

# UML et POO

Xavier Nodet, [xavier.nodet@gmail.com](mailto:xavier.nodet@gmail.com)

Janvier 2022

# Plan du cours

- Introduction
- UML
  - Étude fonctionnelle : acteurs, cas d'utilisation, diagrammes
  - Modélisation statique : classes et objets, attributs, opérations, etc
- Programmation orientée objet
  - Python, Java et C++



Comment le client a exprimé son besoin



Comment le chef de projet l'a compris



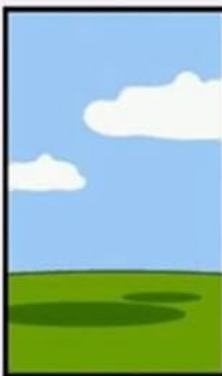
Comment l'ingénieur l'a conçu



Comment le programmeur l'a écrit



Comment le responsable des ventes l'a décrit



Comment le projet a été documenté



Ce qui a finalement été installé



Comment le client a été facturé



Comment la hotline répond aux demandes



Ce dont le client avait réellement besoin

# Introduction

- Communication avec les clients
- Communication avec les developpeurs
- Textes et diagrammes UML
  - Créé en 1996 par Rumbaugh, Jacobson et Booch
- Quelques notions de POO

# Plan du cours

- Introduction
- UML
  - Étude fonctionnelle : acteurs, cas d'utilisation, diagrammes
  - Modélisation statique : classes et objets, attributs, opérations, etc
- Programmation orientée objet
  - Python, Java et C++

# UML : les acteurs

# UML : les acteurs

- Rôle joué par une entité
- Ne fait pas partie du système étudié
- Humain ou non

# UML : les acteurs

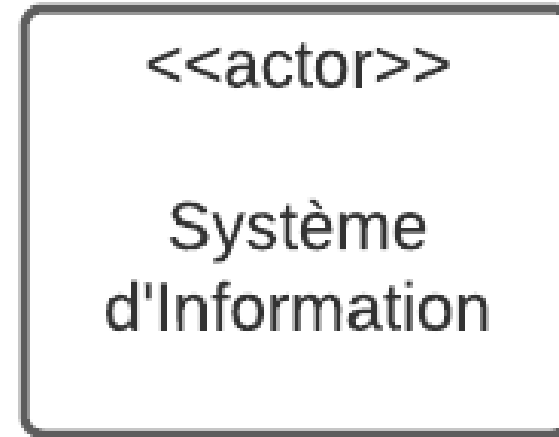
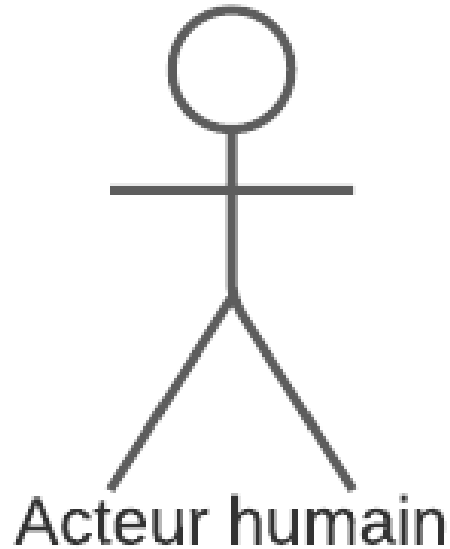
- Rôle joué par une entité
- Ne fait pas partie du système étudié
- Humain ou non

## Exemples:

- L'utilisateur d'une carte de paiement lors d'une transaction sur Internet.
- Le système de gestion des stocks, dans l'étude d'une caisse de supermarché.



# UML : les acteurs





<https://forms.gle/z5gwPvn2E6UDomQZ9>

# UML : les cas d'utilisation

### **Systeme considéré**

Le logiciel exécuté sur l'ordinateur des caisses de paiement du parking. Ni le terminal CB, ni le lecteur de ticket intégrés aux caisses ne font partie du système.

### **Titre**

Validation d'un ticket de parking et paiement du stationnement

### **Liste des acteurs**

Le lecteur de ticket, le terminal de paiement CB, le serveur de

## base de données du parking

### **Pré-conditions**

- La connexion au serveur du parking est établie
- Le terminal CB n'a pas signalé d'erreur de communication lors du dernier paiement

### **Scénario nominal**

- Un ticket de parking est introduit dans le lecteur
- Calculer le montant à payer
- Afficher le montant sur l'écran

- Demander le paiement de ce montant au terminal CB
- Attendre la réponse du terminal CB
- Envoyer un message au serveur en indiquant que le ticket a été payé
- Imprimer la date, l'heure et le montant payé sur le ticket
- Afficher un message et renvoyer le ticket

#### **Scénario alternatif**

- Si l'utilisateur demande l'annulation du paiement, afficher un message et renvoyer le ticket

### Cas d'erreur

- Si le ticket a déjà été payé, afficher un message et renvoyer le ticket
- Si le terminal ne répond pas 'payé' sous 60 secondes, afficher un message et renvoyer le ticket
- Si le terminal répond 'paiement refusé', afficher un message et renvoyer le ticket

# UML : les cas d'utilisation

- Séquence d'événements au cours de laquelle l'acteur principal interagit avec le système pour obtenir un résultat qui l'intéresse
- Description du comportement attendu du système
- Description du *quoi*, et non pas du *comment*



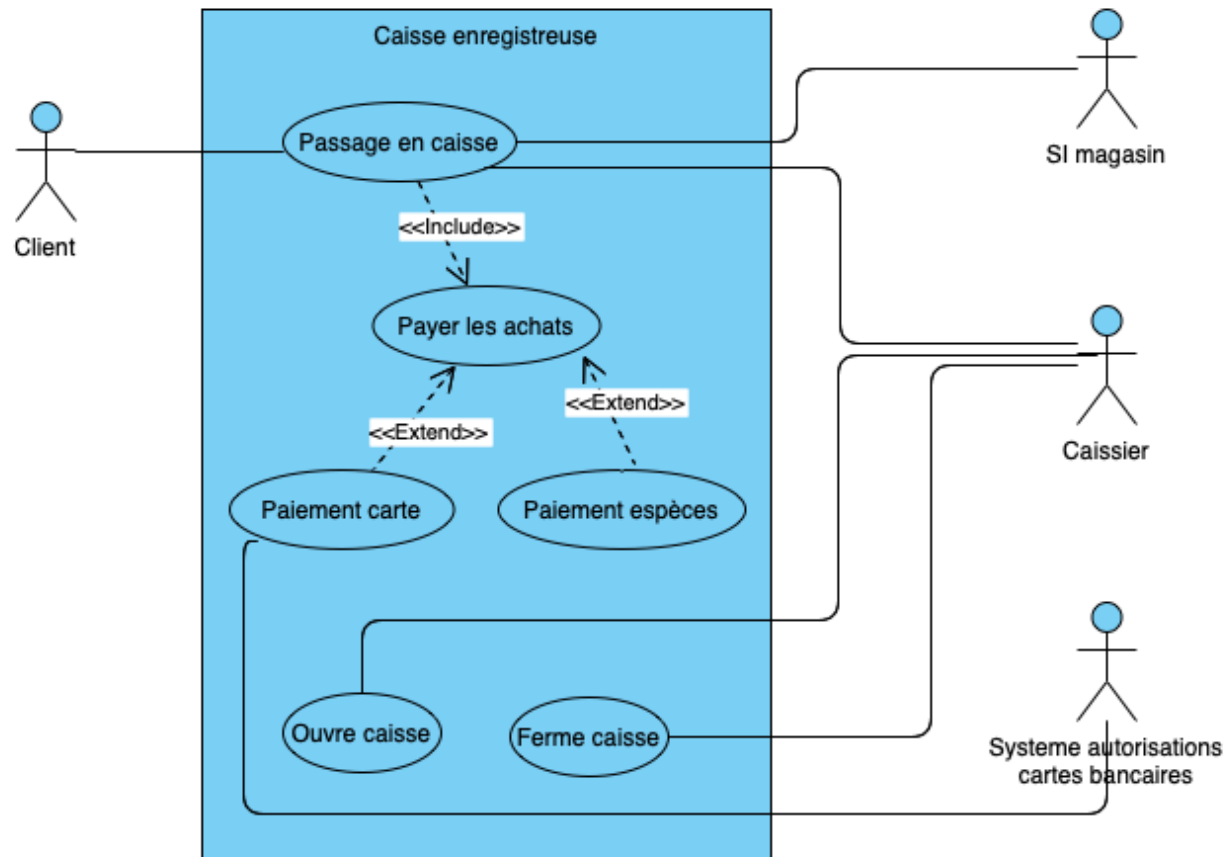
# UML : les cas d'utilisation

- Scénario nominal
- Enchainements alternatifs :
  - Le porteur de carte fait une ou deux (mais pas trois) erreurs de code.
  - Le client présente sa carte de fidélité à la caisse
- Enchainements d'erreur :
  - Pas d'autorisation de retrait
  - Livre déjà réservé

# UML : les cas d'utilisation

- Pré-conditions, post-conditions
- Exigences non fonctionnelles
- Inclusion d'un cas dans un autre : "includes"
- Extension d'un cas par un autre : "extends"

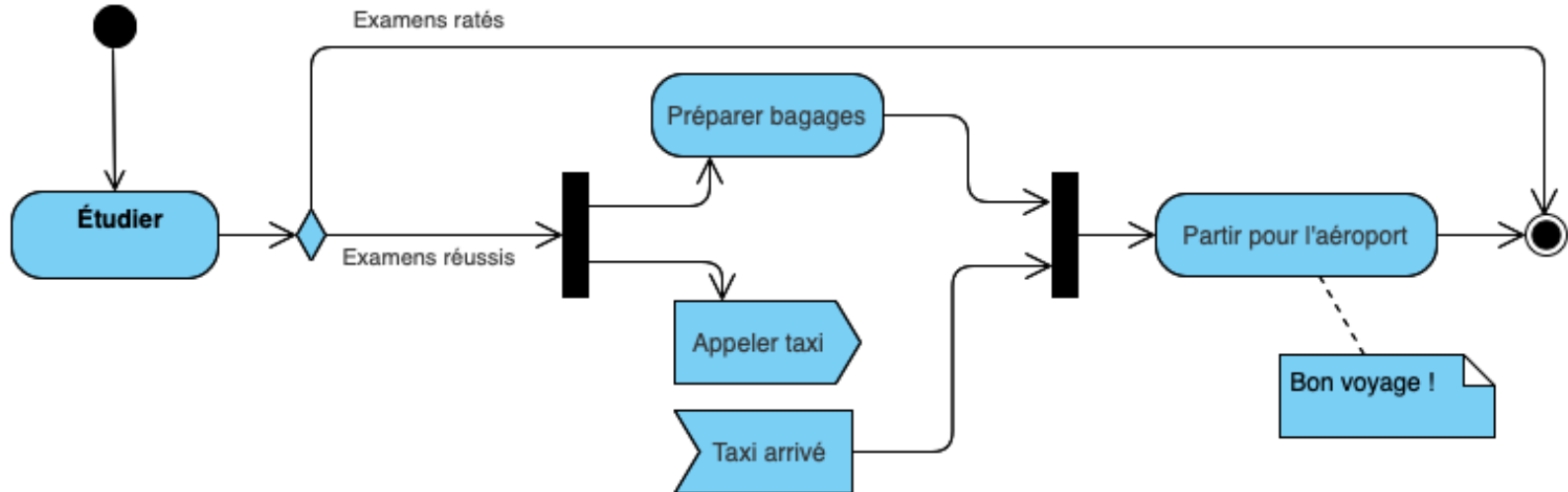
# UML : Diagramme de cas



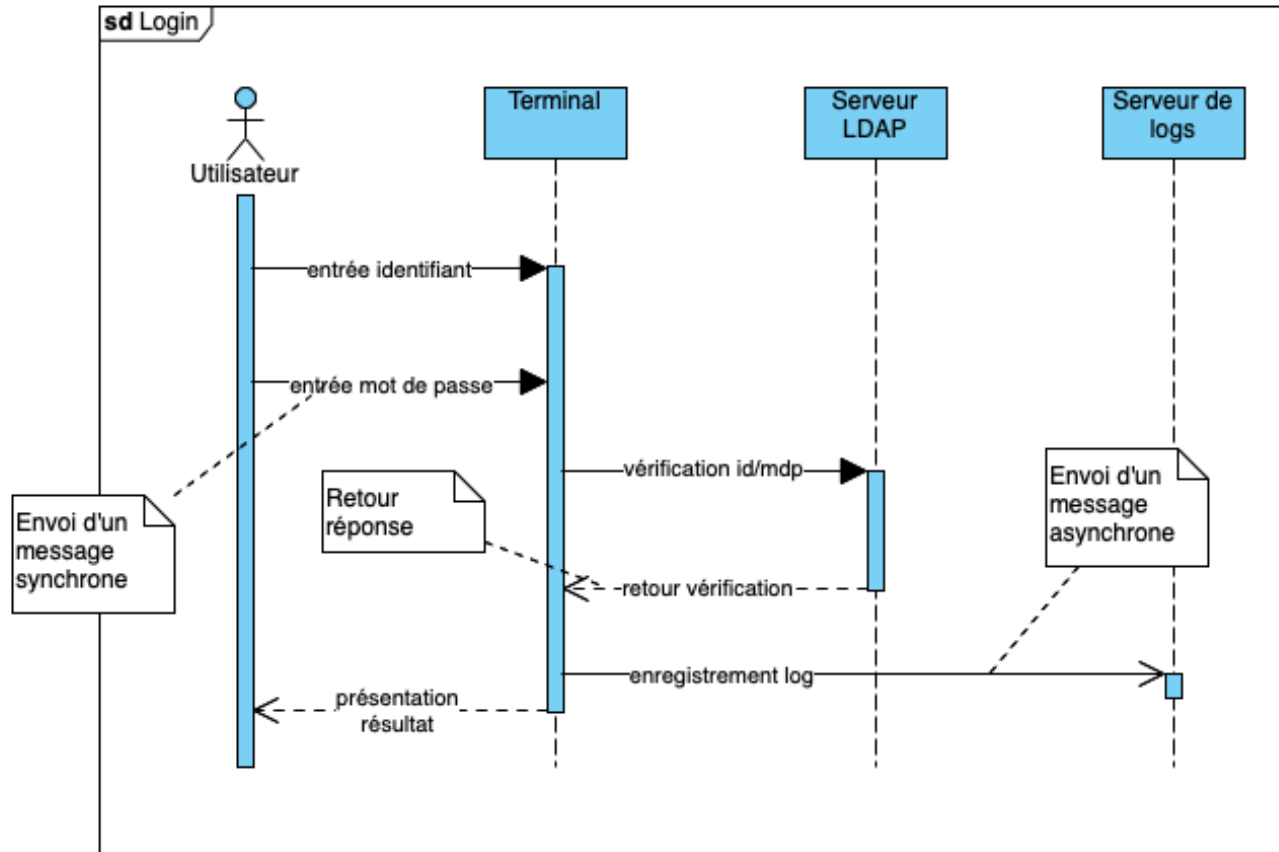


<https://forms.gle/6Ti8UEjQKR2H6c7G9>

# UML : Diagramme d'activité



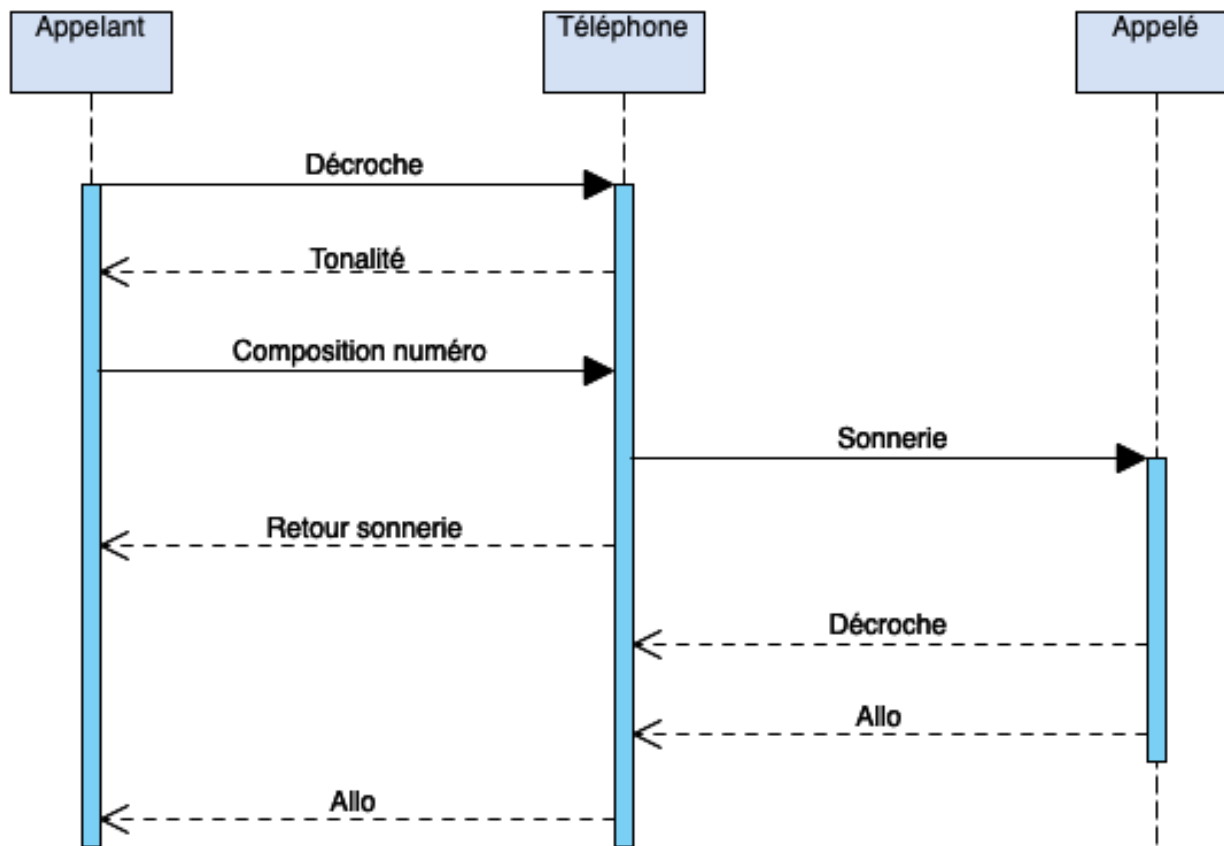
# UML : Diagramme de séquence





# UML : Diagramme de séquence

- Décrit les actions et messages échangés entre les acteurs
- Une *ligne de vie* verticale pour chaque acteur
- Messages synchrones (l'émetteur attend la réponse) ou asynchrones (pas de réponse, ou réponse différée)
- Rectangle sur la ligne de vie pour les traitements



# Plan du cours

- Introduction
- UML
  - Étude fonctionnelle : acteurs, cas d'utilisation, diagrammes
  - Modélisation statique : classes et objets, attributs, opérations, etc
- Programmation orientée objet
  - Python, Java et C++

# UML : Modélisation statique

## Décomposition

- Un système complexe sera décomposé pour faciliter son étude.
- Les composants d'un système deviennent acteurs pour l'étude d'un sous-système.

## Fin de l'étude fonctionnelle

- En théorie, une fois l'étude fonctionnelle terminée, parler au client ne devrait plus être nécessaire.
- En pratique, ce n'est pas aussi simple...

# Cycle en V

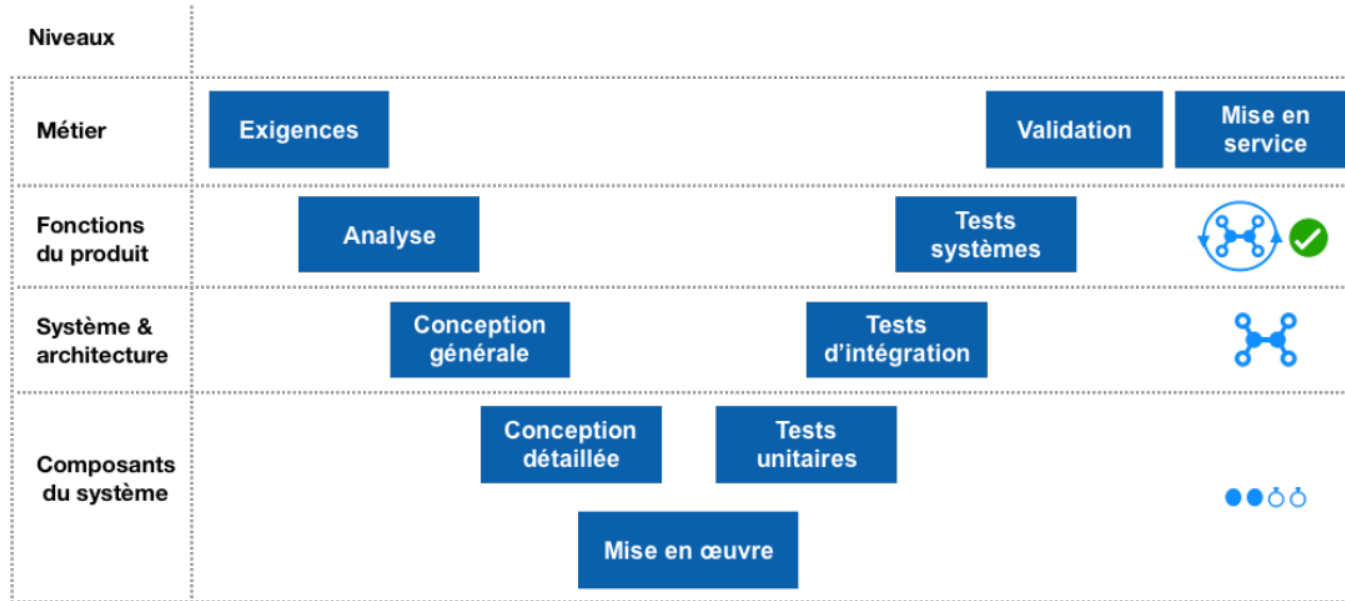
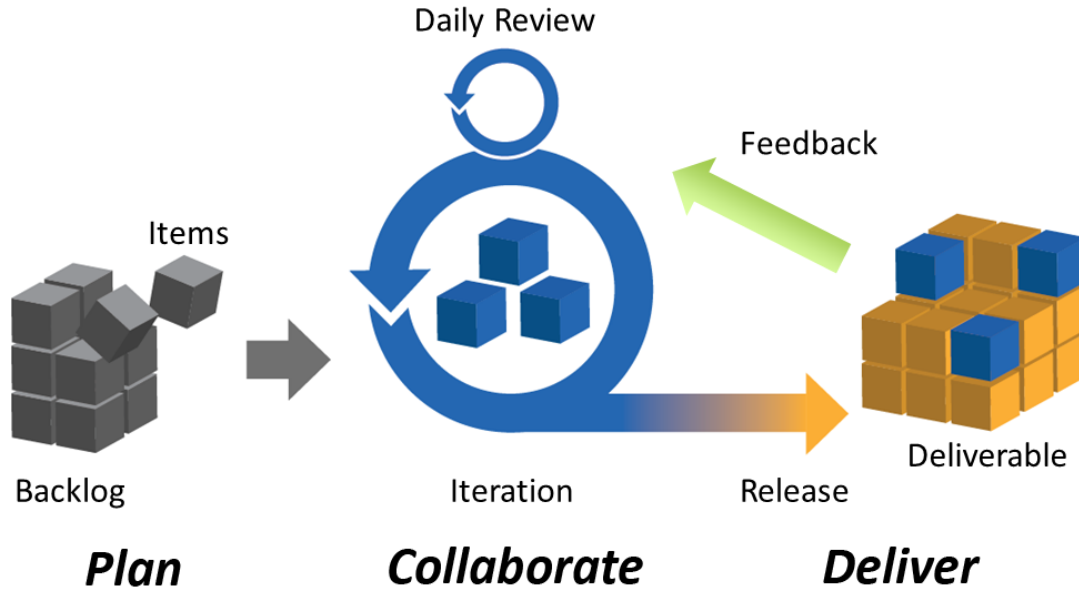


Figure 1. Le cycle en V, par Cth027 — Travail personnel, CC BY-SA 4.0, [link](#)

# Méthodes agiles



Agile Project Management: Iteration

Figure 2. Les méthodes agiles, par Planbox - Travail personnel, CC BY-SA 3.0, [link](#)

# Classes et objets

- Chaque type d'acteur est représenté par une *classe*.
- Les éléments manipulés dans le système étudié seront également représentés par des classes :
  - les livres d'une bibliothèque
  - les voitures d'un concessionnaire automobile

# Classes et objets

- La classe est le patron, le modèle, à partir duquel les objets sont *instanciés*.
- Chaque objet est construit à partir d'une et une seule classe.
- Chaque classe n'est pas nécessairement instanciée plusieurs fois.
- Exemples :
  - l'IHM d'un programme
  - le serveur de base de donnée auquel le système est connecté



# Représentation d'une classe

| Utilisateur  |
|--|
| prénom<br>nom<br>date de naissance<br>/age                               |
| liste_prêts()<br>ajout_prêt()<br>retrait_prêt()<br>envoyer_rappel(livre) |

# Attributs

- Attribut : propriété d'une classe qui associe une *variable* à chaque *instance* de cette classe.
- Exemples :
  - Les prénoms, noms et date de naissance d'un utilisateur
  - Le titre et le nombre de mots d'un livre

# Attributs

- Un attribut peut avoir une type simple (entier, chaîne de caractères, date, etc).
- Les liens avec d'autre objets ne sont pas des attributs, mais des relations.
- Un attribut peut être dérivé, déduit d'informations présentes ailleurs dans le modèle. Il est noté */attribut*.

# Opérations

- Une classe peut aussi définir des *opérations*. Ces opérations représentent des services que peuvent rendre les instances de la classe
- Exemples :
  - *nombre\_emprunts\_en\_cours()*
  - *rendre(livre)*
  - *envoyer\_rappel(utilisateur, livre)*

# Opérations

- Trois types de services :
  - demande d'information
  - enregistrement d'information
  - traitements sans échange d'informations

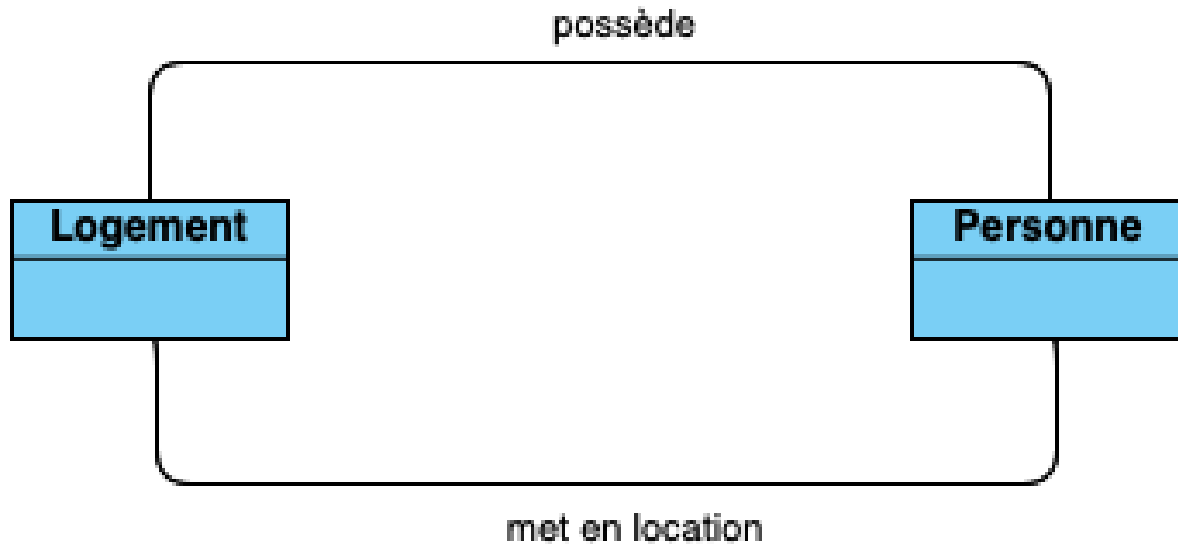
# Représentation d'une classe

| Utilisateur  |
|--|
| prénom<br>nom<br>date de naissance<br>/age                               |
| liste_prêts()<br>ajout_prêt()<br>retrait_prêt()<br>envoyer_rappel(livre) |

# Associations

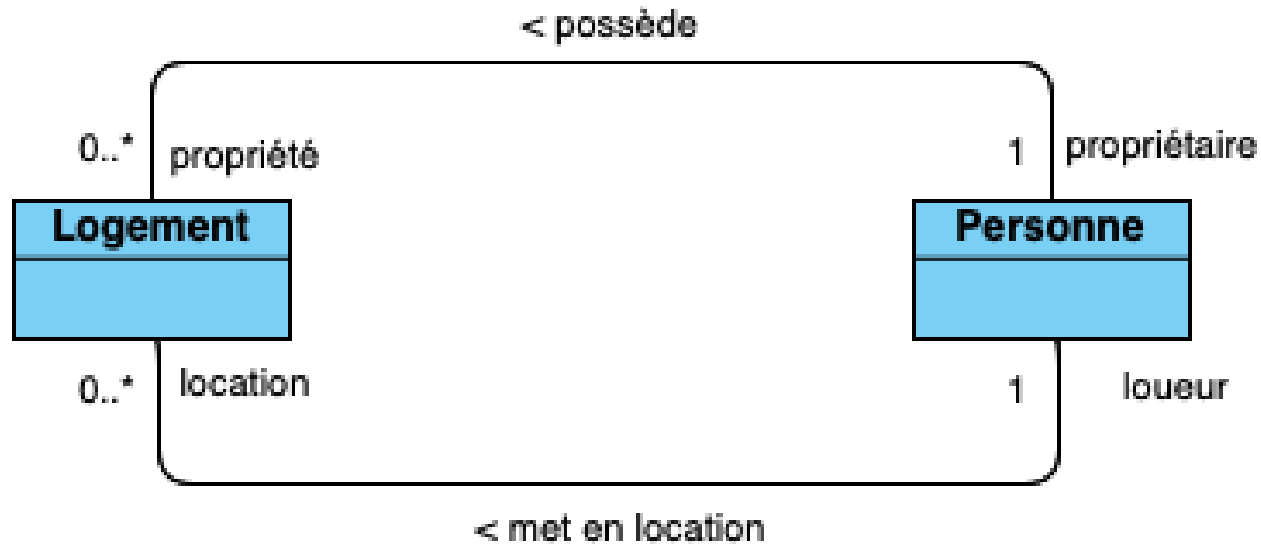
- Association : relation sémantique durable entre deux classes.
- Exemples :
  - Une bibliothèque possède des livres. La relation *possède* est une association entre la classe *Bibliothèque* et la classe *Livre*.
  - Un utilisateur emprunte des livres. La relation *emprunte* est une association entre la classe *Utilisateur* et la classe *Livre*.

# Associations

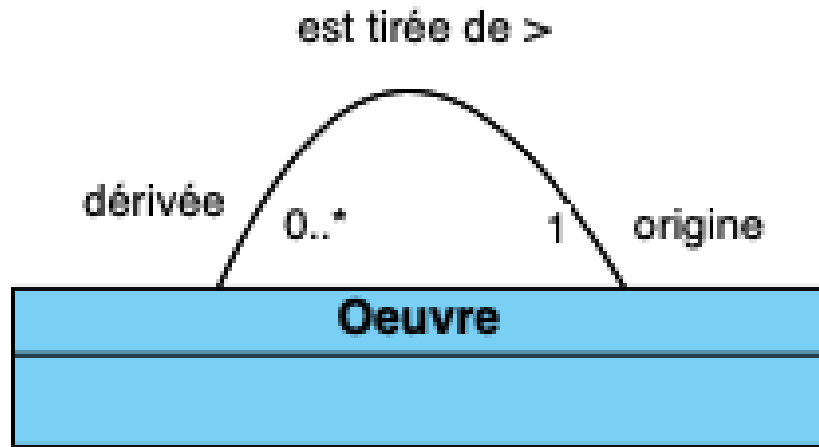




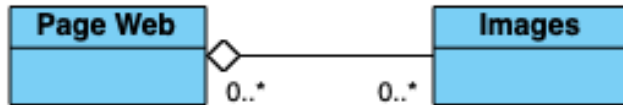
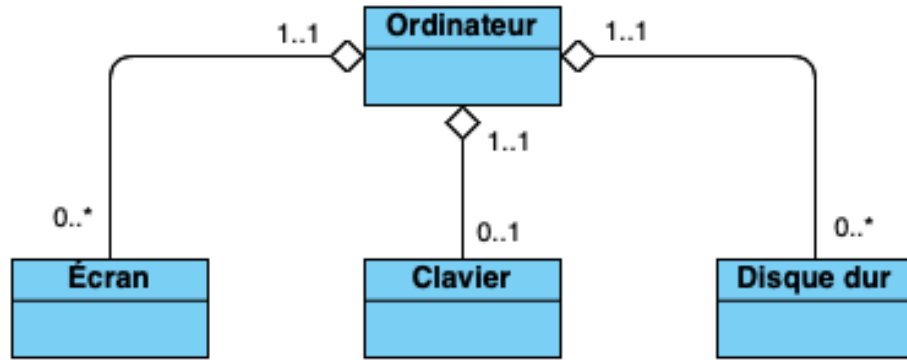
# Associations



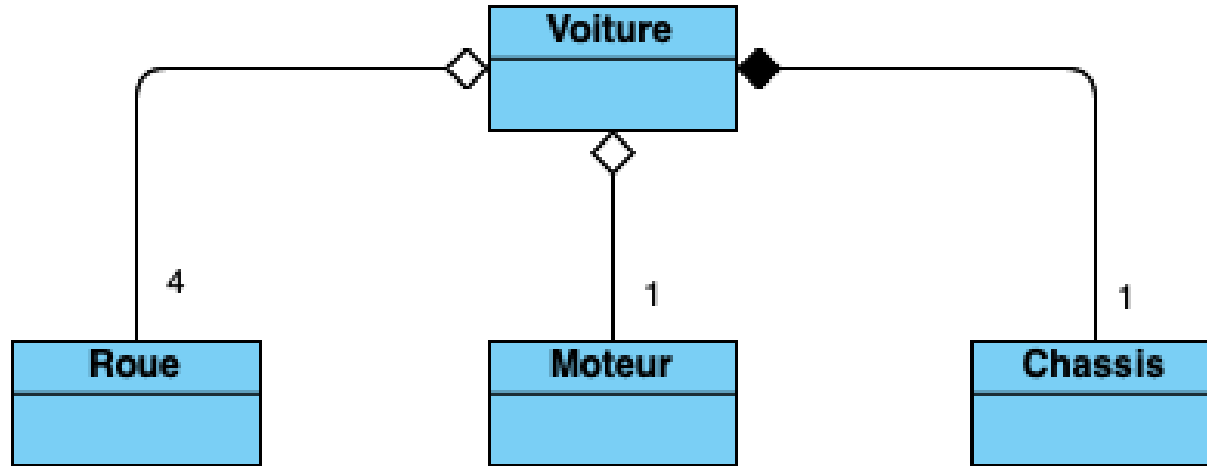
# Associations



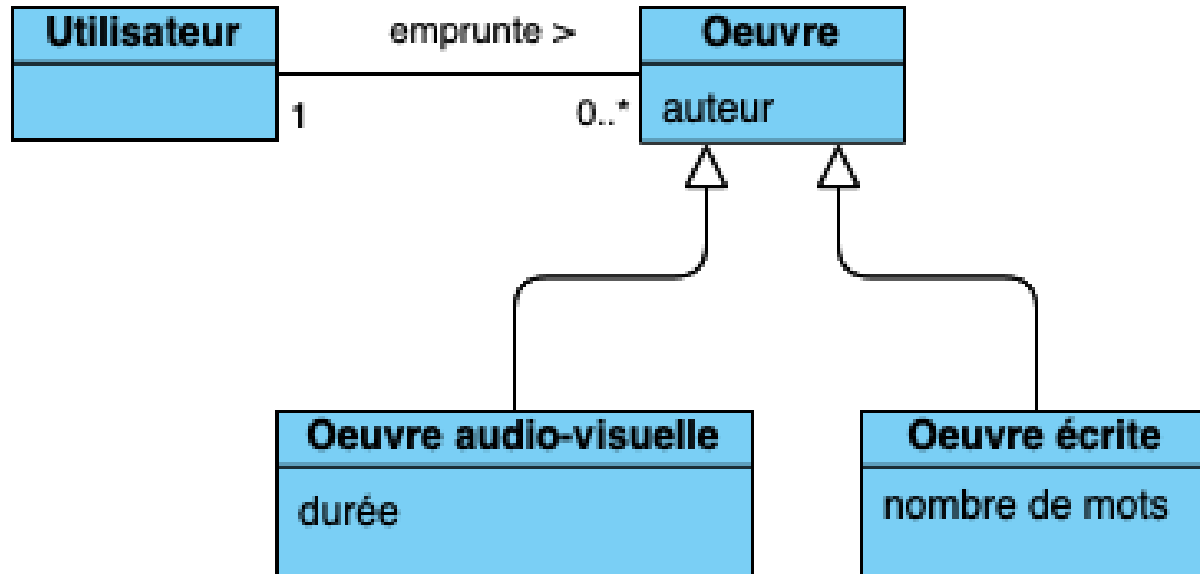
# Agrégation



# Composition



# Généralisation



# Généralisation

- *Oeuvre* est la *généralisation* de *Oeuvre écrite* et *Oeuvre audiovisuelle*.
- Réciproquement, on parle de *spécialisation*.
- Les instances d'une classe spécialisée sont aussi des instances de la *classe de base*, ou *super-classe*.
- La classe spécialisée *hérite* de tous les attributs et méthodes.
- Elle peut rajouter ses propres propriétés.
- Elle respecte le contrat de la classe de base : pré-conditions, post-conditions, invariants.



<https://forms.gle/mFMKuVzfLqxXtbmR8>

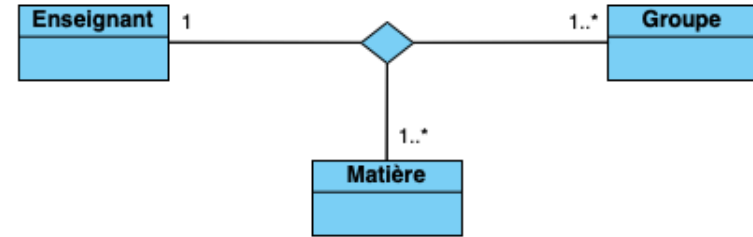
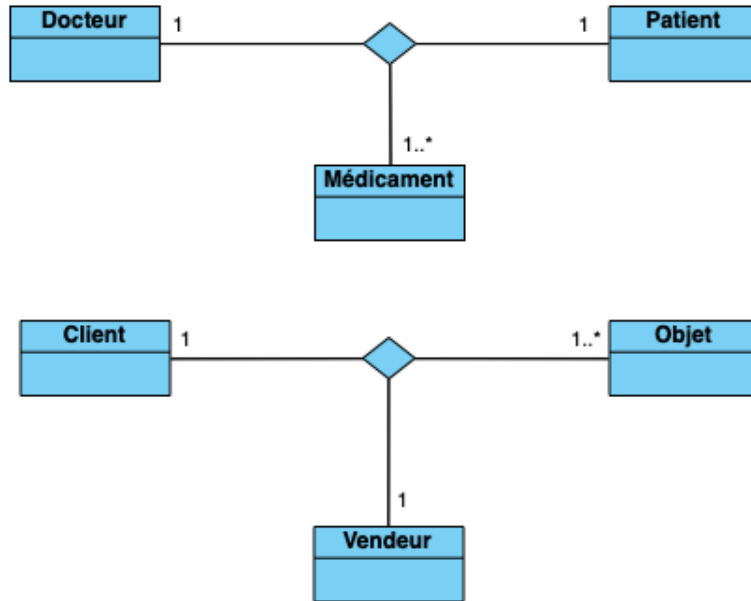
# Relation n-aires

Une relation peut mettre en jeu plus de deux classes.

- Un *docteur* prescrit un *médicament* à un *patient*.
- Un cours est donné par un *enseignant*, pour enseigner une *matière* à un *groupe* d'élèves.
- Un *client* passe une commande pour un *objet* proposé par un *vendeur*.



# Relation n-aires



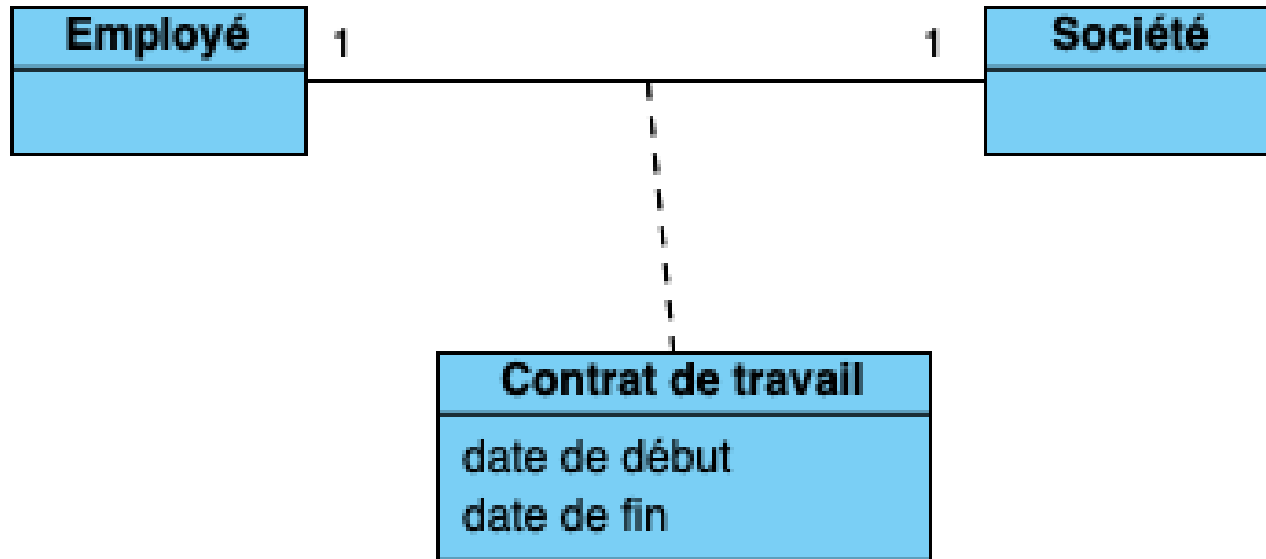
# Relation n-aires

- Ces relations sont représentées par un losange lié aux différentes classes.
- Chaque multiplicité s'entend comme le nombre d'instance quand les autres éléments sont fixés.
- Un client peut commander plusieurs objets, chaque objet peut être proposé par plusieurs vendeurs. Mais un même client ne pourra commander le même objet que chez un seul vendeur.
- Il y aura au plus un enseignant pour une matière et un groupe d'élèves donné.

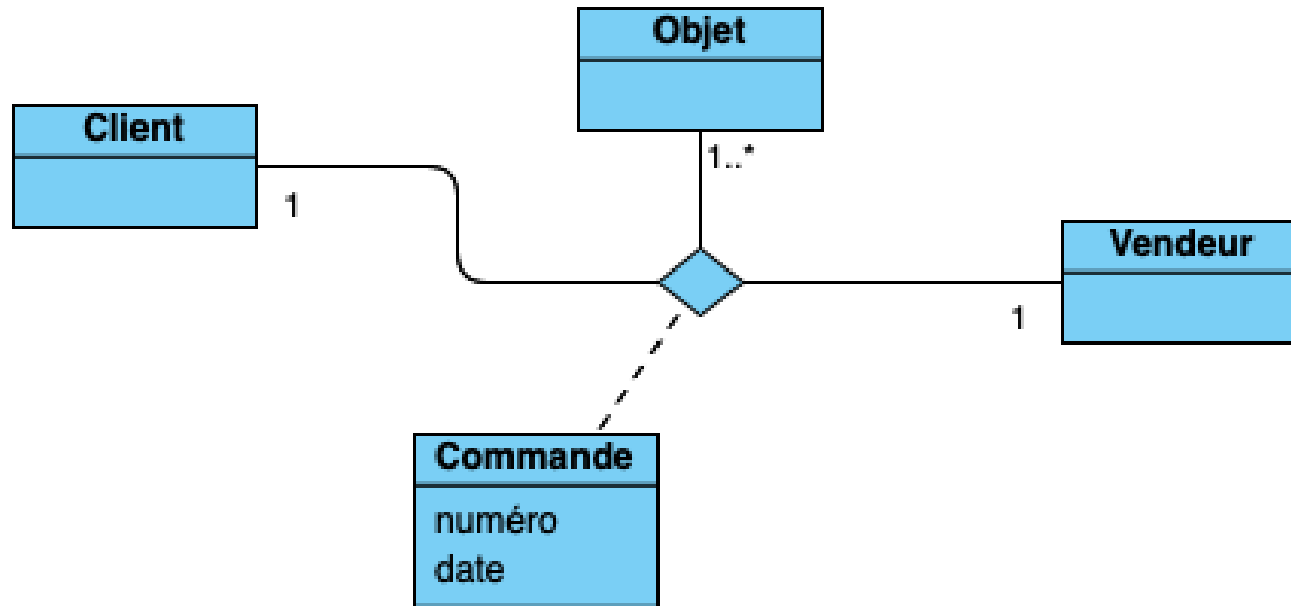
# Classe-association

- Il est souvent nécessaire de stocker des informations à propos d'une relation.
- L'association est alors définie par une classe-association.
- Elle représente l'association.
- Et porte les attributs et les opérations pour l'association.

# Classe-association



# Classe-association



# Plan du cours

- Introduction
- UML
  - Étude fonctionnelle : acteurs, cas d'utilisation, diagrammes
  - Modélisation statique : classes et objets, attributs, opérations, etc
- Programmation orientée objet
  - Python, Java et C++

# Programmation orientée objet

- On peut exprimer les concepts de la modélisation statique (classe, objets, attributs, opérations) dans tout langage informatique. Par exemple, le C.
- Certains langages comme le Python, le Java et le C++, incluent directement ces notions.
- Ce sont des *langages orientés objet*.

```
class MyClass:
    val = None

    def __init__(self, arg1, arg2):
        self._val1 = arg1
        self._val2 = arg2
        self._update_val()

    def _update_val(self):
        self.val = self._val1 + self._val2

    def display_val(self):
        print(f'My value is {self.val}')

the_instance = MyClass(1, 2)
the_instance.display_val()
```



```
class MyClass {  
    int val;  
    int val1;  
    int val2;  
  
    public MyClass(int arg1, int arg2) {  
        val1 = arg1;  
        val2 = arg2;  
        update_val();  
    }  
  
    public void update_val() {  
        val = val1 + val2;  
    }  
}
```

```
public void display_val() {  
    System.out.println("My value is " + val);  
}  
  
public static void main(String args[]) {  
    MyClass the_instance = new MyClass(1, 2);  
    the_instance.display_val();  
}  
}
```

```
#include <iostream>
```

```
class MyClass {
```

```
private:
```

```
    int val_;
```

```
    int val1_;
```

```
    int val2_;
```

```
public:
```

```
    MyClass(int arg1, int arg2)
```

```
        : val1_(arg1)
```

```
        , val2_(arg2)
```

```
    {
```

```
        update_val();
```

```
    }
```

```
void update_val() {  
    val_ = val1_ + val2_;  
}  
  
void display_val() const {  
    std::cout << "My value is " << val_ << std::endl;  
}  
};  
  
int main() {  
    MyClass* the_instance = new MyClass(1, 2);  
    the_instance->display_val();  
    delete the_instance;  
    return 0;  
}
```

```
from datetime import datetime
```

```
class Auteur:
```

```
    def __init__(self, nom, prenom):  
        self.nom = nom  
        self.prenom = prenom
```

```
class Oeuvre:
```

```
    def __init__(self, auteur, titre, date_creation):  
        self.auteur = auteur  
        self.titre = titre  
        self.date_creation = date_creation
```

```
    def describe(self):
```

```
print(f'Auteur: {self.auteur.prenom} {self.auteur.nom}')  
print(f'Titre: {self.titre}')
```

```
def age(self):  
    return datetime.now().year - self.date_creation
```

```
class Livre(Oeuvre):  
    def __init__(self, auteur, titre, annee_creation, nb_mots):  
        super().__init__(auteur, titre, annee_creation)  
        self.nb_mots = nb_mots  
  
    def describe(self):  
        super().describe()  
        print(f'Nombre de mots: {self.nb_mots}')
```

```
jules = Auteur('Verne', 'Jules')
vingtk_lieues = Livre(jules, 'Vingt mille lieues sous la mer', 1869, 142172)
vingtk_lieues.describe()
age = vingtk_lieues.age()
print(f'{vingtk_lieues.titre} a été écrit il y a {age} ans.')
```

```
class Test {  
    public static void main(String args[]) {  
        Auteur jules = new Auteur("Verne", "Jules");  
        Oeuvre vingtk_lieues = new Livre(jules,  
                                           "Vingt mille lieues sous la mer",  
                                           1869, 142172);  
  
        vingtk_lieues.describe();  
        int age = vingtk_lieues.age();  
        System.out.println(vingtk_lieues.titre_ + " a été écrit il y a "  
                             + age + " ans.");  
    }  
}  
  
class Auteur {  
    String nom_;  
    String prenom_;
```



```
public Auteur(String nom, String prenom) {  
    nom_ = nom;  
    prenom_ = prenom;  
}  
}  
  
class Oeuvre {  
    Auteur auteur_;  
    String titre_;  
    int annee_creation_;  
  
    public Oeuvre(Auteur auteur, String titre, int annee_creation) {  
        auteur_ = auteur;  
        titre_ = titre;  
        annee_creation_ = annee_creation;  
    }  
}
```

```
public void describe() {
    System.out.println("Auteur: " + auteur_.prenom_ + " " + auteur_.nom_);
    System.out.println("Titre: " + titre_);
}

public int age() {
    return 2021 - annee_creation_;
}
}

class Livre extends Oeuvre {
    int nb_mots_;

    public Livre(Auteur auteur, String titre, int annee_creation, int nb_mots) {
        super(auteur, titre, annee_creation);
        nb_mots_ = nb_mots;
    }
}
```

```
public void describe() {  
    super.describe();  
    System.out.println("Nombre de mots: " + nb_mots_);  
}  
}
```

```
#include <iostream>
#include <string>

class Auteur {
private:
    std::string nom_;
    std::string prenom_;

public:
    Auteur(std::string nom, std::string prenom)
        : nom_(nom)
        , prenom_(prenom)
    {}
    std::string nom() const {return nom_;}
    std::string prenom() const {return prenom_;}
};
```

```
class Oeuvre {  
  
private:  
    Auteur* auteur_;  
    std::string titre_;  
    int annee_creation_;  
  
public:  
    Oeuvre(Auteur* auteur, std::string titre, int annee_creation)  
        : auteur_(auteur)  
        , titre_(titre)  
        , annee_creation_(annee_creation)  
    {}  
  
    virtual void describe() const {  
        std::cout << "Auteur: " << auteur_->prenom() << " "  

```

```

        << auteur_->nom() << std::endl;
    std::cout << "Titre: " << titre_ << std::endl;
}

int age() const {
    return 2021 - annee_creation_;
}

std::string titre() const {
    return titre_;
}
};

class Livre : public Oeuvre {
private:
    int nb_mots_;
public:

```

```

Livre(Auteur* auteur, std::string titre, int annee_creation, int nb_mots)
    : Oeuvre(auteur, titre, annee_creation)
    , nb_mots_(nb_mots)
{}

void describe() const {
    Oeuvre::describe();
    std::cout << "Nombre de mots: " << nb_mots_ << std::endl;
}

};

int main() {
    Auteur* jules = new Auteur("Verne", "Jules");
    Oeuvre* vingtk_lieues = new Livre(jules,
                                     "Vingt mille lieues sous la mer",
                                     1869, 142172);

    vingtk_lieues->describe();
}

```

```
int age = vingtk_lieues->age();  
std::cout << vingtk_lieues->titre() << " a été écrit il y a "  
          << age << " ans." << std::endl;  
return 0;  
}
```