe

Code

```
class Point:
```

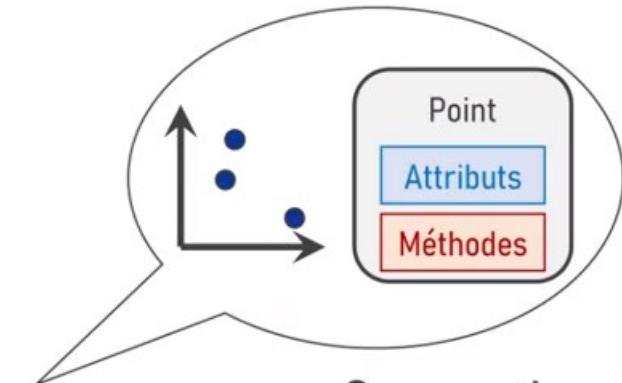
```
...
```



python



Développeur



- Le point, selon Euclide, est ce qui n'a aucune partie.
- Un point désigne un emplacement.  
Sa seule caractéristique est sa position

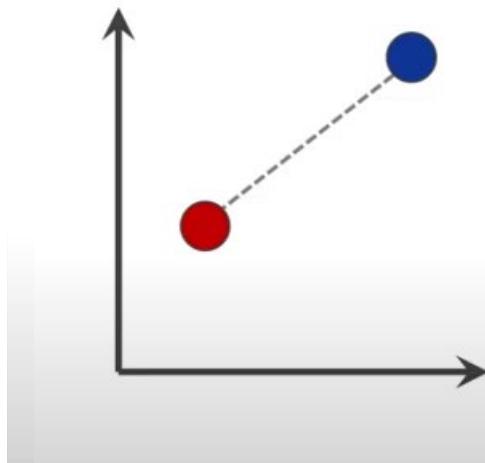


- Le point, selon Euclide, est ce qui n'a aucune partie.
- Un point désigne un emplacement.

Sa seule caractéristique est **sa position**



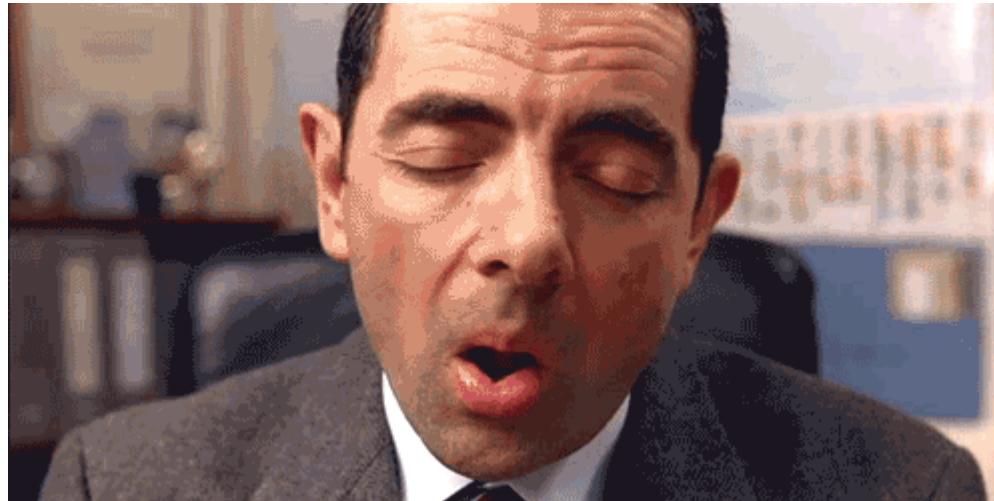
# Passage du concept math à la conception



Point
x
y
reset()
shift(dx, dy)
calculate_distance(...)



- Moment chiantine théorique .... On s'accroche !



## Modélisation des exigences

- Méthode : Fonction d'une classe (procédure aussi)
  - En python : ça fonctionne pareil qu'une fonction
  - Ex : manger(TypeNourriture, Quantité) , marcher, parler; ...
  - Ce sont un peu les services, ce qu'il peut faire
- Méthode de classe : fonction d'une classe avec une forte appartenance à une classe (on verra plus tard)
- Méthode statique : fonction d'une classe mais indépendante de la classe. (classe utilitaire, on utilise pas l'objet)  
Un peu comme un service d'une classe, on instancie aucun objet de la classe , elle est utilisé comme add-on

- On va encapsuler nos données dans notre objet et donc celles-ci ne seront plus « visibles » à l'extérieur de cet « objet ». Il faut donc faire un appel à nos méthodes.
- Obtenir la valeur d'un attribut : On parle de méthode d'accès (getter).
- Modifier la valeur d'un ou plusieurs attributs : On parle de méthodes d'altération (setter).



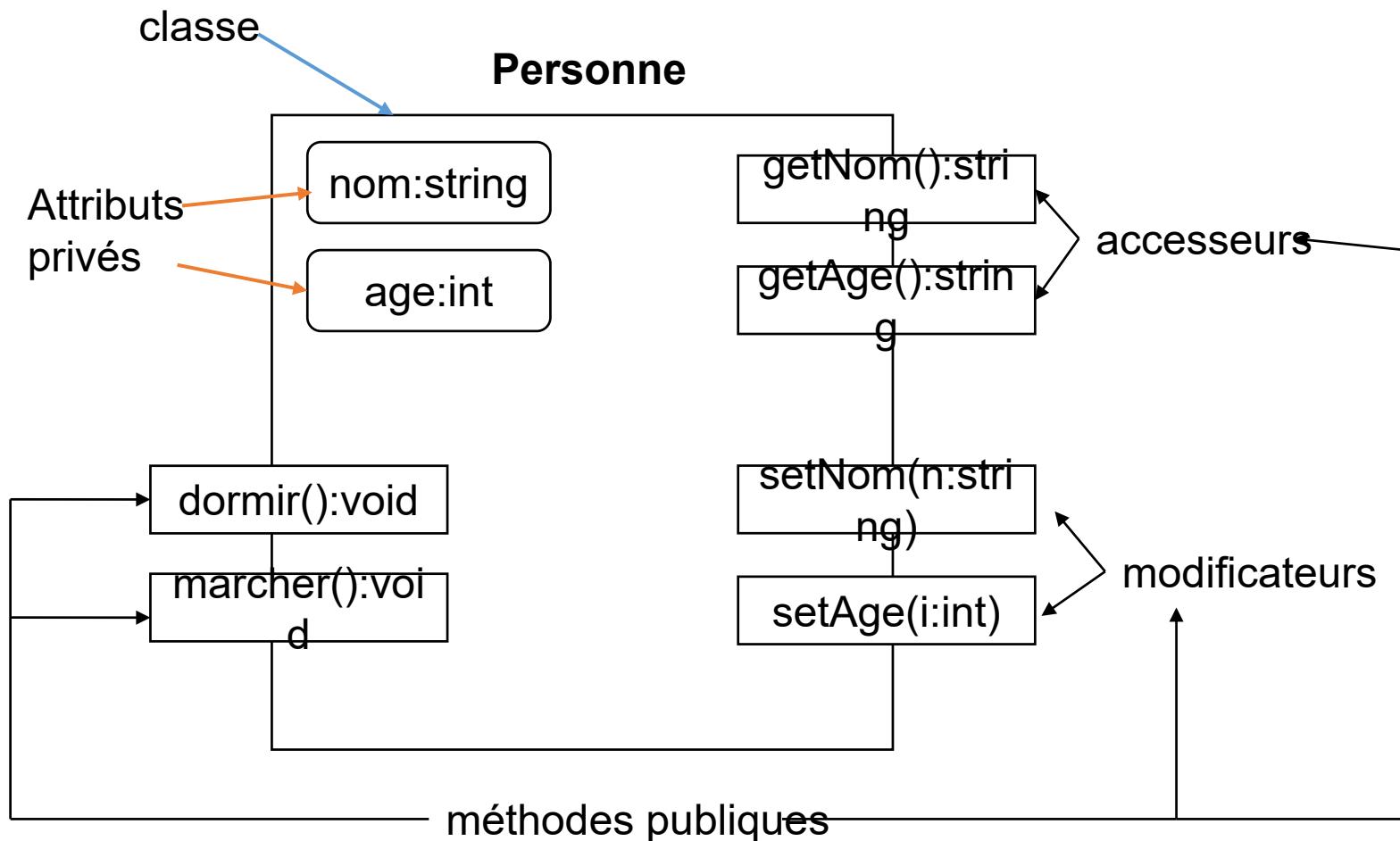
- L'encapsulation permet de définir des niveaux de visibilité des éléments de la **classe**. Ces niveaux de visibilité définissent les **droits d'accès aux données** selon que l'on y accède par une méthode de la classe elle-même, d'une **classe héritière**, ou bien d'une classe quelconque.
- Par défaut, les attributs d'une classe sont privés et ses méthodes publiques

- Niveau 1, 2 ou 3 fais ton choix !
- public : les méthodes et attributs avec ce mot-clé sont accessibles aisément par l'utilisateur de l'objet, mais également par la classe elle-même.
- protected : les méthodes et attributs avec ce mot-clé sont accessibles uniquement depuis la classe elle-même et ses classes héritées (je vous parlerais de l'héritage dans le point suivant)
- private : les méthodes et attributs avec ce mot-clé sont accessibles uniquement depuis la classe elle-même

# Comment choisir le niveau de visibilité des différents éléments d'une classe ?

- Bon niveau c'est quoi ?
  - Protection maximum du code depuis l'extérieur  
→ Niveau minimum accessibilité
- Sensibilité de chaque élément
- Impacts des niveaux d'accès à un élément sur le reste de la classe
- Identification des éléments qui ont besoin d'accès pour fonctionner

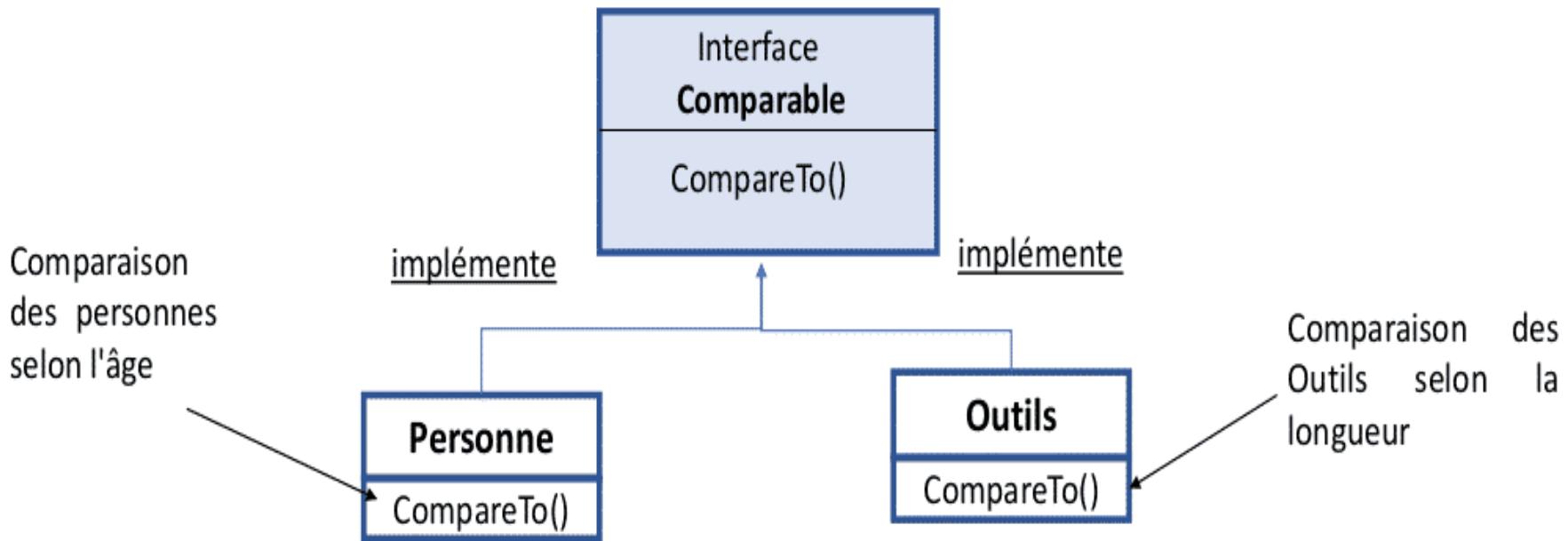
## Exemple



## Interface d'une classe

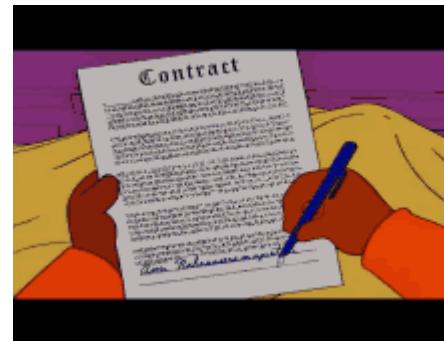
- L'interface d'une classe correspond aux informations dont on doit pouvoir disposer pour savoir utiliser celle-ci. Il s'agit :
  - Du nom de la classe
  - De la signature (nom de la fonction et type des paramètres) et du type de résultat éventuel de chacune des méthodes





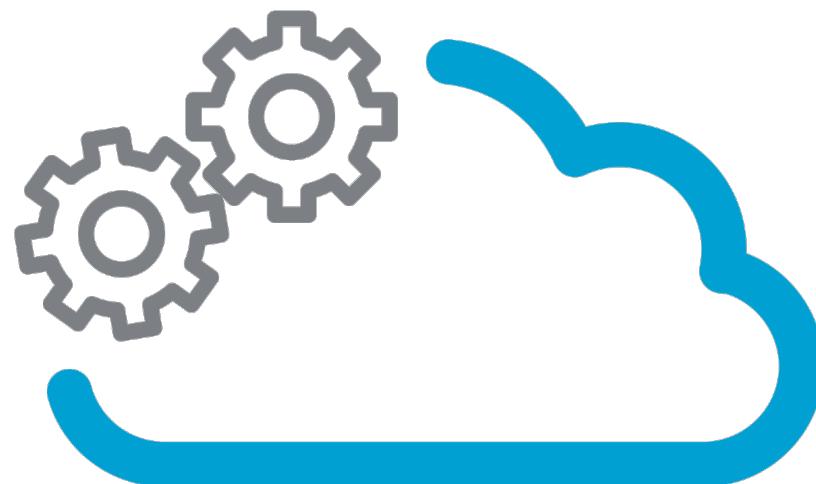
## Le contrat

- Le contrat d'une classe correspond à son interface et à la définition de ses méthodes.



# Implémentation

- L'implémentation d'une classe correspond à l'ensemble des instructions de la classe écrites en vue de réaliser le contrat voulu.



- Constructeur est la première méthode de votre classe



- Il fonctionne comme une fonction
- Il peut prendre des paramètres

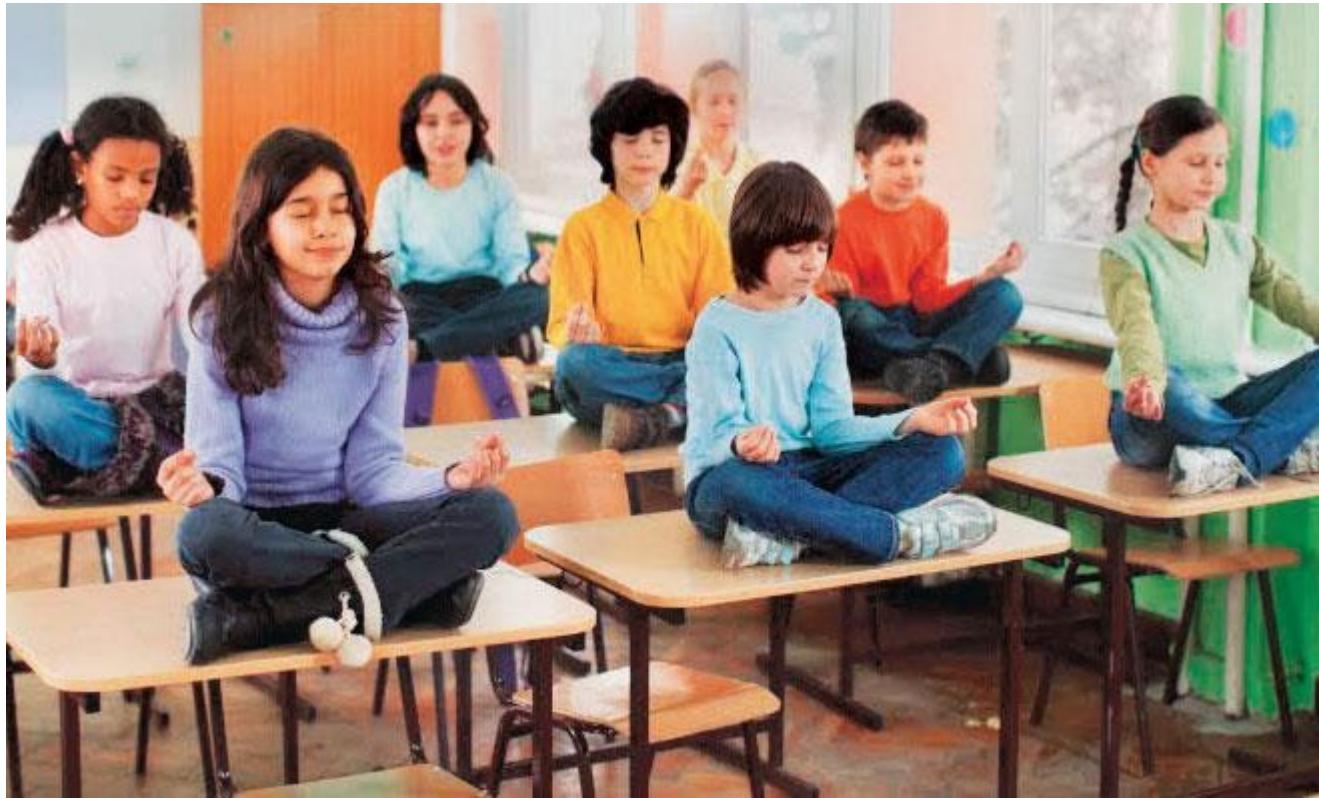
## Constructeur

- Un constructeur se présente comme une méthode particulière de la classe, portant un nom conventionnel ; Nous conviendrons d'utiliser une nomenclature connue qui est d'utiliser le nom de la classe comme nom du constructeur.
- Un constructeur peut disposer de paramètres
- Le constructeur est appelé au moment de la création de l'objet et on peut (si besoin), lui fournir un ou plusieurs paramètres.

- L'instanciation d'une classe fait appel à 3 méthodes spéciales dont la compréhension est très importante :  
**Le constructeur** : on distingue trois types de constructeurs
  - **Le constructeur par défaut** appelé par défaut lors de la création d'un objet (offert par défaut lors de la compilation s'il n'y a pas de constructeur déclaré),
  - **Le constructeur par recopie (ou constructeur de copie)** a un unique argument du même type que l'objet à créer (généralement sous forme de référence constante) et il recopie les attributs depuis l'objet passé en argument sur l'objet à créer.
  - **Le constructeur paramétrique** appelé si la signature correspond à celle du constructeur.

```
classe Point
{
    methode entier Point (entier x, entier y) //constructeur (même nom
que la classe)
    {
        abs :=x
        ord := y
    }
    methode deplace (entier dx, entier dy)
    {
        abs := abs + dx
        ord := ord + dy
    }
    methode affiche
    {
        Ecrire « je suis un point de coordonnées »,abs , « », ord
    }
}
```

- Avantage de la POO :
  - Organisation du code
    - Peut définir des catégories : JV (soigner, attaquer, .... ) sont des choses du joueur par ex (classe joueur qui possède des méthodes)  
classe Mur qui aurait par exemple la densité, ....
    - Réutilisation du code : Une fois une classe défini, on a plus qu'a instancier -> Johan = Erwin = Joakim = Antoine = Denis...  
(enfin presque)







## Rappel

- Encapsulation : Comme une boite noire

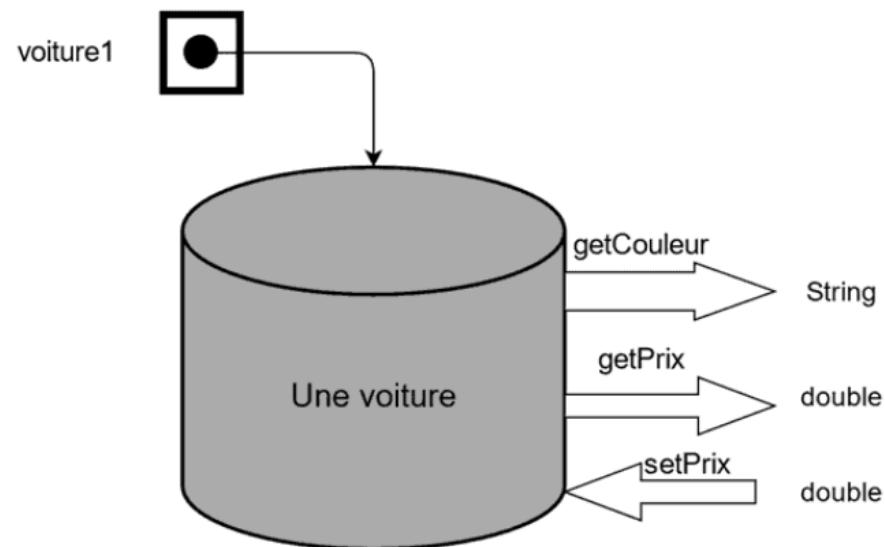


- Cliquons sur un bouton

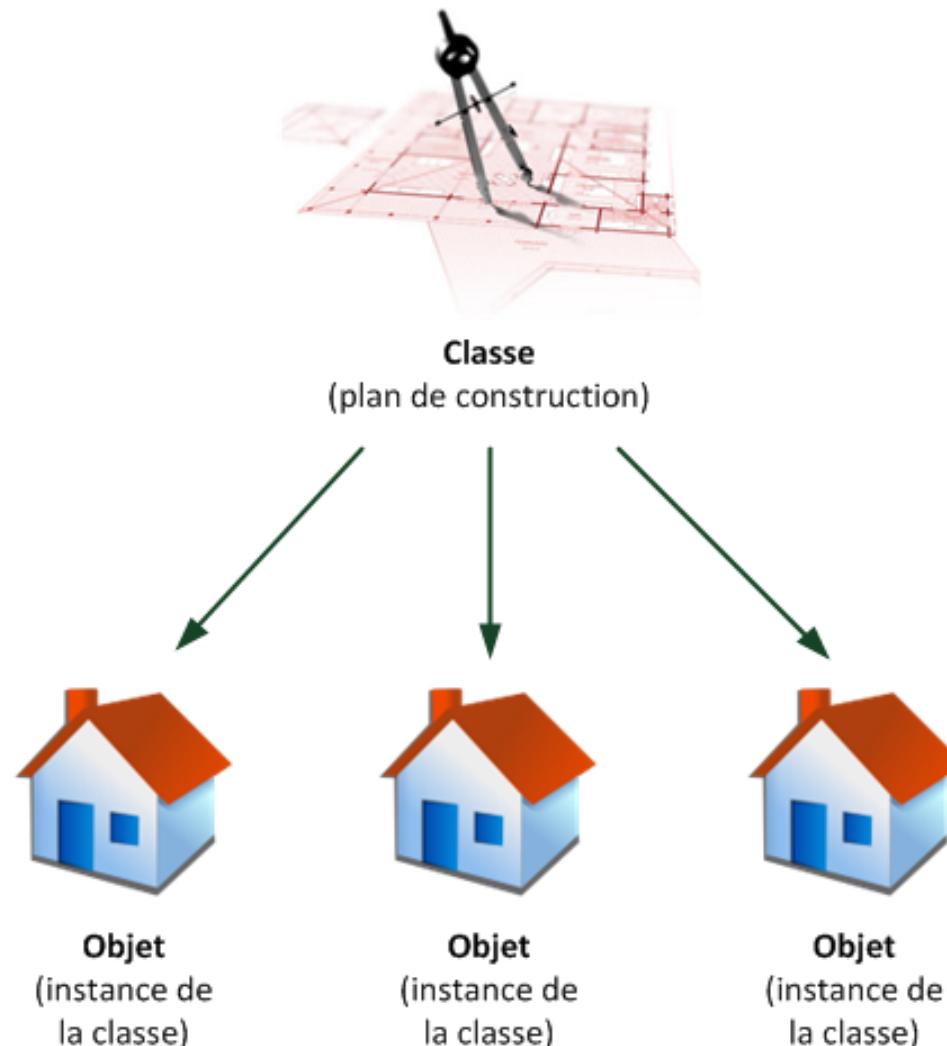
- sortir ou entrer le train d'atterrissage
- augmenter la puissance des moteurs
- modifier l'assiette, le tangage, le roulis
- allumer les phares
- etc...
-

## Rappel par apport aux Q

- **Publique (+)**: les attributs publics sont accessibles à tous
- **Protégée (#)**: les attributs protégés sont accessibles seulement dans la classe elle-même et aux classes dérivées.
- **Privée (-)**: les attributs privés sont accessibles seulement par la classe elle-même



# Classe- objet



## Description minimaliste :

On peut dire qu'une voiture "possède" :

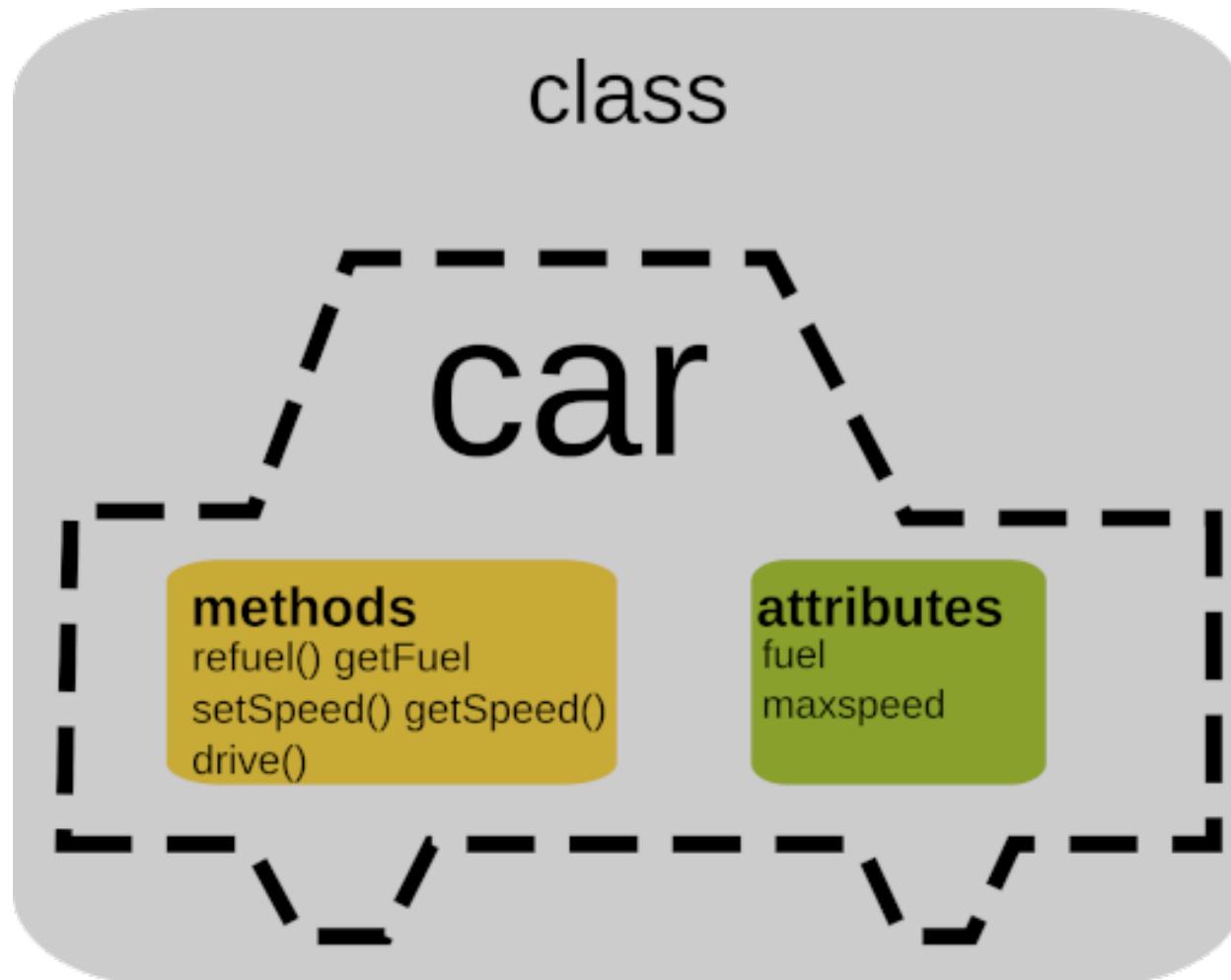
- une marque
- Une couleur

=> *ce sont nos attributs de classe*

On peut dire qu'une voiture "peut" :

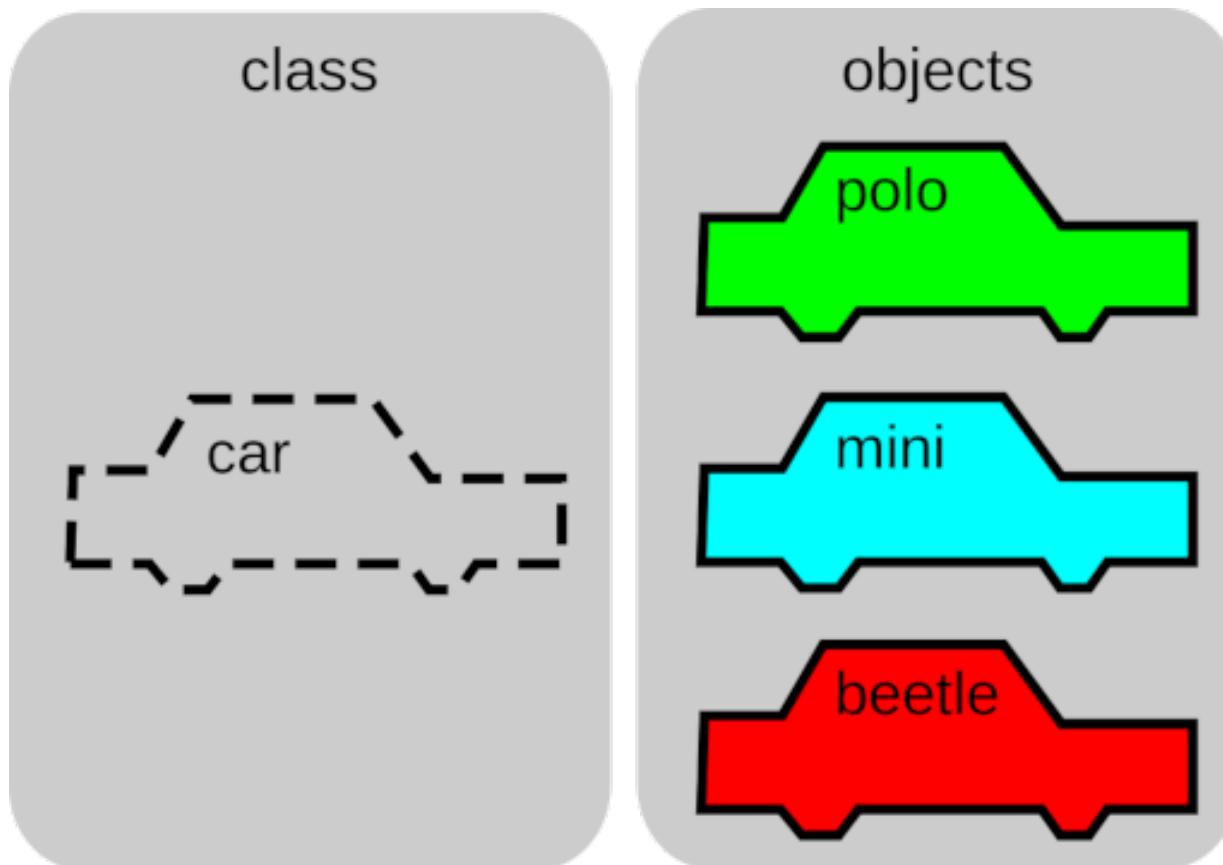
- accélérer
- freiner

=> *ce sont nos méthodes de classe*



```
class Voiture {  
    vitesseMax = 0;  
    couleur = “”;  
    accelerer() {  
        console.log('La voiture accélère !!'); }  
    freiner() {  
        console.log('La voiture freine...');  
    }  
}
```

# Instanciation d'un objet



- les classes sont la description du comportement et de la logique d'une "chose"
- les objets sont la "matérialisation" d'une classe. On dit alors qu'un objet est une instance de classe.

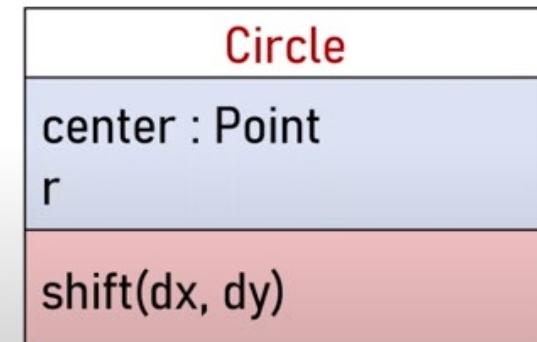
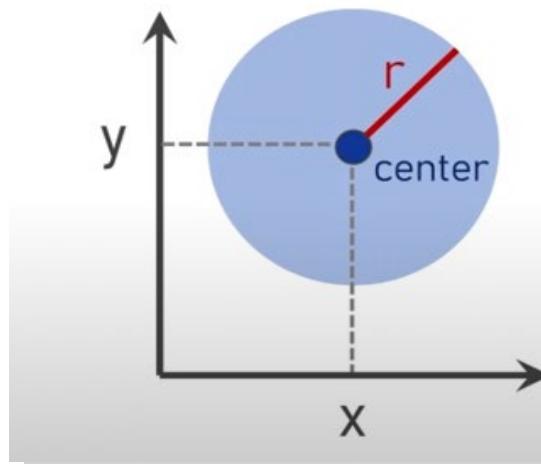
## Rappel

- Le qualificateur Statique permet de déclarer des membres de classe, aussi appelés membres statiques, qui ne dépendront pas d'une instance particulière de la classe. Par exemple la fonction racine carrée, la constante , la fonction Main (première fonction exécutée du programme) ne dépendent pas d'un objet particulier; elles existent même si aucun objet n'est créé.
- Exemple : si on instancie une classe Voiture plusieurs fois, on aura bleu, vert, rouge, ... couleur est lié à l'instance.
- Un attribut statique aura une valeur partagé par tous les objets de la classe contrairement aux attributs standard :
  - chaque voiture a sa propre couleur (rouge, bleu, vert)
  - toutes les voitures ont accès au nombre de participants à la course (par exemple : 8)



## La délégation

- Principe de programmation selon lequel un objet, au lieu d'accomplir une méthode qu'il définit, la délègue à un objet auxiliaire



`Circle.shift(dx, dy) → circle.center.shift(dx, dy)`

# Relation entre deux objets



L'objet A **a un** objet B

① Association

② Agrégation

③ Composition

L'objet A **est un** objet B

④ Héritage

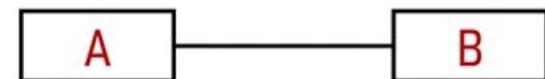
## Association

- Expression sémantique :
  - L'objet A **est lié à** un Objet B (A **a un** B )

Expression syntaxique :

```
class A:  
    def __init__(self, b: "B"):  
        self.b = b
```

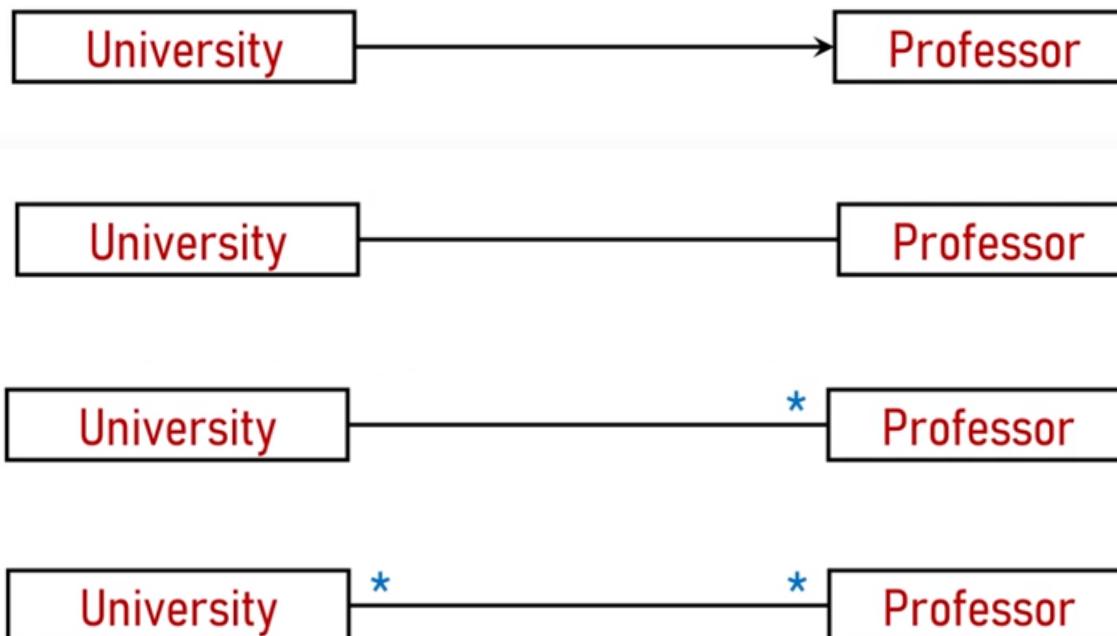
```
class B:  
    def __init__(self, a: "A"):  
        self.a = a
```



## Exemple

Exemple:

- Une université **emploie** un professeur.



```
class University:  
    def __init__(self, professor):  
        self.professor = professor
```

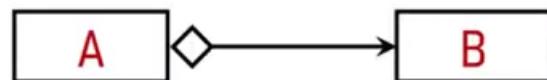
```
class Professor:  
    def __init__(self, university):  
        self.university = university
```

```
class University:  
    def __init__(self, professors):  
        self.professors = professors
```

```
class Professor:  
    def __init__(self, universities):  
        self.universities = universities
```

## Agrégation

- Expression sémantique :
  - L'objet A **est composé** de l'objet B
  - Les cycles de vies des objets A et B sont **indépendants**

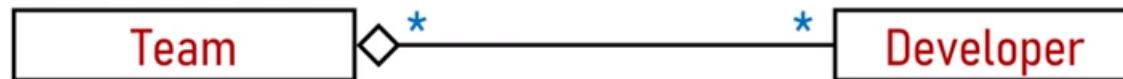
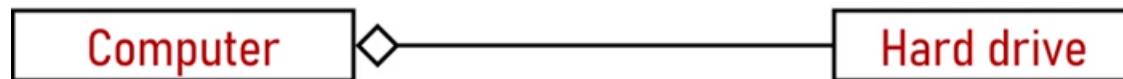


## Expression syntaxique :

```
class A:  
    def __init__(self, b: "B"):  
        self.b = b
```

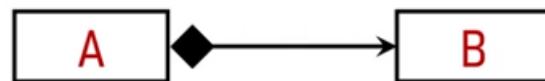
```
class B:  
    pass
```

- Exemple :
  - Un ordinateur est composé d'un disque dur



# Composition

- Expression sémantique :
  - L'objet A est composé de l'objet B
  - Le cycle de vie de l'objet B dépend de A

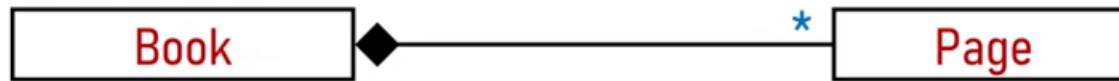


## Expression syntaxique :

```
class A:  
    def __init__(self, b: "B"):  
        self.b = b
```

```
class B:  
    pass
```

- Exemple :
  - Un livre est composé de plusieurs pages



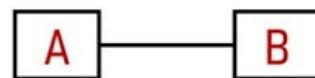
```
class Book:  
    def __init__(self, pages):  
        self.pages = pages
```

```
class Page:  
    pass
```

\* \* =KO

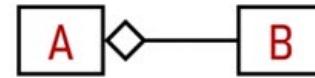
## Résumé

### ① Association



A est B sont reliés

### ② Agrégation



A – composé, B – composant

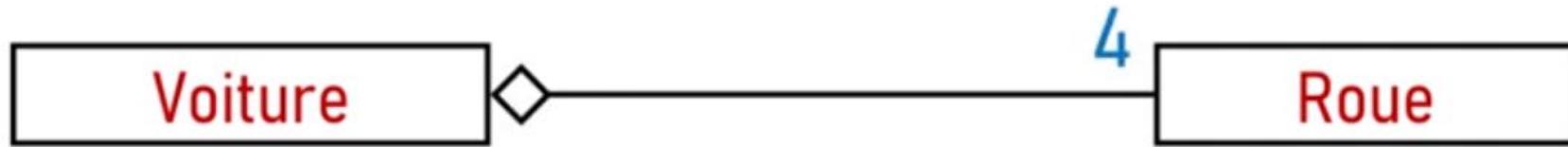
### ③ Composition



A – composé, B – composant  
Le *cycle de vie* de B dépend de A  
B appartient à un seul A

## Quizz

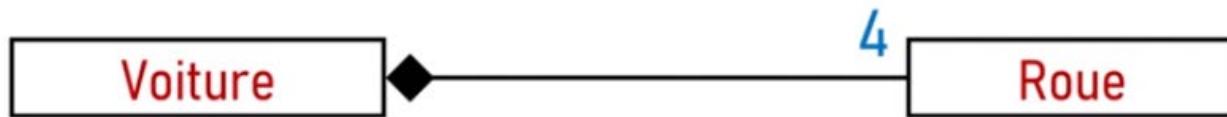
- Une voiture à 4 roues ?



Réponse: agrégation.

Si les roues sont détachables et réutilisables avec une autre voiture.

Une voiture a 4 roues.



Réponse: **composition**.

Si les roues ne sont pas détachables et ne sont pas réutilisables avec une autre voiture.

- Couverture – livre ?
- Molécule - Atomes ?
- Jambe – Humain ?
- Email et titre/fichier/texte/destinataire ?
- Arme - Héros ?
- Etudiant – groupe ?



## Héritage

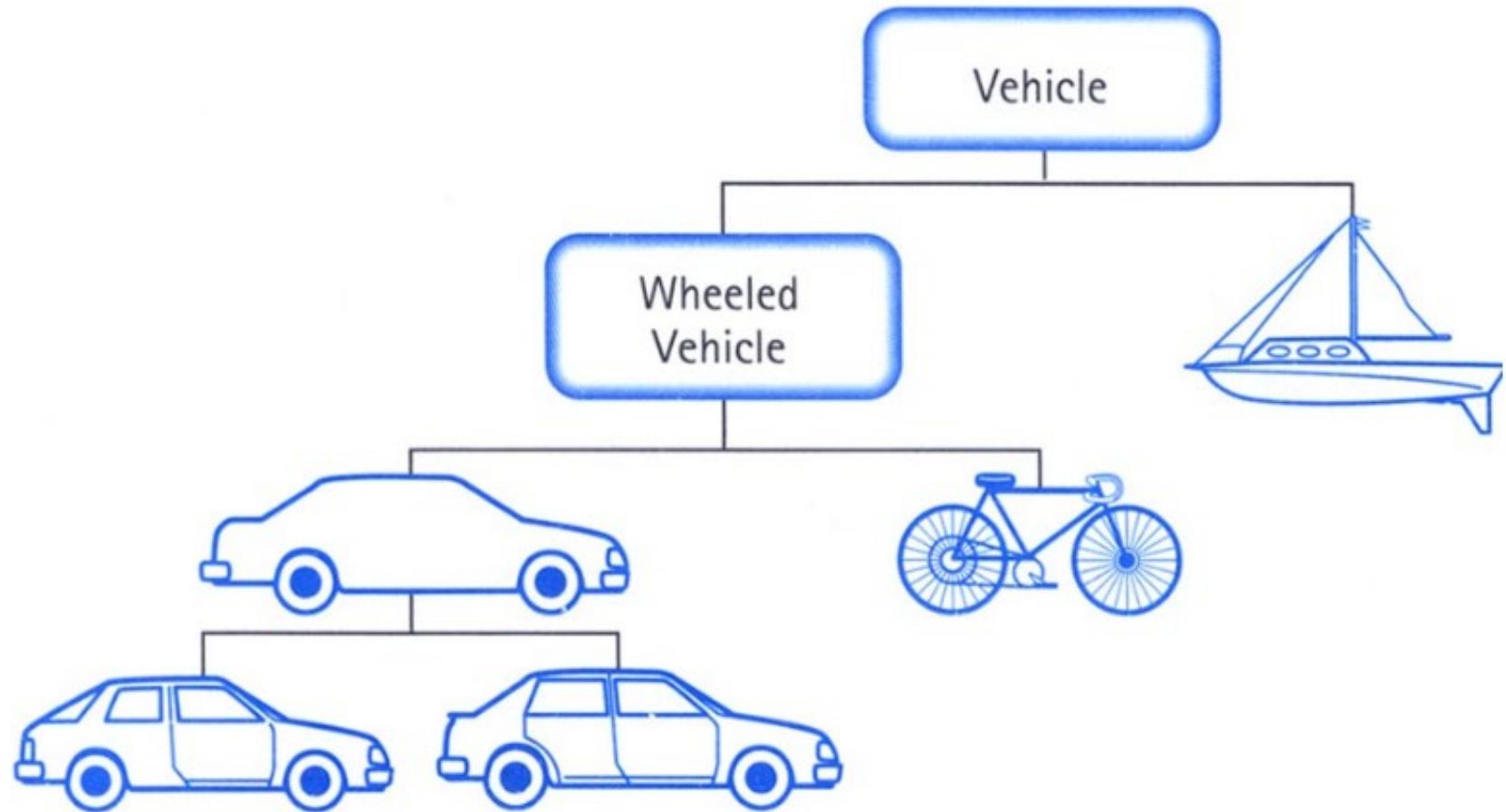
Enfant qui hérite de parents → Animal classe mère

Panda classe fille  
Cheval classe fille ....

Résumé pour retenir : fille est une sorte de mère

→ Panda est une sorte d'animal

La fille est une spécialisation de la mère (elle sait faire des choses en plus )



- Définir un lien de « parenté » entre deux classes
  - Elles sont donc fortement liées (mais différentes)
- Voiture est un véhicule à roues comme moto ou camion
  - Elles sont différentes mais plein de pt commun
- Exemple d'éléments de véhicule :
  - nombre de roues
  - type de carburant
  - année de mise en service
  - démarrer le moteur

- Une classe fille hérite de tous les attributs et méthodes de la classe mère
- Mais chacune à ses particularités aussi :
  - une voiture à un coffre, mais pas la moto ni le camion
  - une moto à un guidon, mais pas la voiture ni le camion
  - un camion a une remorque, mais pas la moto ni la voiture
- Celles-ci iront donc dans la classe fille

- Instantiation d'une classe fille

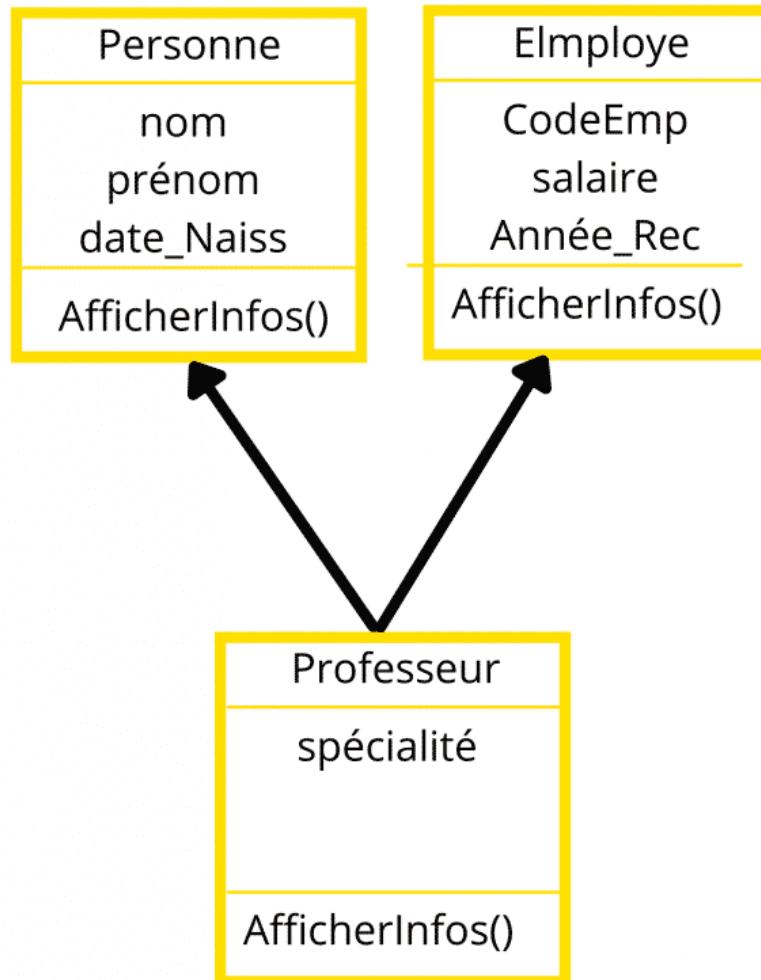


- Concept très puissant et très utilisé.
  - Il permet la granularité du code
  - Une justesse de conception
- Mots clés :
  - **extends** : vous permet de définir une classe comme étant la fille d'une autre. (ex : class Voiture extends Vehicule)
  - **super** : ce mot-clé vous permet d'accéder aux attributs et méthodes d'une classe mère depuis votre classe fille

- Héritage sur autant de niveau qu'on le souhaite
- ex : véhicule → voiture -> 4\*4

## Héritage multiple

- Une classe 1
- Ex: Une classe mères Personne est un employé



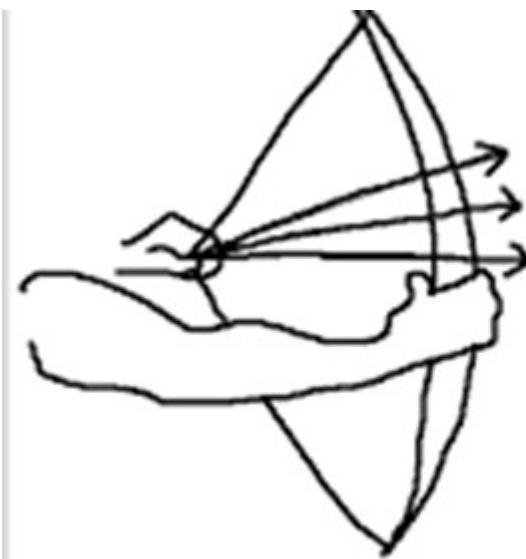
↳ classes  
↳ professeur  
↳ personne :

- Notre professeur à les attributs suivants :
  - nom, prénom et date de naissance.
  - code d'employé , le salaire et l'année de recrutement
  - spécialité ( qui est un attribut spécifique à la classe ; c'est à dire non hérité)
- Il possède la méthode AfficheInfos() hérité 2x ... Aie
- Attention chacune ne fonctionne que sur sa classe mère !  
Re –Aie
- Il faudra donc redéfinir une nouvelle méthode AfficheInfos()  
qui permettra d'afficher la spécialité

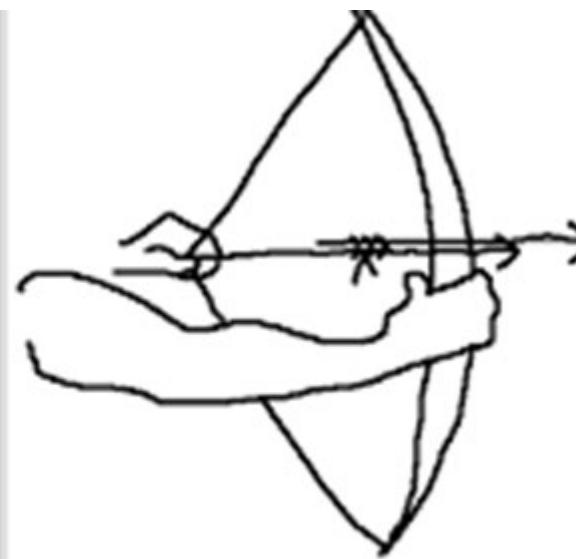


- On peut redéfinir ou surcharger une méthode de la classe mère
  - Modifier le comportement de celles-ci
  - Ajouter des instructions particulières
- Ex : méthode rouler de véhicule :
  - la moto peut rouler sur des chemins, des routes ou des autoroutes
  - la voiture sur des routes ou des autoroutes
  - le camion seulement sur des autoroutes





Overloading



Overriding

## Surcharge

- En apprenant, on apprend de la même chose des autres.
- Même chose.
- Parfois, on peut faire des erreurs.
- Les erreurs sont utiles.
- L'erreur est une déclaration de plusieurs constructeurs dans une même classe.



ssi  
s une  
s avec

ype(s)),

## Redéfinition

- La redéfinition consiste à définir le type de l'objet qui nouvelle information sans changer sa nature.
  - Le même type
  - Même type
  - Même par
- 
- consiste à définir le type de l'information sans changer sa nature.
- Le même type
- Même type
- Même par
- WE ARE TRYING EVERYTHING,  
EVEN MANUAL OVERRIDE.



# Exercice !

- Intro

# Exercice !

- Intro
- Crée une classe Padawan
- Crée des classes enfants Jedi et Sith

## Padawan

nom : string

race : string

midichloriens : int

sePresenter( ) : void

sEntrainier(temp) : void









## Jedi

sabreLaser : string

formesDeCombat: [ ]

sePresenter( ) : void

pousser (Objet) : void

influencerPensee (Personne) : void





## Sith

surnom : string

apprenti : bool

formesDeCombat: [ ]

sePresenter( ) : void

étrangler (Personne) : void





Padawan

Jedi



Sith



Encore plus !

- Créer une classe enfant de Sith : SeigneurNoir



## SeigneurNoir

apprentis : [ ]

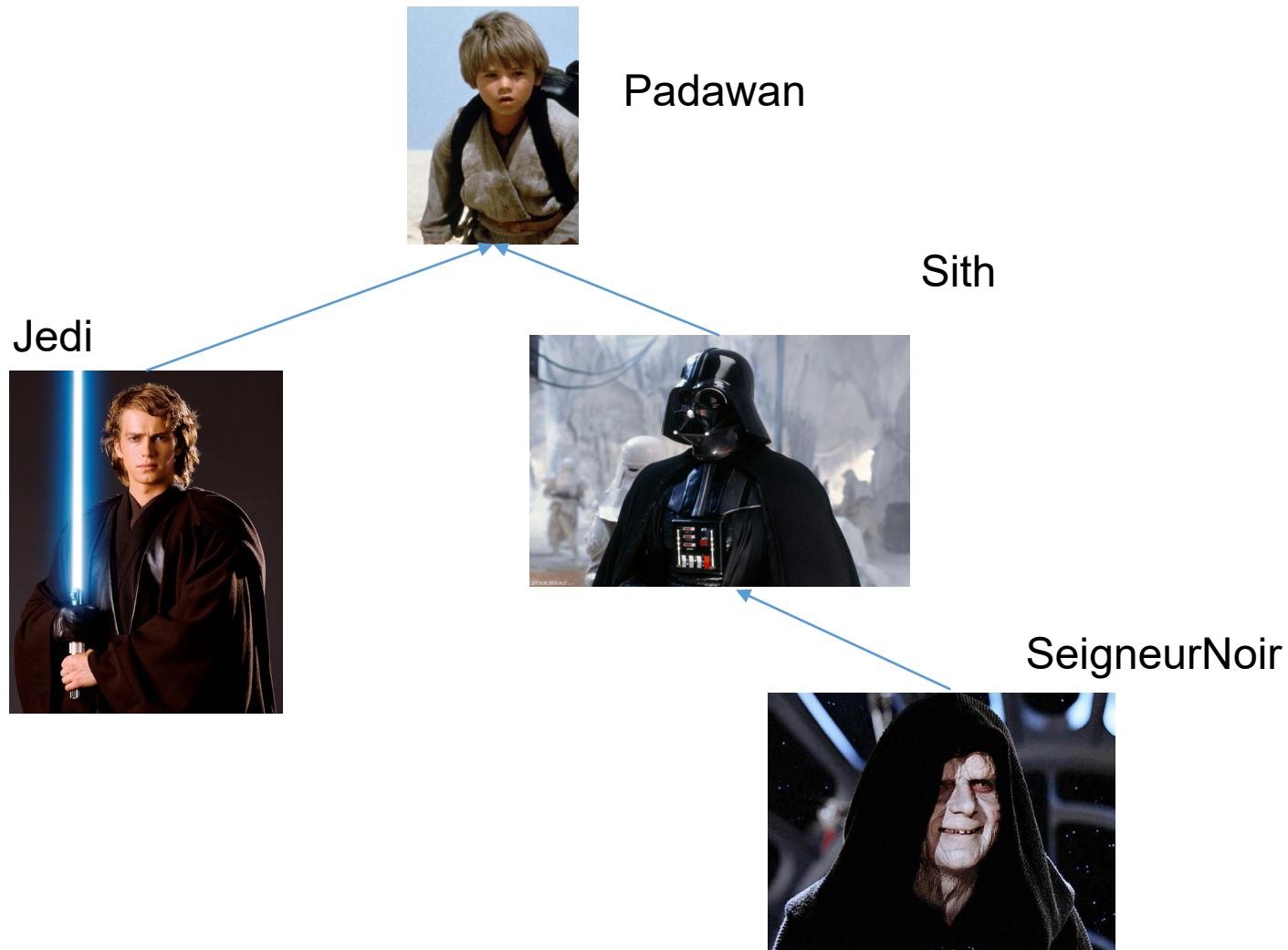
niveauColere : int

sePresenter( ) : void

electrocuter ( Personne ) : void

M. Desmet aka Er-Win





## Ca veut dire quoi ?

- Une instance de SeigneurNoir a également un nom et une race par exemple
- MAIS une instance de Sith NE peut PAS électrocuter les gens



# Polymorphisme

Polym



## Un petit bout de culture G

Le polymorphisme désigne, dans le domaine de la génétique, l'apparition de multiples variantes génétiques au sein d'une population. Les différentes variantes d'un gène particulier sur le même locus peuvent également impacter les allèles mentionnés. Voir aussi le polyallélisme.



[www.aqueportail.com](http://www.aqueportail.com)

## Ok et .... en français ?

- Le mot polymorphisme vient du grec et signifie “**qui peut prendre plusieurs formes**”.
- Dans la programmation orienté objet, le polymorphisme est utilisé en relation avec les fonctions, les méthodes et les opérateurs. Des fonctions et des méthodes de mêmes noms peuvent avoir des comportements différents ou effectuer des opérations sur des données de types différents.

- Le polymorphisme représente la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation.
- On voit donc apparaître ici le concept de polymorphisme : choisir en fonction des besoins quelle méthode appeler et ce au cours même de l'exécution

- On va voir les différents types de polymorphisme qui sont :
  - La redéfinition des méthodes
  - La surcharge des méthodes
  - La surcharge des opérateurs

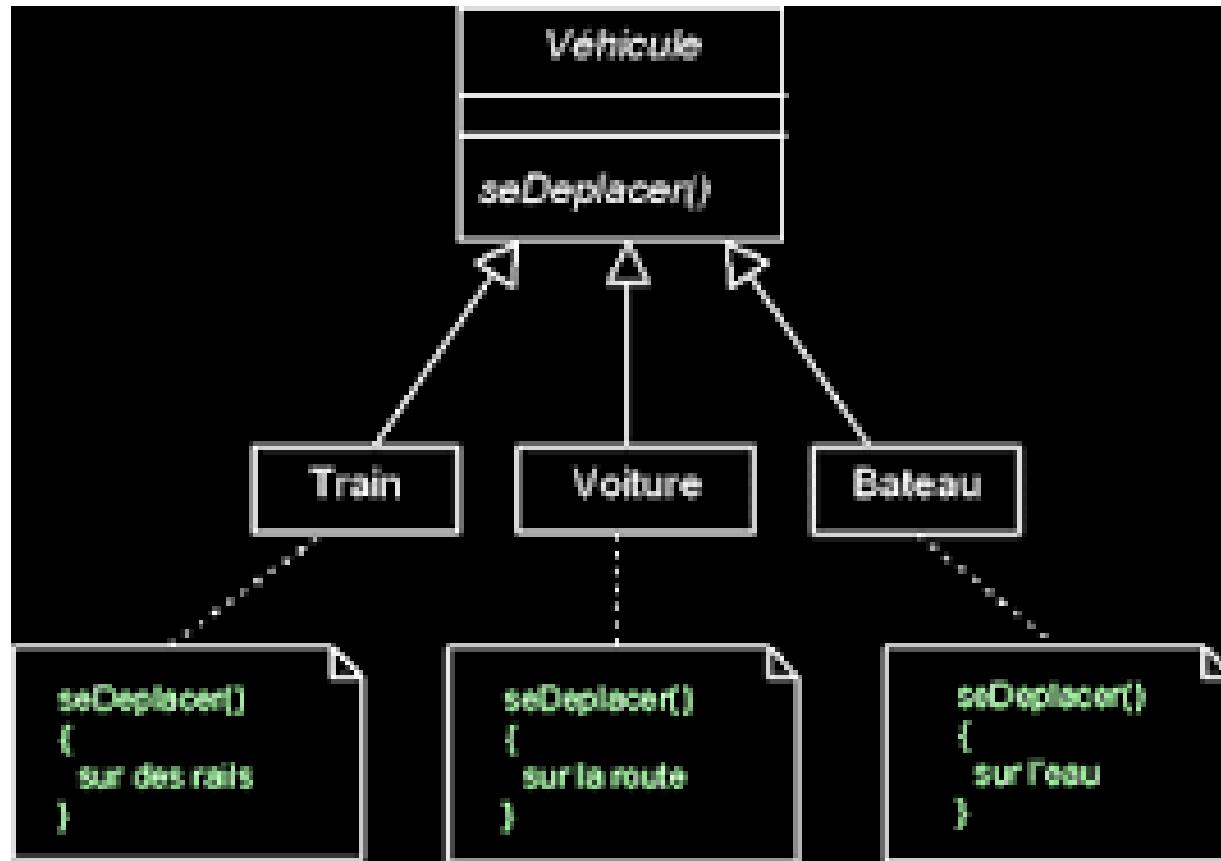
## Le polymorphisme ad hoc (surcharge)

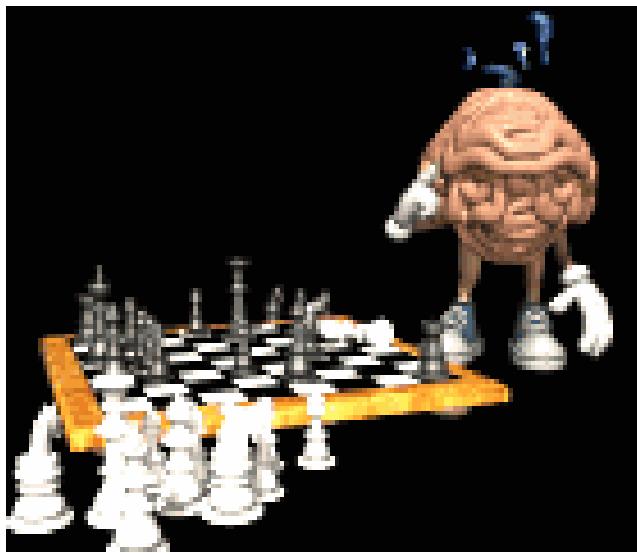
- Le polymorphisme ad hoc permet d'avoir des fonctions de même nom, avec des fonctionnalités similaires, dans des classes sans aucun rapport entre elles (si ce n'est bien sûr d'être des filles de la classe objet). Par exemple, la classe complexe, la classe image et la classe lien peuvent avoir chacune une fonction "afficher". Cela permettra de ne pas avoir à se soucier du type de l'objet que l'on a si on souhaite l'afficher à l'écran.
- Le polymorphisme ad hoc permet ainsi de définir des opérateurs dont l'utilisation sera différente selon le type des paramètres qui leur sont passés. Il est donc possible par exemple de surcharger l'opérateur + et de lui faire réaliser des actions différentes selon qu'il s'agit d'une opération entre deux entiers (addition) ou entre deux chaînes de caractères (concaténation).



## Le polymorphisme d'héritage

- La possibilité de redéfinir une méthode dans des classes héritant d'une classe de base s'appelle la **spécialisation**. Il est alors possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque : il s'agit du **polymorphisme d'héritage**. Ceci permet de faire abstraction des détails des classes spécialisées d'une famille d'objet, en les masquant par une interface commune (qui est la classe de base).
- Imaginons un jeu d'échec comportant des objets *roi*, *reine*, *fou*, *cavalier*, *tour* et *pion*, héritant chacun de l'objet *piece*. La méthode *mouvement()* pourra, grâce au polymorphisme d'héritage, effectuer le mouvement approprié en fonction de la classe de l'objet référencé au moment de l'appel. Cela permettra notamment au programme de dire *piece.mouvement* sans avoir à se préoccuper de la classe de la pièce.





# Le polymorphisme paramétrique

- Le polymorphisme paramétrique, appelé généricité, représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type). Le polymorphisme *paramétrique* rend ainsi possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre.



- Ainsi, on peut par exemple définir plusieurs méthodes homonymes *addition()* effectuant une somme de valeurs.
  - La méthode *int addition(int, int)* pourra retourner la somme de deux entiers
  - La méthode *float addition(float, float)* pourra retourner la somme de deux flottants
  - La méthode *char addition(char, char)* pourra définir au gré de l'auteur la somme de deux caractères
  - etc.
- On appelle *signature* le nombre et le type (statique) des arguments d'une fonction. C'est donc la signature d'une méthode qui détermine laquelle sera appelée.

	<b>Surcharge</b>	<b>Polymorphisme</b>
Héritage	nul besoin	nécessite une arborescence de classes
signature des méthodes	doivent différer	doivent être les mêmes
résolu à	la compilation	l'exécution





# RAPPEL



Joyeux anniversaire au meilleur prof de programmation de l'école!

- Ensuite, en tant que professeur d'élite, vous avez su nous montrer comment traverser les boucles infinies et éviter les pièges des exceptions non gérées. Pour cela, rien de moins qu'une parade à votre honneur, digne des plus grands Gilles, où chaque pas est synchronisé à la perfection et chaque costume reflète l'unicité de son porteur. Imaginez les rues résonnant au son des tambours, les confettis volant comme des octets dans le flux d'une connexion haut débit, chaque sourire reflétant la joie d'un code qui s'exécute sans erreur.

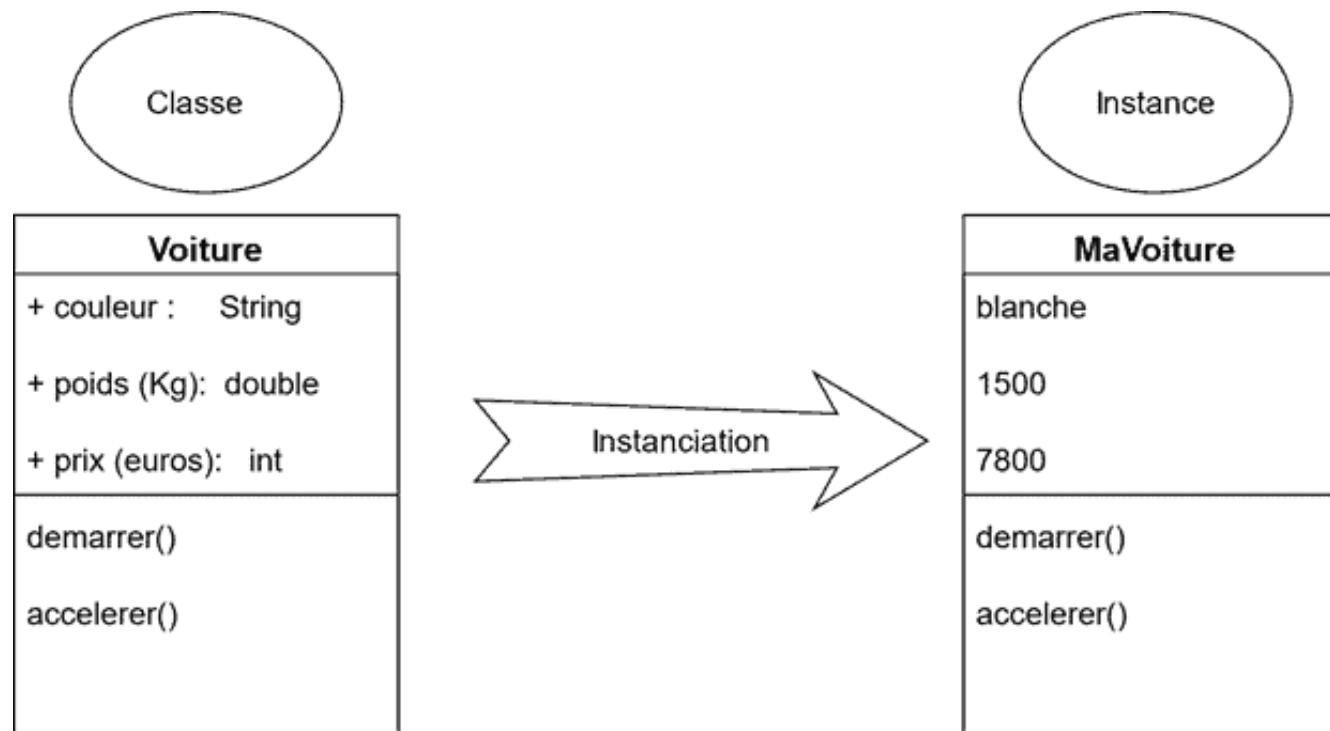
- Enfin, que cette nouvelle année de votre vie soit riche en découvertes et aventures, comme une session épique de jeux en tout genre. Que vous soyez le héros d'un RPG captivant, le stratège d'un jeu de société complexe ou le champion incontesté d'une soirée jeux vidéo, que chaque jour vous apporte son lot de challenges stimulants, de réussites éclatantes et, bien sûr, de bugs... car sans eux, où serait le fun?
- Joyeux anniversaire, cher maestro de la programmation! Merci de compiler notre savoir avec tant de passion et d'humour. Levons nos verres (de débogage) à votre santé et au nombre infini de lignes de code que vous allez encore écrire et enseigner!

à 3, soufflez  
sur l'écran !

L'age heureux !



## Constructeur



## Constructeur

- Un constructeur est, en programmation orientée objet, une fonction particulière appelée lors de l'instanciation. Elle permet d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.

- Si le langage l'autorise, la surcharge est possible et les constructeurs se différencient par le nombre et le type des paramètres passés et renvoyés mais aussi par le formalisme employé. Dans de nombreux langages, on distingue certains constructeurs en particulier :
  - Le constructeur par défaut n'a aucun argument ;
  - Le constructeur par recopie a un unique argument du même type que l'objet à créer (généralement sous forme de référence constante) et il recopie les attributs depuis l'objet passé en argument sur l'objet à créer.

# Constructeur

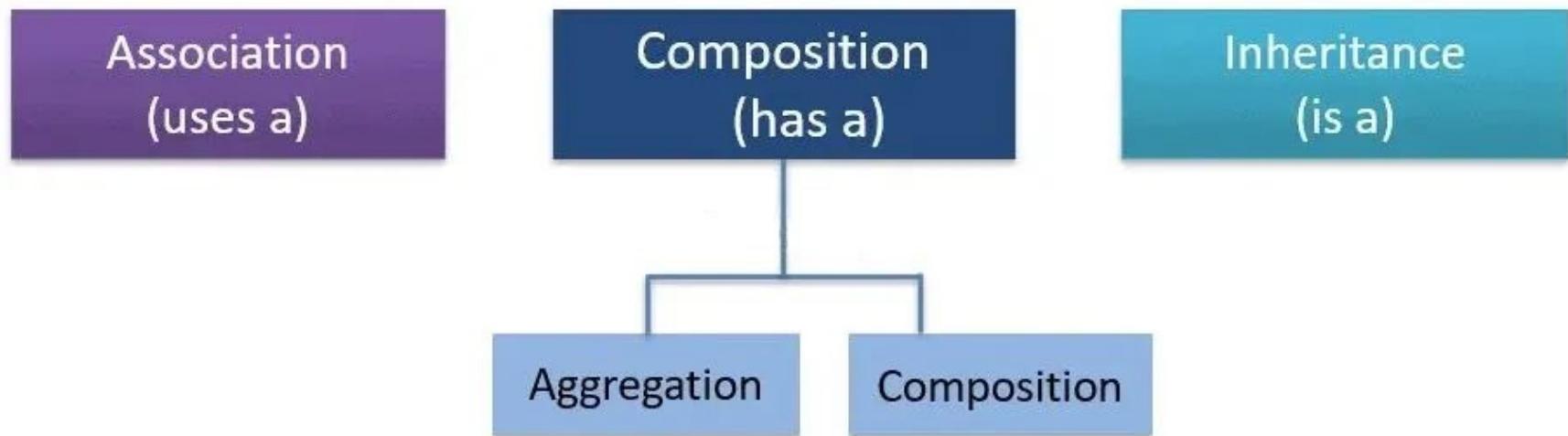
```
public class MaClasse
{
    private int a;
    private string b;

    // Constructeur
    public MaClasse() : this(42, "string")
    {
    }

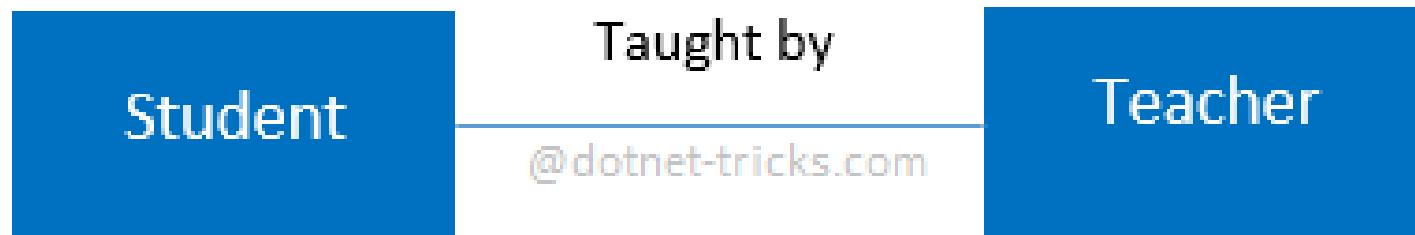
    // Surcharge d'un constructeur
    public MaClasse(int a, string b)
    {
        this.a = a;
        this.b = b;
    }
}
```

```
// Instanciation d'un objet à l'aide du constructeur
MaClasse c = new MyClass(42, "string");
```

# Relations entre classes



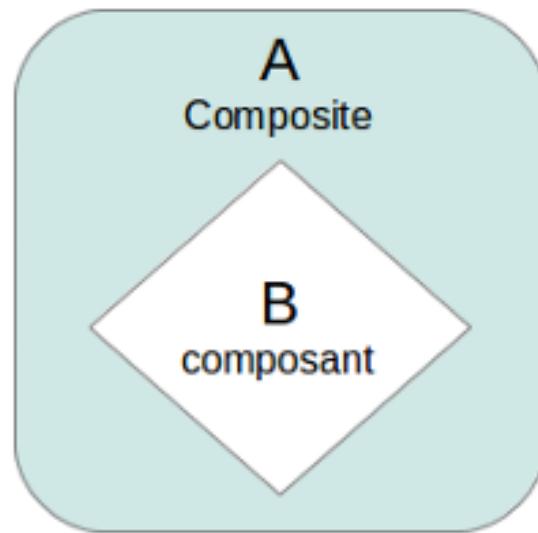
- Une classe utilise une autre classe



Association

## Aggrégation

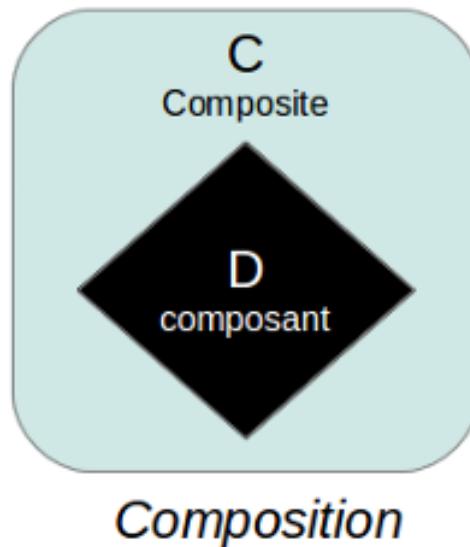
- Une classe est composée d'une autre classe. La composition est faible. Leur cycle de vie ne sont pas liés.



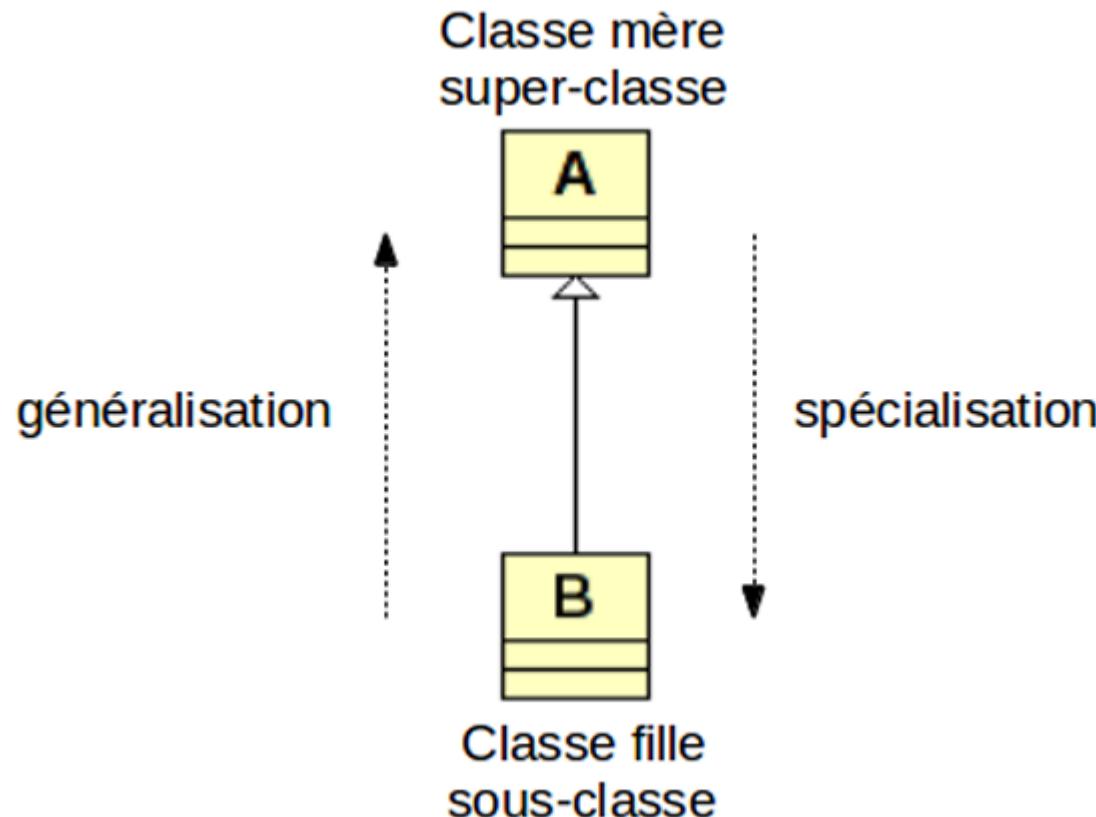
*Agrégation*

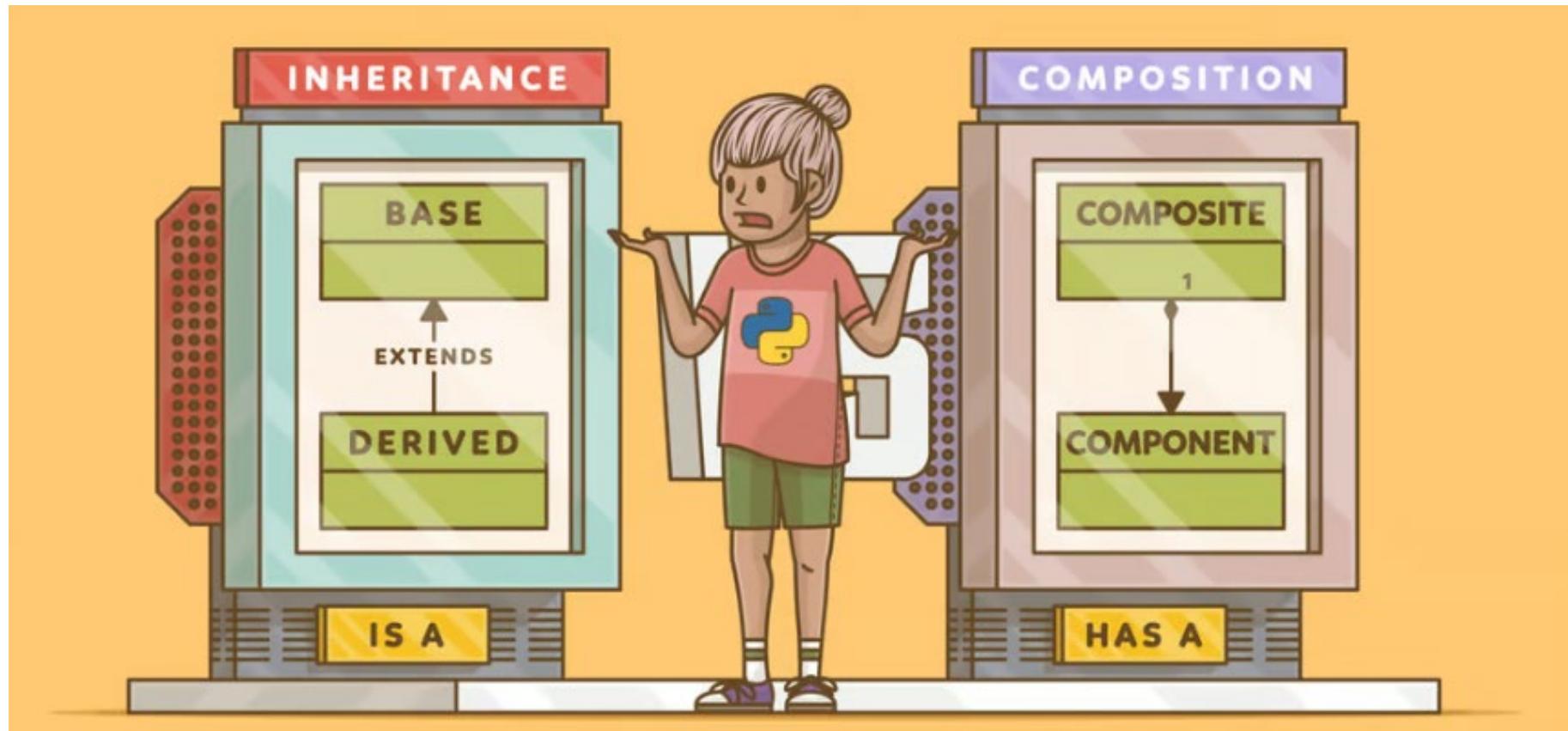
## Composition

- Une classe est composée d'une autre classe.  
La composition est forte. Leur cycle de vie sont liés.



- La classe enfant est une spécialisation de la classe parent.







## Surcharge et Redéfinition

- Surcharge (Overloading) : Réécrire, dans une même classe ou dans des classes héritées, une méthode qui a le même nom mais des paramètres différents.
- Redéfinition (Overriding) : Réécrire, dans une classe héritée, une méthode qui a le même nom et les mêmes paramètres.

# Redéfinition et Surcharge

## Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}

class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name.  
Same parameter

## Overloading

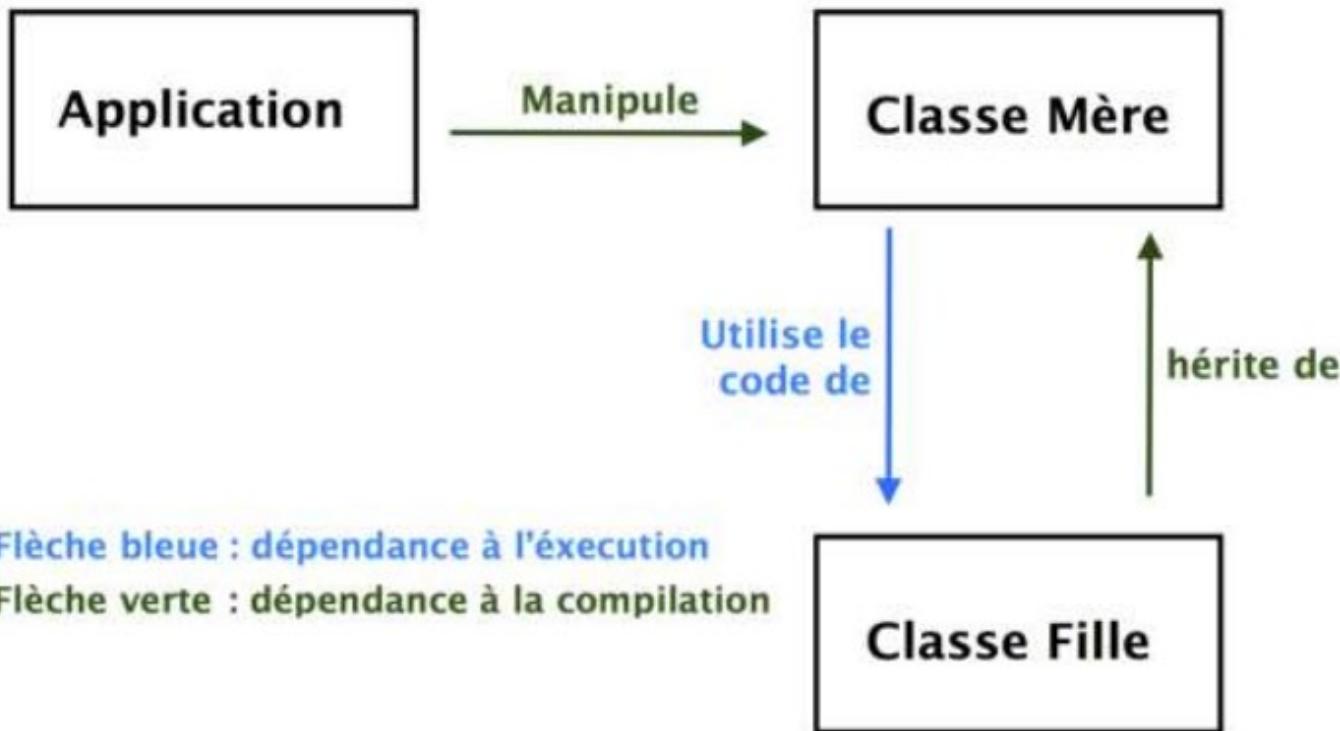
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name.  
Different Parameter

# Polymorphisme





## Polymorphisme

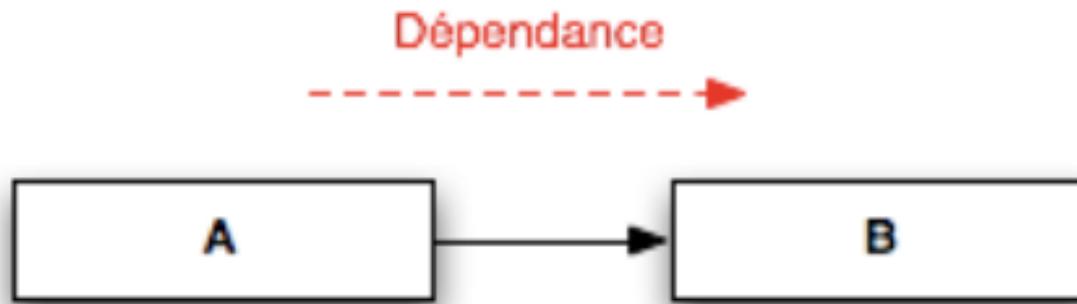


Schéma 1 : Sans polymorphisme, l'application (A) dépend de l'objet (B) qu'elle utilise. En conséquence, toute modification apportée à (B) oblige une re-compilation de (A).

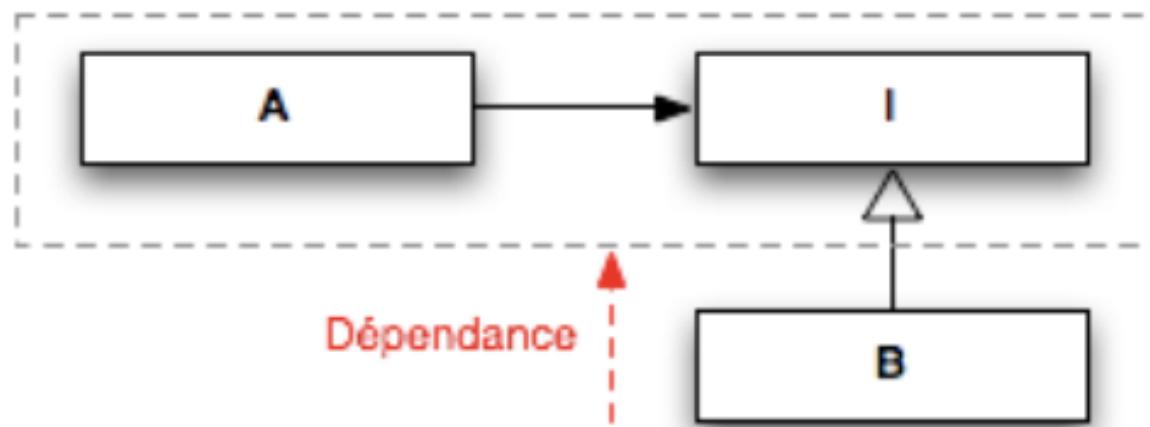
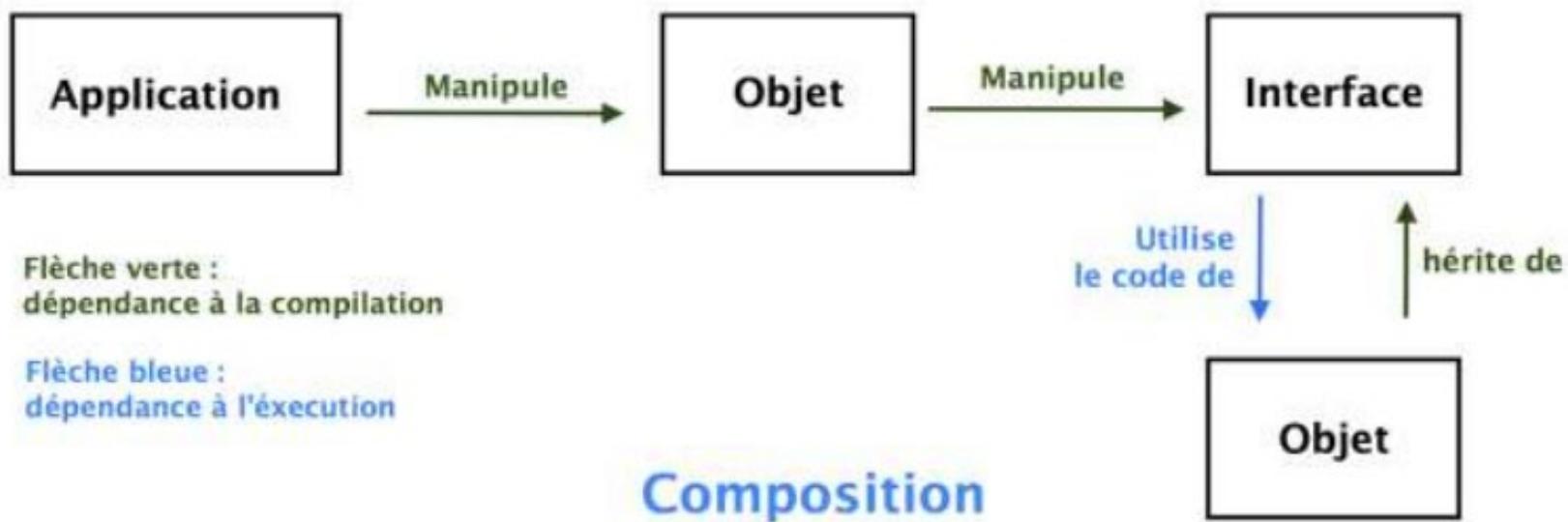


Schéma 2 : Grâce au polymorphisme, l'application (A) ne dépend plus de l'objet (B) qu'elle utilise, mais seulement d'une Interface (I). L'implémentation de (B) peut changer sans que (A) n'ait besoin d'être re-compilée.



- En conclusion, la composition est la meilleure façon du réutiliser des objets car elle :
  - renseigne immédiatement sur les intentions du programmeur,
  - n'induit pas de couplage fort,
  - favorise un design souple et maintenable.
- Son utilisation va dans le sens des objectifs de la POO et ne constitue en rien un aveu de faiblesse de cette dernière. Au contraire, l'utilisation de la composition est une bonne pratique de programmation orientée-objet qui favorise l'utilisation du polymorphisme

- Réutilisation des objets existants
  - mauvaise méthode : héritage d'implémentation
  - bonne méthode : composition + méthode de transfert (forwarding)
- Réutilisation du code client (celui qui utilise des objets)
  - bonne méthode : héritage d'interface (donc polymorphisme)



## Héritage d'implémentation

- L'héritage d'implémentation, aussi connu sous le nom d'héritage de classe, est le type d'héritage le plus couramment utilisé. Il permet à une classe d'hériter d'une autre classe, souvent appelée classe parente ou superclasse, de sorte que la classe enfant (ou sous-classe) obtienne toutes les propriétés et méthodes publiques et protégées de la classe parente. Cela facilite la réutilisation du code et le polymorphisme, permettant aux développeurs de construire des architectures logicielles plus modulaires et flexibles.

## Exemple

```
class Parent {  
    public void methodeParent() {  
        // Implémentation  
    }  
}  
  
class Enfant extends Parent {  
    // La classe Enfant hérite de Parent  
}
```

## Héritage d'interface

- L'héritage d'interface, d'autre part, ne concerne pas l'héritage direct des implémentations de méthode, mais plutôt l'héritage de signatures de méthode. Une interface est un contrat qui spécifie ce qu'une classe doit faire, mais pas comment elle doit le faire. Les classes qui implémentent une interface doivent fournir des implémentations pour toutes les méthodes définies dans l'interface. Cela permet un niveau élevé de modularité, rendant le code plus flexible et plus facile à maintenir, en permettant à différentes classes d'implémenter les mêmes interfaces de manières différentes.

```
interface MonInterface {  
    void maMethode();  
}  
  
class MaClasse implements MonInterface {  
    public void maMethode() {  
        // Implémentation spécifique  
    }  
}
```

## Les différences

- **Nature:** L'héritage d'implémentation est un mécanisme par lequel une classe reçoit le comportement (méthodes) et l'état (champs) d'une autre classe. L'héritage d'interface est un contrat qui spécifie un ensemble de méthodes qu'une classe doit implémenter.
- **Utilisation:** L'héritage d'implémentation est utilisé pour exprimer une relation "est-un" entre la classe enfant et la classe parente, indiquant que la classe enfant est un type spécialisé de la classe parente. L'héritage d'interface est utilisé pour spécifier une capacité ou un ensemble de capacités qu'une classe doit fournir, sans imposer la manière dont ces capacités sont réalisées.

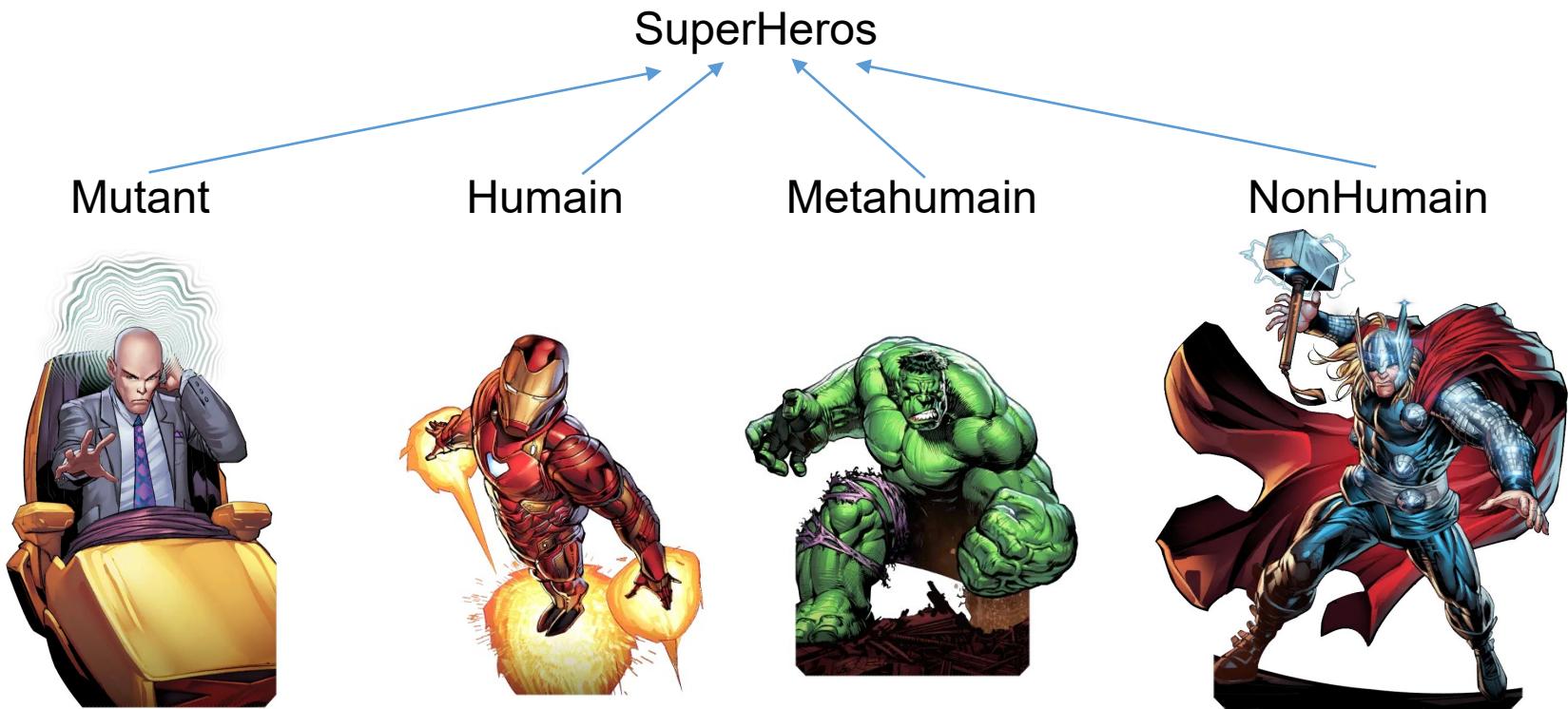
- **Flexibilité:** Les interfaces offrent une plus grande flexibilité, permettant à une classe d'implémenter plusieurs interfaces, alors que l'héritage d'implémentation dans de nombreux langages de programmation ne permet pas à une classe d'hériter de plusieurs superclasses (héritage multiple).



## Exercice !



- But : Créer une classe SuperHeros et des classes-enfants pour les différentes types de super-héros



## SuperHeros

nom : string

equipes : [ ]

puissance : int

sePresenter( ) : void

rejoindreEquipe(string) : void

taperSuperVilain(SuperVilain) : void

## Mutant

mutations : [ ]

sousType : string

sePresenter( ) : void

taperSuperVilain(SuperVilain) : void

## Humain

technologies : [ ]

capacités : [ ]

sePresenter( ) : void

taperSuperVilain(SuperVilain) : void

## Metahumain

pouvoirs : [ ]

origine : string

sePresenter( ) : void

taperSuperVilain(SuperVilain) : void

## NonHumain

capacités : [ ]

race : string

sePresenter( ) : void

taperSuperVilain(SuperVilain) : void

Et donc ?



- On peut créer une liste 'avengers' qui contient des objets Mutant, Humain, MetaHumain et NonHumain.

avengers = [ ironMan, thor, captainAmerica, wolverine]

Et ... ?

- On peut appeler la méthode sePresenter( ) pour chaque élément de la liste.



- Et on obtiendra alors :

"Je suis Iron-Man, je suis un humain et j'ai une armure"

"Je suis Thor, je suis un non-humain et j'ai un marteau"

"Je suis Captain America, je suis un métahumain et j'ai une force surhumaine"

"Je suis Wolverine, je suis un mutant et j'ai un facteur régénérant accéléré"

Et c'est pas tout !

## "Somehow, Palpatine returned"

nom :  
equip  
puissa  
sePre  
rejoin  
taperS

void



- Si on fait appelle, à la méthode sePresenter( ) de Jedi et de SuperHeros... ils utiliseront leur méthodes respectives, alors que les deux classes n'ont aucun rapport entre-elles !



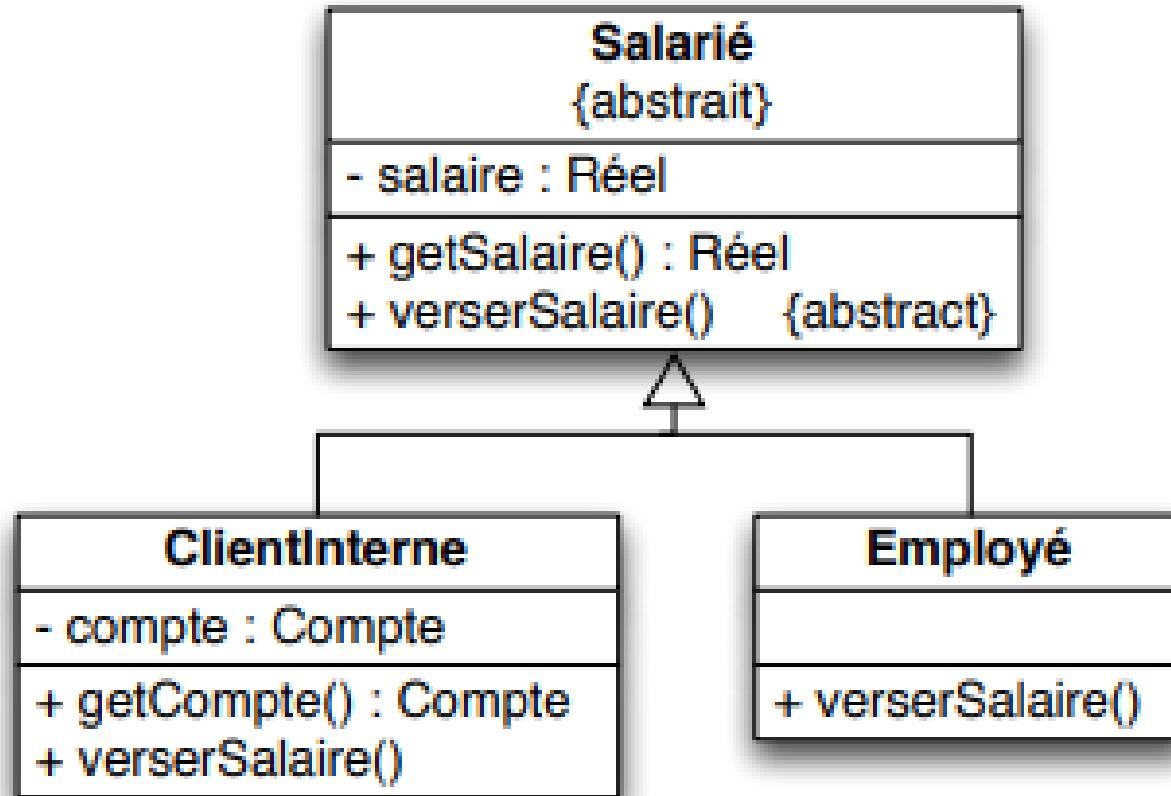




## Classe abstraite

- Une classe abstraite est un type de classe en POO, qui déclare une ou plusieurs méthodes abstraites. Ces classes peuvent avoir des méthodes abstraites ainsi que des méthodes concrètes. Elle ne peut donc pas être directement instanciée. Une classe abstraite encapsule des attributs et méthodes qui peuvent être utilisés par les instances des classes qui en héritent. L'intérêt des classes abstraites est de regrouper plusieurs classes sous un même nom de classe (par polymorphisme) ainsi que de décrire partiellement des attributs et méthodes communs à plusieurs classes.

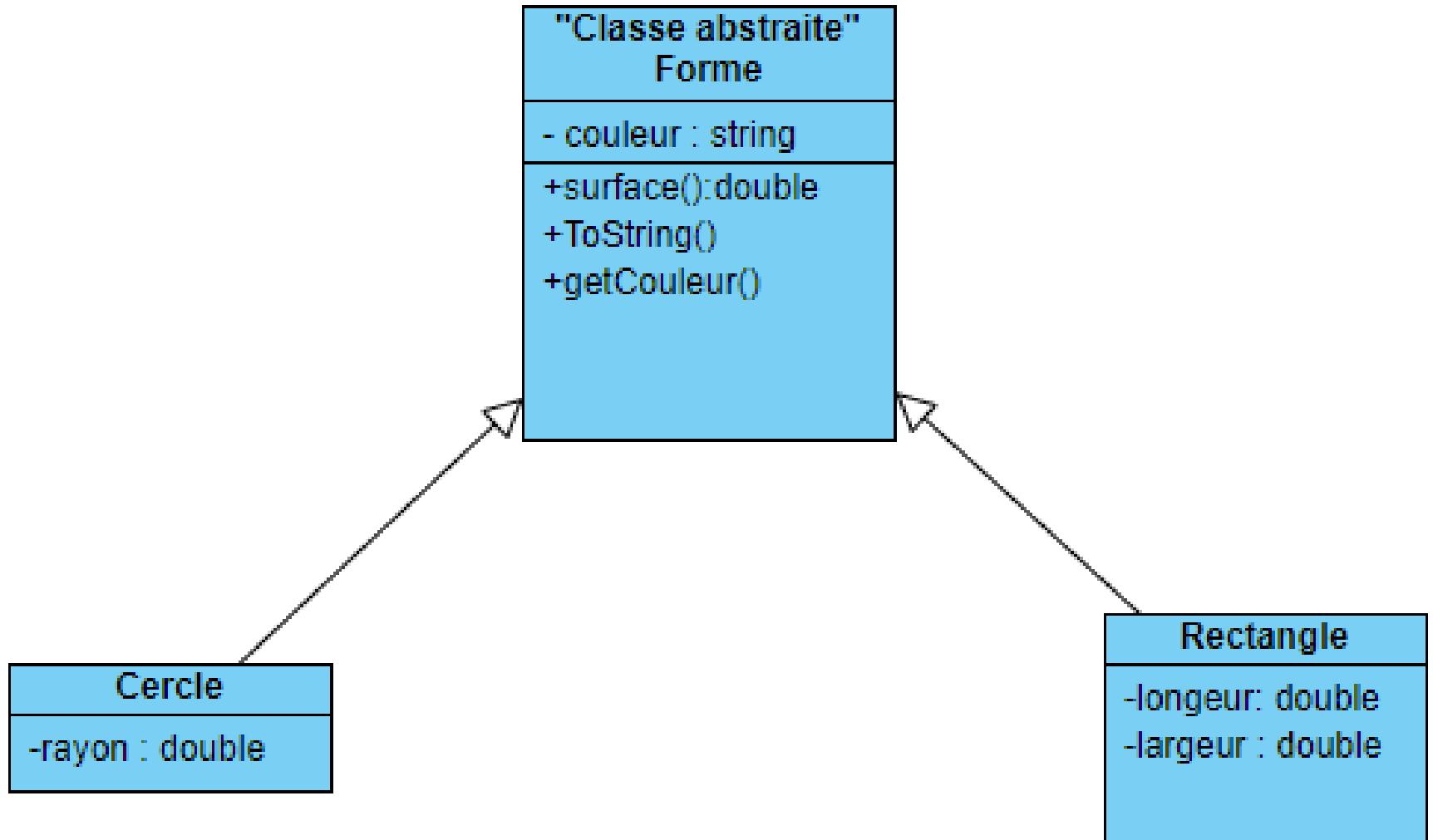
# Classe abstraite



- Encapsulation des méthodes abstraite → Traitements n'est pas dans la classe

```
public abstract class Salarie {  
    private double salaire;  
  
    public double getSalaire() {  
        return salaire;  
    }  
  
    public abstract void verserSalaire();  
}
```

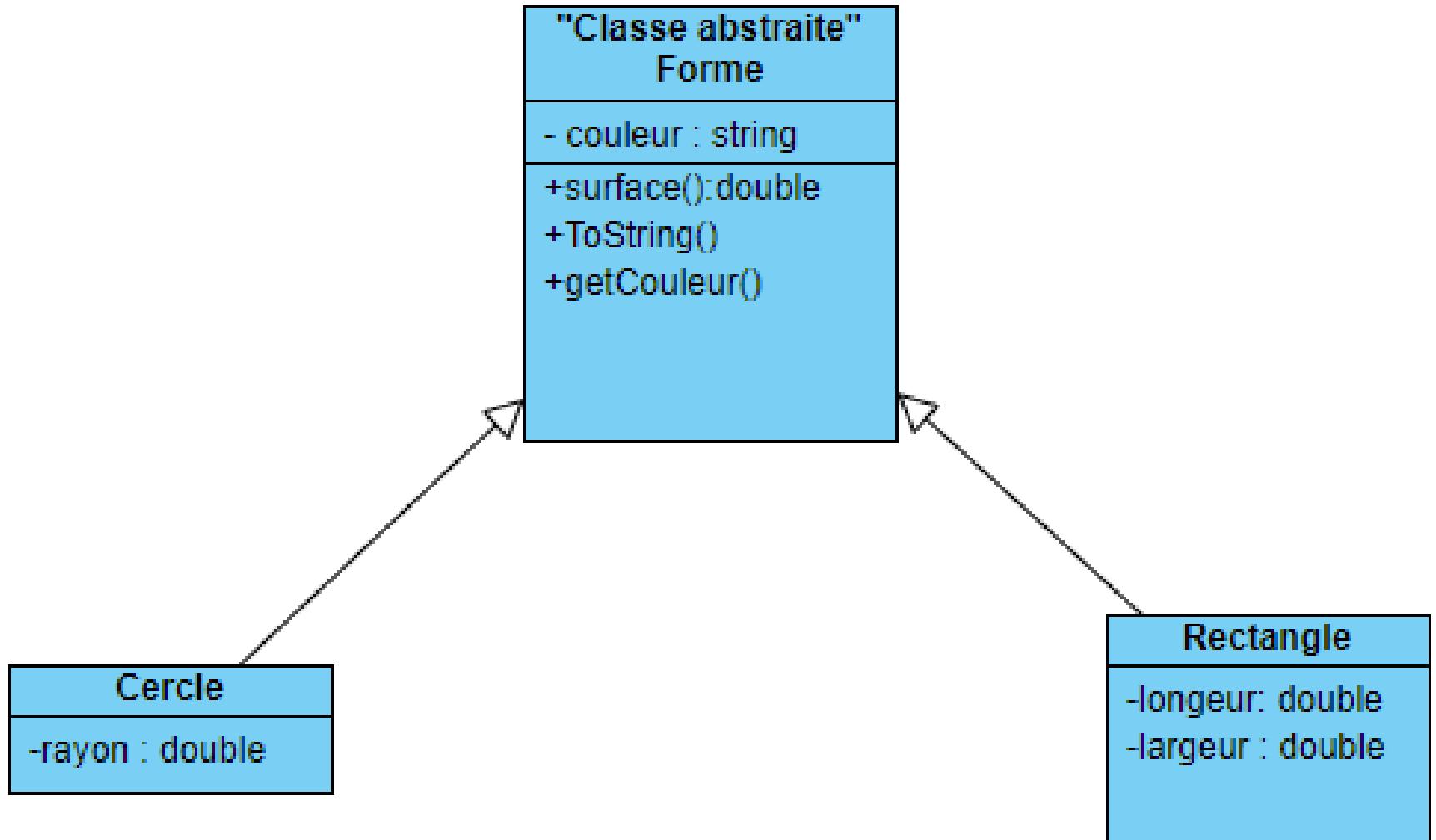
- Réduire la complexité.
- Évite la duplication de code et augmente la possibilité de réutilisation.
- Aide à renforcer la sécurité d'une application ou d'un programme car seuls les détails importants sont fournis à l'utilisateur.



## Exercices : Forme –Rect-Cercle

- Commençons par créer la superclasse Forme. Notez l'utilisation du mot clé « abstract » dans la définition de la classe. Ceci marque la classe comme abstraite, ce qui signifie qu'elle ne peut pas être instanciée directement. Nous définissons deux méthodes appelées surface() et toString() en tant que méthode abstraite. De cette façon, vous laissez l'implémentation de ces méthodes aux sous-classes de la classe Forme.

```
1. abstract class Forme
2. {
3.     String couleur;
4.
5.     // Méthodes abstraites
6.     abstract double surface();
7.     public abstract String toString();
8.
9.     // Classe abstraite peut avoir un constructeur
10.    public Forme(String couleur) {
11.        System.out.println("Constructeur de la classe Forme est appelé");
12.        this.couleur = couleur;
13.    }
14.
15.    // Méthode concrète
16.    public String getCouleur() {
17.        return couleur;
18.    }
19. }
```



- La classe « Cercle » hérite toutes les propriétés de la classe mère « Forme », mais doit fournir sa propre implémentation pour la méthode surface() et toString(). Dans ce cas, nous multiplions la valeur de  $\pi$  par le carré de son rayon, et on définit un message spécifique dans la classe Cercle avec la méthode toString().

```
1. class Cercle extends Forme
2. {
3.     double rayon;
4.
5.     public Cercle(String couleur, double rayon) {
6.
7.         // appel du constructeur de la classe Forme
8.         super(couleur);
9.         System.out.println("Constructeur de la classe Cercle est appelé");
10.        this.rayon = rayon;
11.    }
12.
13.    @Override
14.    double surface() {
15.        return Math.PI * Math.pow(rayon, 2);
16.    }
17.
18.    @Override
19.    public String toString() {
20.        return "La couleur de la cercle est " + super.couleur +
21.               " et la surface est : " + surface();
22.    }
23.
24. }
```

- La classe « Rectangle » a également sa propre implémentation de la méthode surface() et toString(). Dans ce cas, il suffit de multiplier longueur par largeur. Ainsi un message spécifique dans la classe Rectangle avec la méthode toString().

```
1. class Rectangle extends Forme{  
2.  
3.     double longueur;  
4.     double largeur;  
5.  
6.     public Rectangle(String couleur, double longueur, double largeur) {  
7.         // appel du constructeur de la classe Forme  
8.         super(couleur);  
9.         System.out.println("Constructeur de la classe Rectangle est  
appelé");  
10.        this.longueur = longueur;  
11.        this.largeur = largeur;  
12.    }  
13.  
14.    @Override  
15.    double surface() {  
16.        return longueur * largeur;  
17.    }  
18.  
19.    @Override  
20.    public String toString() {  
21.        return "La couleur de la rectangle est " + super.couleur +  
22.               " et la surface est : " + surface();  
23.    }  
24.  
25. }
```

## Testons notre code

```
1.  public class Test
2.  {
3.      public static void main(String[] args)
4.      {
5.          Forme f1 = new Cercle("Bleu", 3.2);
6.          Forme f2 = new Rectangle("Rouge", 3, 6);
7.
8.          System.out.println(f1.toString());
9.          System.out.println(f2.toString());
10.     }
11. }
```

La sortie est comme suite:

Constructeur de la classe Forme est appelé

Constructeur de la classe Cercle est appelé

Constructeur de la classe Forme est appelé

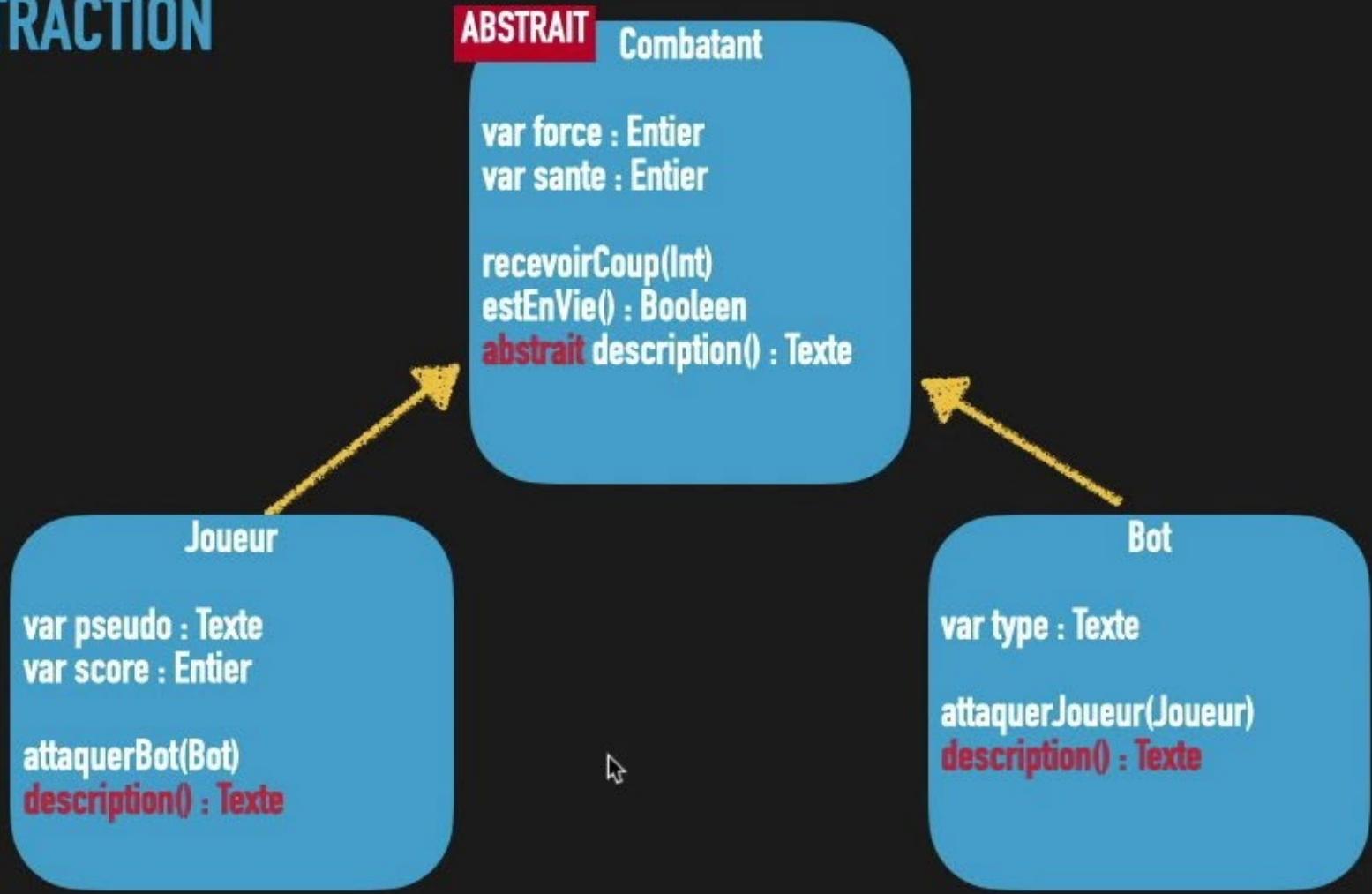
Constructeur de la classe Rectangle est appelé

La couleur de la cercle est Bleu et la surface est : 32.169908772759484

La couleur de la rectangle est Rouge et la surface est : 18.0

## Exemple qui vous parle

### ABSTRACTION



- L'abstraction est un concept général que vous pouvez trouver dans le monde réel ainsi que dans les langages POO. Tous les objets du monde réel, comme votre voiture, ou votre maison, masquant les détails internes fournissent une abstraction.
- Ces abstractions facilitent beaucoup la gestion de la complexité en les divisant en petites parties. Dans le meilleur des cas, vous pouvez les utiliser sans comprendre comment ils fournissent les fonctionnalités. Et cela vous aide à diviser la complexité de votre logiciel en parties contrôlable.



- THEORIE ABSTRAIT

- Les classes abstraites sont un concept fondamental dans la programmation orientée objet (POO), servant de base pour la création d'une hiérarchie de classes. Une classe abstraite est une classe qui ne peut pas être instanciée directement. Autrement dit, vous ne pouvez pas créer d'objets à partir d'une classe abstraite. Au lieu de cela, elle est destinée à être une classe parente à partir de laquelle d'autres classes dérivent et héritent. Voici les caractéristiques principales des classes abstraites et leur utilisation :

## Caractéristiques classe abstraite

- **Instantiation interdite** : Vous ne pouvez pas créer d'objets d'une classe abstraite directement.
- **Méthodes abstraites** : Une classe abstraite peut contenir des méthodes abstraites. Une méthode abstraite est une méthode qui est déclarée dans la classe abstraite, mais elle doit être implémentée par les classes dérivées. Les méthodes abstraites n'ont pas de corps ; elles se contentent de définir la signature de la méthode.

## Caractéristiques classe abstraite

- **Méthodes concrètes** : Les classes abstraites peuvent également contenir des méthodes concrètes avec une implémentation complète. Cela permet une certaine forme de réutilisation du code, puisque les classes dérivées peuvent hériter de ces méthodes concrètes et les utiliser directement ou les surcharger si nécessaire.
- **Constructeurs** : Bien qu'une classe abstraite ne puisse pas être instanciée directement, elle peut avoir des constructeurs. Ces constructeurs ne sont pas utilisés pour créer des instances de la classe abstraite elle-même, mais ils sont utilisés lors de l'instanciation des sous-classes pour initialiser l'état commun hérité de la classe abstraite.

## Utilisation

- Les classes abstraites sont utilisées comme une base pour d'autres classes. Elles permettent de définir un modèle (ou contrat) que toutes les sous-classes doivent suivre, tout en permettant une certaine flexibilité dans la manière dont les sous-classes implémentent les détails spécifiques. Cela est particulièrement utile dans les situations suivantes :

## Utilisation

- **Lorsque vous avez des parties de votre classe qui sont communes et d'autres qui doivent être spécifiques à chaque sous-classe.** Les parties communes peuvent être implémentées dans la classe abstraite (comme des méthodes concrètes), tandis que les parties spécifiques sont définies comme des méthodes abstraites pour être implémentées par les sous-classes.
- **Pour fournir un cadre commun pour un ensemble de sous-classes,** assurant que toutes les sous-classes partagent une structure et une méthodologie communes, tout en ayant la liberté de personnaliser et d'étendre cette base selon leurs besoins spécifiques.

```
abstract class Animal {  
    // Méthode abstraite  
    abstract void makeSound();  
  
    // Méthode concrète  
    public void eat() {  
        System.out.println("This animal eats food.");  
    }  
}  
  
class Dog extends Animal {  
    // Implémentation de la méthode abstraite  
    void makeSound() {  
        System.out.println("Bark");  
    }  
}
```



## Interface

- Une interface décrit des objets mais uniquement en terme de méthodes abstraites. Une interface ne peut contenir ni attribut, ni méthode implémentée. Le terme d'héritage n'est pas utilisé entre classes et interfaces : on dit qu'une classe implémente une interface. Lorsqu'une classe implémente une interface, elle doit redéfinir toutes ses méthodes abstraites ou bien être abstraite. Une interface peut hériter d'une autre interface

- **Définition 1 :**

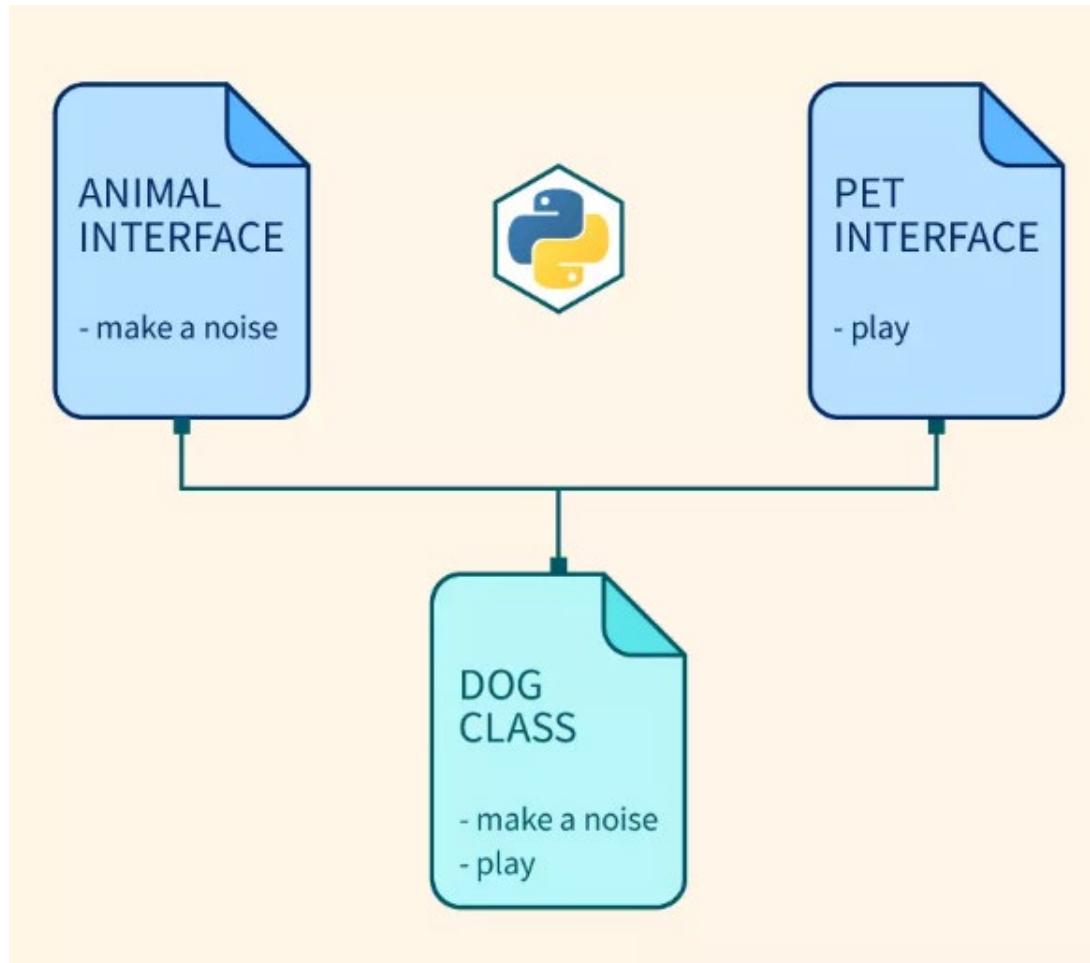
Une interface est une forme particulière de classe où **toutes les méthodes sont abstraites**.

- **Définition 2 :**

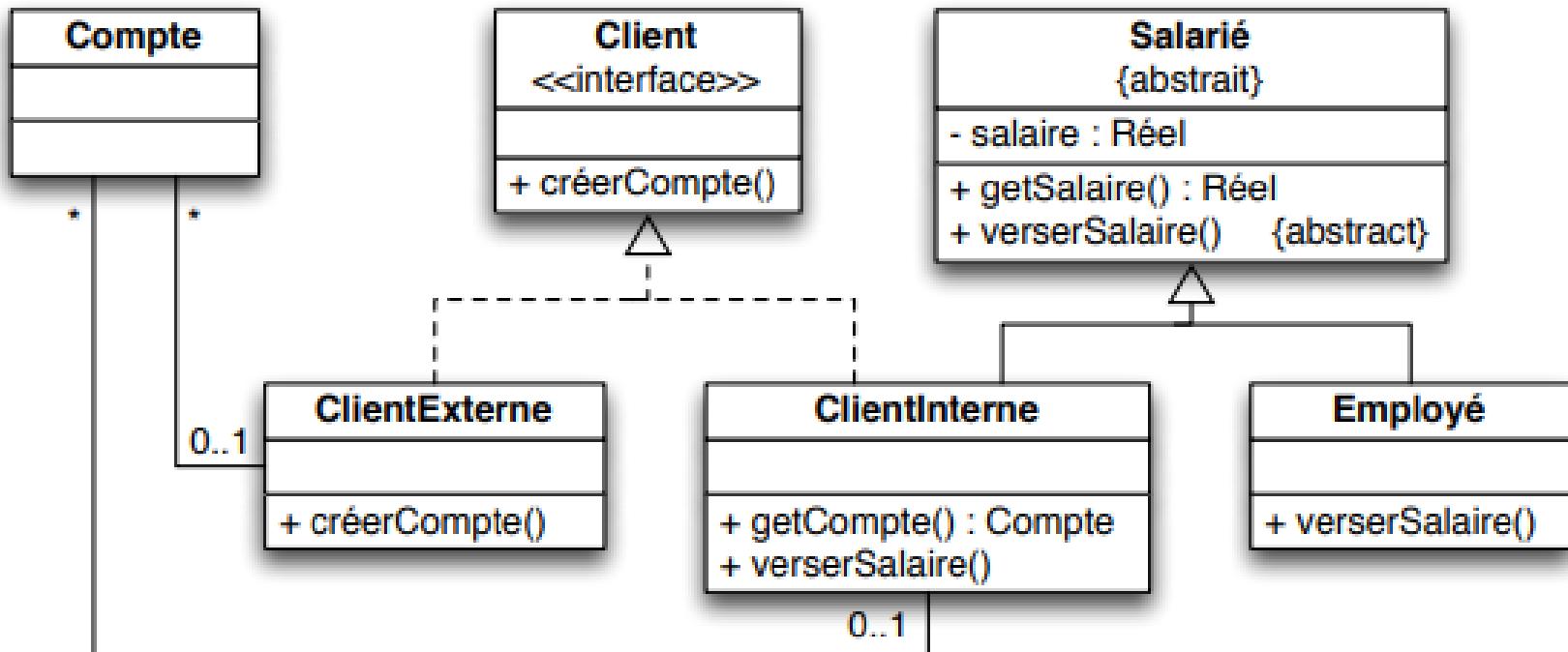
Lorsqu'une classe **implémente** une interface, elle indique ainsi qu'elle **s'engage** à fournir une implémentation (c'est-à-dire un corps) pour chacune des méthodes abstraites de cette interface.

- **Règle :**

Si une classe **implémente** plus d'une interface, elle doit implémenter **toutes les méthodes abstraites de chacune des interfaces**.



# Interface et classe abstraite



```
public interface Client {  
  
    public void creerCompte();  
  
}
```

```
public class ClientInterne extends Salarie implements Client {  
    private Compte compte;  
  
    public void creerCompte() {  
        compte = new Compte();  
    }  
  
    public void verserSalaire() {  
        compte.credite(salaire);  
    }  
}
```

## Interface

I only know method names that I will require for my job to be done.  
You have to provide body for those methods.

Interface



Sure, I will definitely provide body to all your methods but in my way.

Implementer

## Abstract class

Some methods I know.  
Some methods I don't know and I will depend upon you to provide.

Implementer 2

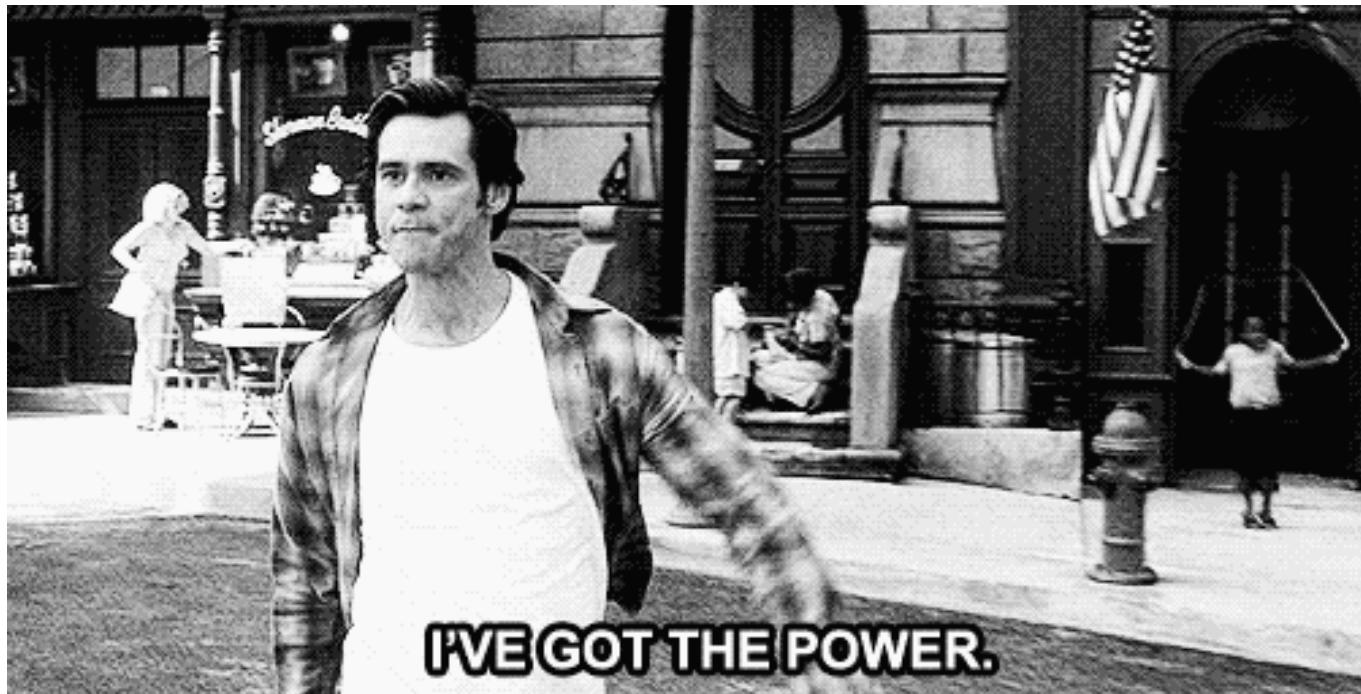
abstract class

Some methods I know.  
Some methods I don't know and I will depend upon you to provide.

Implementer 1









- Les interfaces jouent un rôle crucial dans la programmation orientée objet (POO), particulièrement en ce qui concerne l'abstraction et le polymorphisme. Une interface est un contrat qui définit un ensemble de méthodes publiques sans implémentations. Les classes qui choisissent d'implémenter une interface doivent fournir des implémentations concrètes pour toutes les méthodes déclarées dans l'interface.

## Caractéristiques (orienté java)

- **Méthodes abstraites** : Les interfaces se composent principalement de méthodes abstraites, c'est-à-dire des méthodes sans corps. Cela définit un "contrat" que les classes implémentantes doivent respecter, en fournissant des implémentations concrètes pour ces méthodes.
- **Pas d'état** : Les interfaces ne peuvent pas contenir d'état. Elles ne peuvent pas avoir de champs d'instance. Toutefois, à partir de Java 8, les interfaces peuvent avoir des champs statiques qui sont finaux et immuables.

- **Méthodes par défaut (Java 8 et ultérieures)** : Java 8 a introduit la possibilité pour les interfaces d'avoir des méthodes par défaut, qui ont un corps. Cela permet aux développeurs d'ajouter de nouvelles méthodes à une interface sans affecter les classes qui l'implémentent déjà.
- **Méthodes statiques (Java 8 et ultérieures)** : Les interfaces peuvent contenir des méthodes statiques avec une implémentation depuis Java 8. Ces méthodes appartiennent à l'interface elle-même plutôt qu'à l'instance d'une classe qui l'implémente.
- **Héritage multiple** : Contrairement aux classes, Java permet à une interface d'hériter de plusieurs autres interfaces. Une classe peut également implémenter plusieurs interfaces, offrant une forme d'héritage multiple.



## Utilisations

- Les interfaces sont utilisées pour définir un ensemble de méthodes que différentes classes peuvent implémenter, permettant à des objets de types différents d'interagir de manière uniforme. Voici quelques utilisations courantes des interfaces :
- **Définir un contrat commun** : Les interfaces établissent un contrat que différentes classes peuvent suivre, permettant à ces classes d'interagir avec le reste du système de manière prévisible.

- **Découplage** : En programmant pour des interfaces plutôt que pour des implémentations spécifiques, les composants de votre système deviennent moins couplés, ce qui facilite la maintenance et l'évolution du code.
- **Polymorphisme** : Les interfaces permettent le polymorphisme, où une référence d'interface peut pointer vers des objets de n'importe quelle classe qui implémente cette interface, permettant ainsi à différentes implémentations d'être interchangeables.

```
interface Animal {  
    void makeSound();  
}  
  
class Dog implements Animal {  
    public void makeSound() {  
        System.out.println("Bark");  
    }  
}  
  
class Cat implements Animal {  
    public void makeSound() {  
        System.out.println("Meow");  
    }  
}
```

# Warning PYTHONNNNNNN



- En Python, les interfaces ne sont pas implémentées de la même manière que dans des langages statiquement typés comme Java, car Python est un langage dynamiquement typé. Python utilise un concept appelé "duck typing", où les types d'objets ne sont pas aussi importants que les méthodes ou les attributs qu'un objet possède. Cela signifie que si un objet peut effectuer une action requise (c'est-à-dire, il a les méthodes ou attributs nécessaires), il peut être utilisé indépendamment de sa classe.

- Toutefois, pour définir une interface en Python, on peut utiliser les "Abstract Base Classes" (ABC) du module abc. Les ABC permettent de définir des méthodes abstraites qui doivent être implémentées par les sous-classes, offrant ainsi une manière de créer des interfaces.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Bark")

class Cat(Animal):
    def make_sound(self):
        print("Meow")
```

- Dans cet exemple, Animal est une classe abstraite qui sert d'interface, avec une méthode abstraite `make_sound`. Les classes Dog et Cat héritent de Animal et fournissent des implémentations concrètes pour la méthode `make_sound`.
- L'utilisation des classes abstraites en Python avec le module abc force les sous-classes à implémenter les méthodes abstraites définies dans la classe parente abstraite, simulant ainsi le comportement des interfaces dans des langages statiquement typés.



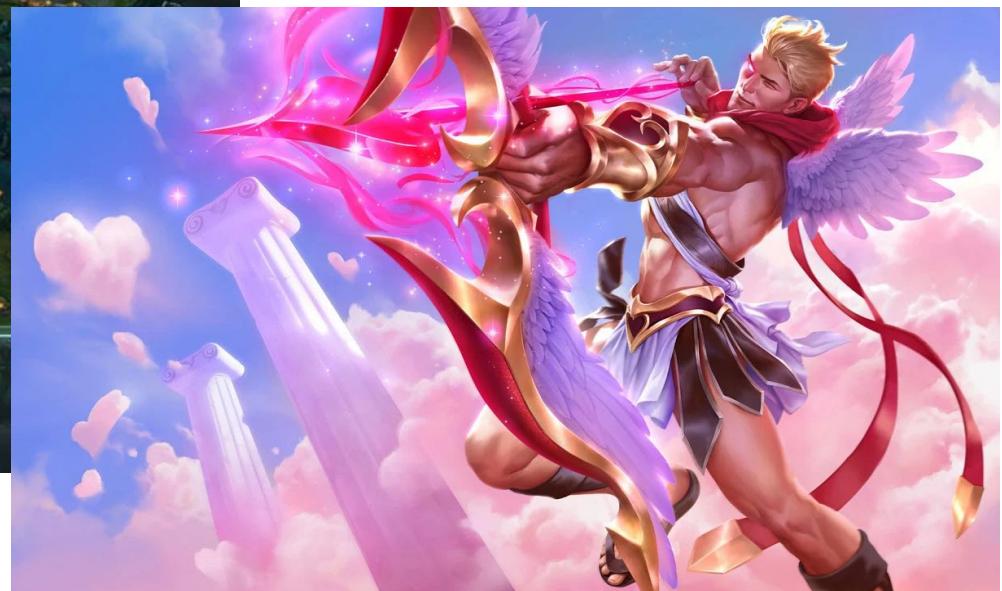
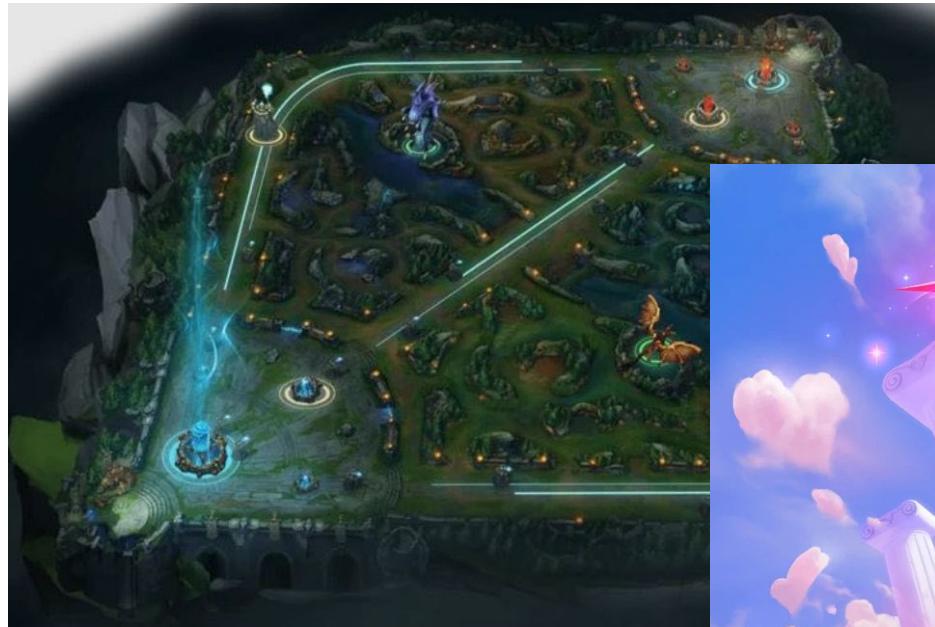


# Exercice !



- But : Imaginer, de manière simplifiée, le fonctionnement de LoL.





# Champion

## Champion (abstrait)

- vie : int
- niveau: int
- passif : string
- spells : [Spell]
- +toString()
- +seDeplacer( )
- +autoAttack( )
- +lancerSpell( )

```
from abc import ABC, abstractmethod

class Champion(ABC):
    def __init__(self, vie=0, passif="", spells=None):
        if spells is None:
            spells = []
        self._vie = vie
        self._niveau = 1
        self._passif = passif
        self.spells = spells

    @abstractmethod
    def toString(self):
        pass

    @abstractmethod
    def seDeplacer(self):
        pass

    @abstractmethod
    def autoAttack(self, target):
        pass

    @abstractmethod
    def lancerSpell(self, spell, target=0):
        pass
```

```
class Champion(ABC):...

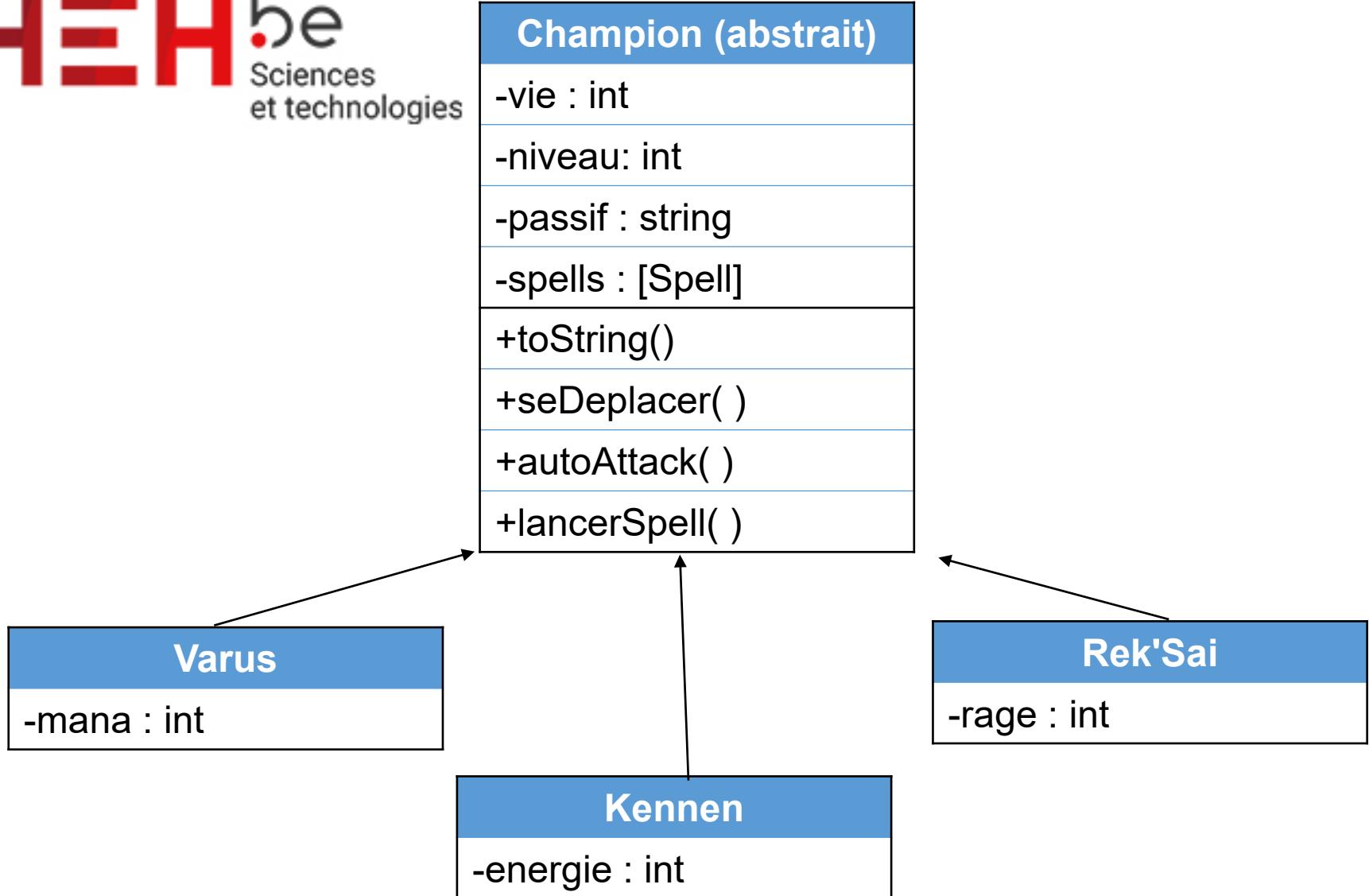
class Varus(Champion):
    def __init__(self):
        super().__init__(600, "Vengeance incarnée",
                         ["Flèche perforante", "Carquois meurtri", "Pluie de flèches", "Chaîne corruptrice"])
        self._mana = 360

    def toString(self):
        print("Je suis Varus, Flèche de la Vengeance")

    def seDeplacer(self):
        # Code pour se déplacer sur la map
        pass

    def autoAttack(self, target):
        # Code pour auto-attaquer un champion ennemi
        pass

    def lancerSpell(self, spell, target=0):
        # Code pour lancer un spell
        pass
```



- On peut alors imaginer une classe Map

Map
-tours : [Tour]
-nexus: [Nexus]
-temps: time
-champions: [Champion]
+toString()
+finDePartie( )
+deplacerChampion(Champion)

Map
-tours : [Tour]
-nexus: [Nexus]
-temps: time
-champions: [Champion]
+toString()
+finDePartie( )
+deplacerChampion(Champion)



Champion (abstrait)
-vie : int
-niveau: int
-passif : string
-spells : [Spell]
+toString()
+seDeplacer( )
+autoAttack( )
+lancerSpell( )

Varus
-mana : int



***Tell me more! Tell me more!***

- On pourrait imaginer une interface EntiteDestructible qui serait implémentée par les objets sur la map pouvant être détruit par le joueur
- Exemple :
  - Nexus
  - Tour
  - Plante

## IEntiteDestructible

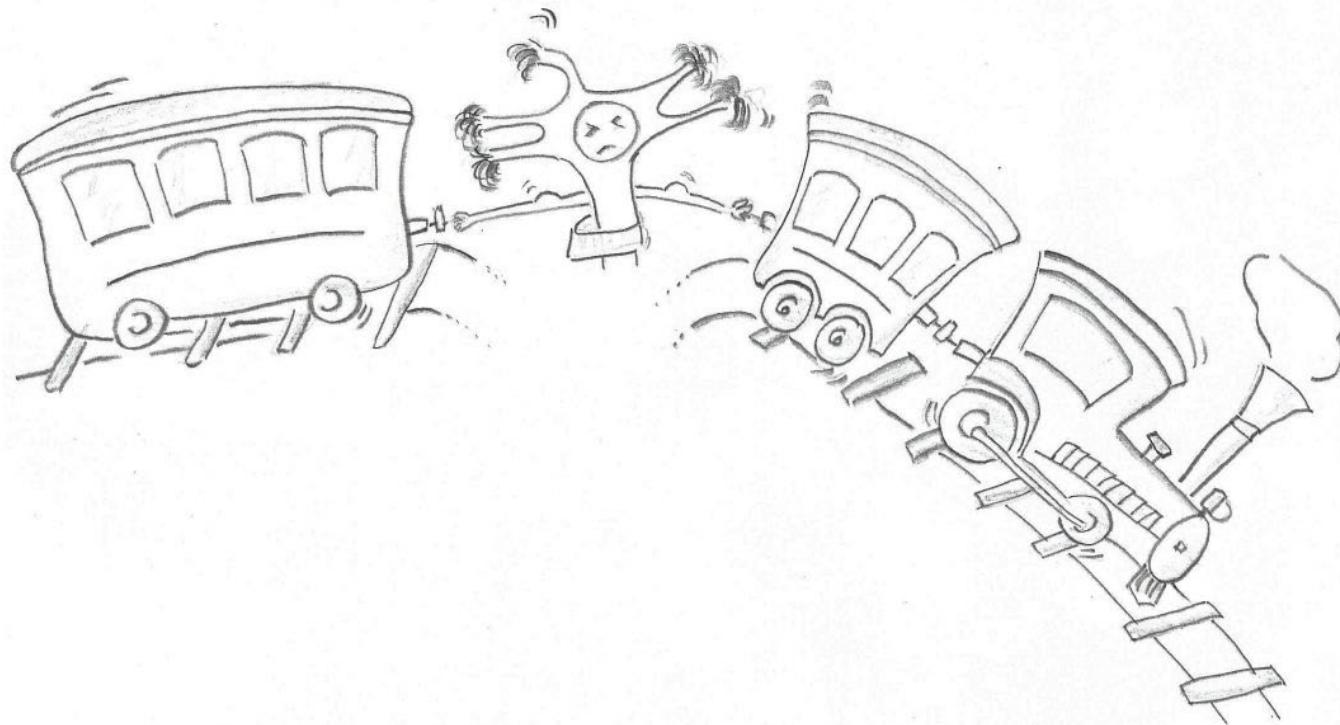
+estDetruit( )

+changeSkin( )





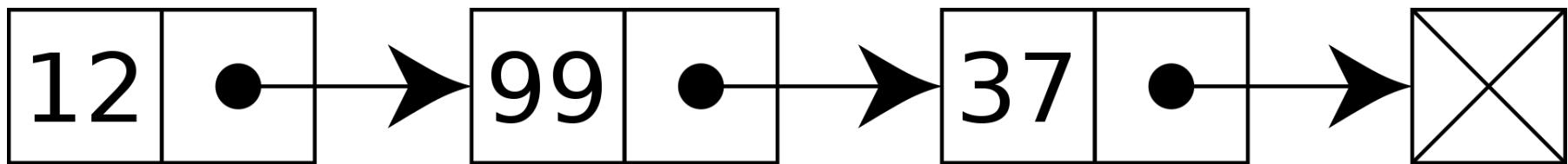
# Pourquoi ?



On se souvient...

- Les listes simplement chaînées, ...

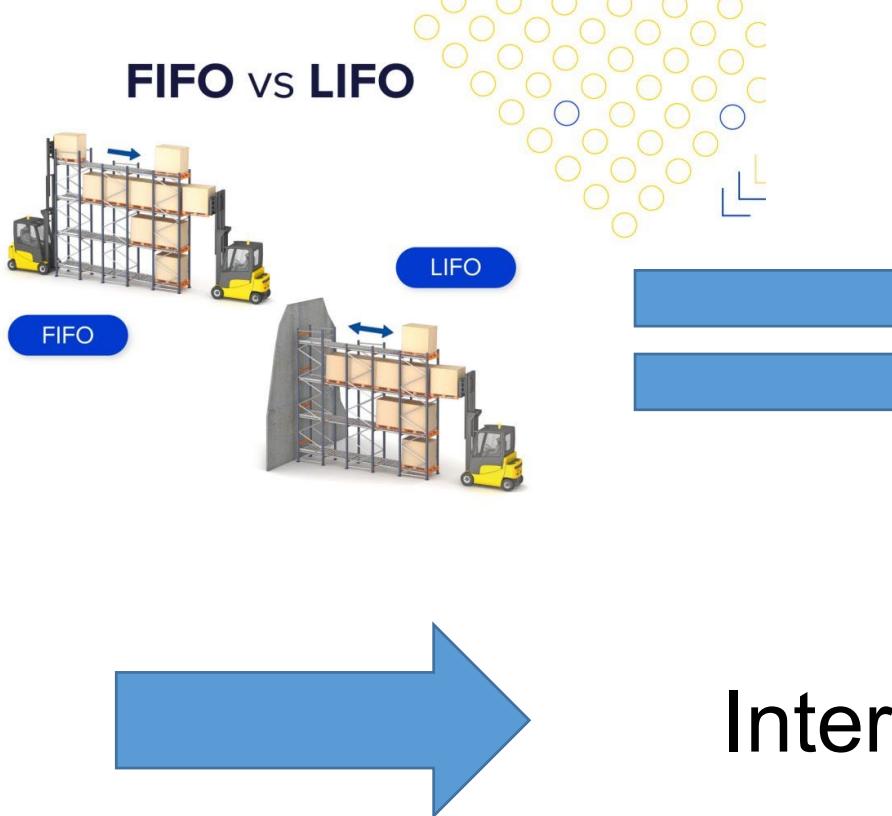




**HOW FASCINATING**



**PLEASE TELL ME MORE**



IFile
+enfiler(Element)
+defiler( ) : Element
+estVide( ) : bool
+seCompter( ) : int

IPile
+enpiler(Element)
+depiler( ) : Element
+estVide( ) : bool
+seCompter( ) : int



OU







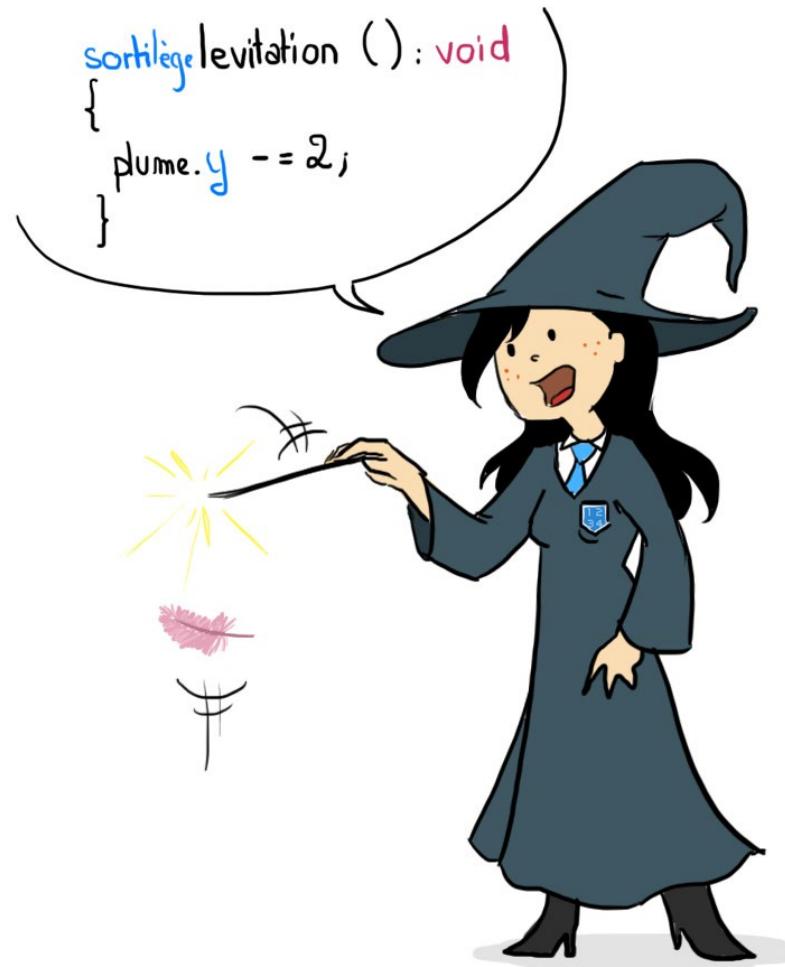


# Programmation = magie



- Formule
- Connaissance
- Temps et logique





# L'invocations des golems

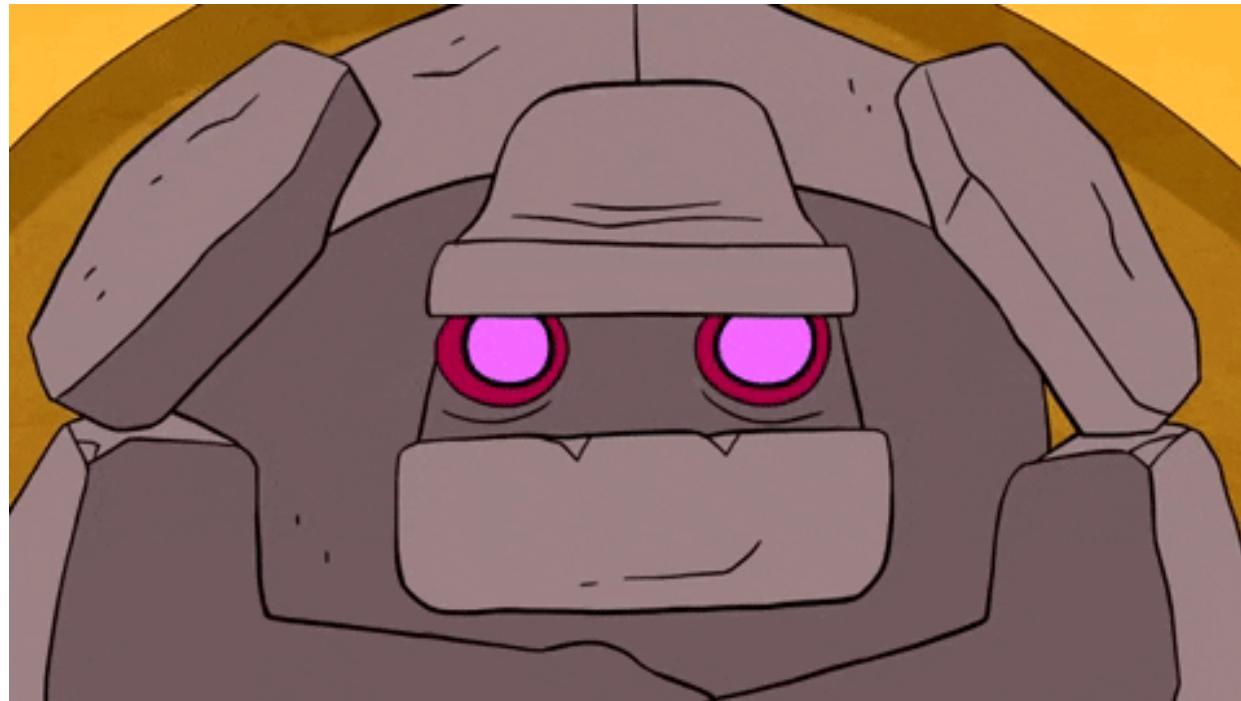


python<sup>®</sup>



# Formule magique

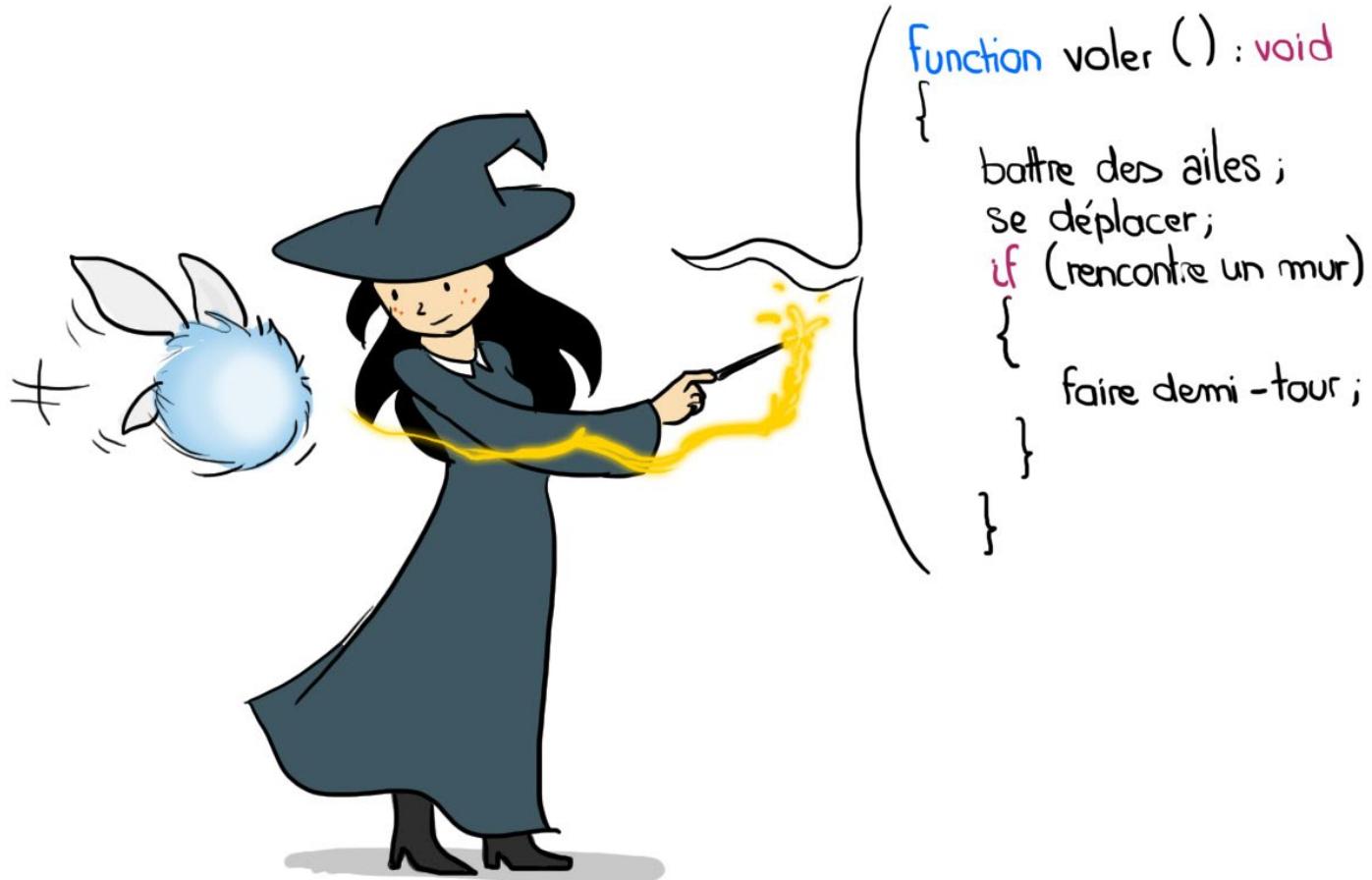




## Enchantement de base

```
var taille : Number = 80 cm;  
var densitéPoids : Number = 67 %;  
var poids : Number = 26 kg;  
var paroles : String = "Hey!"
```





## PROJECTILE

### VARIABLES :

vitesseDeplacement  
dureeDeVie  
puissance

### FONCTIONS :

deplacement();  
toucherEnnemi();  
autodestruction();

## HEROS

### VARIABLES :

vitesseDeplacement  
pointsDeVie  
inertie

### FONCTIONS :

deplacement();  
collision();  
affecterVie();

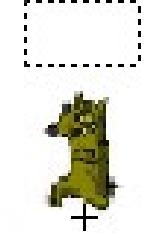
## ENNEMI

### VARIABLES :

vitesseDeplacement  
pointsDeVie  
puissance

### FONCTIONS :

choisirCible();  
pathfinding();  
mort();



# Les familles





## AKITA

### VARIABLES :

portee  
puissance  
pointsDeVie

### FONCTIONS :

detecterCible();  
affecterVie();  
attaqueBavure();

## ADJ

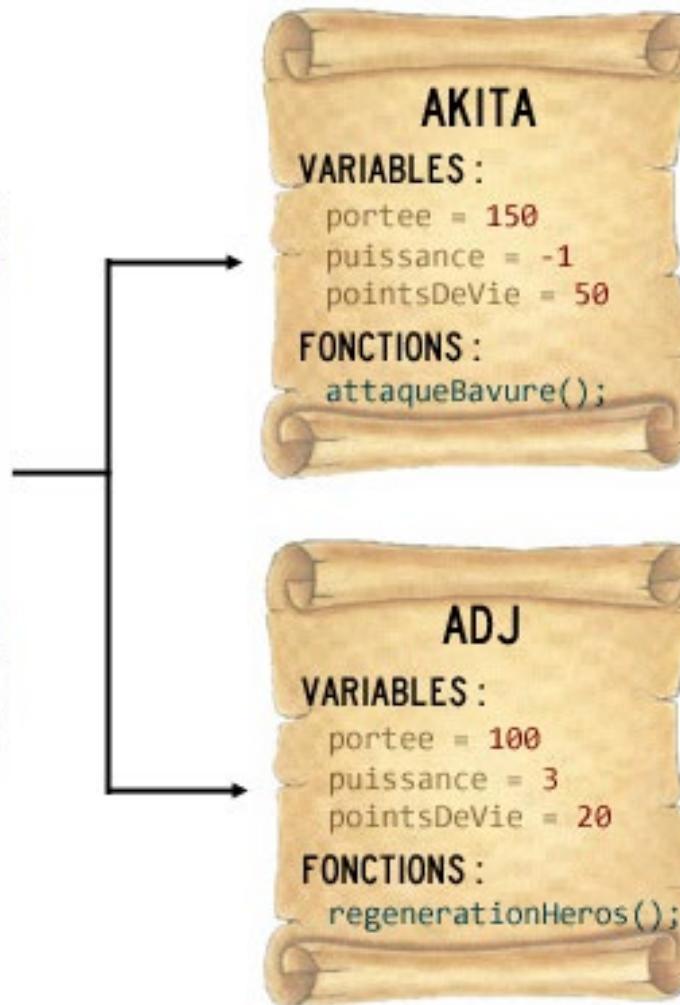
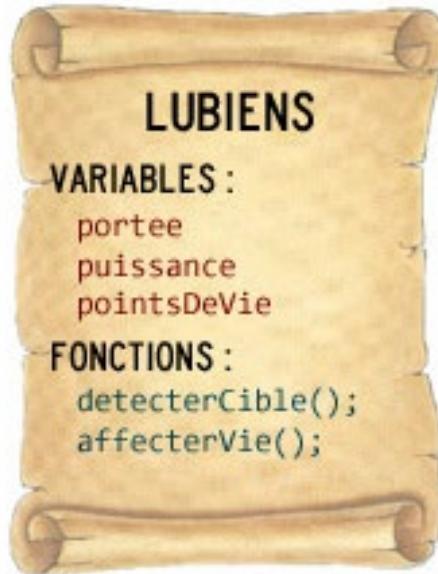
### VARIABLES :

portee  
puissance  
pointsDeVie

### FONCTIONS :

detecterCible();  
affecterVie();  
regenerationHeros();





# Senior ~~Developer~~ Wizard





