



# Analyse et Conception Orienté Objet

## *Unified Modeling Language*

**Module : Programmer en Orienté Objet**

**Filière : Développement Digital**

**Formatrice : Meriem Onzy**

**E-mail : [meriem.onzy@ofppt.ma](mailto:meriem.onzy@ofppt.ma)**

# CHAPITRE 1



## 01 ● Génie Logiciel

● Conduite de projet informatique

● Phases de développement

● Méthodes d'analyse/conception

● Modèles de développement

# Génie Logiciel

L'ingénierie logicielle, est une discipline qui concerne l'application systématique des principes, des méthodes et des pratiques de l'ingénierie pour le développement, la maintenance et l'évolution de logiciels de qualité.

# Qualité du logiciel

## Facteurs internes (concepteur)

- Réutilisabilité

- Aptitude d'un logiciel à être réutilisé, en tout ou en partie, pour d'autres applications

- Vérifiabilité

- aptitude d'un logiciel à être testé (optimisation de la préparation et de la vérification des jeux d'essai)

- Portabilité

- aptitude d'un logiciel à être transféré dans des environnements logiciels et matériels différents

- Lisibilité

## Facteurs externes (utilisateur)

- **Validité**
  - aptitude à répondre aux besoins et à remplir les fonctions définies dans le cahier des charges
- **Extensibilité**
  - facilité avec laquelle de nouvelles fonctionnalités peuvent être ajoutées à un logiciel
- **Sécurité**
  - aptitude d'un logiciel à protéger son code contre des accès non autorisés.
- **Efficacité**
  - utilisation optimale des ressources matérielles (processeur, mémoires, réseau, ...)

# CHAPITRE 1



Génie Logiciel

**Conduite de projet informatique**

Phases de développement

Méthodes d'analyse/conception

Modèles de développement

# Un Projet

Ensemble d'actions à entreprendre afin de répondre à un besoin défini dans des délais fixés, mobilisant des ressources humaines et matérielles, possédant un coût.

# Acteurs d'un projet

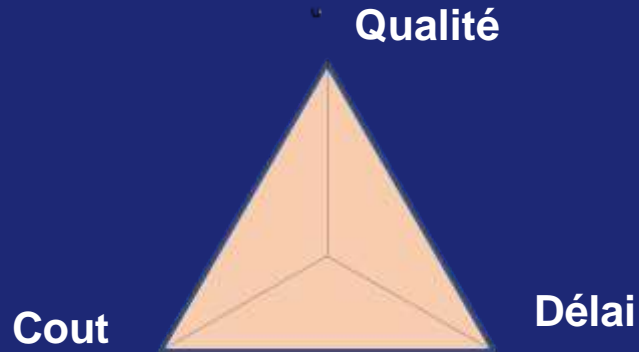
**Maitre d'ouvrage** : personne physique ou morale propriétaire de l'ouvrage. Il détermine les objectifs, le budget et les délais de réalisation.

**Maître d'œuvre**: personne physique ou morale qui reçoit mission du maître d'ouvrage pour assurer la conception et la réalisation de l'ouvrage.



# Conduite de projet

Organisation méthodologique mise en œuvre pour faire en sorte que l'ouvrage réalisé par le maître d'œuvre **réponde** aux attentes du maître d'ouvrage dans les contraintes de **délai**, **coût** et **qualité**.



# CHAPITRE 1



Génie Logiciel

Conduite de projet informatique

**Phases de développement**

Méthodes d'analyse/conception

Modèles de développement

# Activité en binôme

Proposez un ordre des différentes étapes d'un projet informatique

- Spécification des besoins
- Maintenance
- Planification
- Livraison
- Implémentation (Codage)
- Conception (Spécification technique)
- Tests unitaires
- Intégration et tests
- Analyse (Spécification formelle)
- Étude de la faisabilité

# Cycle de vie d'un logiciel

- Étude de la faisabilité
- Planification
- Spécification des besoins
- Analyse (Spécification formelle)
- Conception (Spécification technique)
- Implémentation (Codage)
- Tests unitaires
- Intégration et tests
- Livraison
- Maintenance

# Planification

- **Objectifs** : identification de plusieurs solutions et évaluation des coûts et bénéfices de chacune d'elles
- **Activités** : simulation de futurs scénarios de développement
- **Sortie** : un schéma directeur contenant
  - la définition du problème
  - les différentes solutions avec les bénéfices attendus
  - les ressources requises pour chacune d'elles (délais, livraison, etc.)

# Spécification des besoins

- Objectifs :

À partir du cahier des charges, description du problème à traiter

- identification des besoins de l'utilisateur
- spécification du "quoi" fait par le logiciel : informations manipulées, services rendus, interfaces.

- Sorties:

- Modèle des besoins
- Manuel utilisateur provisoire pour les non informaticiens
- Plans de tests du système futur (cahier de validation)

# Analyse

- Objectifs :

- Répondre au « Que fait le système ? »
- Modélisation du domaine d'application
- Analyse du métier et des contraintes de réalisation

- Activités :

- Abstraction et séparation des problèmes

- Sorties :

- Modèle conceptuel (diagrammes de classes etc.)

# Conception

- Objectifs :

- Répondre au « Comment faire le système ? »
- Décomposition modulaire

- Activités :

- Définition de l'architecture du logiciel
- Définition de chaque constituant du logiciel : informations traitées, traitements effectués, résultats fournis, contraintes à respecter

- Sorties :

- Modèle logique (diagrammes de composants etc.)



# Implémentation

- Objectifs :

- Réalisation des programmes dans un (des) langage(s) de programmation
- Tests selon les plans définis lors de la conception

- Activités :

- Écriture des programmes • Tests • Mise au point (débugage)

- Sorties :

Modèle physique

- Collection de modules implémentés, non testés
- Documentation de programmation qui explique le code

# Tests Unitaires

- Objectifs :

- Test séparé de chacun des composants du logiciel en vue de leur intégration

- Activités :

- réalisation des tests prévus pour chaque module
- les tests sont à faire par un membre de l'équipe n'ayant pas participé à la fabrication du module

- Sorties :

Rapport de cohérence logique

# Intégration et test du système

## Objectifs :

- Intégration des modules et test de tout le système

## • Activités :

- Assemblage de composants testés séparément
- Tests Alpha : l'application est mise dans des conditions réelles d'utilisation, au sein de l'équipe de développement (simulation de l'utilisateur final)

## • Sorties :

- Rapport de conformité
- Documentation des éléments logiciels

# Livraison, maintenance, évolution

## Objectifs :

- Livraison du produit final à l'utilisateur,
- Suivi, modifications, améliorations après livraison.

## Activités :

- Distribution du produit sur un groupe de clients avant la version officielle (version d' évaluation)
- Livraison à tous les clients
- Maintenance

## Sorties :

- Produit et sa documentation

# CHAPITRE 1



Génie Logiciel

Conduite de projet informatique

Phases de développement

04

**Modèles de développement**

Méthodes d'analyse/conception

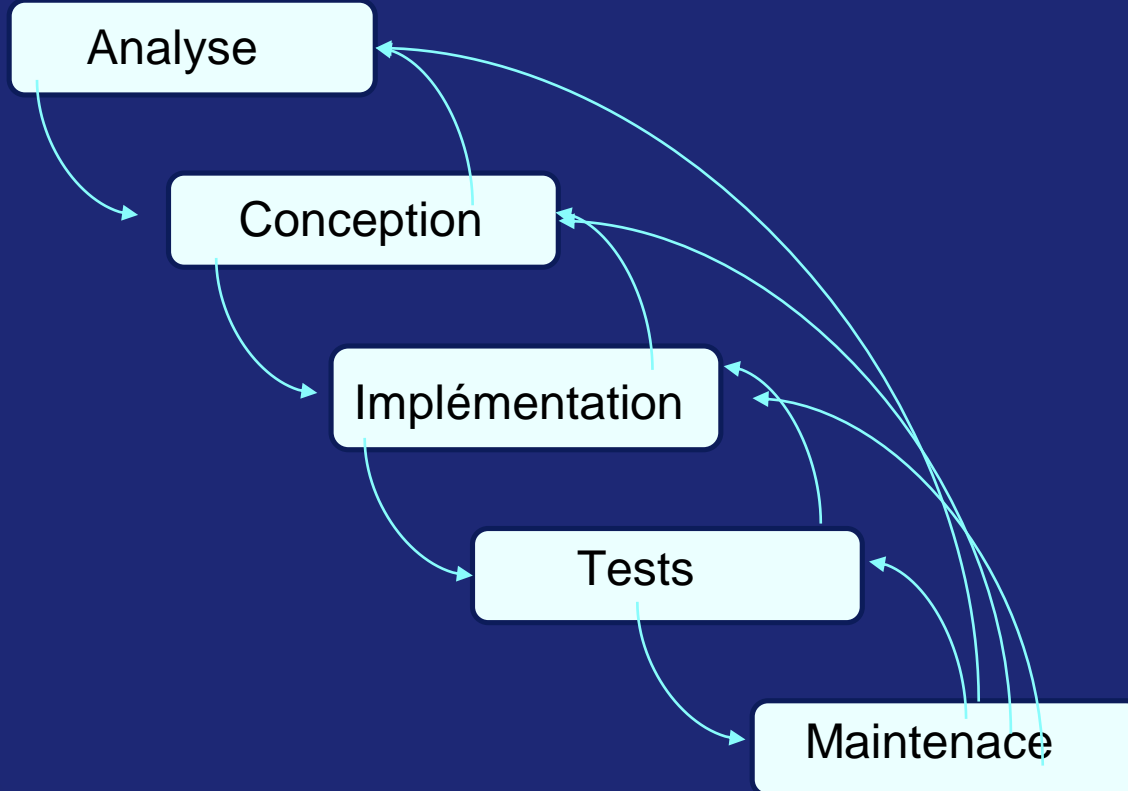
# Modèles de développement

## Objectifs :

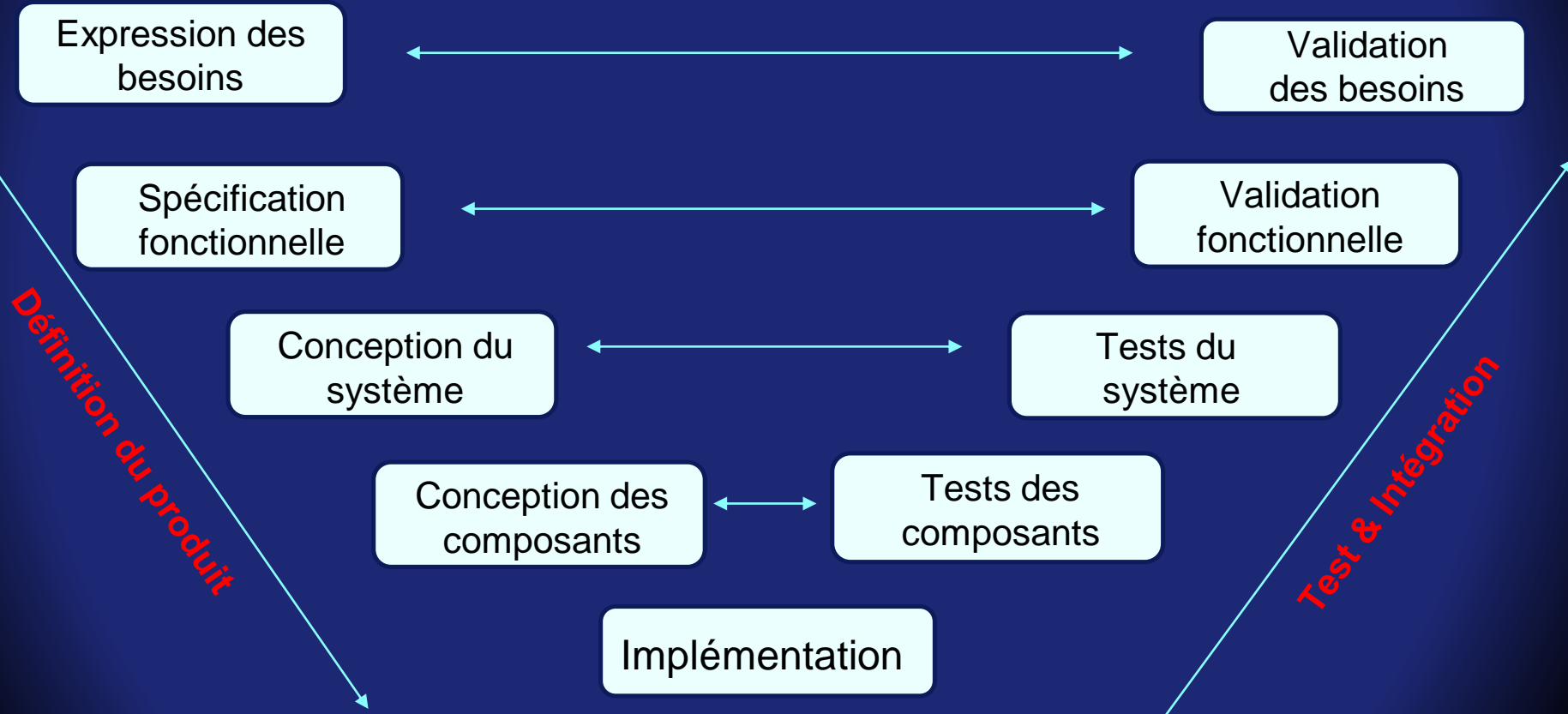
- Organiser les différentes phases du cycle de vie pour l'obtention d'un logiciel fiable, adaptable et efficace
- Guider le développeur dans ses activités techniques
- Fournir des moyens pour gérer le développement et la maintenance (ressources, délais, avancement, etc.)

• Modèle (linéaire) en cascade • Modèle en V • Modèle en spirale • Processus Unifié

# Modèle en cascade

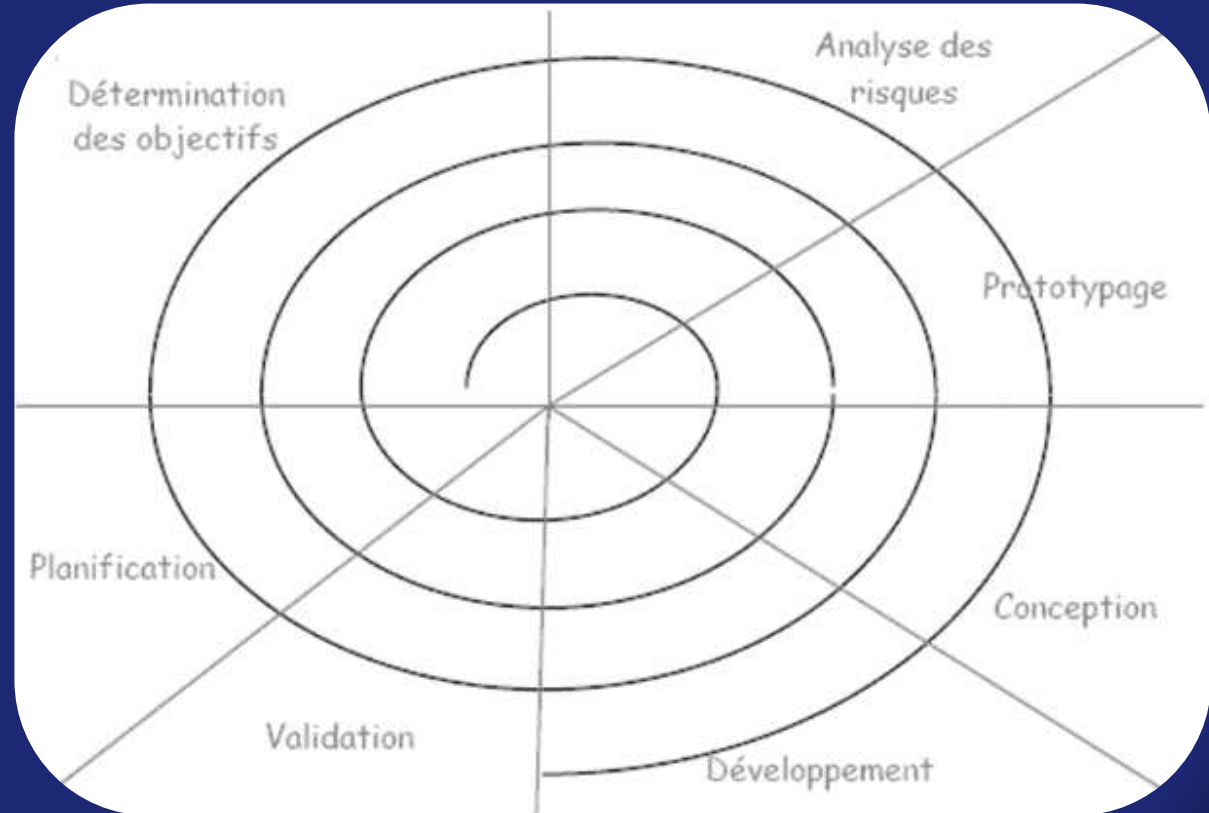


# Modèle en V





# Modèle en spirale



# CHAPITRE 1



Génie Logiciel

Conduite de projet informatique

Phases de développement

Modèles de développement

**05 Méthodes d'analyse/conception**

UML

# Méthodes d'analyse et de conception

- Proposition d'une démarche distinguant les étapes du développement dans le cycle de vie du logiciel
- Utilisation d'un formalisme de représentation qui facilite la communication, l'organisation et la vérification
- Production de documents (modèles) qui facilitent les retours sur conception et l'évolution des applications

# De nombreuses Méthodes

- Méthodes données :
  - Entité-Relation, MERISE, ...
- Méthodes comportements
  - SA-RT, Réseaux de Pétri, ...
- Méthodes objets
  - OMT, OOA, Classe-Relation, OOD,

# CHAPITRE 1



Génie Logiciel

Conduite de projet informatique

Phases de développement

Modèles de développement

Méthodes d'analyse/conception

06

UML

# Unification des méthodes Objet

- Au début des années 90, il existait une cinquantaine de **méthodes objet**, liées uniquement par un consensus autour d'idées communes (objet, classe, sous-systèmes,...)
- Chacune possède sa propre notation, SANS arriver à remplir **tous les besoins** et à modéliser correctement les **divers** domaines d'application.

# Recherche d'un langage commun unique

D'où la recherche d'un langage qui est:

- Utilisable par toute méthode objet, dans **toutes les phases** du cycle de vie
- **Compatible** avec les techniques de réalisation actuelles.

L'unification des notations donne Naissance de **UML**  
(signifie Unified Modeling Language)

# UML

- Unified Modeling Language
- UML est un langage de modélisation objet  
Support des phases d'Analyse et de Conception orientée objet
- UML est un langage de communication  
Utilisation d'un même formalisme par tous les intervenants  
Permet de lever les ambiguïtés du langage naturel
- UML est un langage simple de haut niveau  
Facile à appréhender  
Indépendant de tout langage de programmation



# Types de Diagrammes UML

Dans UML on distingue **trois** types de diagrammes :

## **Diagrammes Structurels/ Statiques** (ce que le système est )

- diagrammes de classes
- diagrammes d'objets
- diagrammes de composants
- diagrammes de déploiement
- diagrammes de paquetages

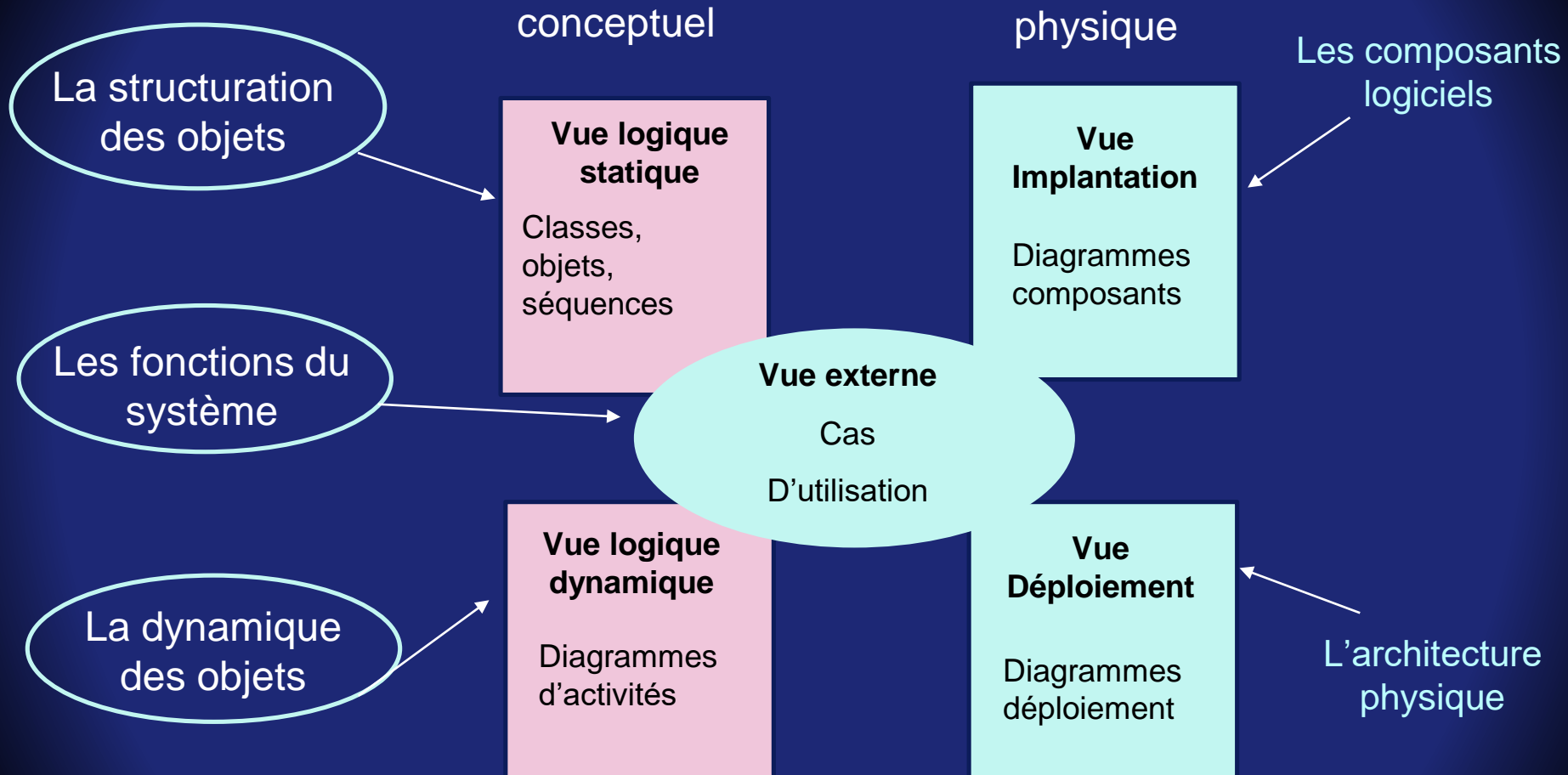
# Types de Diagrammes UML

Diagrammes Comportementaux ou Fonctionnel  
(ce que le système fait)

- diagramme de cas d'utilisation
- diagramme états – transitions
- diagramme d'activité

Diagrammes d'interactions / interactions dynamiques  
(comment le système évolue)

- diagrammes de séquence
- diagramme de communication
- diagramme global d'interaction

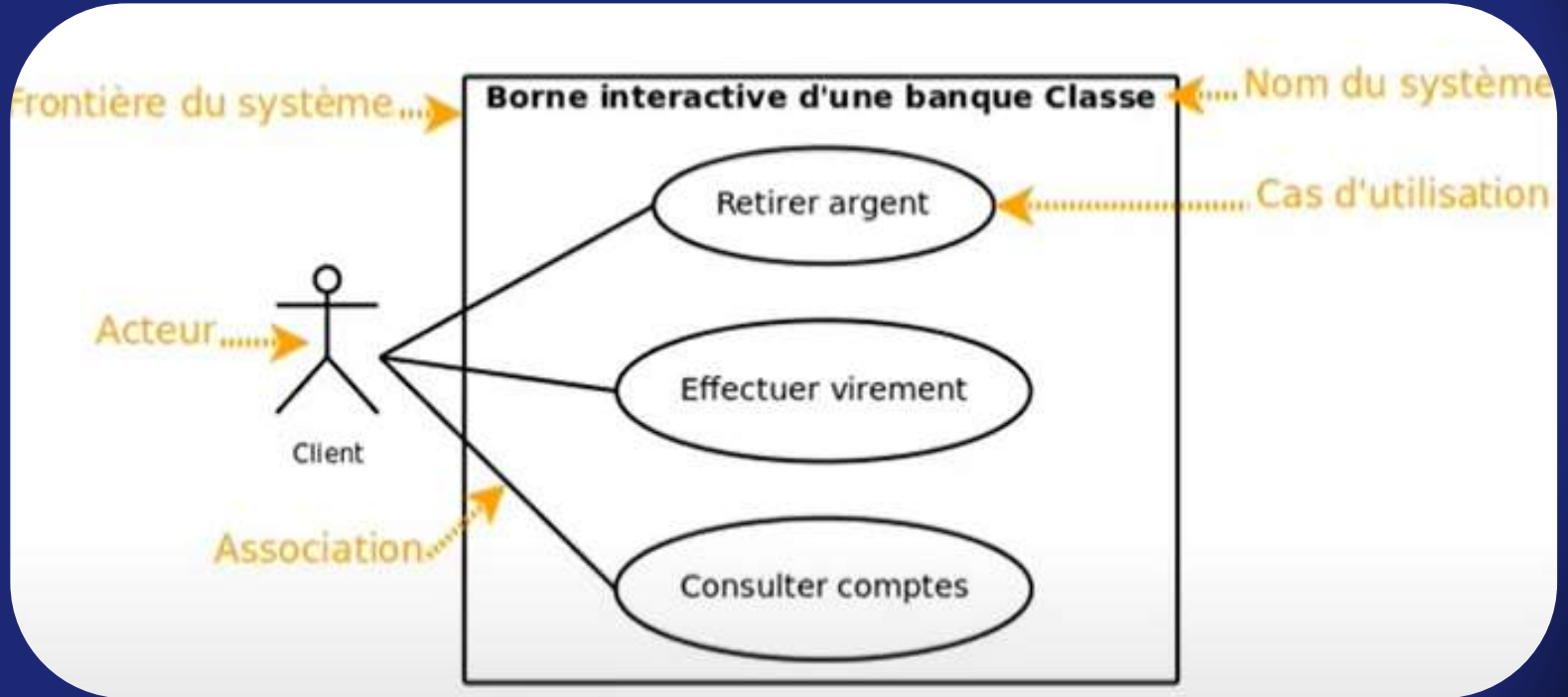


## CHAPITRE 2



# Diagrammes de cas d'utilisation (Use Case Diagram)

# Exemple de diagramme de cas d'utilisation



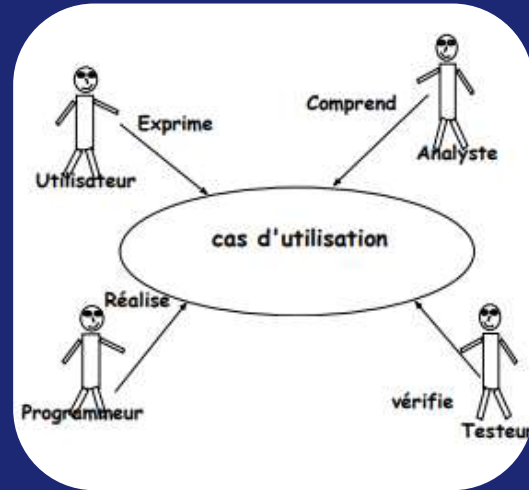
# Diagramme de cas d'utilisation

- Modélise les besoins des utilisateurs.
- Précisent le but à atteindre.
- Permet d'identifier les fonctionnalités principales (critiques) du système.



# Un Cas d'utilisation

- Facilite la structuration des besoins des utilisateurs.
- Exprime les limites et les objectifs du système.
- Une suite d'interactions entre un acteur et le système.
- Correspond à une fonction visible par l'utilisateur.



# Les Acteurs

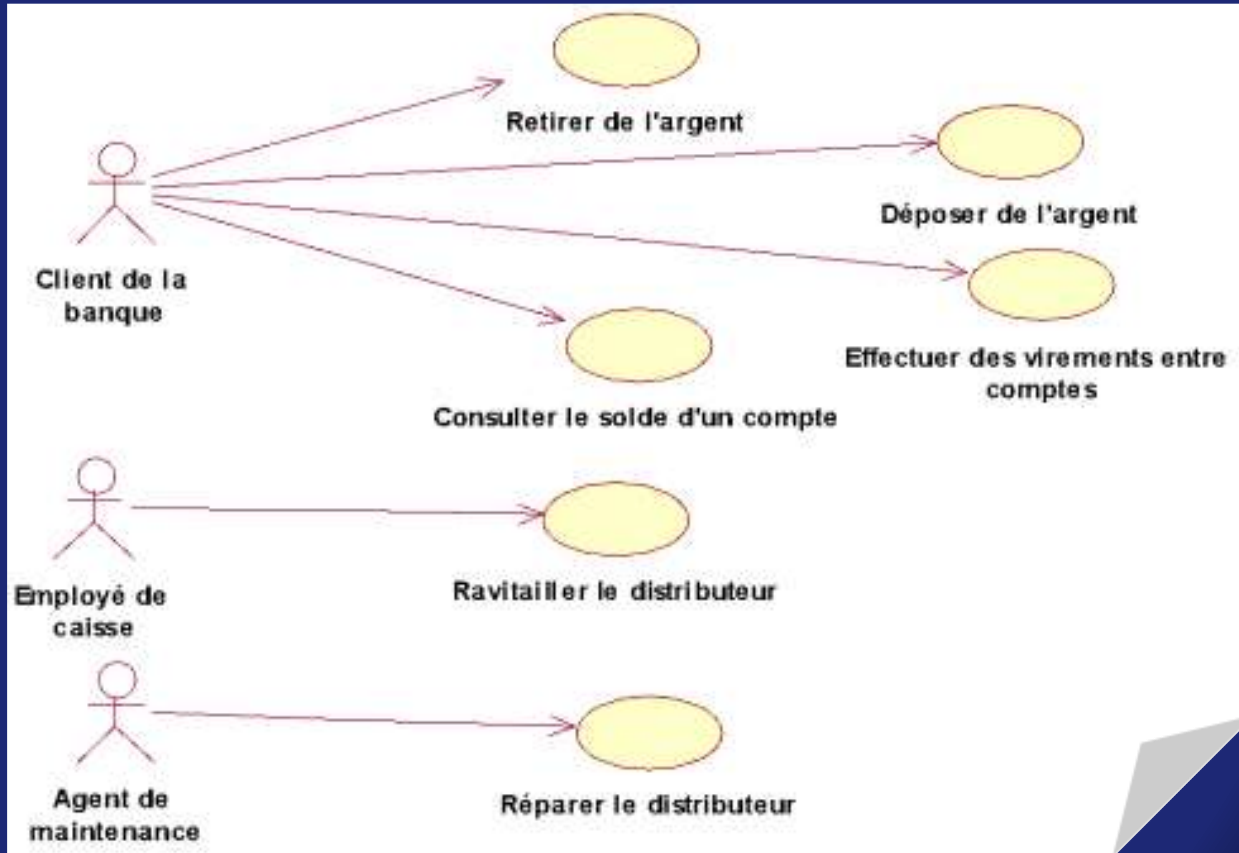
## Acteur :

Une entité externe qui agit sur le système et prend les décisions contrairement à un élément logiciel possède un rôle par rapport au système (utilisateur ou un autre système)

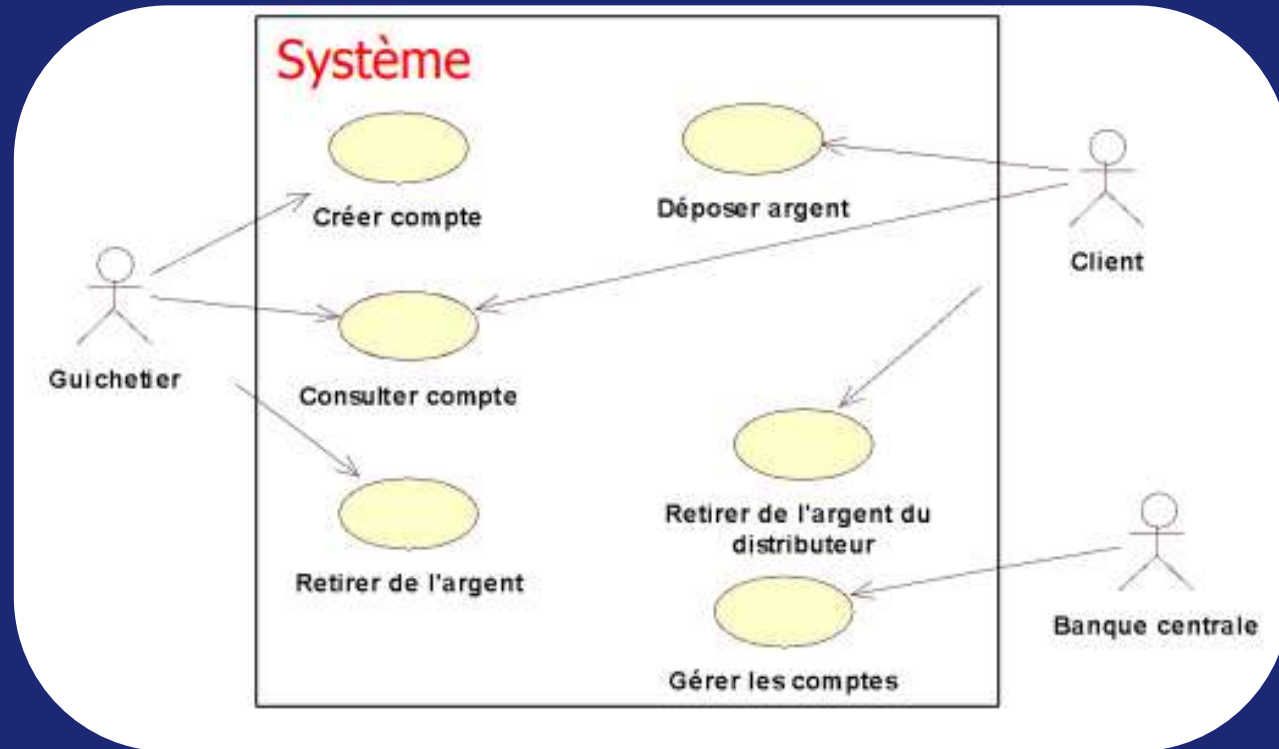




# Diagramme de cas d'utilisation



# Diagramme de cas d'utilisation



# Description d'un Use Case

Description textuelle (informelle) :

Exemple :

Use case : “ Retrait en espèce ” :

1. Le guichetier saisit le n° de compte du client.
2. L'application valide le compte auprès du système central.
3. L'application demande le type d'opération au guichetier.
4. Le guichetier sélectionne un retrait d'espèces de 2000 DH.
5. Le système “ guichetier ” interroge le système central pour s'assurer que le compte est suffisamment approvisionné.
6. Le système central effectue le débit du compte.
7. Le système notifie au guichetier qu'il peut délivrer le montant

# Comment trouver les acteurs

Pour Dégager les acteurs d'un Système , on peut poser les questions suivantes:

- Qui utilise le système.?
- Qui maintient le système?
- Qui administre le système?
- Quels autres systèmes qui interagissent avec le système?
- Qui a besoin d'information venant du système?

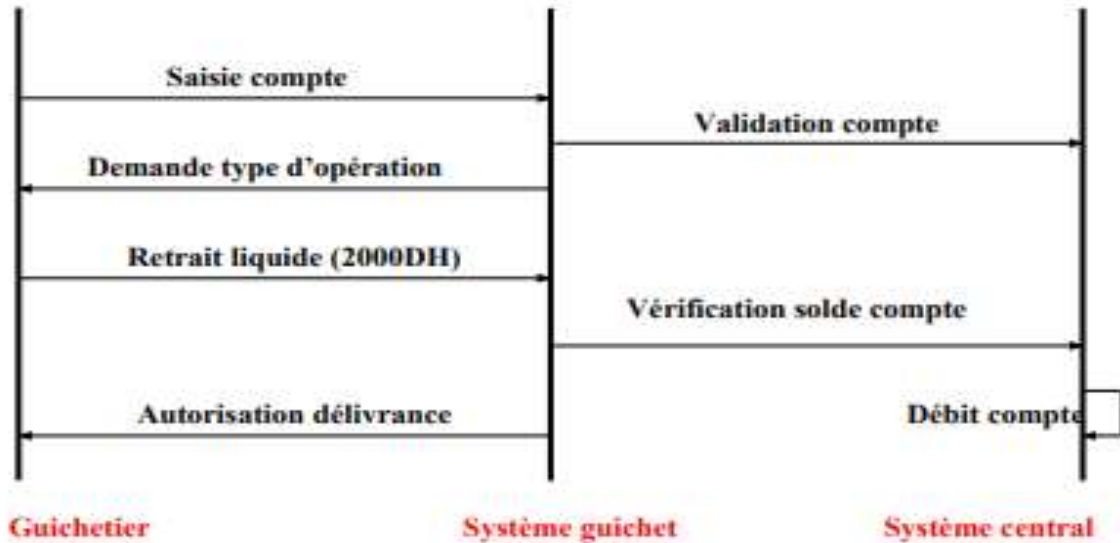
# Activité en binôme

Un opérateur Télécom désire disposer d'une plateforme d'achat en ligne où les clients peuvent consulter ou/et commander les produits qui sont disponibles sur un catalogue.

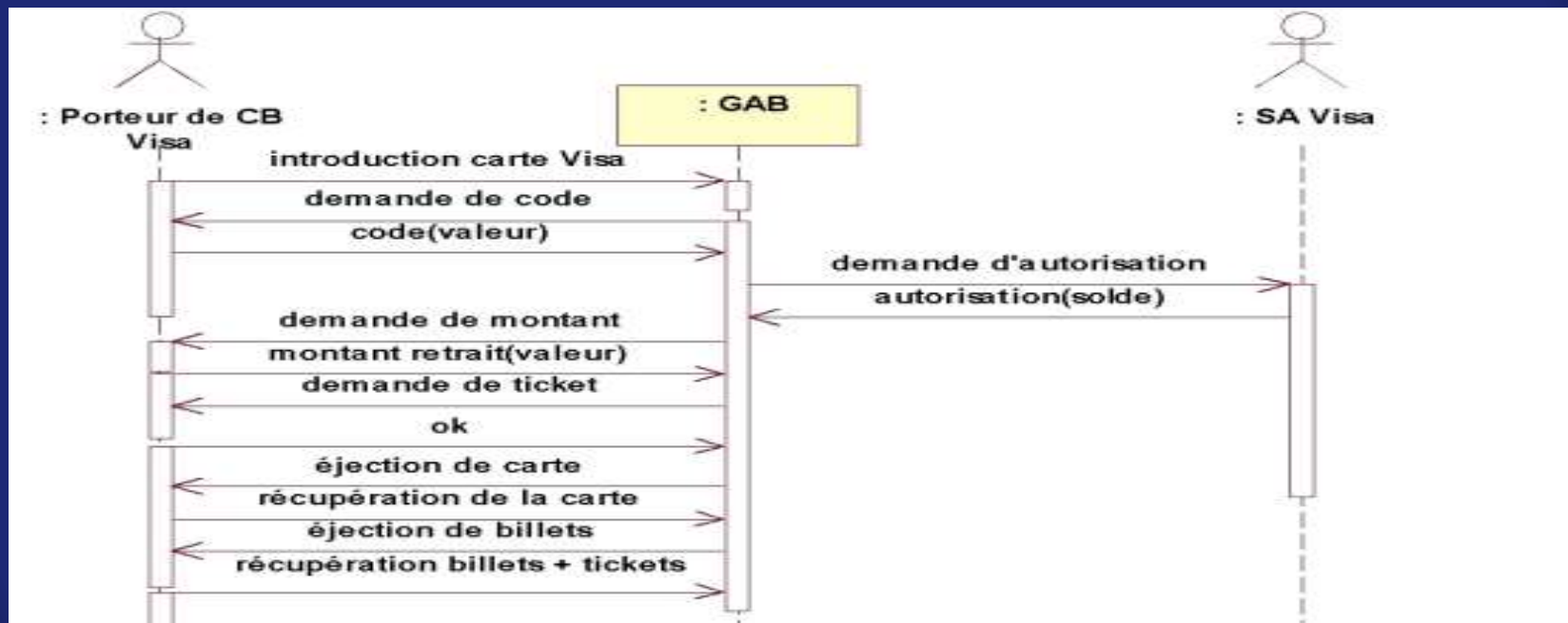
Les administrateurs de la plateforme sont responsables de la mise à jour des produits et du traitement des commandes effectuées par les clients.

Proposez un diagramme de cas d'utilisation

# Descriptions à l'aide de diagrammes de séquences



# Activité



## CHAPITRE 3



# Diagrammes de classes



# Objectifs d'un diagramme de classes

- Déterminer les données qui seront manipulées par le système
- Ces données sont organisées en **classes**
- Donner la structure **statique** de ces données
- Ce diagramme permet de décrire la **structure interne** de chacune des **classes**
- Représenter les relations statiques existantes entre différentes données du système

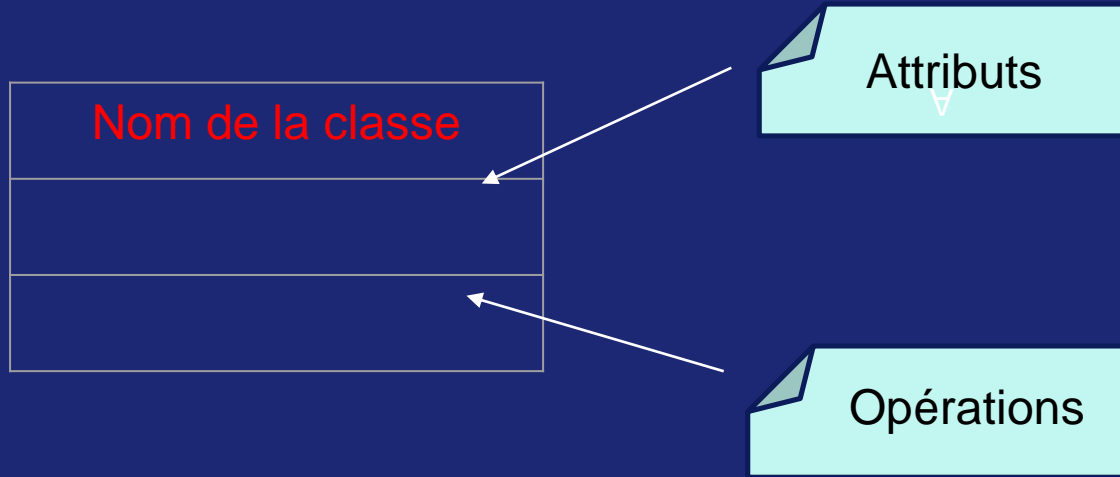
# Classe et objet

Une **classe** est une description abstraite (modèle) d'un ensemble d'**objets** ayant :

- des propriétés similaires
- un comportement commun
- des relations communes avec d'autres objets
- des sémantiques communes

Un **objet** est représentation individuelle(**instance**) d'une **classe**

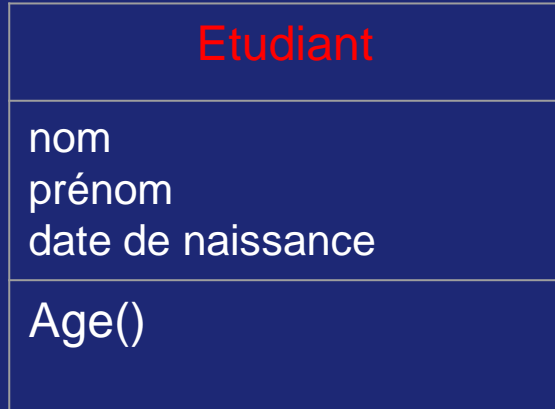
# Représentation d'une classe en UML



**Note:** Les compartiments d'une classe peuvent être omis si leur contenu n'est pas intéressant dans le contexte d'un diagramme

# Représentation d'une classe en UML

## Exemples



# Attributs de classes

Un attribut de classe définit une propriété commune aux objets d'une classe.

Nom de la classe
Nom de l'attribut : Type
Age()

# Attributs de classes

## Exemples

Personne
Nom : chaine Prénom : chaine Date de naissance : Date
Age()

Voiture
Immatriculation : chaine Puissance : chaine Marque : chaine

# Opérations de classe

- Une opération définit une **fonction** appliquée à des objets d'une classe

Personne
Nom d'opération (liste d'arguments)

- Elle représente le service que la classe doit fournir à ces utilisateurs

# Opérations de classes

## Exemples

Etudiant
nom prénom date de naissance
Age() changerAdresse()

ObjetGéométrique
Couleur : chaine
déplacer(dx:vecteur) Sélectionner(p:point):boolean



# Accessibilité aux attributs et opérations d'une classe

## Trois niveaux de protection :

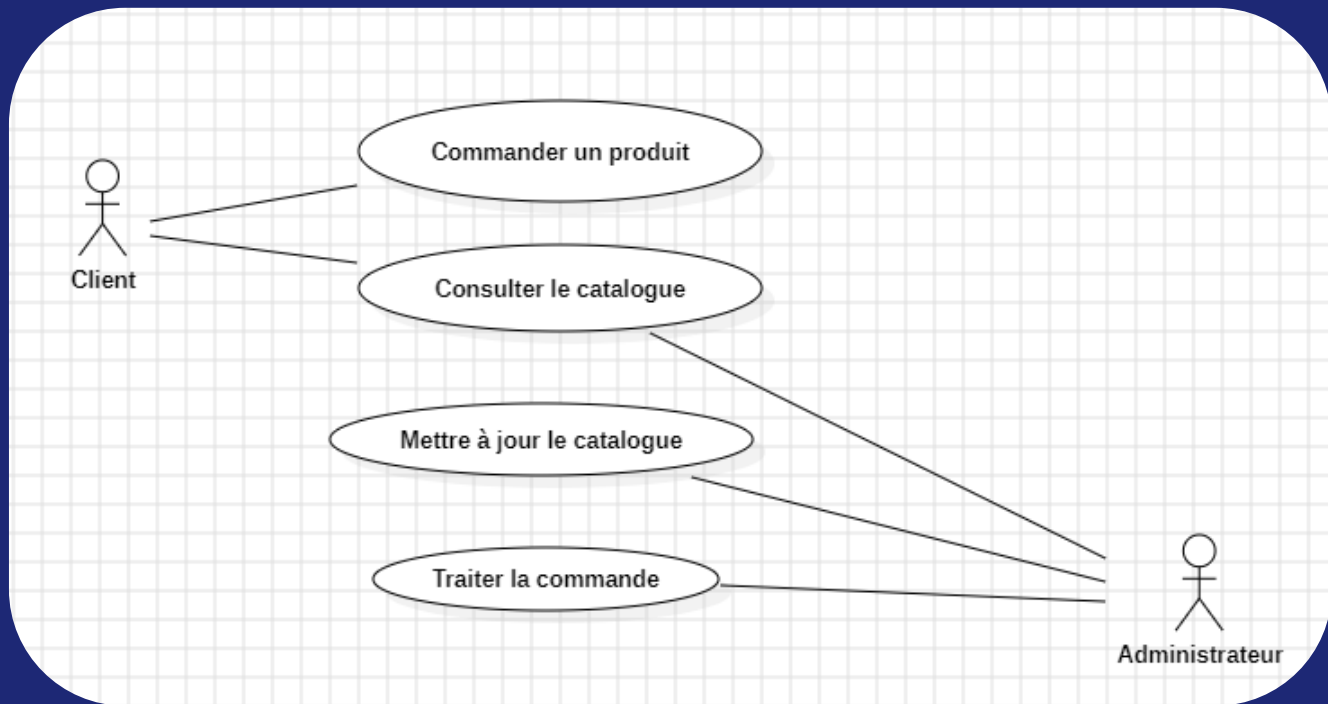
**Public (+)** : accès à partir de toute entité interne ou externe à la classe

**Protégé (#)** : accès à partir de la classe ou des sous-classes

**Privé (-)** : accès à partir des opérations de la classe

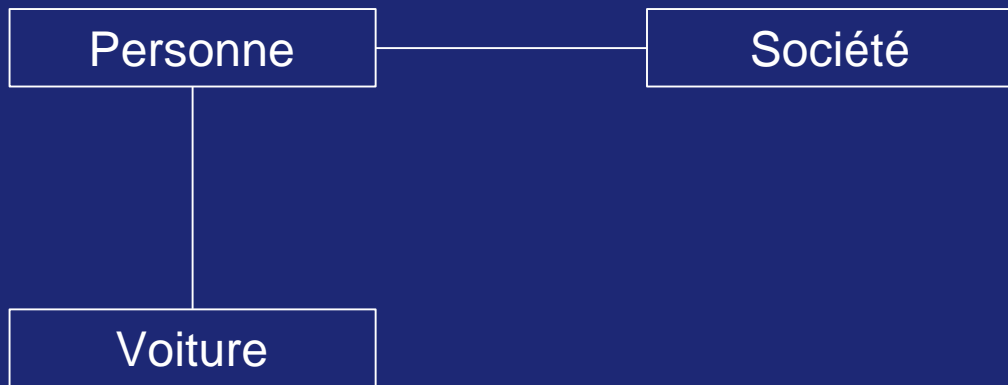
Etudiant
- nom - prénom - date de naissance
+ Age() + CalculMoyenne()

# Activité en binôme



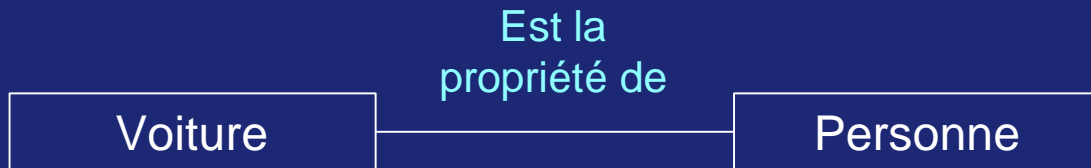
# Associations

Les **associations** représentent les **liens** unissant les instances des classes



# Nommage des associations

On note en général les associations par une forme verbale, soit active, soit passive



# Multiplicité des associations



1 : Une personne travaille pour une et une seule société

1 .. \* : Une société emploie de un à plusieurs personnes

# Multiplicité des associations

La **multiplicité** est une information portée par le rôle, qui indique le nombre **d'objets** successibles de **participer** à une association

1	Un et un seul
0..1	Zéro ou Un
M..N	De M à N (entiers naturels)
*	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	De Un à plusieurs

# Association particulière: agrégation



-une ou plusieurs couleurs peuvent figurer dans une voiture

# Association particulière: agrégation

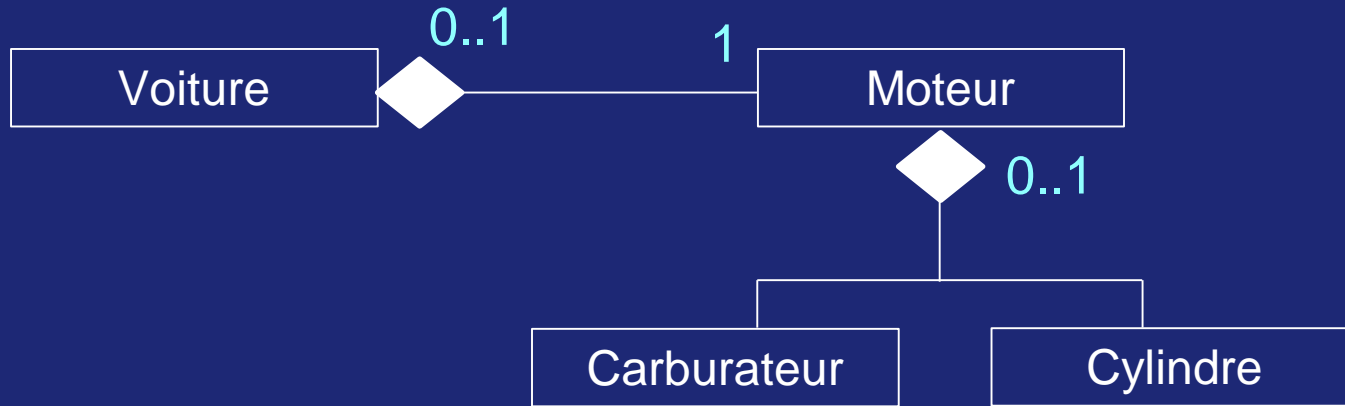
- Une **agrégation** est une **association non symétrique** : l'une des extrémités joue un rôle prédominant par rapport à l'autre.



- Une classe B « **fait partie** » **intégrante** d'une classe A
- **Les valeurs d'attributs** de la classe B se **propagent dans les valeurs d'attributs** de la classe B



# Association particulière: Composition



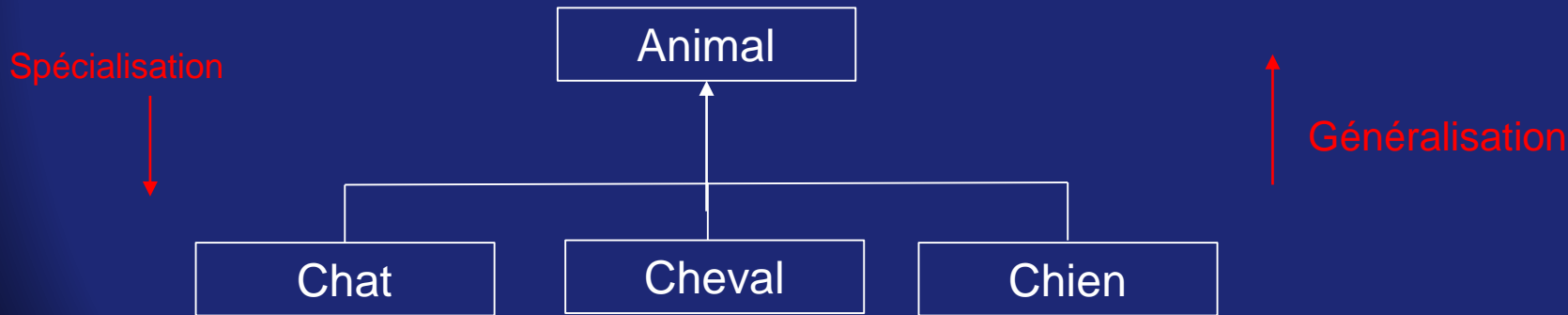
# Association particulière: Composition

La composition est une forme particulière d'agrégation. Le composant est « physiquement » contenu dans l'agrégat. La composition implique une contrainte sur la valeur de la multiplicité du côté de l'agrégat : (0 ou 1)



# Généralisation - Spécialisation

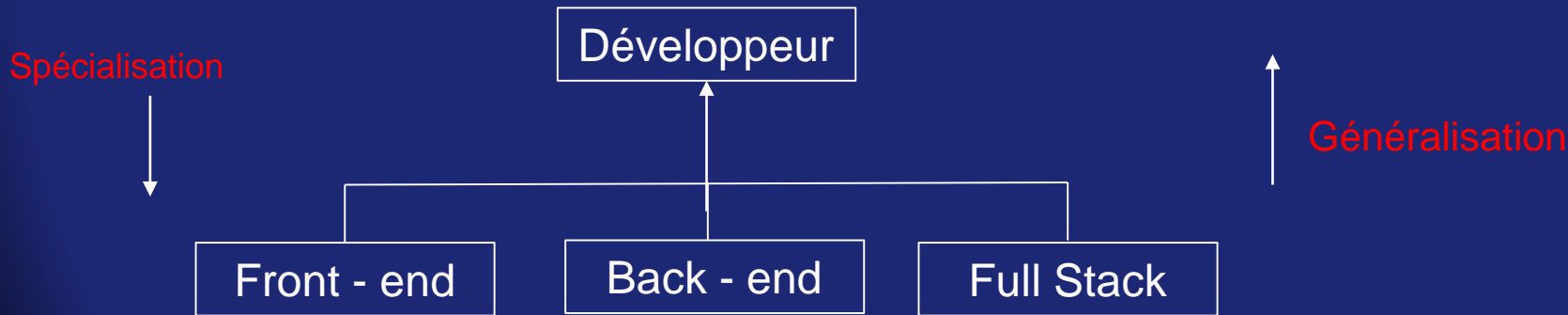
**Exemple** : un animal est un concept plus **général** qu'un chat ou un chien. Inversement un chien est un concept plus **spécialisé** qu'un animal. La classe Animal est une **généralisation** de la classe Chat ou la classe Chien. La classe Chien est une **spécialisation** de la classe Animal.



# Généralisation - Spécialisation

## Définition :

Relation (irréflexive, antisymétrique, transitive) entre une classe plus **générale** et une classe plus **spécifique** (signifie “est un” ou “est une sorte de”). **Ce n’est pas une association.**



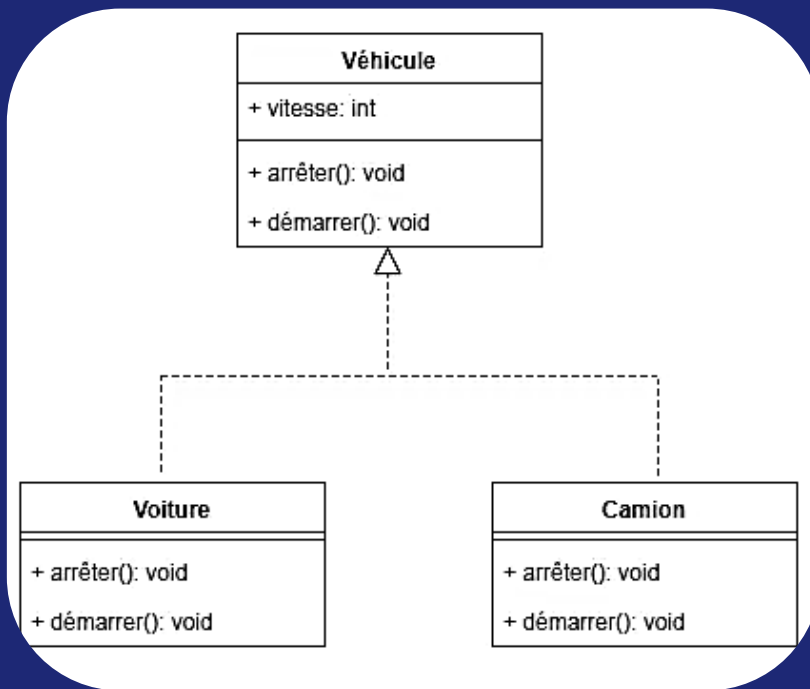
# Activité

Proposez des sous-classes de la classe véhicule

Véhicule
vitesse : int
Démarrer() Arrêter()

# Activité

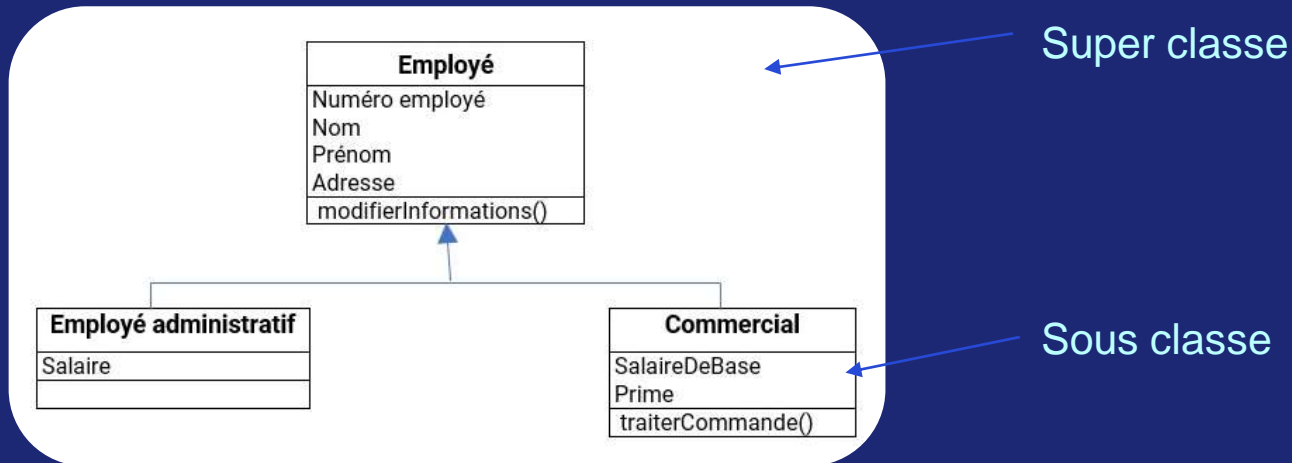
Proposez des sous-classes de la classe véhicule



# Héritage

L'héritage est mécanisme permettant à une classe d'utiliser les membres de sa classe mère sans avoir à les redéfinir.

L'héritage est un mécanisme de la programmation objet.



# Activité

Une personne est caractérisée par son nom, son prénom, son sexe et son âge. Les objets de classe Personne doivent pouvoir calculer leurs revenus et leurs charges. Les attributs de la classe sont privés ; le nom, le prénom ainsi que l'âge de la personne doivent être accessibles par des opérations publiques.

- 1) Donnez une représentation UML de la classe Personne, en remplissant tous les compartiments adéquats.



# La classe Personne

Personne
- nom : String - prénom : String - sexe : String - âge : Integer
+getNom() : String + getPrénom() : String + getAge() : Integer + calculRevenu(): float + calculCharge(): float

# Rappel

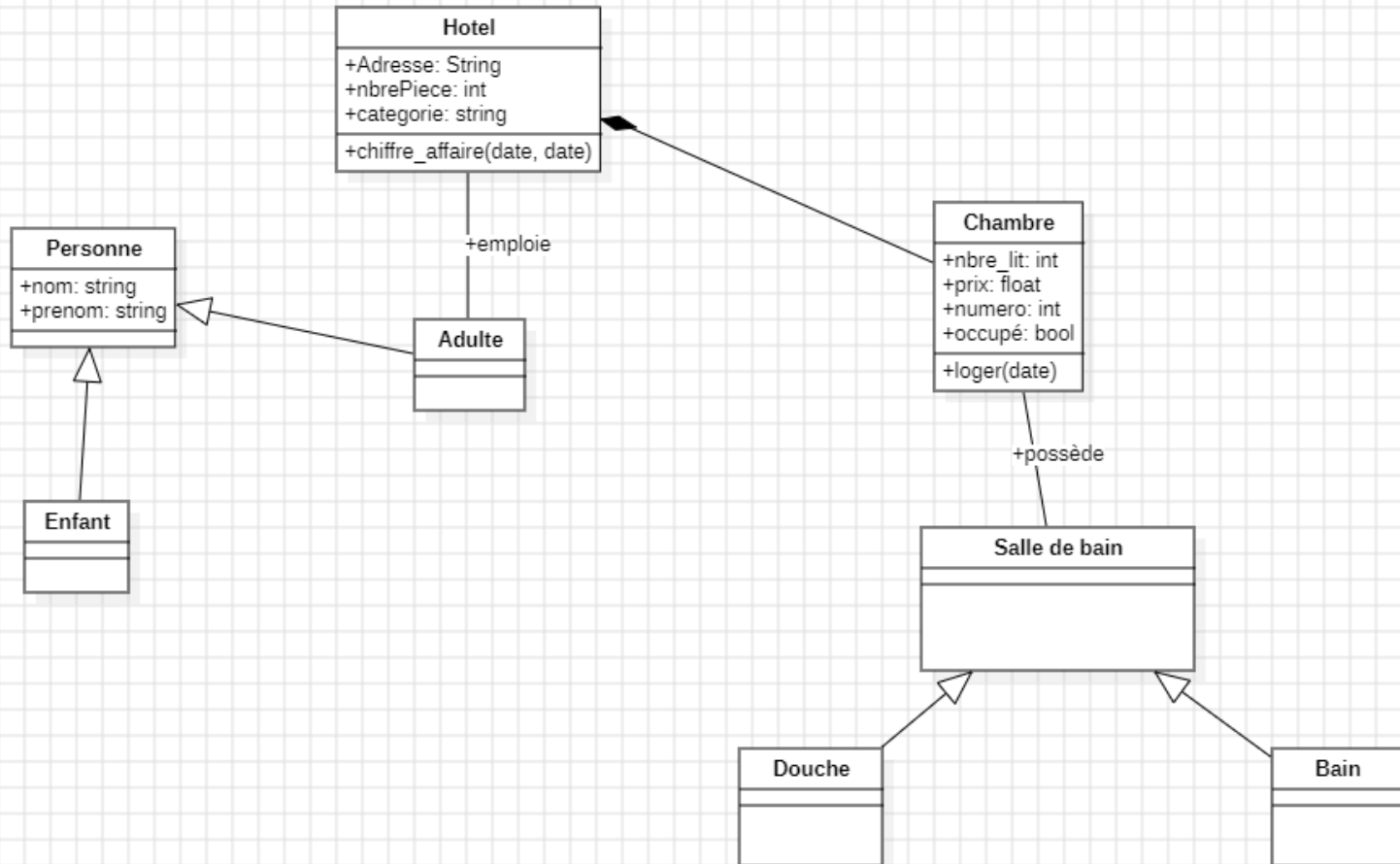
- Un répertoire contient des fichiers
- Une pièce contient des murs
- Les modems et les claviers sont des périphériques d'entrée/sortie
- Une transaction boursière est achat ou une vente

Déterminer la relation statique appropriée (généralisation, composition, agrégation ou association) dans chaque phrase.  
Proposez le diagramme de classe correspondant.

# Activité (diagramme de classes)

Un hôtel est composé d'au moins deux chambres. Chaque chambre dispose d'une salle d'eau : douche ou bien baignoire. Un hôtel héberge des personnes. Il peut employer du personnel et il est impérativement dirigé par un directeur. On ne connaît que le nom et le prénom des employés, des directeurs et des occupants. Certaines personnes sont des enfants et d'autres des adultes (faire travailler des enfants est interdit). Un hôtel a les caractéristiques suivantes : une adresse, un nombre de pièces et une catégorie.

Une chambre est caractérisée par le nombre et de lits qu'elle contient, son prix et son numéro. On veut pouvoir savoir qui occupe quelle chambre à quelle date. Pour chaque jour de l'année, on veut pouvoir calculer le loyer de chaque chambre en fonction de son prix et de son occupation (le loyer est nul si la chambre est inoccupée). La somme de ces loyers permet de calculer le chiffre d'affaires de l'hôtel entre deux dates.



# Diagramme De Cas D'utilisation

MonAuto est une entreprise qui fait le commerce, l'entretien et les réparations de voitures.

MonAuto désire exploiter un logiciel de gestion des réparations; elle dispose déjà d'un logiciel comptable.

Les factures de réparations seront imprimées et gérées par le logiciel comptable.

Le logiciel de gestion des réparations devra communiquer avec le logiciel comptable pour lui transmettre les réparations à facturer.

Le logiciel de gestion des réparations est destiné en priorité au chef d'atelier, il devra lui permettre de saisir les fiches de réparations et le travail effectué par les divers employés de l'atelier.

Pour effectuer leur travail, les mécaniciens et autres employés de l'atelier vont chercher des pièces de rechange au magasin.

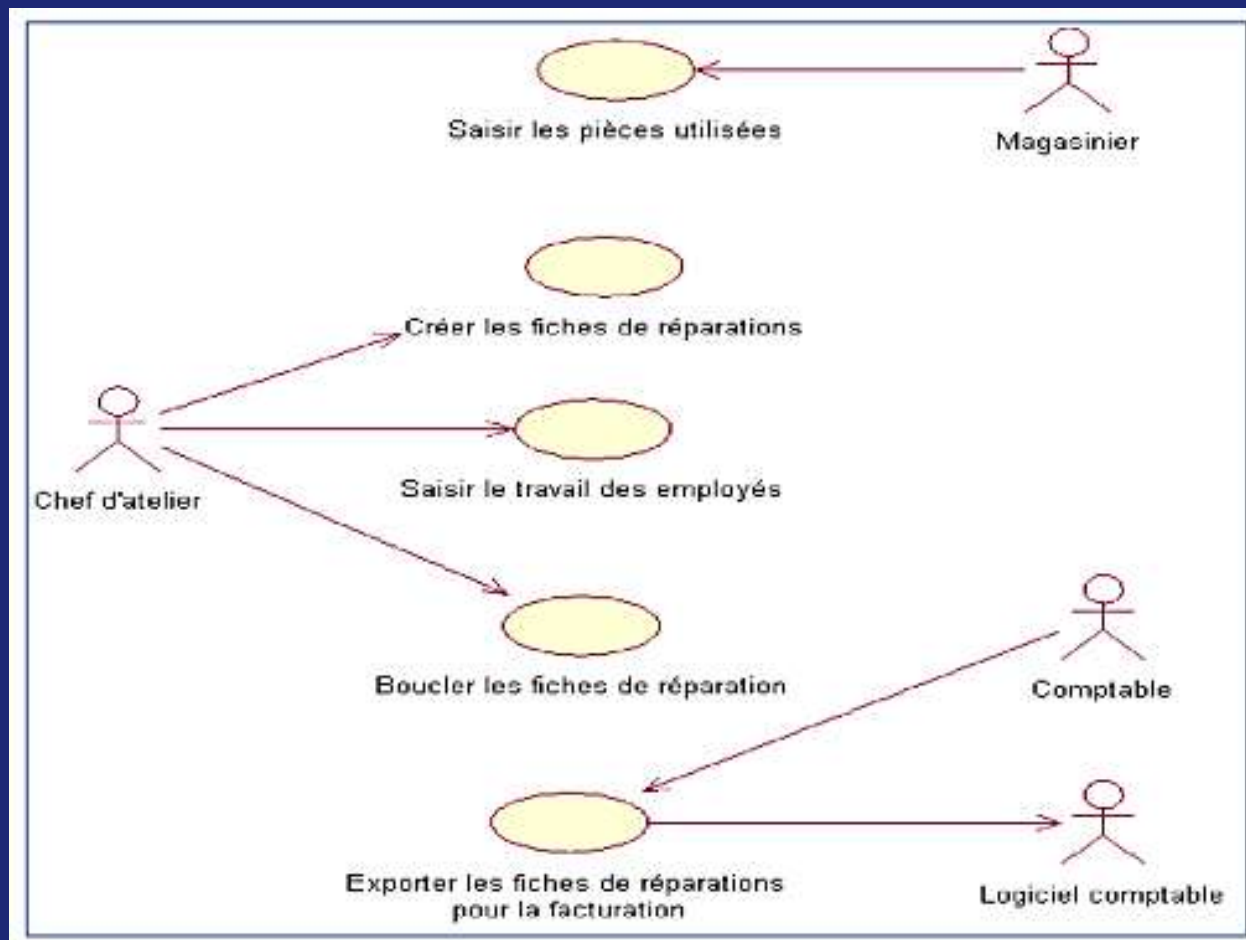
Lorsque le logiciel sera installé, les magasiniers ne fourniront des pièces que pour les véhicules pour lesquels une fiche de réparation est ouverte; ils saisiront directement les pièces fournies depuis un terminal installé au magasin.

Lorsqu'une réparation est terminée, le chef d'atelier va essayer la voiture. Si tout est en ordre, il met la voiture sur le parc clientèle et bouclera la fiche de réparation informatisée.

Les fiches de réparations bouclées par le chef d'atelier devront pouvoir être importées par le comptable dans le logiciel comptable.

Donner la liste des acteurs et indiquer pour chacun s'il est principal ou secondaire.

# Solution





# La Programmation Orienté Objet

**Module : Programmer en Orienté Objet**

**Filière : Développement Digital**

**Formatrice : Meriem Onzy**

**E-mail : [meriem.onzy@ofppt.ma](mailto:meriem.onzy@ofppt.ma)**



# PLAN DU COURS

**01**

**Introduction &  
Historique**

**02**

**Les notions classe, objet  
et méthodes**

**03**

**Modélisation  
d'une classe**

**04**

**Activités**



# Historique

## Les années 70

Simula  
Simula I  
Smalltalk

## Les années 90

Java

## Les années 80

C++  
Objective C

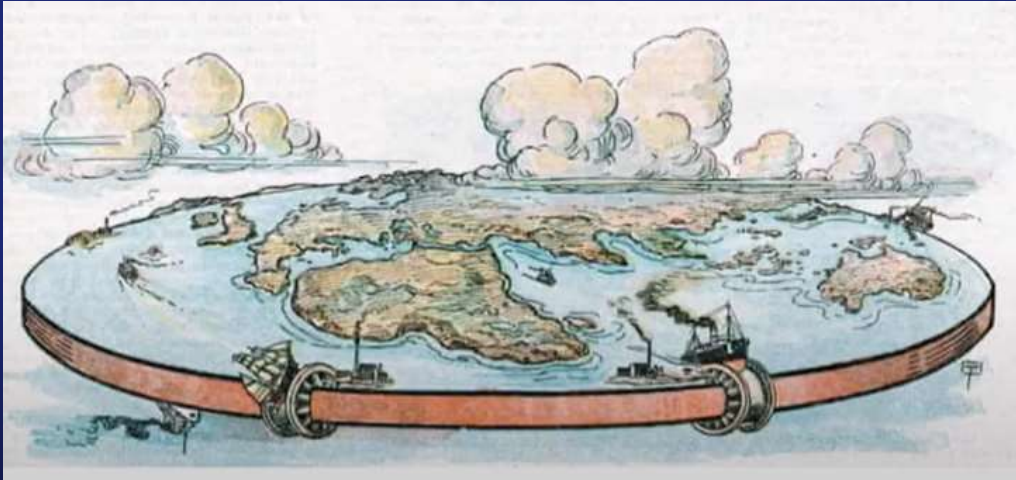
## De nos jours

PHP v5  
C#  
Ruby  
Python

# Le Paradigme de l'orienté objet

Paradigme = Représentation du monde

Modèle de la terre plate



# Le Paradigme de l'orienté objet

Un **paradigme** de programmation est une **façon de concevoir**  
**votre code**

Un moyen de **formuler un problème**, imaginer sa solution et  
l'implémenter

# Le Paradigme de l'orienté objet

## Les paradigmes courants

Procédural

Orienté Objet

Fonctionnel

# Le Paradigme de l'orienté objet

## Les paradigmes courants

Procédural

Orienté Objet

Fonctionnel

# Le Paradigme de l'orienté objet

## Les paradigmes courants

1

Procédural

2

Orienté Objet

Fonctionnel

# Le paradigme Procédural

Programmation Impérative Structurée

1

Procédural

if

else

while

For

# Le paradigme Procédural

Programmation Impérative Structurée

Fait appel à des procédures réutilisables

1

Procédural



# Le paradigme Procédural

1

Procédural

$A = (0, 0)$

$B = (1, 1)$

**distance**(A,B)

Données

Procédure

# Le paradigme Procédural



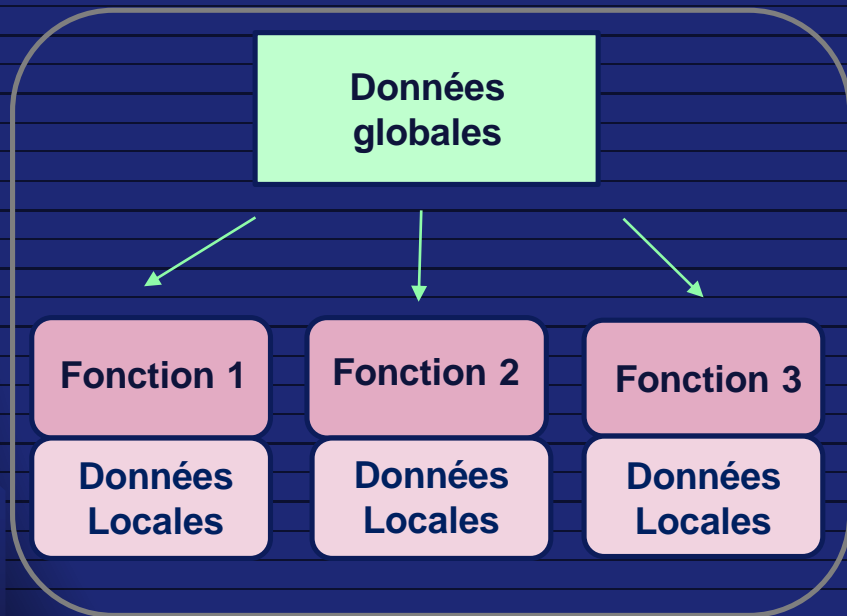
$A = (0, 0)$

$B = (1, 1)$

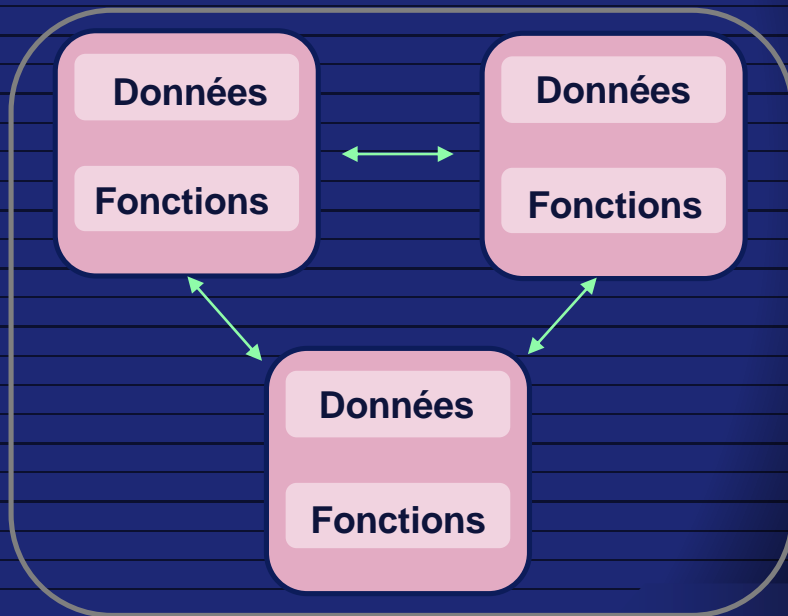
`distance`(A,B)

# Programmation Procédurale vs P00

Programmation Procédurale



Programmation Orientée Objet



# Le Procédural comparé à L'Orienté Objet

A = (0, 0)

B = (1, 1)

**Distance**(A,B)

Priorité à l'algorithme

A = **Point**(0, 0)

B = **Point**(1, 1)

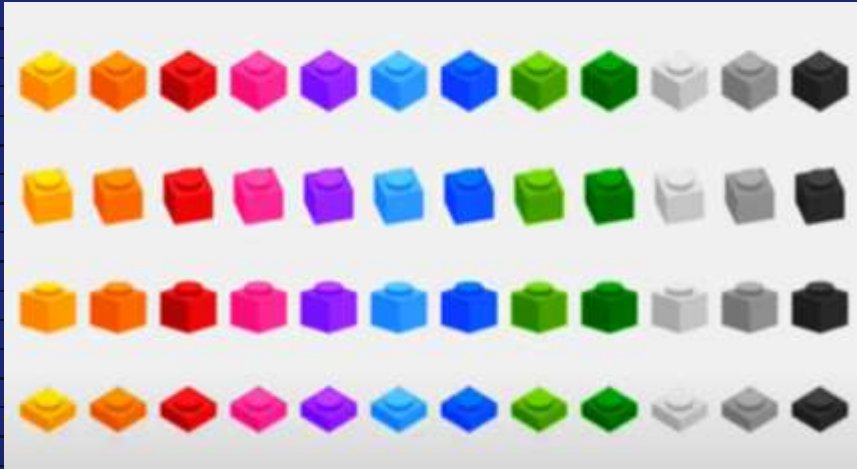
**A.Distance**(B)

Priorité à l'objet

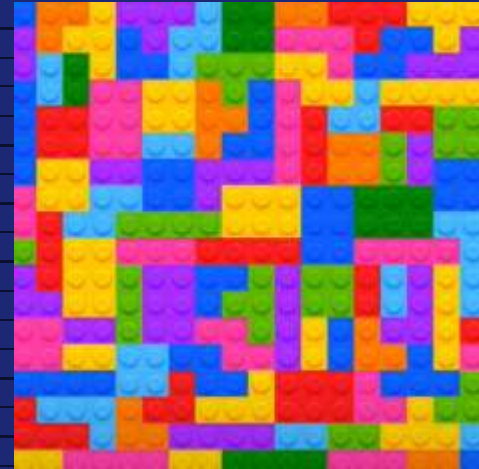
# Notion d'objet



Des Objets



Application



# Notion d'objet

Objet = Structure de données

Type de données

int

float

str



python

# Notion d'objet


```
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
```



python

# Notion d'objet

Objet = Structure de données

A light gray rounded rectangle with the word 'Objet' inside, underlined.

Objet

Se trouve dans un état

[propriétés]

Répond à des messages

[comportements]



# Notion d'objet

## Notes du 1<sup>er</sup> Contrôle

Développer des Sites Statiques	16,0
Manipuler les Base de données	17,5
Programmer en Javascript	6,0



Nada

# Notion d'objet

## Notes du 2<sup>e</sup> Contrôle

Développer des Sites Statiques	16,0
Manipuler les Base de données	17,5
Programmer en Javascript	13,0



Nada

# Notion d'objet

Etat  
1<sup>er</sup> Contrôle

16,0
17,5
6,0

étudier

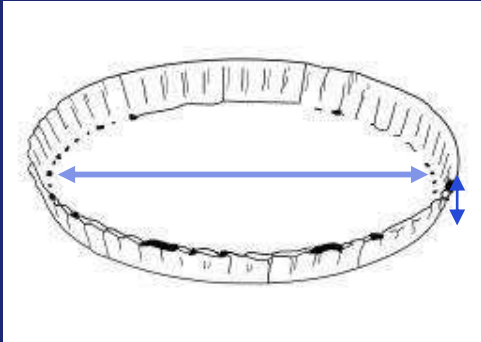
Etat  
2<sup>e</sup> Contrôle

17,0
17,5
13,0



# Notion d'une classe

Une **classe** est une **description** d'un ensemble d'objets qui partagent les **mêmes attributs et méthodes**



# Concept



# UNE CLASSE



**CLASSE**

**OBJET**

**OBJET**

**OBJET**

**OBJET**

**OBJET**

**OBJET**

# Exemple

## Objets étudiants



Sara

16.0



Nada

13,0



Ali

17,0

Identité  
Propre

## Classe Etudiant



**Etudiant**

prénom  
note

passer\_un\_examen()

Abstraction  
Concept d'étudiant

# En POO un objet est une **instance** de sa classe

Classe Etudiant



Etudiant
prénom note
passer_un_examen()

instanciation

instanciation

instanciation

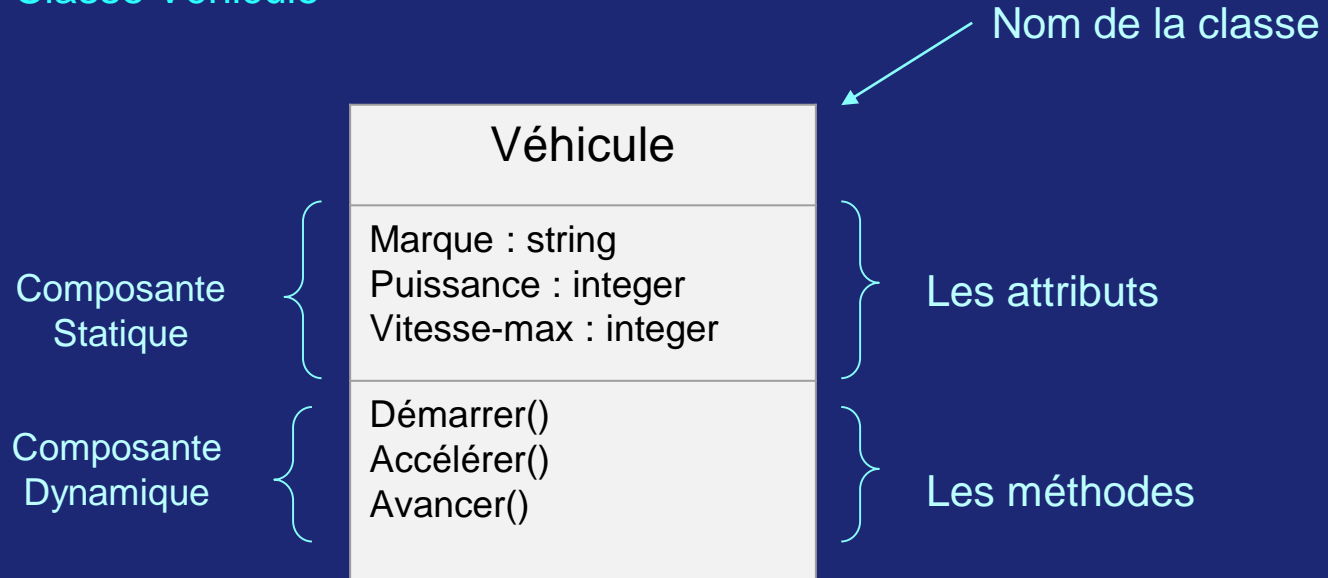
Objets (instances)





# Modélisation d'une classe

Classe Véhicule



# Instances de la classe véhicule



**Marque :** Kia  
**Modèle :** Ceed  
**Couleur :** bleu  
**Année :** 2019



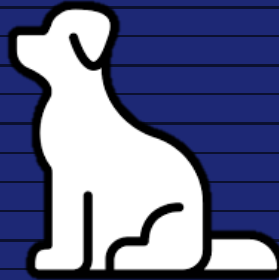
**Marque :** Fiat  
**Modèle :** 500  
**Couleur :** bleu ciel  
**Année :** 2020

# EXEMPLE



Dans un jeu ou on va s'occuper des chiens

## CLASSE



# UNE CLASSE



Classe

Propriétés

Méthodes

## CHIEN

nom

taille

race

poids

est\_allonge

dormir()

# UNE CLASSE

## CLASSE

### CHIEN

nom  
taille  
race  
poids  
est\_allonge

dormir()



## OBJET

### mon\_chien

SAM  
110 cm  
Bouvier Bernois  
45 Kg  
**FAUX**

dormir()

# UNE CLASSE

`mon_chien.dormir()`

CLASSE

CHIEN
nom
taille
race
poids
est_allonge
dormir()



OBJET

mon_chien
SAM
110 cm
Bouvier Bernois
45 Kg
VRAI
dormir()

# Travail de réflexion

## La classe **camion**



### Camion

capacité  
vitesse maximale  
cout d'amortissement

procéder au paiement

### Camion

temps de fabrication  
cout de fabrication

fabriquer  
stocker

### Camion

délai de disponibilité  
Mode de livraison

vendre  
calculer une remise  
de prix

# Travail de réflexion

## La classe **camion**



La classe « Camion »  
Abstraction du **client**

Camion
capacité vitesse maximale cout d'amortissement
procéder au paiement

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

Camion
temps de fabrication cout de fabrication
fabriquer stocker

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

Camion
délai de disponibilité Mode de livraison
vendre calculer une remise de prix



## Remarques sur l'exemple

- La même classe peut avoir
- Le concept du Camion dans le même programme peut être différent
- La conception d'une classe n'est pas unique

# Synthèse

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

# QUIZ

a



b



Est-ce que ces objets sont des instances :

☐

des classes différentes

☐

de la même classe

Proposez la/les classes

# Les avantages de la POO



**Modularité** : les objets forment des modules compacts regroupant des données et un ensemble d'opérations ce qui réduit la complexité de l'application (classe = module)



- **Réutilisabilité** : La POO favoriser la réutilisation de composants logiciels et même d'architectures complexes

# Les Avantages de la POO



## Abstraction

- Les détails qui ne sont pas utiles à l'utilisateur lui sont masqués
- Regrouper un certain nombre de classes selon des caractéristiques communes

# Les principes de l'orienté objet



Encapsulation

**L'objectif de l'encapsulation :** Ne laisser accessible que le strict nécessaire pour que la classe soit utilisable

Restreindre l'accès à certains éléments d'une classe (le plus souvent ses attributs)

# Les principes de l'orienté objet



Héritage



# Les principes de l'orienté objet



## Héritage

Conception et organisation des armes





# Les principes de l'orienté objet



Héritage

**ARME**

\_poids

\_taille

\_valeur

attaquer()



**Hache**



**Couteau**



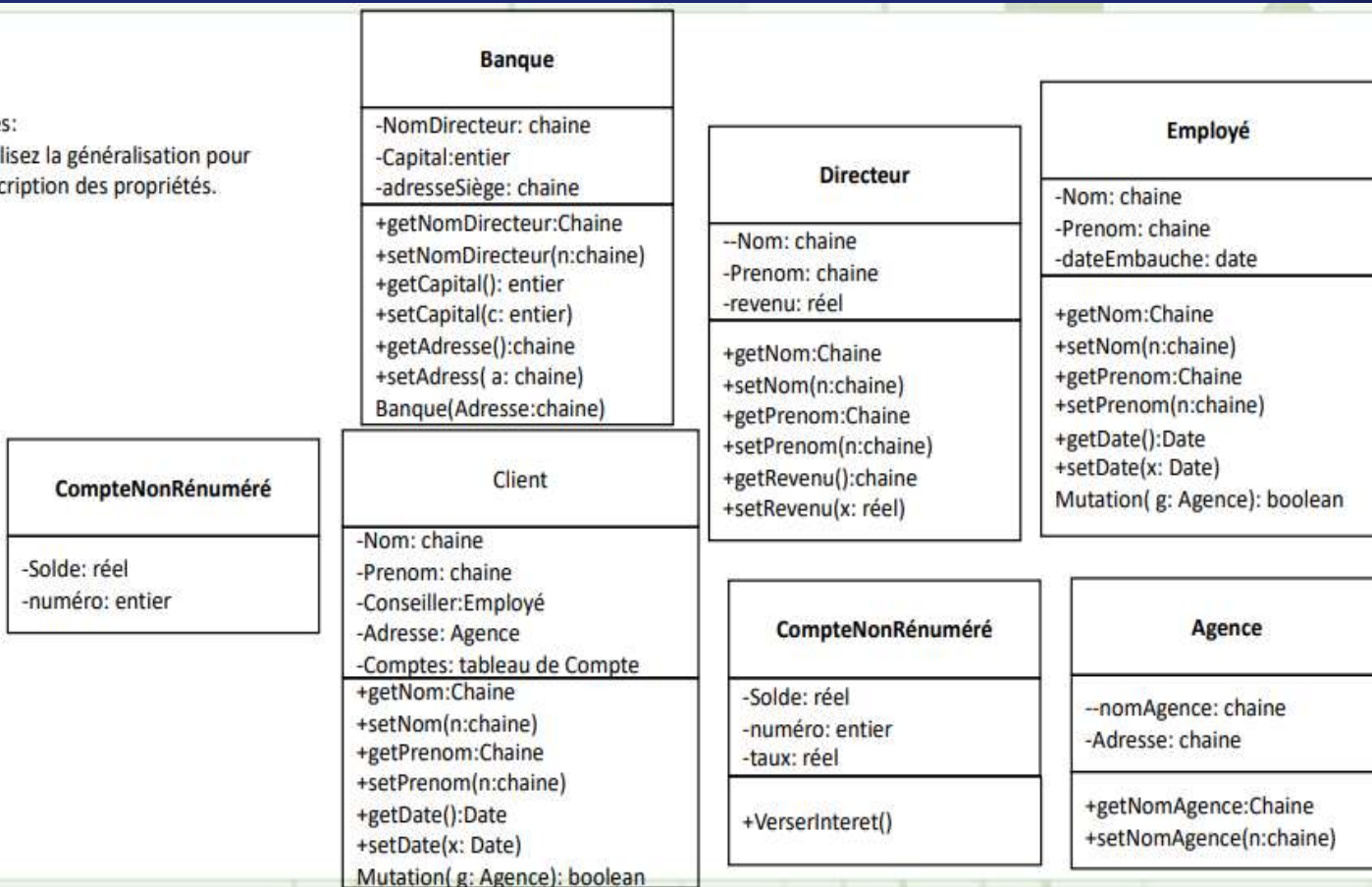
**Corde**

\_type\_corde

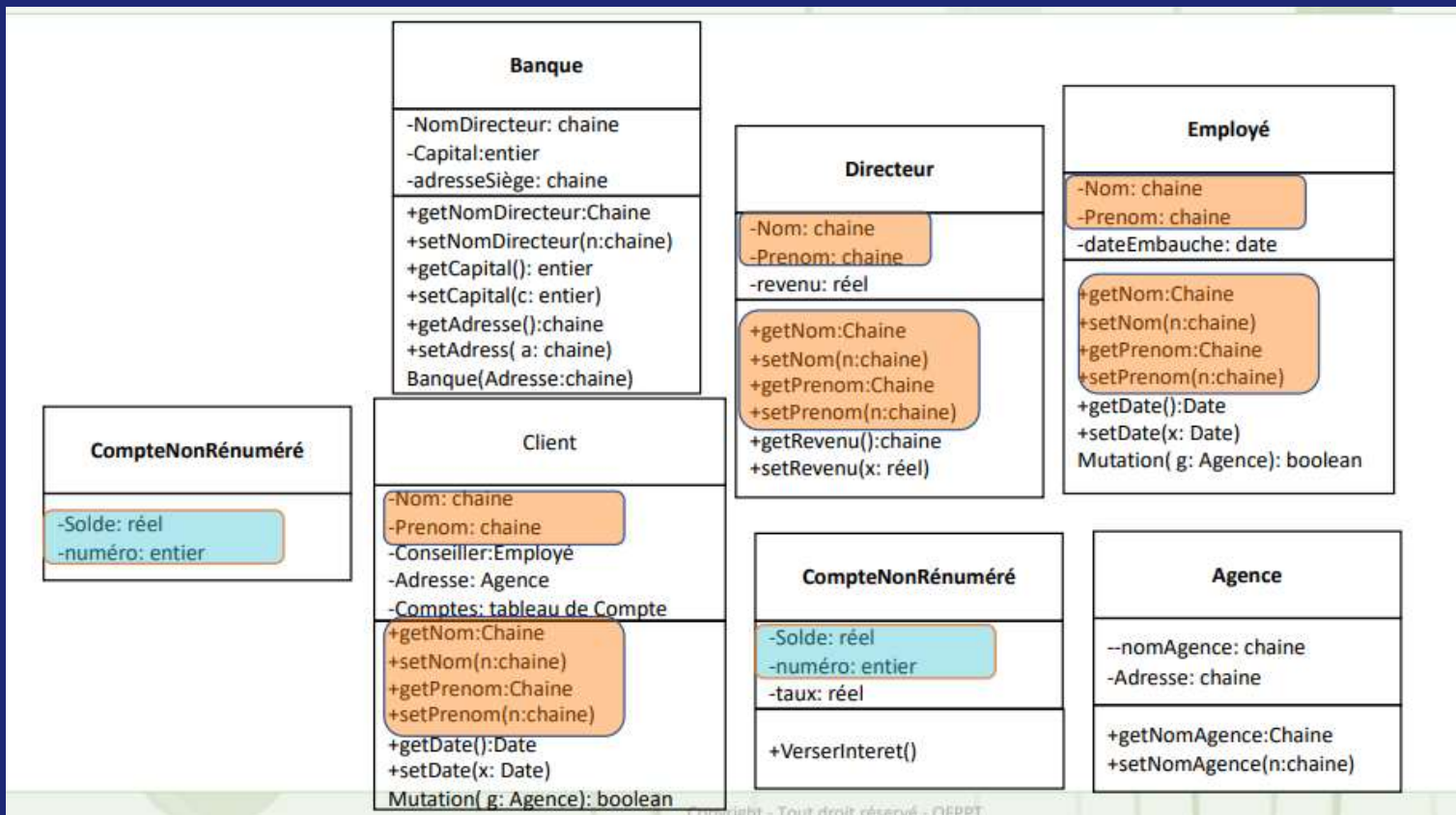


# TP

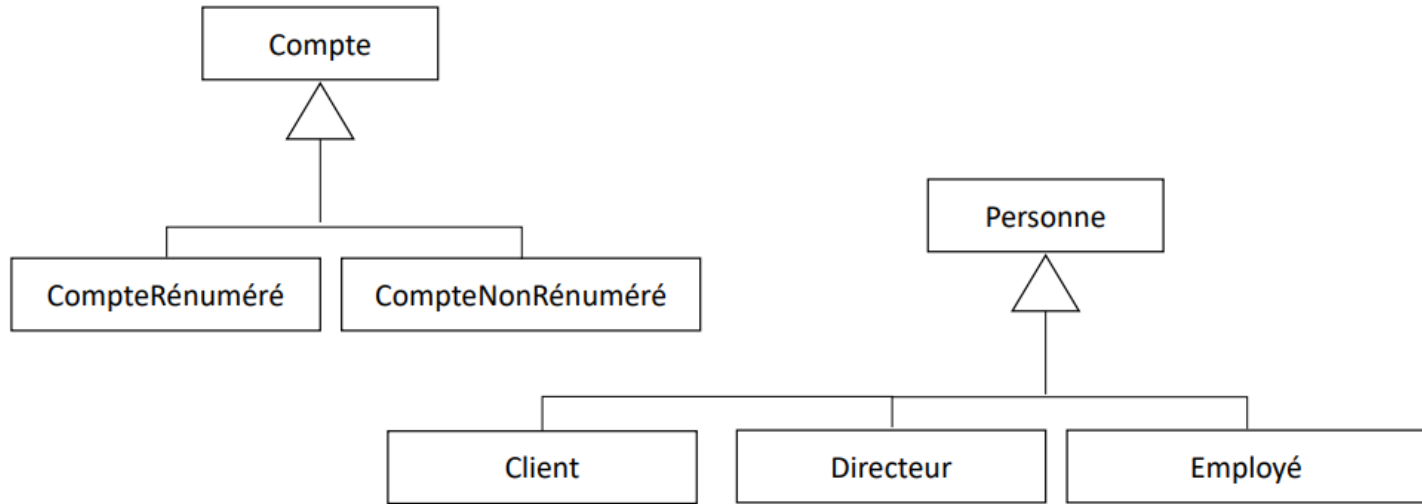
- Soient les classes suivantes:
- Analysez ces classes et utilisez la généralisation pour factoriser au mieux la description des propriétés.



# TP



# TP Corrigé



# Constructeur par Défaut

```
#Sous Python : Pour Chaque classe 1 seul Constructeur

class Stagiaire : # Classe Stagiaire

    def __init__(self): # Constructeur Par défaut
        #Les attributs
        self.nom = " "
        self.prenom = " "
        self.age = " "
```

# Constructeur Paramétrés

```
class Personne :
    def __init__(self,n ,p ,a): # Constructeur Paramétrés
        self.nom = n
        self.prenom = p
        self.age = a

    def Affiche(self):
        print("Nom :",self.nom,"Prenom:",self.prenom,"Age",self.age)

class Etudiant :
    def __init__(self,n = " ",p = " ",a = 0): # Constructeur Paramétrés avec initialisation
        self.nom = n
        self.prenom = p
        self.age = a

    def Affiche(self):
        print("Nom :",self.nom,"Prenom:",self.prenom,"Age",self.age)

e1 = Etudiant()
e1.Affiche()
```

# Implémenter une solution orientée objet

---



## Exemple 1

**Ecrire en Python une classe «Rectangle» ayant deux variables « a » et « b » et une fonction membre « surface() » qui retournera la surface du rectangle.**



## Exercice 2

Écrire en Python une classe « **Somme** » ayant deux variables « **n1** » et « **n2** » et une fonction membre « **som()** » qui calcule la somme.

**Vous devez :**

- demandez à l'utilisateur d'entrer deux entiers
- passez-les au **constructeur par défaut** de la classe « **Somme** »
- afficher le résultat de l'addition des deux nombres.

# Exemple 1

Ecrire en Python une classe «Rectangle» ayant deux variables « a » et « b » et une fonction membre « surface() » qui retournera la surface du rectangle.

```
class Rectangle:
    def __init__(self,a=0,b=0):
        self.a=a
        self.b=b
    def surface(self):
        return self.a*self.b

r1 = Rectangle ()
r2 = Rectangle (5,4);
print("la surface est:",r1.surface())
print("la surface est:",r2.surface());
```

# Corrigé

```
class Somme:
    def __init__(self,nbr1=0,nbr2=0):
        self.n1=nbr1
        self.n2=nbr2

    def som(self):
        return n1 + n2

n1 = int(input("Entrer N1:"))
n2 = int(input("Entrer N1:"))
obj = Somme(n1,n2);
print("Le resultat de l'addition est :",obj.som())
```

# Corrigé

```
class Etudiant:
    def __init__(self,nom,note1,note2):
        self.nom = nom
        self.note1 = note1
        self.note2 = note2

    def calc_moy(self):
        return (self.note1 + self.note2)/2

    def afficher (self):
        print("Etudiant: ",self.nom, " moyenne: ",self.calc_moy())

nom = input("Entrer le nom: ")
note1= int(input("Entrer la note 1: "))
note2= int(input("Entrer la note 2: "))
E = Etudiant (nom, note1, note2)
E.afficher()
```

# Constructeurs

- Un constructeur est une méthode particulière invoquée implicitement lors de la création d'un objet
- Il permet d'initialiser les données des objets (les attributs) de la classe dont elle dépend
- Le constructeur ne doit pas avoir un type de retour
- Une classe peut posséder plusieurs constructeurs, mais un objet donné n'aura pu être produit que par un seul constructeur

# Constructeur et instantiation

- Le constructeur d'une classe se présente comme une méthode et suit la même syntaxe à ceci près que son nom est imposé : `__init__`
- Hormis le premier paramètre, invariablement `self`, il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

## Déclaration d'un constructeur

```
class nom_classe :  
    def __init__(self, param_1, ..., param_n):  
        # code du constructeur
```

Déclaration d'un constructeur à n paramètres

## Appel d'un constructeur

```
x = nom_classe (valeur_1,...,valeur_n)
```

## Exemple :

```
class classe1:  
    def __init__(self):  
        # pas de paramètre supplémentaire  
        print("constructeur de la classe classe1")  
        self.n = 1 # ajout de l'attribut n  
  
x = classe1()# affiche constructeur de la classe classe1  
print(x.n) | # affiche 1
```

```
class classe2:  
    def __init__(self, a, b):  
        # deux paramètres supplémentaires  
        print("constructeur de la classe classe2")  
        self.n = (a + b) / 2 # ajout de l'attribut n  
  
x = classe2(5, 9) # affiche constructeur de la classe classe2  
print(x.n) # affiche 7
```

# Constructeur et instantiation

## Constructeur et instantiation

- Par défaut, toute classe en Python a un constructeur par défaut sans paramètre
- Le constructeur par défaut n'existe plus si la classe présente un constructeur à paramètre
- Contrairement à d'autres langues, la classe de Python n'a qu'un seul constructeur. Cependant, Python permet à un paramètre de prendre une valeur par défaut.
- Remarque : Tous les paramètres requis doivent précéder tous les paramètres qui ont des valeurs par défaut.

Exemple :

```
class Person :  
  
    # Les paramètres d'âge (age) et de genre (gender) ont une valeur par défaut.  
    def __init__(self, name, age = "1", gender = "Male" ):  
        self.name = name  
        self.age = age  
        self.gender= gender  
  
    def showInfo(self):  
        print (self.name + "    "+self.age + "    "+self.gender)
```

```
if __name__ == '__main__':  
    aimee = Person("Aimee", "21", "Female")  
    aimee.showInfo() #affiche Aimee 21 Female  
  
    alice = Person("Alice") # age, gender par défaut.  
    alice.showInfo() #affiche Alice 1 Male  
    |  
    tran = Person("Tran", "37") # gender par défaut.  
    tran.showInfo() #affiche Tran 37 Male
```

# Rappel

Quel code ne va pas me retourner d'erreur après exécution

1:

```
class Counter:
    def __init__(self, count):
        self.count = count
        self.name = name

c = Counter(0, "My counter")
print(c.count)
```

3:

```
class Counter:
    self.count = 5
    self.name = "My counter"

c = Counter(0, "My counter")
print(c.count)
```

2:

```
class Counter:
    def __init__(self, count, name):
        self.count = 5
        self.name = name

c = Counter(0, "My counter")
print(c.count)
```

4:

```
class Counter:
    def __init__(self, count, name):
        self.name = name

c = Counter(0, "My counter")
print(c.count)
```



# Destructeur

- Les destructeurs sont appelés lorsqu'un objet est détruit.
- En Python, les destructeurs ne sont pas aussi nécessaires qu'en C++ (par exemple), car Python dispose d'un ramasse-miettes qui gère automatiquement la gestion de la mémoire.
- La méthode `__del__()` est une méthode appelée destructeur en Python. Il est appelé lorsque toutes les références à l'objet ont été supprimées, c'est-à-dire lorsqu'un objet est nettoyé.

## Syntaxe de destructeur

```
def __del__(self):  
    # actions
```

## Exemple :

```
class Person :  
  
    def __init__(self, name, age = 1, gender = "Male" ):  
        self.name = name  
        self.age = age  
        self.gender= gender  
  
    def showInfo(self):  
        print (self.name + " " +self.age + " " + self.gender)  
  
    def __del__(self):  
        print("je suis le destructeur")
```

```
if __name__ == '__main__':  
    aimee = Person("Aimee", "21", "Female")  
    aimee.showInfo()  
  
    del aimee #affiche je suis le destructeur  
    print(aimee) #NameError: name ' aimee ' is not defined  
|
```

# Les Méthodes

- Les méthodes sont des fonctions qui sont associées de manière explicite à une classe. Elles ont comme particularité un accès privilégié aux données de la classe elle-même.
- Les méthodes sont des fonctions pour lesquelles la liste des paramètres contient obligatoirement un paramètre explicite (self) qui est l'instance de la classe à laquelle cette méthode est associée.
- Ce paramètre est le moyen d'accéder aux données de la classe.
- Déclaration d'une méthode

## Déclaration d'une méthode

```
class nom_classe :  
    def nom_methode(self, param_1, ..., param_n):  
        # corps de la méthode...
```

Self désigne l'instance courante de la classe

## Appel d'une méthode

```
cl = nom_classe() # variable de type nom_classe  
t = cl.nom_methode (valeur_1, ..., valeur_n)
```

# Affichage de l'instance

```
class Person :  
  
    # Les paramètres d'âge (age) et de genre (gender) ont une valeur par défaut.  
    def __init__(self, name, age=1, gender = "Male" ):  
        self.name = name  
        self.age = age  
        self.gender= gender  
  
if __name__ == "__main__":  
    aimee = Person("Aimee", 21, "Female")  
    print(aimee) # affiche: <__main__.Person object at 0x0000024FE3FE7F40>
```

- Il est également possible de vérifier qu'une instance est bien issue d'une classe donnée avec la fonction isinstance()

```
class Person :  
  
    # Les paramètres d'âge (age) et de genre (gender) ont une valeur par défaut.  
    def __init__(self, name, age=1, gender = "Male" ):  
        self.name = name  
        self.age = age  
        self.gender= gender  
  
if __name__ == "__main__":  
    aimee = Person("Aimee", 21, "Female")  
    print(isinstance(aimee, Person)) # affiche: True
```

# Attributs

- Les attributs sont des variables qui sont associées de manière explicite à une classe.
- Les attributs de la classe se comportent comme des variables globales pour toutes les méthodes de cette classe.

## Déclaration d'un attribut

```
class nom_classe :  
    def nom_methode (self, param_1, ..., param_n) :  
        self.nom_attribut = param_1
```

Déclaration d'un attribut en précédant son nom par **self**

# Attributs Spéciales

## L'attribut spécial `__dict__`

- Cet attribut spécial donne les valeurs des attributs de l'instance :

```
if __name__ == '__main__':  
    ma_voiture = Voiture()  
    print(ma_voiture.__dict__)  
    # affiche {'couleur': 'rouge'}
```

## Fonction `dir`

- La fonction `dir` donne un aperçu des méthodes de l'objet :

```
if __name__ == '__main__':  
    ma_voiture = Voiture()  
    print(dir(ma_voiture))  
    # affiche ['__doc__', '__init__', '__module__', 'get_couleur', 'set_couleur']
```

# Attributs Spéciales

## L'attribut spécial `__dict__`

- Cet attribut spécial donne les valeurs des attributs de l'instance :

```
if __name__ == '__main__':  
    ma_voiture = Voiture()  
    print(ma_voiture.__dict__)  
    # affiche {'couleur': 'rouge'}
```

## Fonction `dir`

- La fonction `dir` donne un aperçu des méthodes de l'objet :

```
if __name__ == '__main__':  
    ma_voiture = Voiture()  
    print(dir(ma_voiture))  
    # affiche ['__doc__', '__init__', '__module__', 'get_couleur', 'set_couleur']  
    |
```

# Attributs d'instance

## Ajout d'un attribut d'instance

- L'ajout d'un attribut depuis l'extérieur de la classe avec une syntaxe `instance.nouvel_attribut = valeur`, créera ce nouvel attribut uniquement pour cette instance :

```
if __name__ == "__main__":
    ma_voiture = Voiture()
    print("Attributs de ma voiture")
    print(ma_voiture.__dict__)
    sa_voiture = Voiture()
    sa_voiture.matricule=1235
    print("Attributs de sa voiture")
    print(sa_voiture.__dict__)
#affiche
#Attributs de ma voiture
#{'couleur': 'rouge'}
#Attributs de sa voiture
#{'couleur': 'rouge', 'matricule': 1235}
```

# Attributs de classe

- Les **attributs de classe** sont différents des attributs d'instance.
- Un attribut dont la valeur est la même pour toutes les instances d'une classe est appelé un attribut de classe. Par conséquent, la valeur de l'attribut de classe est partagée par tous les objets.
- Les attributs de classe sont définis au niveau de la classe plutôt qu'à l'intérieur de la méthode `__init__()`.
- Contrairement aux attributs d'instance, les attributs de classe sont accessibles à l'aide du nom de la classe ou le nom d'instance.

```
class Citron:  
    forme=""  
    def donnerForme(self, params):  
        Citron.forme = params
```

forme est un attribut statique

Exemple :

```
class Fruit:  
    nom = 'fruit'  
    def __init__(self, couleur, poids_g):  
        print("J'aime manger des fruits")  
        self.couleur = couleur  
        self.poids_g = poids_g  
if __name__ == '__main__':  
    pomme = Fruit("verte", 100)  
    print(pomme.nom) #affiche fruit  
    print(Fruit.nom) #affiche fruit
```



# Attributs de classe

- La modification de l'attribut de classe à l'aide du nom de la classe affectera toutes les instances d'une classe
- La modification de l'attribut de classe à l'aide de l'instance n'affectera pas la classe et les autres instances. Cela n'affectera que l'instance modifiée.

Exemple :

```
class Fruit:
    nom = 'fruit'
    def __init__(self, couleur, poids_g):
        print("J'aime manger des fruits")
        self.couleur = couleur
        self.poids_g = poids_g
if __name__ == '__main__':
    pomme = Fruit("verte", 100)
    banane = Fruit("jaune", 100)

    print(banane.nom) #affiche fruit
    pomme.nom="fruit d'été"
    print(banane.nom) #affiche fruit
    pomme.nom="fruit d'été"
    print(pomme.nom) # affiche fruit d'été
    print(Fruit.nom) #affiche fruit
    print(banane.nom) #affiche fruit
```

# Rappel

## Méthodes Spéciales

### 1) Une classe peut instancier

☐ des variables    ☐ des objets    ☐ des attributs    ☐ des méthodes

### 2) Que retourne un constructeur

☐ Rien    ☐ des arguments    ☐ True Or false

### 3) Comment ajoutez-vous un élément à la fin d'une liste en Python?

☐ list.add()    ☐ list.append()    ☐ list.insert()

# Les tuples

Les tuples sont **immuables**, une fois créés, vous ne pouvez pas ajouter, supprimer ou modifier des éléments.

```
# Déclaration d'un tuple pour les informations d'un livre
livre_info = ("Introduction to Python", "John Smith", 2022)

# Accès aux éléments du tuple
titre_livre = livre_info[0]
auteur_livre = livre_info[1]
annee_publication = livre_info[2]

# Affichage des informations du livre
print("Titre du livre :", titre_livre)
print("Auteur du livre :", auteur_livre)
print("Année de publication :", annee_publication)
```

# Les tuples

## Fonctions

Index()    len()

```
livre_info = ("Introduction to Python", "John Smith", 2022)
index = livre_info.index("John Smith")
print("Index de 'John Smith' :", index)
```

```
livre_info = ("Introduction to Python", "John Smith", 2022)
longueur_tuple = len(livre_info)
print("Longueur du tuple :", longueur_tuple)
```

# \_\_str\_\_

```
class Personne :  
    def __init__(self, prenom = " " ,age = 0, adresse = " "): # Constructeur Paramétrés  
        self.prenom = prenom  
        self.age = age  
        self.adresse = adresse
```

```
p1 = Personne("Ahmed",30, "Rue de Rome,Casablanca")  
print(p1)
```

<\_\_main\_\_.Personne object at 0x00000268E5267150>

## \_\_str\_\_

```
class Personne :  
    def __init__(self, prenom = "" ,age = 0, adresse = ""): # Constructeur Paramétrés  
        self.prenom = prenom  
        self.age = age  
        self.adresse = adresse  
  
    def __str__(self):  
        return "Prenom :"+self.prenom + " Age :" + str(self.age) + " Adresse :" + self.adresse  
  
p1 = Personne("Ahmed",30, "Rue de Rome,Casablanca")  
print(p1)
```

Prenom : Ahmed Age : 30 Adresse : Rue de Rome,Casablanca

# Exercice 3

Implémentez la classe « Etudiant » avec les membres suivants:

**nom** : (de type String)

**note1, note2** : (de type float)

**calc\_moy()** : calcule la note moyenne.

**afficher ()** : affiche le nom et la note moyenne.

Demandez à l'utilisateur d'entrer le nom et les notes d'un étudiant. et affiche leur nom et la note moyenne.

# Exercice

ce programme simule la gestion d'un simple compte bancaire. Le compte est créé avec un solde initial. Il est possible de déposer et de retirer des fonds, d'ajouter des intérêts et de connaître le solde actuel. Cela devrait être implémenté dans une classe nommée `Compte` qui comprend:

- 1) Un constructeur qui accepte une balance initiale comme paramètre.
- 2) Une fonction `getBalance` qui renvoie le solde actuel.



# Exercice

ce programme simule la gestion d'un simple compte bancaire. Le compte est créé avec un solde initial. Il est possible de déposer et de retirer des fonds, d'ajouter des intérêts et de connaître le solde actuel. Cela devrait être implémenté dans une classe nommée `Compte` qui comprend:

- 1) Un constructeur qui accepte une balance initiale comme paramètre.
- 2) Une fonction `getBalance` qui renvoie le solde actuel.
- 3) Une méthode `deposer` pour déposer un montant spécifié.
- 4) Une méthode `retirer` pour retirer un montant spécifié.
- 5) Une méthode `ajouter_Interet` pour ajouter de l'intérêt au compte.

La méthode `ajouter_Interet` prend le taux d'intérêt comme paramètre et modifie le solde du compte en  $\text{solde} * (1 + \text{taux d'intérêt})$ .

# Corrigé

```
class Compte:
    def __init__(self, balance_initial=0):
        self.balance = balance_initial

    def getBalance(self):
        return self.balance

    def deposer(self, montant):
        if montant > 0:
            self.balance += montant
            print(f'Dépôt de {montant} effectué. Nouveau solde : {self.balance}')
        else:
            print("Montant de dépôt invalide.")

    def retirer(self, montant):
        if 0 < montant <= self.balance:
            self.balance -= montant
            print(f'Retrait de {montant} effectué. Nouveau solde : {self.balance}')
        else:
            print("Montant de retrait invalide ou solde insuffisant.")

    def ajouter_Interet(self, taux_interet):
        if taux_interet >= 0:
            self.balance *= (1 + taux_interet)
            print(f'Intérêt ajouté. Nouveau solde : {self.balance}')
        else:
            print("Taux d'intérêt invalide.")
```

# Corrigé

```
# Exemple d'utilisation de la classe
compte1 = Compte(1000)
print(f'Solde initial : {compte1.getBalance()}')

compte1.deposer(500)
compte1.retirer(200)
compte1.ajouter_Interet(0.05)

print(f'Solde final : {compte1.getBalance()}')
```

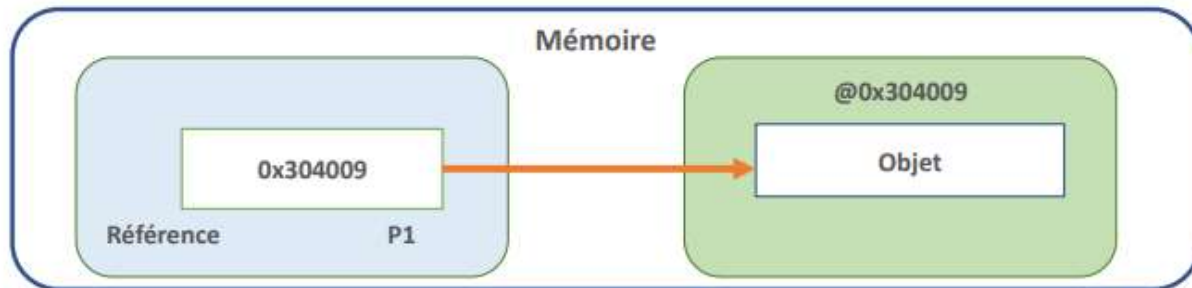
```
ripts/python.exe d:/Manuels/P00/Exo1.py
Solde initial : 1000
Dépôt de 500 effectué. Nouveau solde : 1500
Retrait de 200 effectué. Nouveau solde : 1300
Intérêt ajouté. Nouveau solde : 1365.0
Solde final : 1365.0
```

# Passage Par référence

## Notion d'objet

OBJET= Référent + Etat + Comportement

- Chaque objet doit avoir un nom (référence) « qui lui est propre » pour l'identifier
- La référence permet d'accéder à l'objet, mais n'est pas l'objet lui-même. Elle contient l'adresse de l'emplacement mémoire dans lequel est stocké l'objet.



# Visibilités

## Nom de Classe

---

+ Attribut publique

- Attribut privé

# Attribut protégé

---

+ Méthode publique

- Méthode privé

# Méthode protégé

# Getters & Setters

Pour interagir avec les attributs d'un objet de l'extérieur, il suffit de créer des méthodes publiques dans la classe appelées **Accesseurs et Modificateurs**



# Getters et Setters

```
class Geek:  
    def __init__(self, age = 0):  
        self._age = age  
    def get_age(self):  
        return self._age  
    def set_age(self, x):  
        self._age = x
```

```
if __name__ == '__main__':  
    raj = Geek()  
    raj.set_age(21)  
    print(raj.get_age())  
    print(raj._age)  
  
#affiche:  
#21  
#21
```

Les Getters et Setters en Python sont souvent utilisés pour éviter l'accès direct à un champ de classe, c'est-à-dire que les variables privées ne peuvent pas être accessibles directement ou modifiées par un utilisateur externe.

# Encapsulation

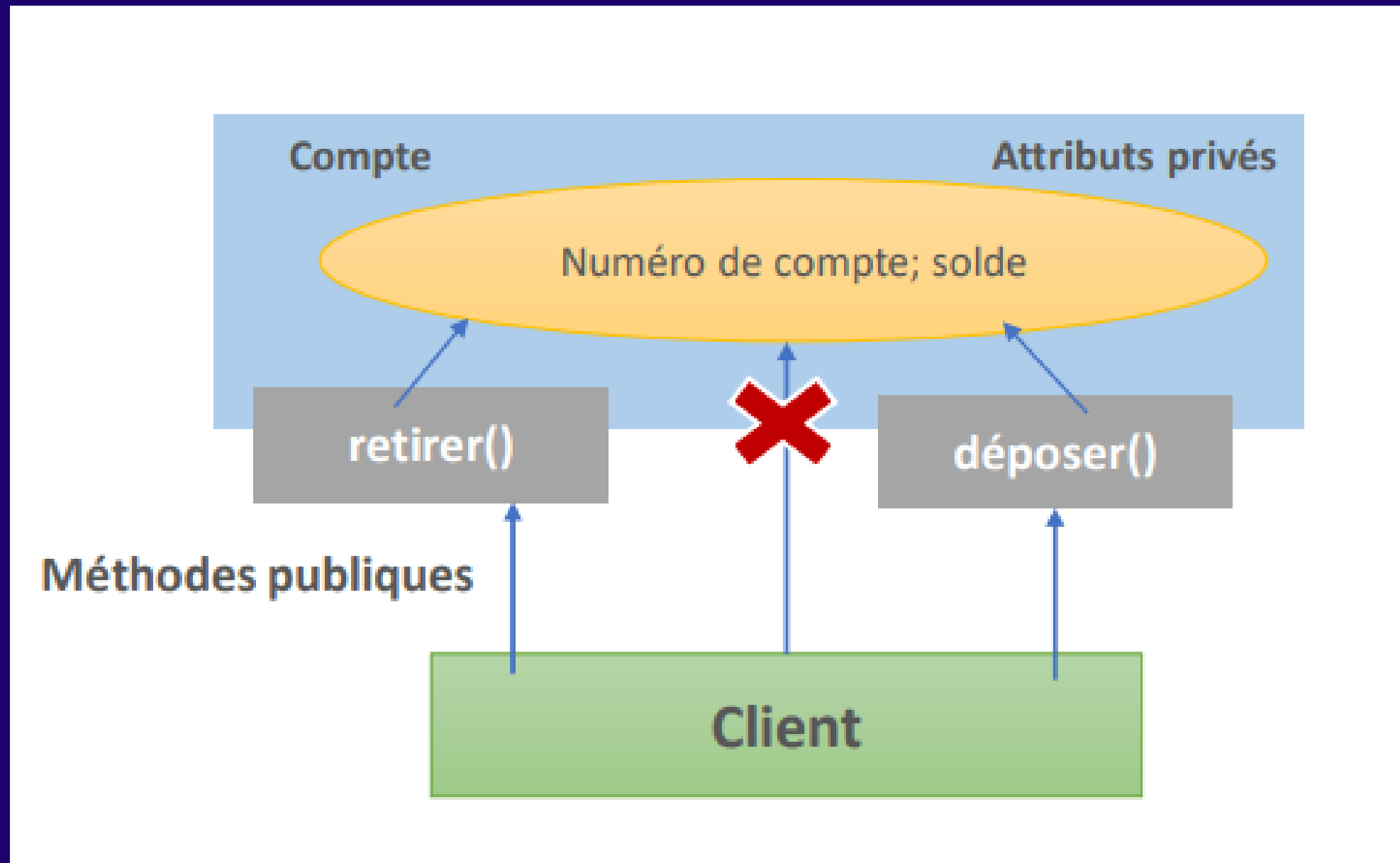
**PROTECTION DES DONNÉES**

**MODULARITÉ**

**OBJET = CODE (MÉTHODES) + DONNÉES (ATTRIBUTS)**



# Encapsulation & Privilèges



# PROPERTY

- En Python `property()` est une fonction intégrée qui crée et renvoie un objet de propriété.
- Un objet de propriété a trois méthodes, `getter()`, `setter()` et `delete()`.
- La fonction `property()` en Python a trois arguments **`property(fget, fset, fdel, doc)`**
  - **`fget`** est une fonction pour récupérer une valeur d'attribut
  - **`fset`** est une fonction pour définir une valeur d'attribut
  - **`fdel`** est une fonction pour supprimer une valeur d'attribut
  - **`doc`** est une chaîne contenant la documentation (docstring à voir ultérieurement) de l'attribut

```
class Stagiaire:
    def __init__(self, name = " ", cin = " ", age=0):
        self.name = name
        self.age = age
        self.__cin = cin

    #getter method
    def get_cin(self):
        return self.__cin

    #setter method
    def set_cin(self, c):
        self.__cin = c

    #destructeur
    def del_cin(self):
        del self.__cin

    #Property
    cin = property(get_cin, set_cin, del_cin, "la cin du stagiaire")
```

## PROPERTY

```
s1 = Stagiaire("Ahmed", "BE1567", 30)
s2 = Stagiaire("Amina", "BL9811", 24)
#Changing cin using setter
s1.set_cin("BE1234")
print(s1.get_cin())
s1.del_cin()
print(s2.get_cin())
```

```
#Using property
s1.cin = "BE1234"
print(s1.cin)
del s1.cin
print(s1.cin)
```

# PROPERTY

```
class Geeks:
    def __init__(self):
        self._age = 0
    def get_age(self):
        print("getter method called")
        return self._age
    def set_age(self,a):
        print("setter method called")
        self._age = a
    def del_age(self):
        del self._age
    age = property(get_age,set_age, del_age)
```

appel

```
if __name__ == '__main__':
    mark = Geeks()
    mark.age = 10
    print(mark.age)

#affiche:
#setter method called
#setter method called
#10|
```

# DÉCORATEURS

## Utilisation de @property

- @property est un décorateur qui évite d'utiliser des fonctions setter et getter explicites

Exemple :

```
class Etudiant:

    def __init__(self, nom_famille, prenom):
        self.nom_famille = nom_famille
        self.prenom = prenom

    @property
    def nomcomplet(self):
        return self.prenom + ' ' + self.nom_famille
```

```
if __name__ == '__main__':
    E1 = Etudiant('Richard', 'Marie')
    print('Le nom complet de E1 est :', E1.nomcomplet)

    # nous allons modifier le nom de l'objet E1
    print("\n La modification : ")
    E1.nom_famille = 'Bernard'
    print('Le nom complet de E1 est :', E1.nomcomplet)
```

nomcomplet() fonctionne comme un getter à cause du décorateur @property.  
Appel de nomComplet au lieu de nomcomplet()

le décorateur @property est utilisé avec la fonction nomcomplet()

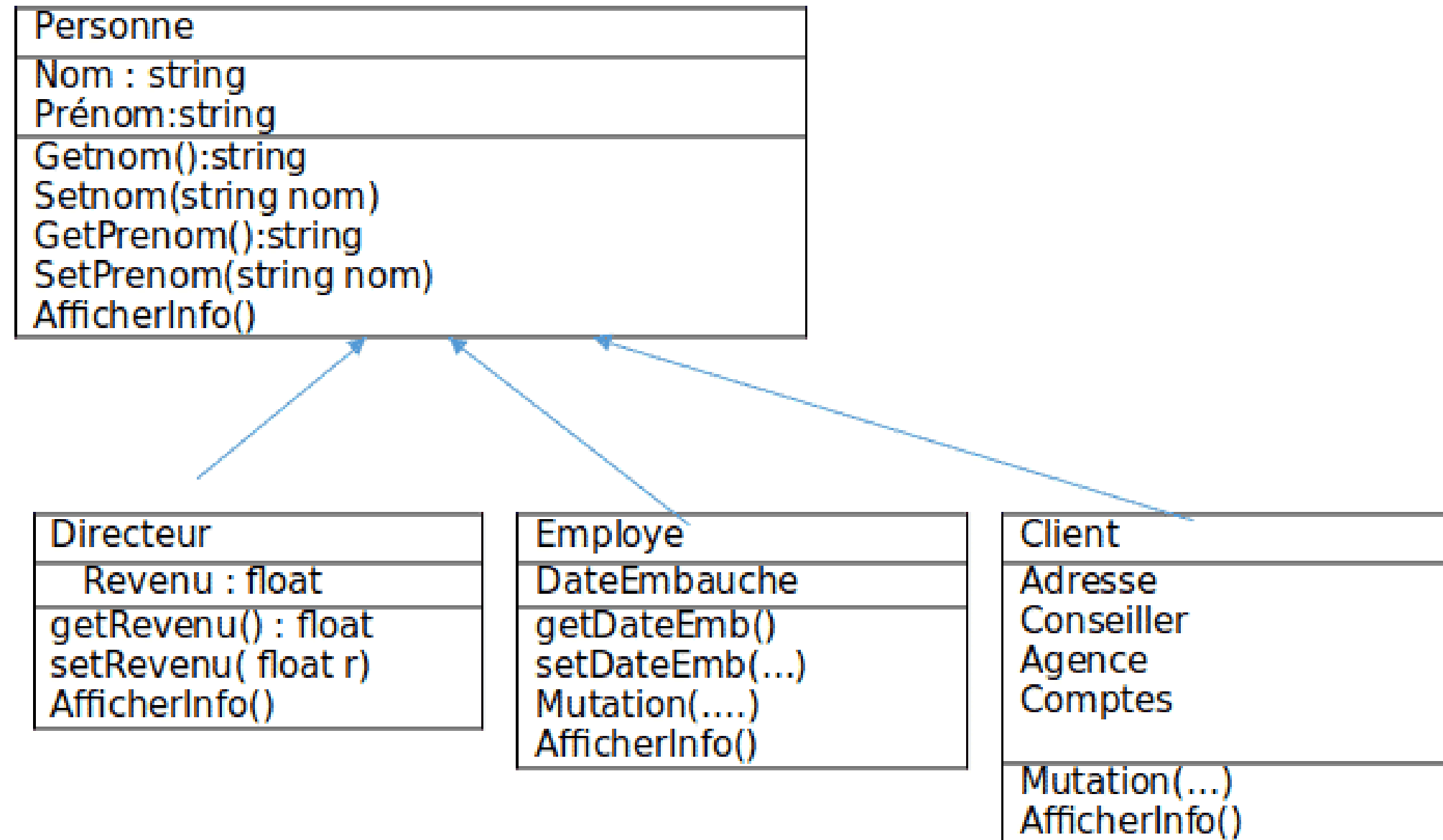
# DÉCORATEURS

```
@property
def cin(self):
    |     return self.__cin

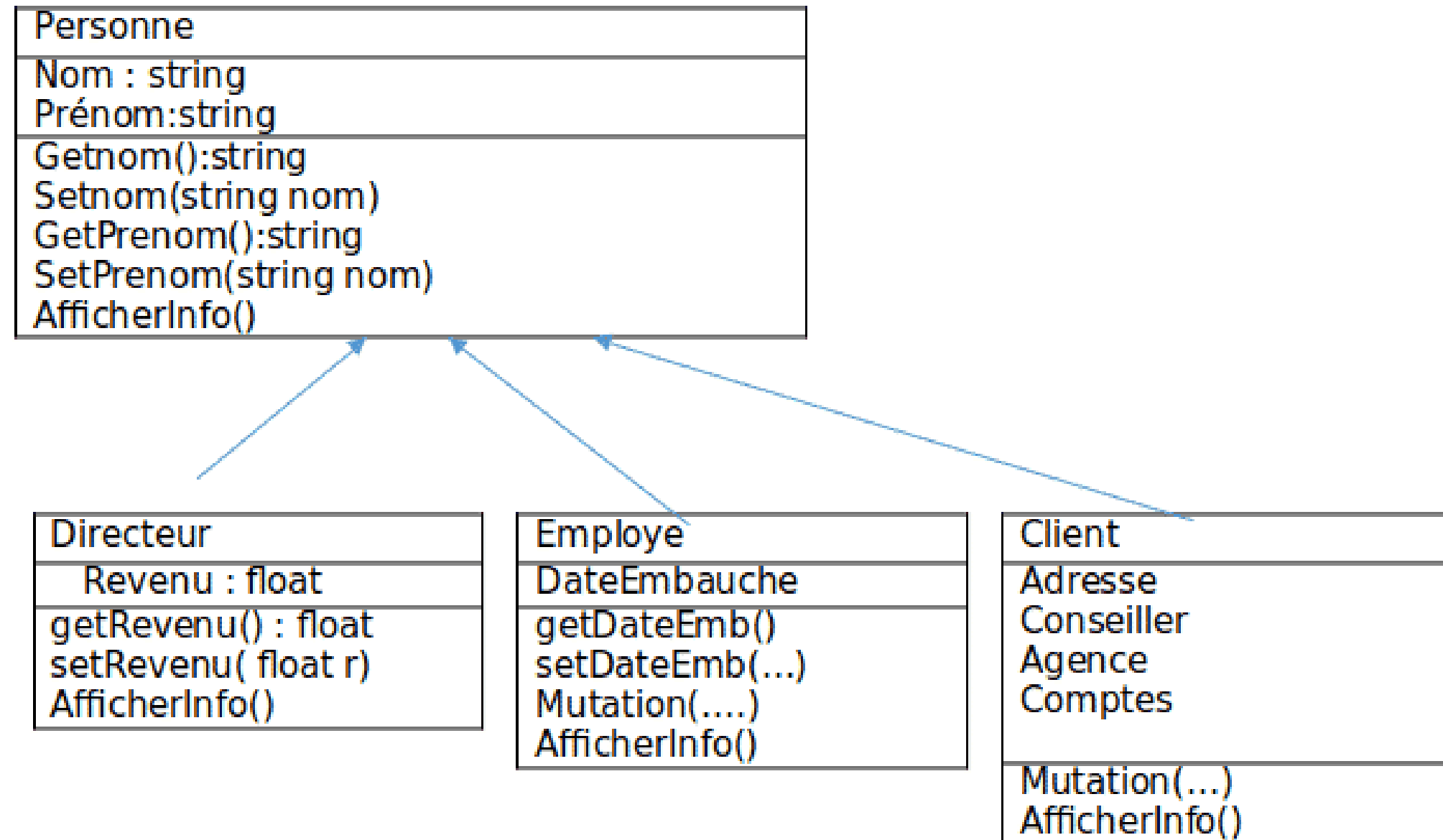
# Setter method
@cin.setter
def cin(self, c):
    |     self.__cin = c

# Deleter method
@cin.deleter
def cin(self):
    |     del self.__cin
```

# Héritage



# Héritage Simple





# Héritage Simple : Exemple

```
class Personne:
    def __init__(self,n,p,d):
        self.nom=n
        self.prenom=p
        self.dateNaiss=d

    def AfficherInfos(self):
        print("nom: ",self.nom,"prénom: ",self.prenom,"date de naissance : ",self.dateNaiss)
```

```
# class Class_fille(Class_Mère)
class Stagiaire(Personne):
    #Constructeur de la sous classe
    def __init__(self,n,p,notes):
        #Appeler le constructeur de la super classe
        Personne.__init__(self,n,p) #self.nom=n #self.prenom=p
        super().__init__(self,n,p)
        #Ajout d'un Attribut de la classe Stagiaire
        self.notes=notes

    #Méthode de la classe Stagiaire
    def AfficherNotes(self):
        print(self.notes)

    def AfficherInfos(self): #Redéfinition de la méthode AfficheInfos
        #super() : Accéder aux méthodes de la classe mère
        super().AfficherInfos()
        #print("nom: ",self.nom,"prénom: ",self.prenom)
        self.AfficherNotes() # print(self.notes)
```