# Bases de la programmation

Licence Professionnelle Jeux Vidéo

*Paris 13*


*Simon Chauvin*

*chauvin.simon@gmail.com*

2012 - 2013

# Présentation



- Initiation 20h :
  - Bases de la programmation
  - Initiation à la programmation en AS3
  - Initiation à Flixel
  - Réalisation d'un petit prototype de jeu avec Flixel
  - *Évaluation :* devoir surveillé d'une heure
- Spécialisation 30h :
  - Programmation avancée avec Flixel
  - Soutien pour le projet de fin d'année
  - *Évaluation :* rendu de TP

# Présentation



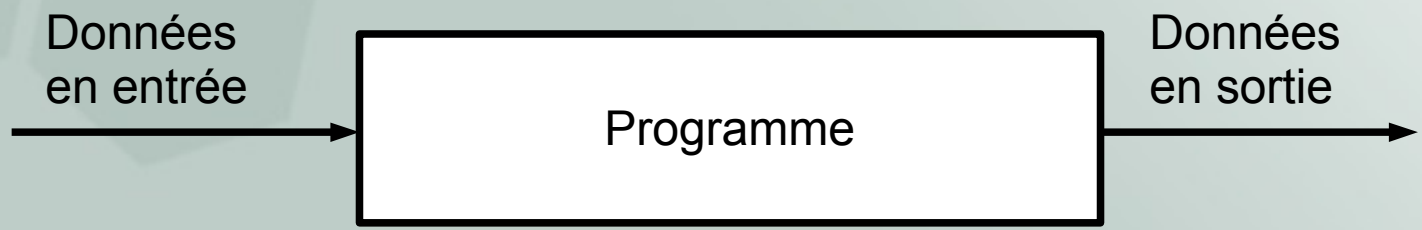
- Programmation :
  - *Principes, objectifs, description*
- Programmation orientée objet :
  - *Concepts, objectifs, description*

# Programmation

# Programmation



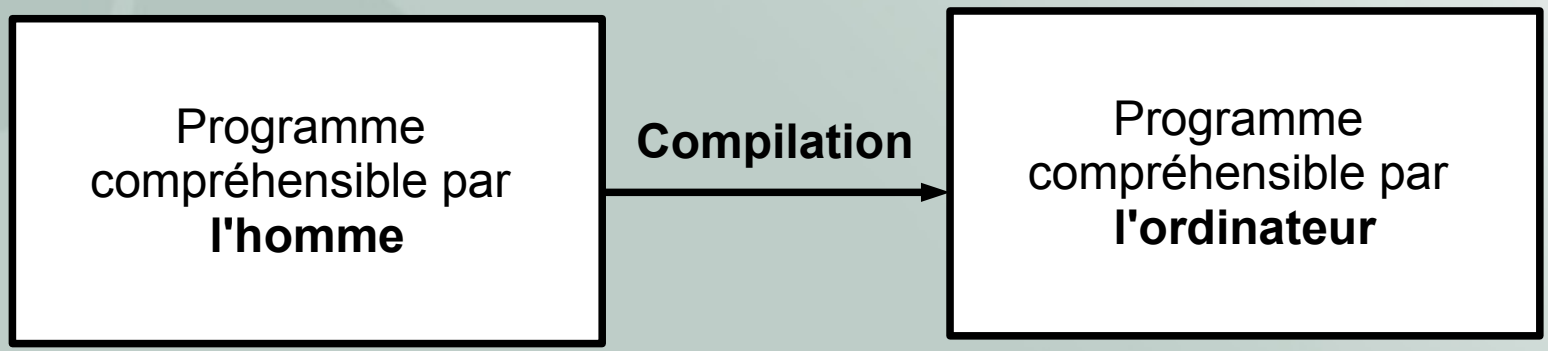
- La programmation :
  - Consiste à créer des **programmes**
  - Les programmes **automatisent** des processus de traitement de données
  - Ils sont **écrits** dans un certain langage
  - Ils sont **compréhensibles** par l'homme
  - Et peuvent être **traduits** dans un langage compréhensible par l'ordinateur
  - La programmation permet de donner des **ordres complexes** à l'ordinateur



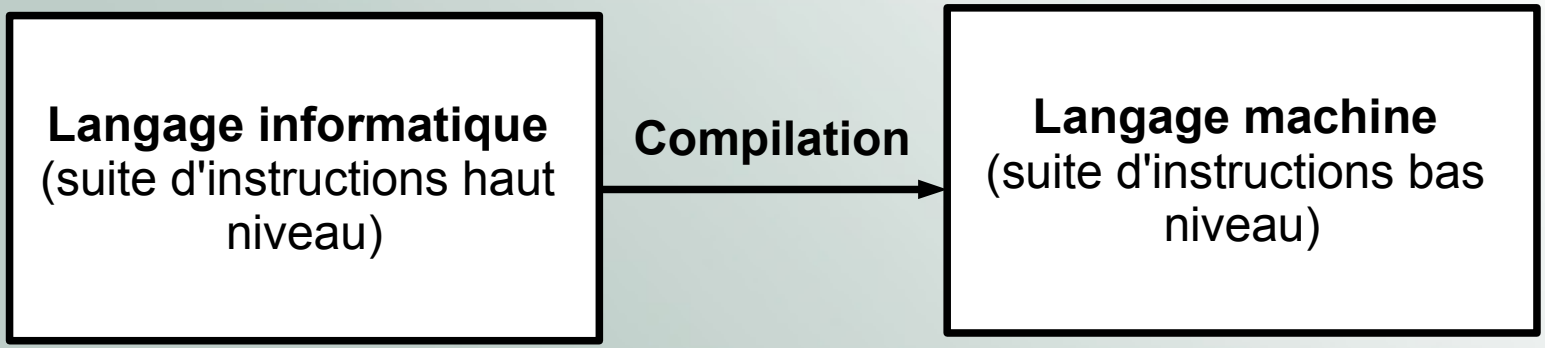
# Programmation



- Un programme :
  - Un programme est écrit dans un **langage** particulier :
    - C, C++, C#, Java, JavaScript, ActionScript, Python, etc.
  - Un programme utilise les **instructions** fournis par le langage dans lequel il est écrit :
    - Conditions, boucles, déclaration de variables, fonctions, classes
  - Un programme est **compilé** pour que l'ordinateur puisse le comprendre et l'exécuter

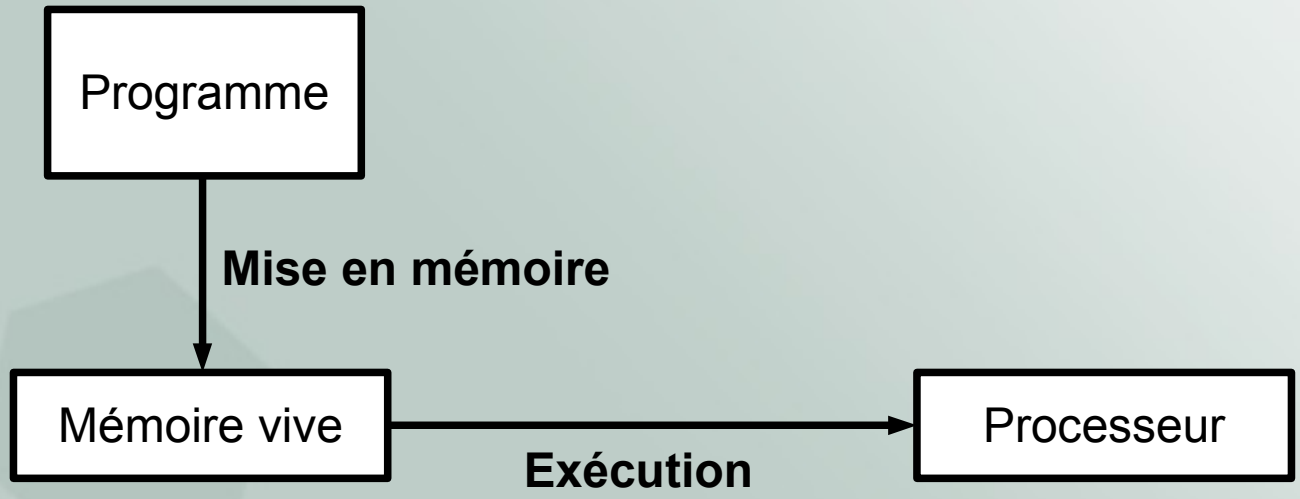


- La compilation :
  - Processus par lequel un programme est **traduit** dans un langage compréhensible par l'ordinateur




  - Pour que la compilation fonctionne un langage informatique **impose une syntaxe** qui doit être respectée sous peine de ne pas pouvoir être compilé

- Exécution d'un programme :
  - Mise en mémoire des **données** du programme
  - Mise en mémoire des **instructions** bas niveau
  - **Exécution** des instructions par le processeur sur les données en mémoire



- Exécution d'un programme :
  - La quantité et la **complexité** des **instructions** influent sur la vitesse **d'exécution**
  - La quantité et la **structure** des **données** influent sur la **quantité** de mémoire nécessaire au programme
  - Les **jeux** sont extrêmement **gourmands** en **mémoire** :
    - Assets graphiques et sonores
  - Et très gourmands en **processeur** également :
    - Temps réel, affichage, animations, collisions, etc.

# Programmation



- Un programme est donc constitué de :
  - **Données** et **d'instructions**
- Un programme est écrit dans un certain **langage**
- Un programme doit être **compilé** pour pouvoir être **exécuté** par l'ordinateur
- Un programme utilise des **ressources** mémoires et processeurs

- Les données :
  - Sont **variables** ou **constantes**
  - Elles peuvent être **simples** :
    - Entiers, réels, booléens
  - Ou **complexes** :
    - Tableaux, chaînes de caractères, objets, etc.
  - Elles peuvent subir des **opérations** :
    - Comparaison, addition, soustraction, etc.

- Les données simples :
  - *Entiers, réels :* permettent de **stocker** des **informations simples** (score, vie, munitions, etc.)
  - *Booléens :* permettent de **stocker** un **état** (si un bouton est enfoncé ou non, etc.)
- Les données complexes :
  - *Tableaux :* permettent de **stocker** des **ensembles** de données (les ennemis vivants, les bonus du niveau, les armes du joueur, etc.)
  - *Chaînes de caractères :* nom du joueur, nom du niveau, dialogues, etc.
  - *Objets :* permettent de **représenter** et **manipuler** des **entités très complexes** (joueur, ennemis, etc.) composés de données simples et complexes

- Les Tableaux :
  - Un tableau :
    - tableau = [0, 8, 1, 9, 10, 15, 7], longueur de 7, indices allant de 0 à 6
  - Éléments du tableau :
    - tableau[4] retourne 10
    - tableau[0] retourne 0
    - tableau[6] retourne 7

- Opérations :
  - Sur les entiers et réels :
    - Affectations, additions, soustractions, etc.
    - Comparaisons
  - Sur les booléens :
    - Affectations, comparaisons
  - Sur les chaînes de caractères :
    - Affectations, comparaisons
  - Sur les tableaux :
    - Affectations, ajout/suppression d'éléments
  - Sur les objets :
    - Affectations, opérations spéciales définies par l'objet

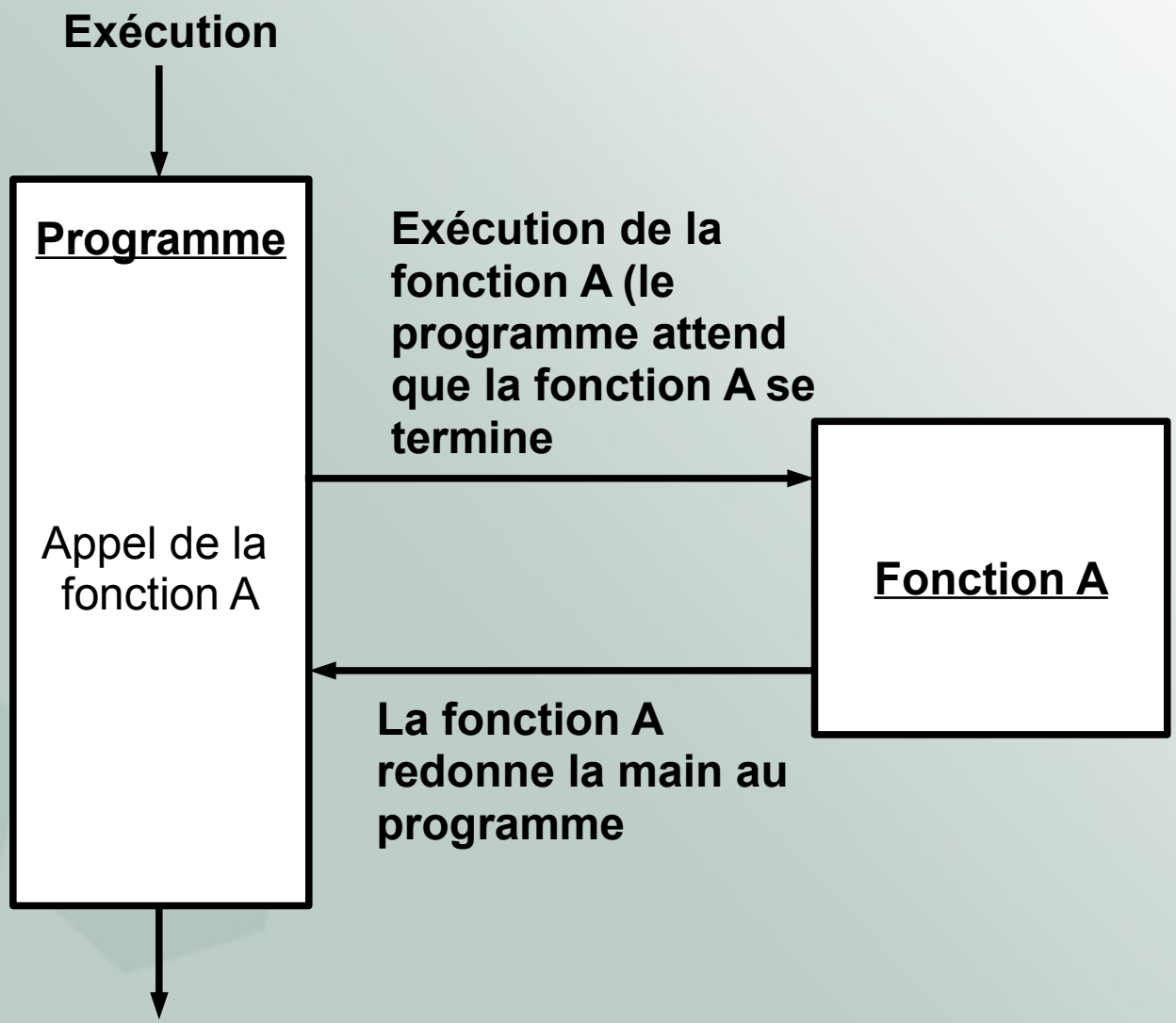
- Les instructions :
  - Sont **propres** au langage de programmation utilisé
  - Mais la plupart des langages utilisent les mêmes **instructions** :
    - Conditions, boucles, déclarations de variables, constantes, fonctions, etc.
  - Permettent de **construire** le programme et de **manipuler** les données :
    - Diminuer la vie du joueur, passer au niveau suivant, savoir si un bouton a été enclenché, etc.

- Les instructions :
  - Déclarations de variables :
    - Permet de **créer** une **variable** d'un certain type de données (entier, réel, etc.)
  - Déclarations de constantes :
    - Permet de **créer** une **constante** d'un certain type de données (entier, réel, etc.)
  - Une fois **déclarée** une **variable** peut être **manipulée** :
    - Initialisée, modifiée, additionnée, soustraite, comparée, etc.
  - Une fois **déclarée** une **constante** peut être **manipulée** :
    - Comparée, additionnée, soustraite, etc.

- Les instructions :
  - Déclarations de fonctions :
    - Une **fonction** permet de **rassembler** un ensemble d'**instructions** répondant à un **objectif précis** (augmenter le nombre de munitions du joueur, changer l'arme principale, diminuer la vie de l'adversaire, etc.)
    - Une fonction peut **renvoyer** des données en sortie et/ou prendre des **données** en entrée
    - C'est une manière simple et élégante de **structurer** un programme (plus lisible et modifiable par la suite)

- Les fonctions :
  - Une fonction doit être **appelée** pour s'exécuter
  - Une fonction peut nécessiter des **arguments**
  - Une fonction peut **renvoyer** des données
  - *Exemple :* augmenterScore(10) appel la fonction augmenterScore(entier) qui nécessite une certaine valeur en argument
  - *Exemple :* calculerDegats():entier appel la fonction calculerDegats qui doit renvoyer la valeur calculée des dégâts à infliger

- Les fonctions



- Les fonctions :
  - L'appel :

```
fonction addition (a, b) {
   retourner a + b ;
}
-----------------------------
resultat = addition(10, 5);
afficher (resultat);
-----------------------------
// Affiche 15
```

# Programmation



- Les instructions :
  - Conditionnelles :
    - Si, sinon (if ... else)
    - Permettent de **tester** des **valeurs** afin de savoir si l'on se trouve dans tel ou tel cas et éventuellement opérer un changement
    - *Exemple :* si le joueur n'a plus de vie alors déclencher game over
  - Boucles :
    - Tant que (while)
    - De … jusqu'à (for)
    - Permettent de **parcourir** des tableaux, d'exécuter une action un certain nombre de fois
    - *Exemple :* pour chaque ennemi à l'écran déclenchez son animation de marche

# Programmation orientée objet

# Programmation orientée objet

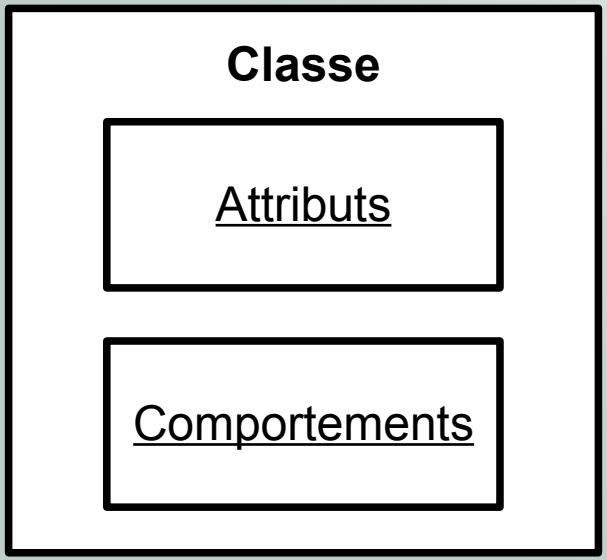


- Programmation objet :
  - **Structuration** des données en objets, intuitif et modulable
  - Les classes permettent de **créer** des **objets**
  - Une classe permet de définir des **attributs** et **comportements**
  - Une classe peut **hériter** des attributs et comportements d'une autre classe
  - Une classe peut être **associée** à une autre classe
  - Une classe **encapsule** l'information

# Programmation orientée objet



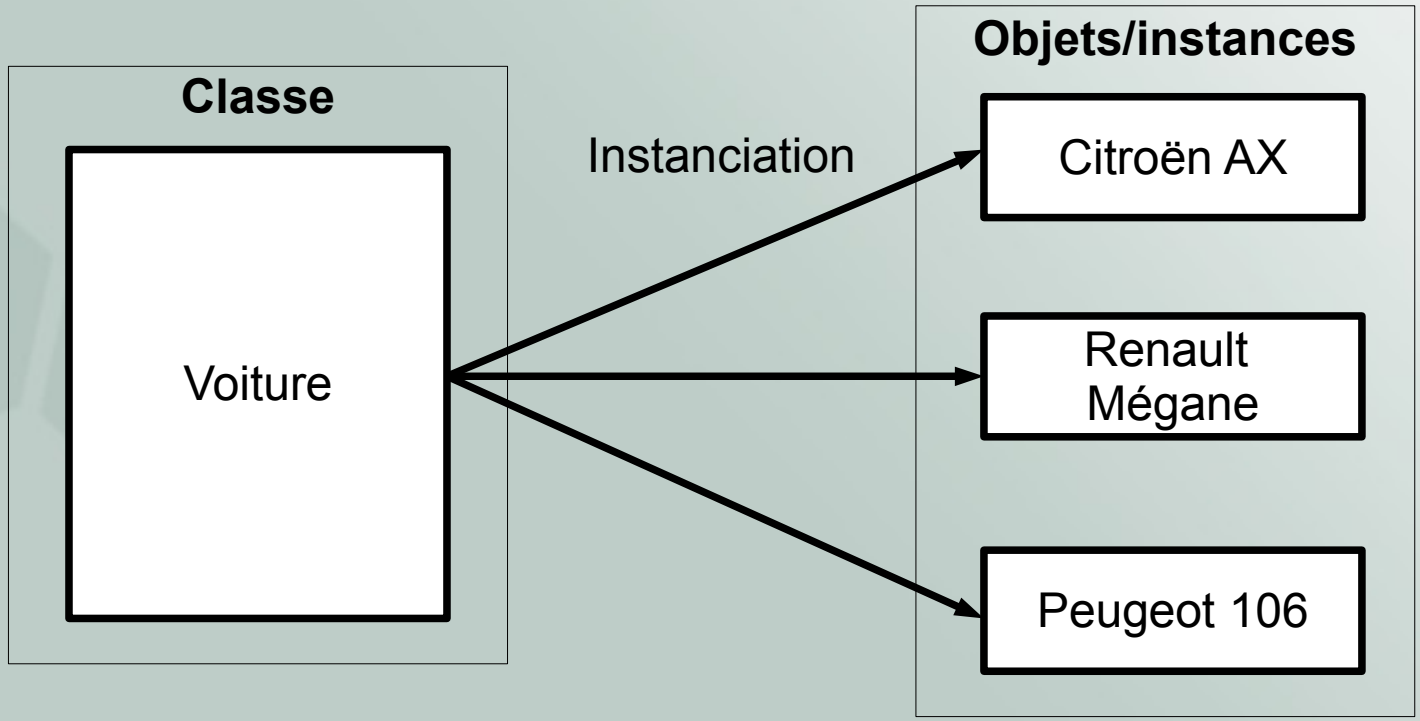
- Principe :
  - Boite noire fournissant un **modèle** pour un type d'objet particulier, composé **d'attributs** et de **comportements**
  - Pas besoin de connaître le fonctionnement interne pour l'utiliser
  - Les comportements disponibles suffisent



# Programmation orientée objet



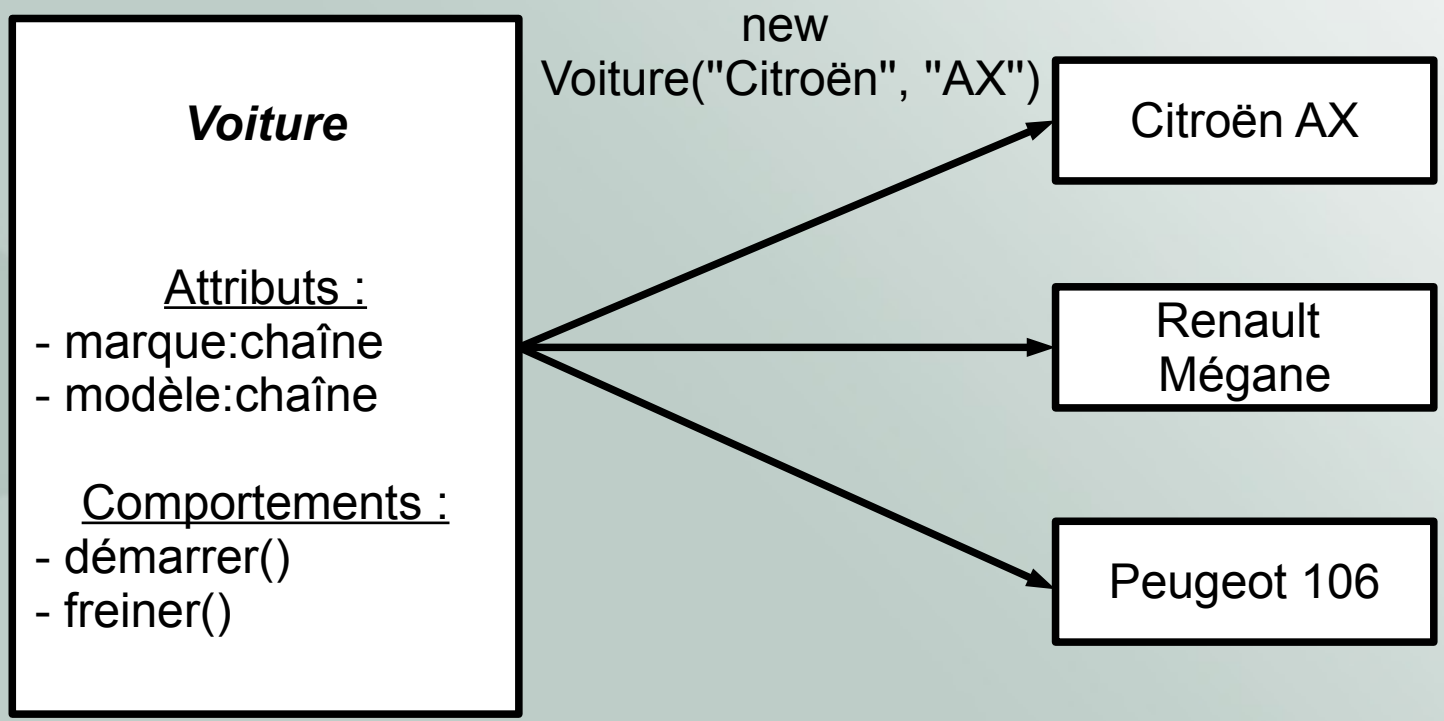
- Classe et objet :
  - *Un moule :* la classe
  - *Une instance :* l'objet
  - Une classe permet de **créer** des **objets** avec des **caractéristiques** différentes
  - Une classe Voiture permet de **générer** des centaines de **modèles** de voitures différents



# Programmation orientée objet



- Classe et objet :
  - *Attributs :* les **caractéristiques** des instances à créer
  - *Comportements :* les **actions** possibles sur un objet
  - *Constructeur :* permet **d'instancier** la classe



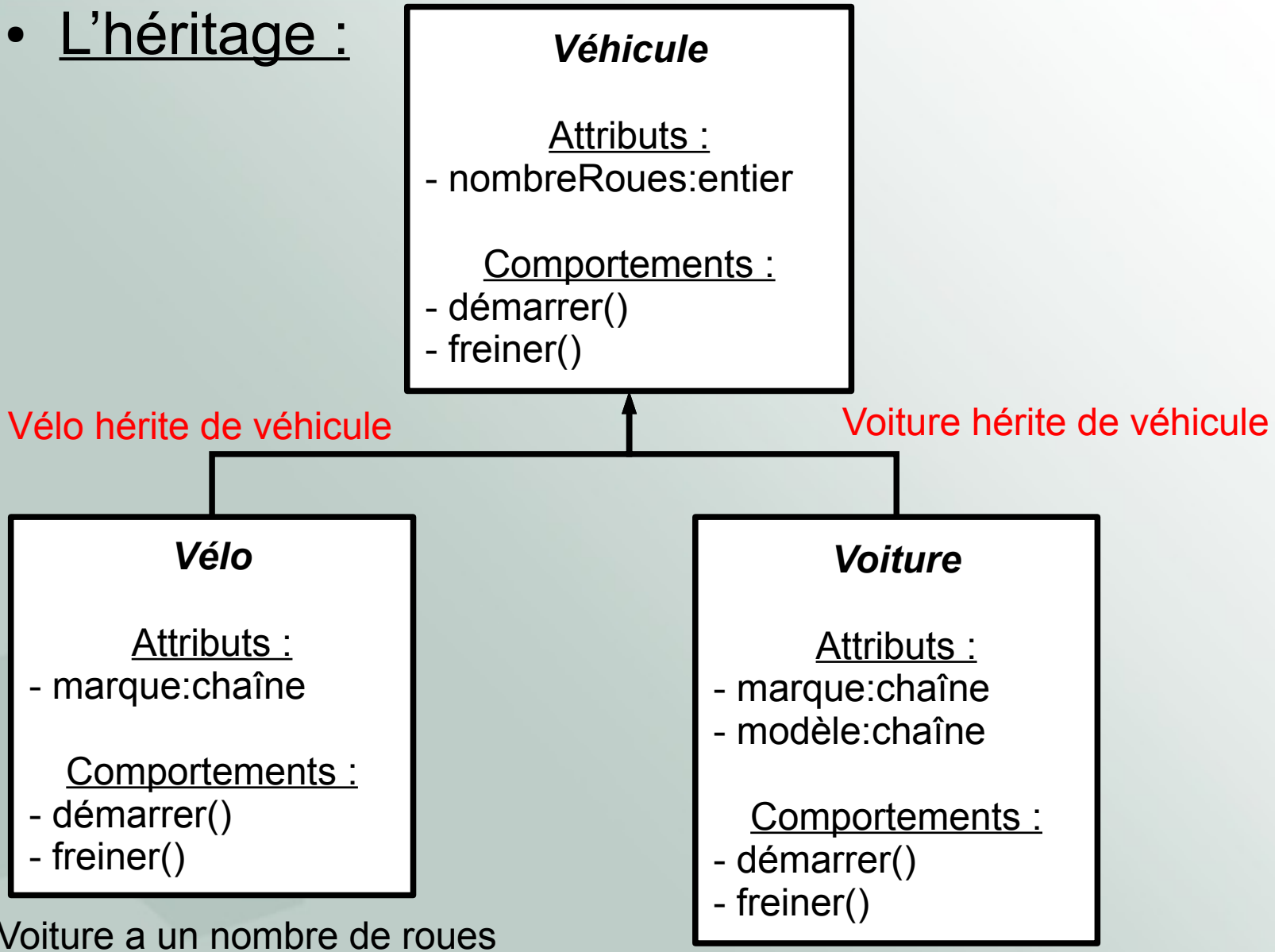
- Classe et objet :
  - Une **classe** est une **structure**
  - **L'objet** (ou **l'instance**) est l'élément présent en **mémoire** que le programme va effectivement **manipulé**
  - Une classe non instanciée ne sert à rien
  - **L'accès** aux **comportements** et aux **attributs** d'un objet ne peut se faire que si celui-ci à été **instancié**

- L'héritage :
  - Une classe **hérite** des **attributs** et **comportements** d'une classe mère
  - Elle peut **accéder** ses éléments **héritées**
  - L'héritage se résume à une relation « **est un** »
  - *Exemple :* voiture hérite de véhicule, une voiture est donc un véhicule mais véhicule n'est pas nécessairement une voiture

# Programmation orientée objet



- L'héritage :

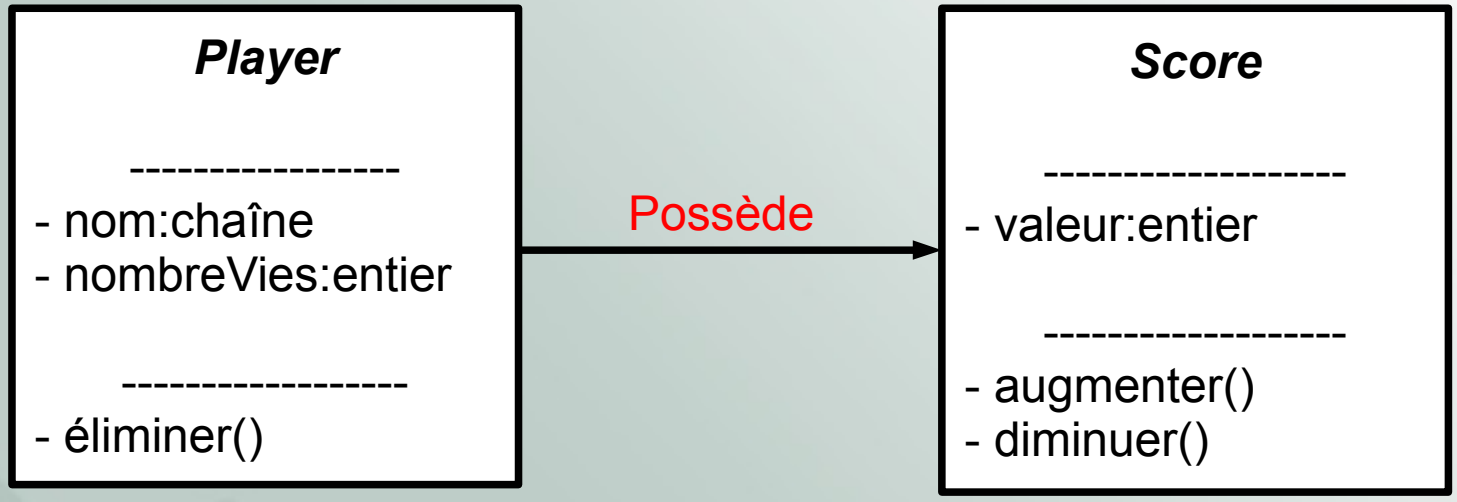



Voiture a un nombre de roues particulier et redéfinie les comportements démarrer et freiner

Voiture a un nombre de roues particulier et redéfinie les comportements démarrer et freiner

- L'héritage :
  - Dans le cas précédent Vélo et Voiture **redéfinissent** les **comportements** « démarrer() » et « freiner() »
  - Il est en effet possible de **redéfinir** des **comportements** afin de les **spécialiser**
    - Une voiture ne démarre pas de la même façon qu'un vélo

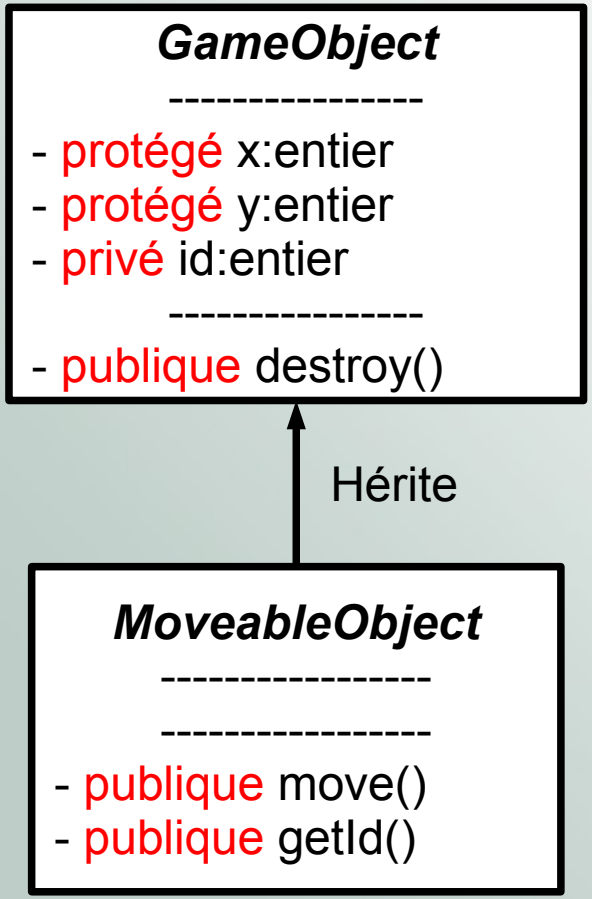
- L'association :
  - Permet **d'associer** une classe B à une classe A
  - La classe A peut alors **accéder** aux **données** et **comportements** de la classe B




  - Ici, la classe Player peut diminuer et augmenter l'attribut valeur de la classe Score
  - L'association permet donc de **distribuer la logique du jeu** dans différentes classes pour une meilleure **lisibilité** et **modularité**

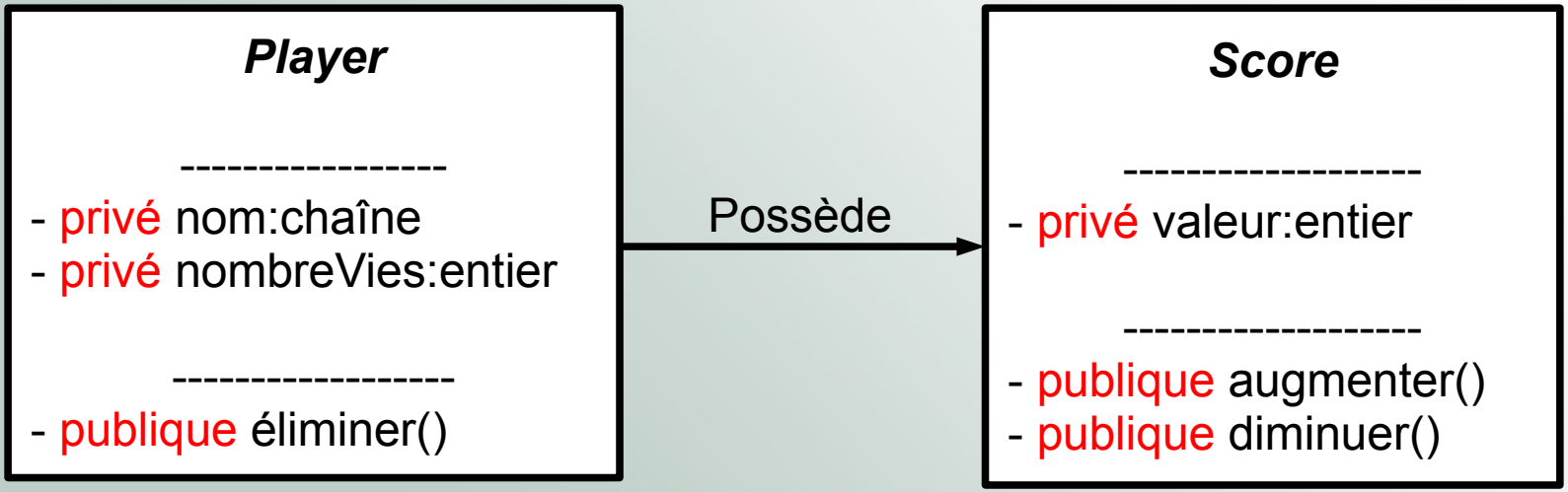
- L'encapsulation :
  - Il est possible **d'accéder**, ou non, à certains attributs et comportements d'une classe
    - *Publique :* visible et accessible de **tous** (utilisé pour les constantes et les comportements)
    - *Privé :* visible de la **classe seule** (utilisé pour les attributs)
    - *Protégé :* visible de la classe et de ses **sous classes** (ainsi que du package) (utilisé pour les attributs dont les sous classes doivent hériter)

- L'encapsulation :




- MoveableObject n'a pas accès à l'attribut id mais a accès aux attributs x et y et aux fonctions destroy() et getId()

- L'encapsulation :




- La classe Player ne peut pas accéder à l'attribut valeur de la classe Score mais peut en revanche accéder aux méthodes diminuer() et augmenter() qui permettent de changer l'attribut valeur

- Exercice :
  - Créer les classes (attributs et méthodes) que vous pensez être utiles pour développer un Tetris

# Programmation orientée objet



- Exercice :

**Game**

privé player:Player
privé blocks:Tableau

publique start()
publique gameOver()
publique lineCompleted()

**Player**

privé name:chaîne
privé score:Score

publique getName():chaîne

**Score**

privé valeur:entier

publique increaseScore()
publique decreaseScore()
publique getScore():entier

**Block**

privé color:entier
privé velocity:entier

publique moveLeft()
publique moveRight()
publique moveDown()
publique rotateLeft()
publique rotateRight()

- Design et programmation OO :
  - La **séparation** attributs/comportements facilite les **réglages** gameplay par **modification** de **constantes**
  - Pensez à utiliser des **constantes** pour :
    - Vitesse des objets mobiles (avatar, ennemis, etc.)
    - Hauteur, vitesse, longueur de saut
    - La vie maximum (ennemis, de l'avatar, etc.)
    - Les dégâts (armes, etc.)