



DÉVELOPPEMENT DIGITAL

PROGRAMMER EN ORIENTÉ OBJET

```
1 'use strict';
2
3 let mongoose = require('libs/mongoose');
4 let ValidationError = mongoose.Error.ValidationError;
5
6 let imagick = require('libs/imagick');
7
8 let FileModel = require('../models/file_model');
9 let EnumErrors = require('../core/error').EnumErr;
10 let ResultError = require('../core/error').Result;
11
12 function validationErrorToResult(error)
13 {
14   for (let errorMessage in error.errors)
15   {
16     let valError = error.errors[errorMessage];
17     if (valError.kind === 'unique')
18     {
19       return new ResultError(EnumErrors,
20     }
21
22     let enumError = valError.properties;
23     return (enumError) ? new ResultError(
24       : valError;
25   }
26 }
27
28 class File
29 {
30
31   static create(ownerId, oldName, name, path)
32   {
33     let fileModel = null;
34
```





OBJECTIF

A l'issue de ce module de compétence, vous serez capable de programmer en Orientée Objet (POO)

90 heures



PARTIE 1

Appréhender le paradigme de la POO

Chapitre 1 : Introduire la POO

Chapitre 2 : Définir un objet

PARTIE 2

Connaître les principaux piliers de la POO

Chapitre 1 : Définir l'héritage et le polymorphisme

Chapitre 2 : Connaître l'encapsulation

Chapitre 3 : Caractériser l'abstraction

PARTIE 3

Appliquer la POO dans le langage de programmation Python

Chapitre 1 : Présenter l'environnement de programmation Python 3

Chapitre 2 : Implémenter une classe

Chapitre 3 : Administrer les exceptions

Chapitre 4 : Gérer les interfaces graphiques

PARTIE 1

APPRÉHENDER LE PARADIGME DE LA POO



CHAPITRE 1

INTRODUIRE LA POO

1 - Programmation procédurale

2 - De la programmation procédurale à la POO

3 - Avantages de la POO

4 - Historique de la POO

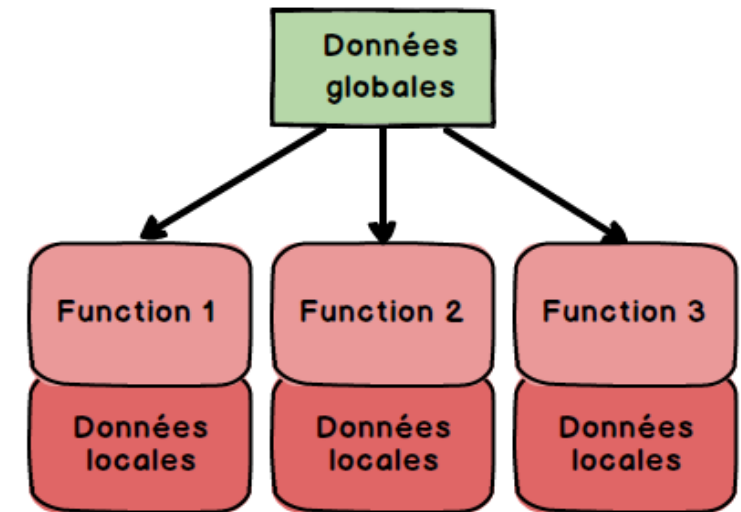
Programmation procédurale



Dans la **programmation procédurale**, le programme est divisé en petites parties appelées **procédures** ou **fonctions**.

- Ensuite, pour résoudre chaque partie, une ou plusieurs procédures/fonctions sont utilisées
- Dans la **programmation procédurale**, les notions de données et de traitement de ces données sont **séparées**

Programmes = Procédures/Fonctions + Données



Exemple d'algorithme



Algorithme CalculSomme100PremierNombre

Variable

I,S: entier

Procédure Somme

Début /*début de la procédure*/

S ← 0

Pour i ← 1 à 100 Faire

 S ← S+1

FinPour

Ecrire(" La somme des 100 premiers nombres est ",S);

Fin /*Fin de la procédure*/

Début /*début algorithme*/

Somme

Fin /*fin algorithme*/

Programmation procédurale



Dans la **programmation procédurale**:

- Les données constituent **la partie passive du programme**.
- Les procédures et les fonctions **constituent la partie active**.

Programmer dans ce cas revenait à:

- définir un certain nombre de variables (entier, chaîne de caractères, structures, tableaux, etc.)
- écrire des procédures pour les manipuler.

Inconvénients:

- Difficulté de réutilisation du code
- Difficulté de la maintenance de grandes applications



CHAPITRE 1

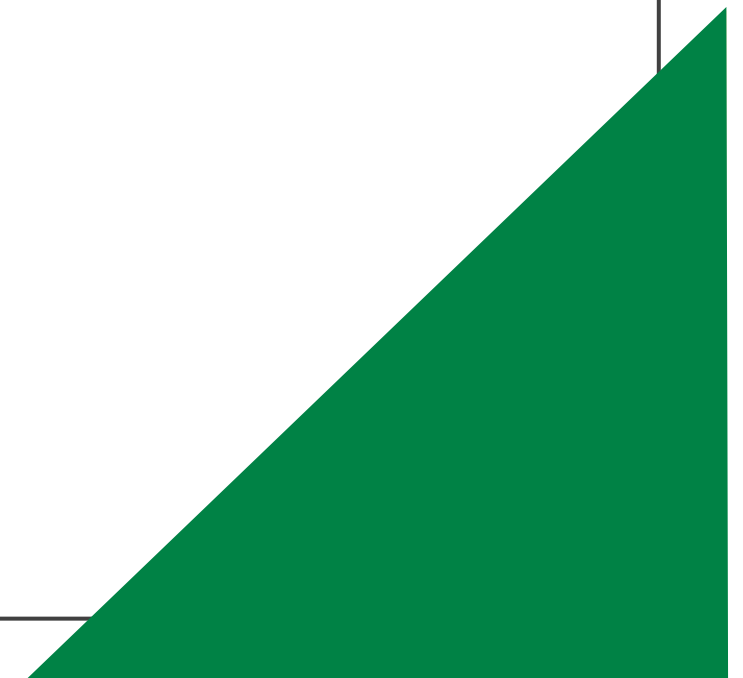
INTRODUIRE LA POO

1 - Programmation procédurale

2 - De la programmation procédurale à la POO

3 - Avantages de la POO

4 - Historique de la POO



Programmation orientée objet



- Séparation (données, procédures) est-elle utile ?
- Pourquoi privilégier les procédures sur les données ?
- Pourquoi ne pas considérer que les programmes sont avant tout des ensembles objets informatiques caractérisé par les opérations qu'ils connaissent?

Les **langages orientés objets** sont nés pour répondre à ces questions.

Ils sont fondés sur la connaissance d'une seule catégorie d'entités informatiques : **LES OBJETS**

Programmation orientée objet



- Un **objet** incorpore des aspects **statiques** et **dynamiques** au sein **d'une même notion**
- Un programme est constitué d'un ensemble d'objets chacun disposant d'une partie procédures (traitement) et d'une partie données.

Programmation orientée objet



Que doit faire le système ?

Programmation procédurale



Sur quoi doit-il le faire?

De quoi doit être composé mon programme?

Programmation Orientée Objet

Appréhender le paradigme de la POO

Connaître les principaux piliers de la POO

La POO en Python

Programmation orientée objet



Créditer un
compte?
Débiter un
compte?

Programmation procédurale

**Programme de Gestion des
comptes bancaires**



Comment se
présente (structure)
d'un compte
bancaire ?

Programmation Orientée Objet

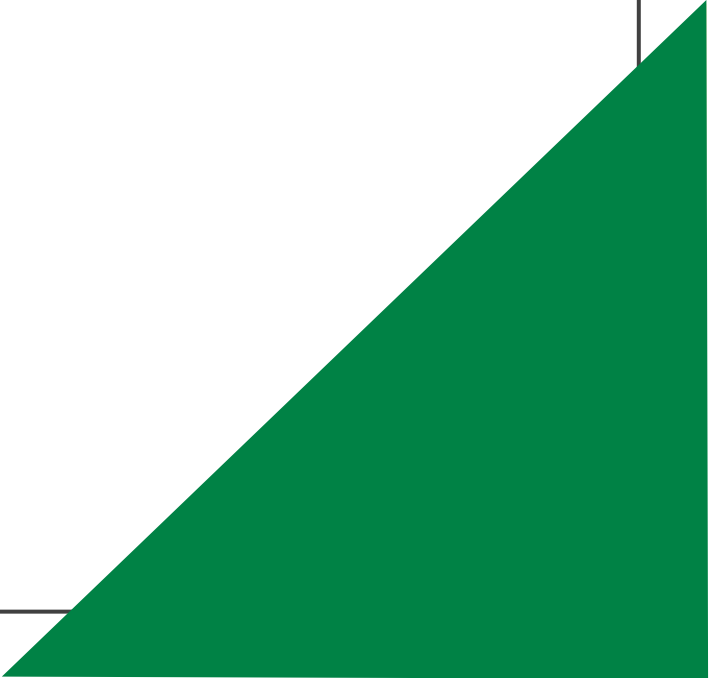
Appréhender le paradigme de
la POO

Connaître les principaux
piliers de la POO

La POO en Python

CHAPITRE 1

INTRODUIRE LA POO

- 1 - Programmation procédurale
 - 2 - De la programmation procédurale à la POO
 - 3 - Avantages de la POO**
 - 4 - Historique de la POO
- 

Avantages de la POO



Motivation : concevoir, maintenir et exploiter facilement de gros logiciels

Avantages:

- **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)
- **Abstraction** :
 - Les entités objets de la POO sont proches de celles du monde réel (objet est un compte, stagiaire, formateur, etc.).
 - La POO se base sur un processus qui consiste à masquer des détails non pertinents à l'utilisateur.

Exemple: Pour envoyer un SMS, vous tapez simplement le message, sélectionnez le contact et cliquez sur Envoyer → Ce qui se passe réellement en arrière-plan est masqué car il n'est pas pertinent à vous.

Motivation et avantage de la POO



Réutilisabilité:

- La POO favorise la réutilisation de composants logiciels et même d'architectures complexes
- La définition d'une relation d'héritage entre les entités logicielles évite la duplication de code
 - Facilité de la **maintenance logicielle** et amélioration de la **productivité**

CHAPITRE 1

INTRODUIRE LA POO

- 1 - Programmation procédurale
- 2 - De la programmation procédurale à la POO
- 3 - Avantages de la POO
- 4 - Historique de la POO**

Historique évolution de la POO



- Les concepts de la POO naissent au cours des années 1970 dans des laboratoires de recherche en informatique.
- Les premiers langages de programmation véritablement orientés objet ont été **Simula**, puis **Smalltalk**.
 - **Simula** (1966) regroupe données et procédures.
 - **Simula I** (1972) formalise les concepts d'objet et de classe. Un programme est constitué d' une collection d'objets actifs et autonomes.
 - **Smalltalk** (1972) : généralisation de la notion d'objet.

Historique évolution de la POO



- A partir des années 80, les principes de la POO sont appliqués dans de nombreux langages
 - **Eiffel** créé par le Français Bertrand Meyer
 - **C++** est une extension objet du langage C créé par le Danois Bjarne Stroustrup
 - **Objective C** une autre extension objet du C utilisé, entre autres, par l'iOS d'Apple
- Les années 1990 ont vu l'avènement des PPOs dans de nombreux secteurs du développement logiciel, et la création du langage **Java** par la société Sun Microsystems
- De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés tels que **PHP** (à partir de la version 5), **VB.NET**, **PowerShell**, **Python**, etc.

CHAPITRE 2

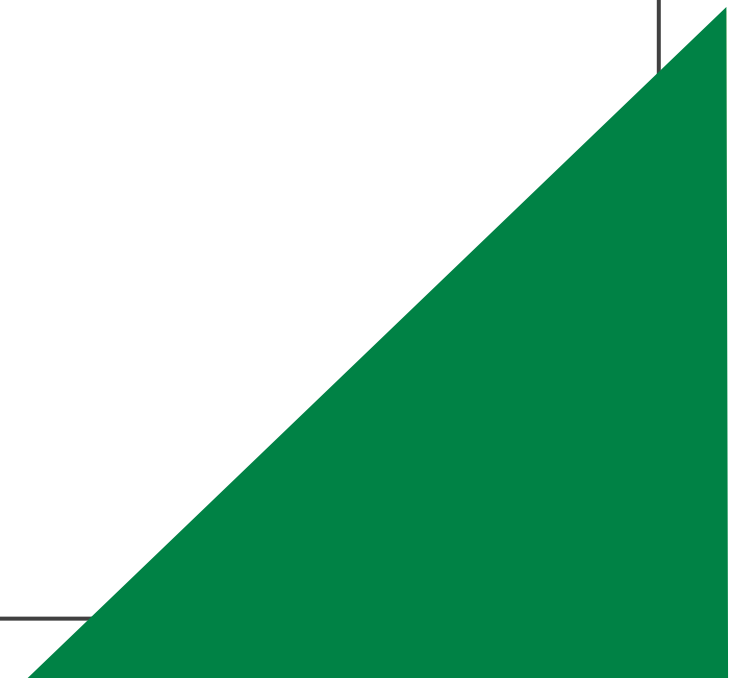
DÉFINIR UN OBJET

1 - Notion d'objet

2 - Notion de classe

3 – Constructeur/Destructeur

4 – Objets en interaction

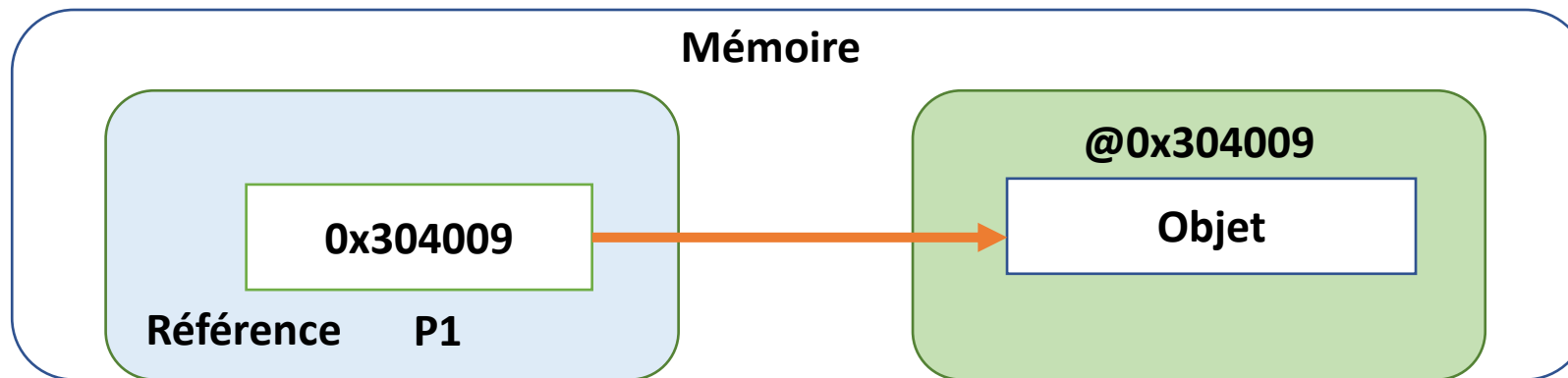


Notion d'objet



OBJET= Référent + Etat + Comportement

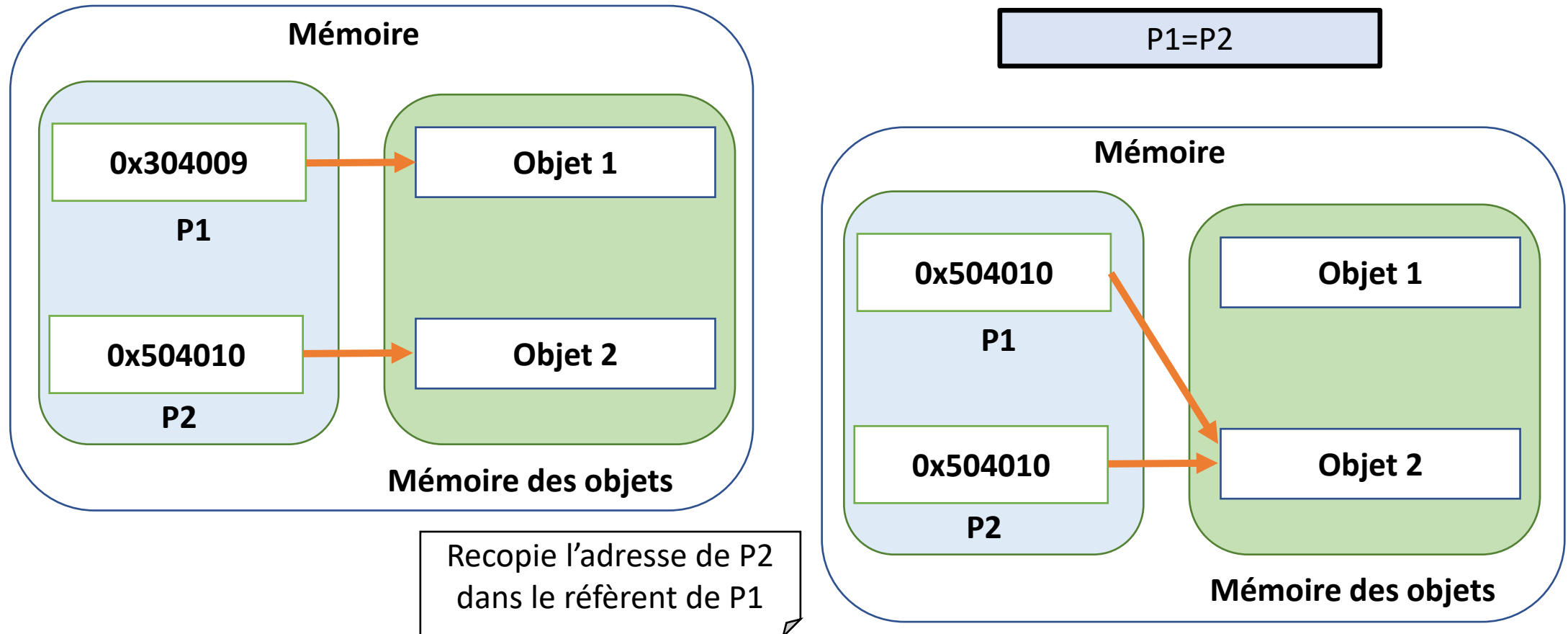
- Chaque objet doit avoir un nom (référence) « qui lui est propre » pour l'identifier
- La référence permet d'accéder à l'objet, mais n'est pas l'objet lui-même. Elle contient l'adresse de l'emplacement mémoire dans lequel est stocké l'objet.



Notion d'objet



- Plusieurs référents peuvent référer un même objet → **Adressage indirect**



Notion d'objet

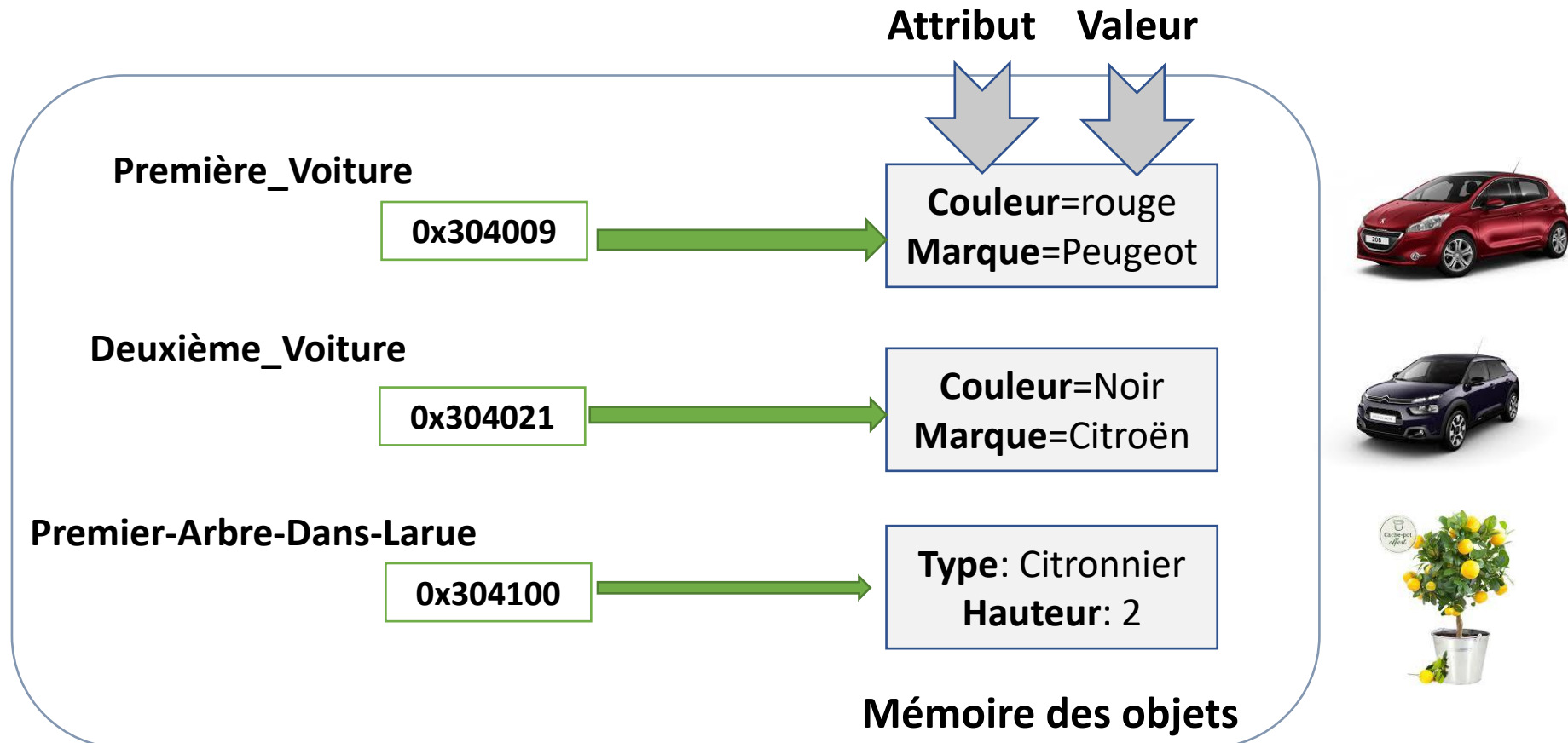


- Un objet est capable de sauvegarder un état c'est-à-dire un ensemble d'information dans des variables internes (**attributs**)
- **Attributs** sont l'ensemble des informations permettant de représenter l'état de l'objet.

Notion d'objet



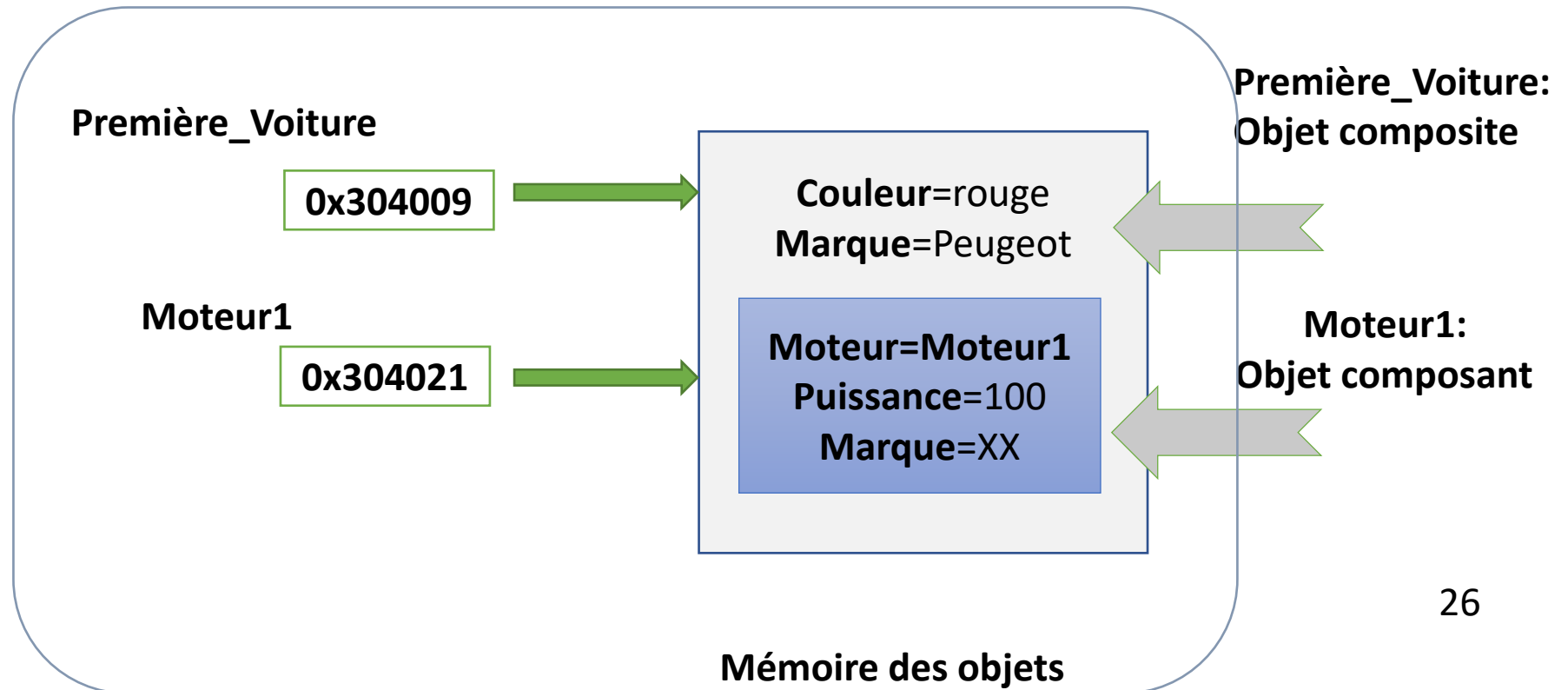
Exemple d'objets où chaque attribut est d'un type dit « primitif » ou « prédéfini », comme entier, réel, caractère...



Notion d'objet



Un objet stocké en mémoire peut être placé à l'intérieur de l'espace mémoire réservé à un autre. Il s'agit d'un **Objet Composite**

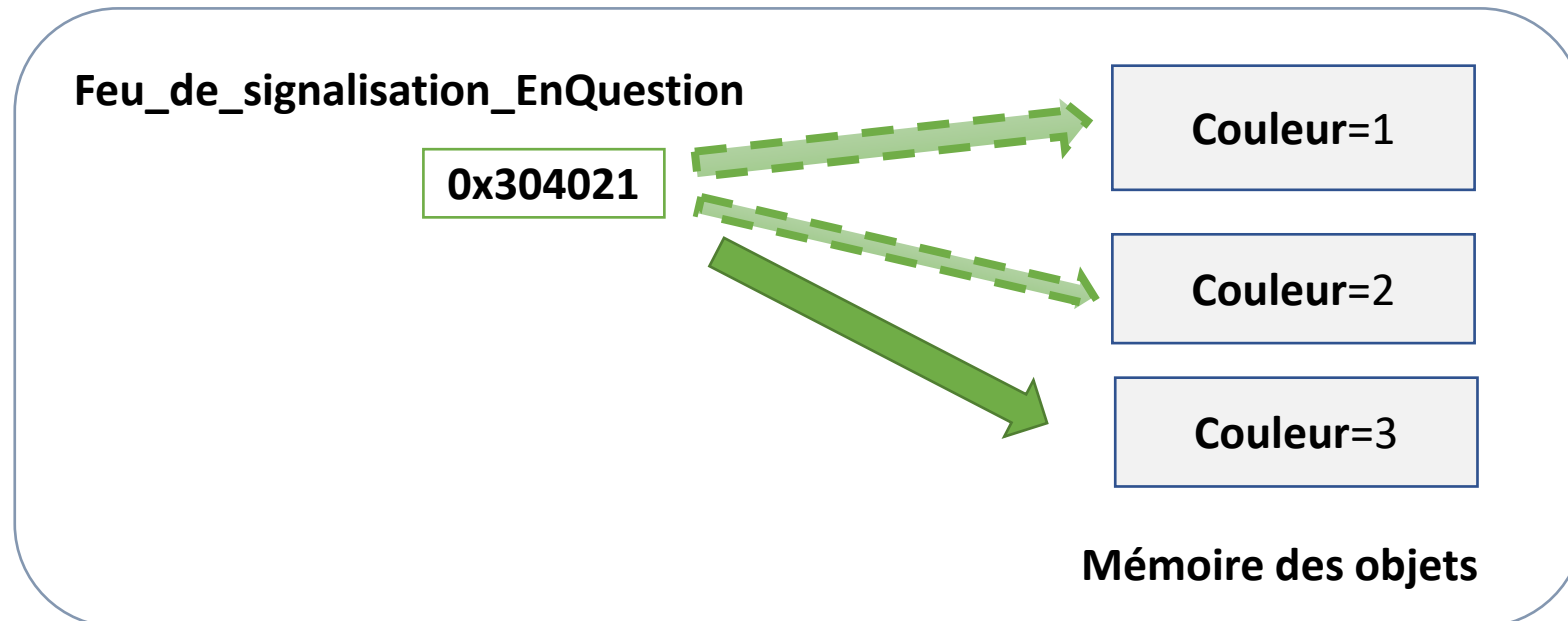


26

Notion d'objet



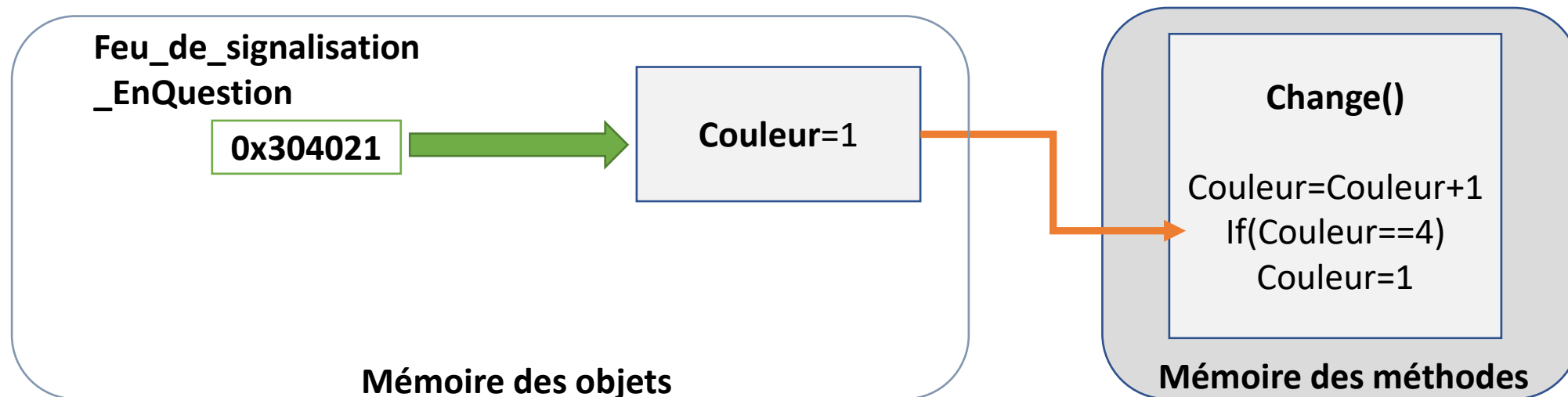
- Les objets changent d'état
- Le cycle de vie d'un objet se limite à une succession de changements d'états
- Soit l'objet `Feu_de_signalisation_EnQuestion` avec sa couleur et ses trois valeurs



OBJET= Référent + Etat + Comportement

Mais qui est la cause des changements d'états ?

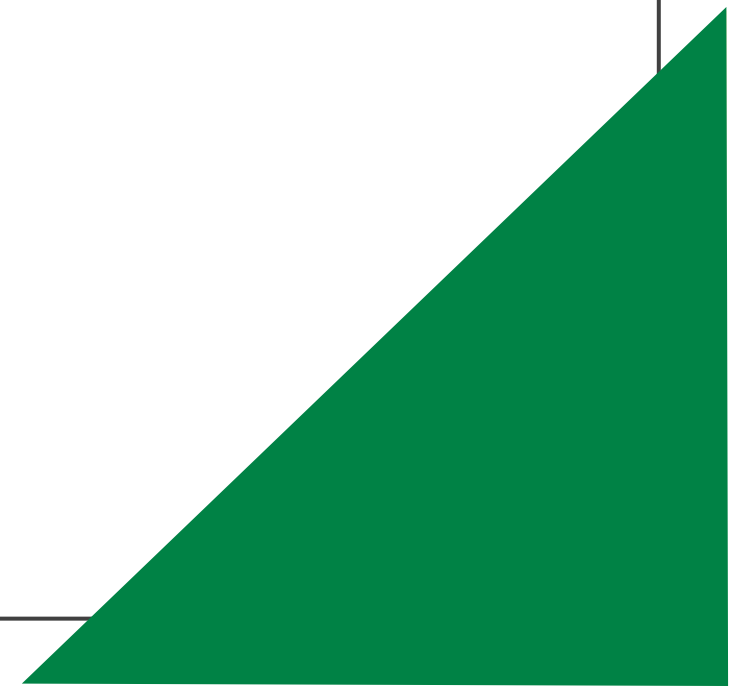
– LES METHODES



CHAPITRE 2

DÉFINIR UN OBJET

- 1 - Notion d'objet
- 2 - Notion de classe**
- 3 – Constructeur/Destructeur
- 4 – Objets en interaction



Notion de classe

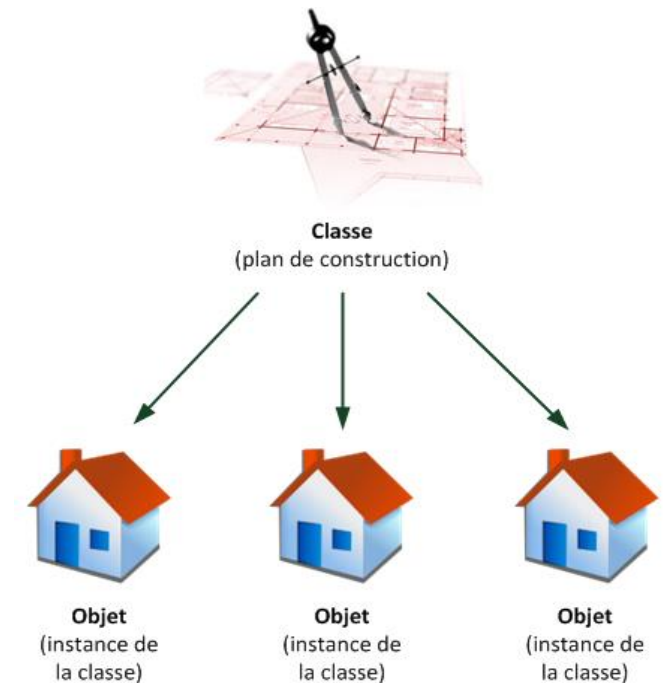


Une **classe** unit les méthodes aux attributs

CLASSE= Attributs + Méthodes

Une **classe** est un **modèle** de définition pour les objets ayant:

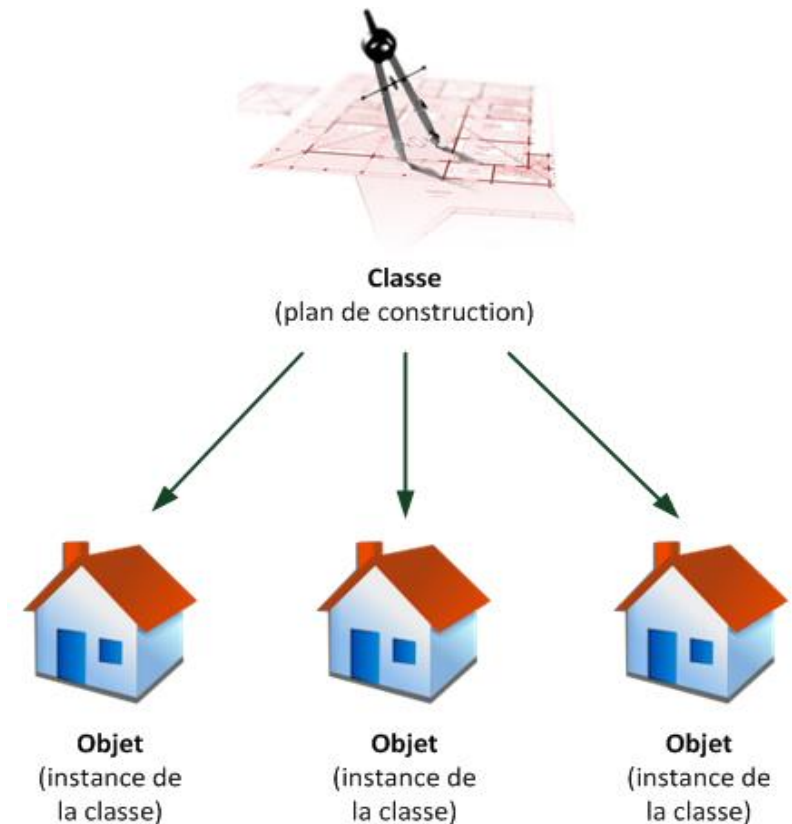
- même structure (**même ensemble d'attributs**)
- même comportement (**mêmes méthodes**)



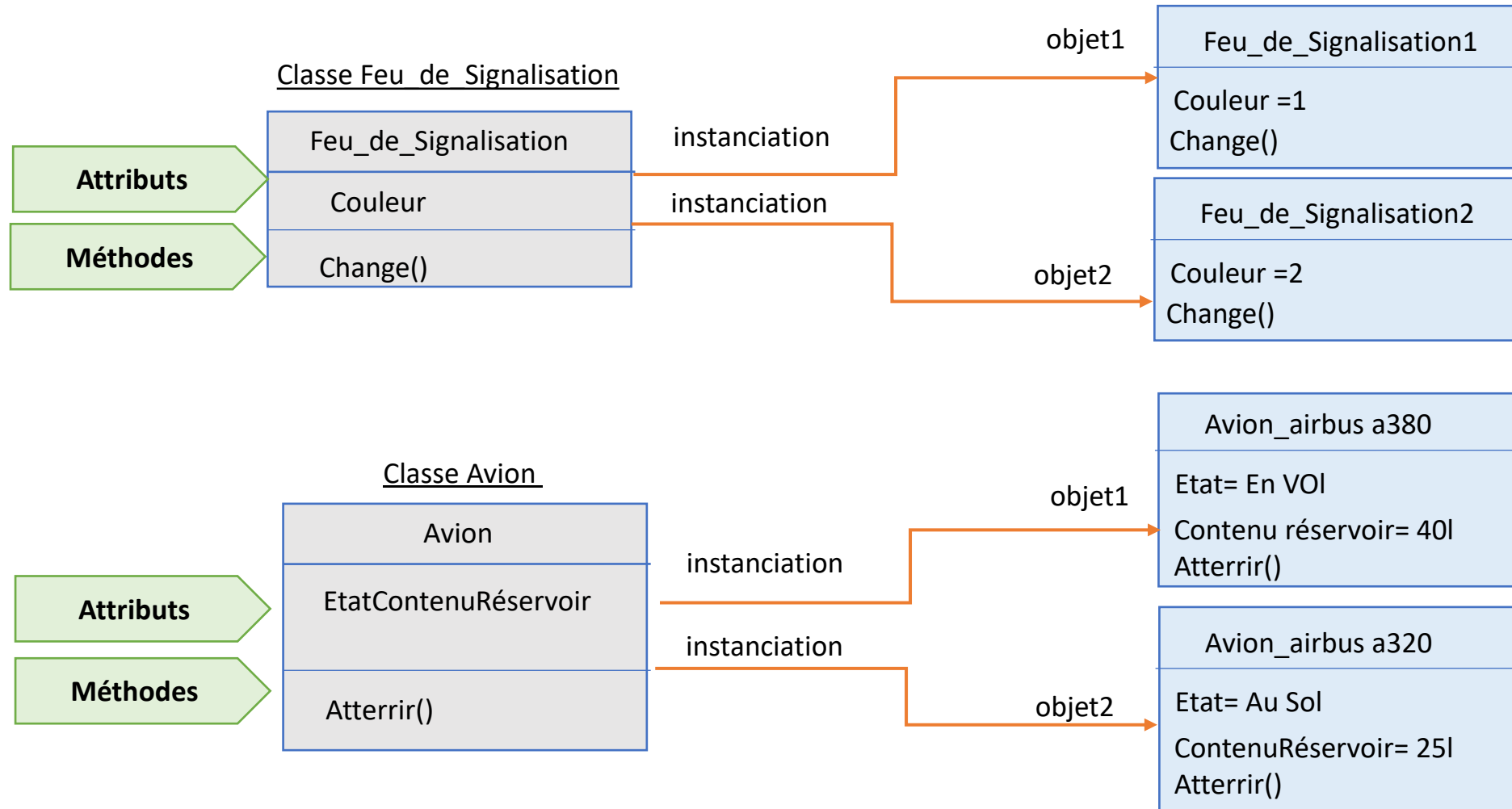
Notion de classe



- Les objets sont des représentations dynamiques (instanciation), « vivantes » du modèle défini pour eux au travers de la classe.
- Une classe permet **d'instancier** (créer) plusieurs objets
- Chaque objet est une instance (un exemplaire) **d'une (seule) classe**



Notion de classe



Objet: instance d'une classe



- Chaque objet correspond à une instance de la classe à laquelle il fait référence
- La création d'un objet est constituée de deux phases:
 - Une phase de ressort de la classe: allouer de la mémoire et un contexte d'exécution minimaliste.
 - Une phase du ressort de l'objet: initialiser ses attributs d'instances
- Dans les langages, ces deux phases ne sont pas différenciées.
→ Appel à une méthode spéciale : **le constructeur**

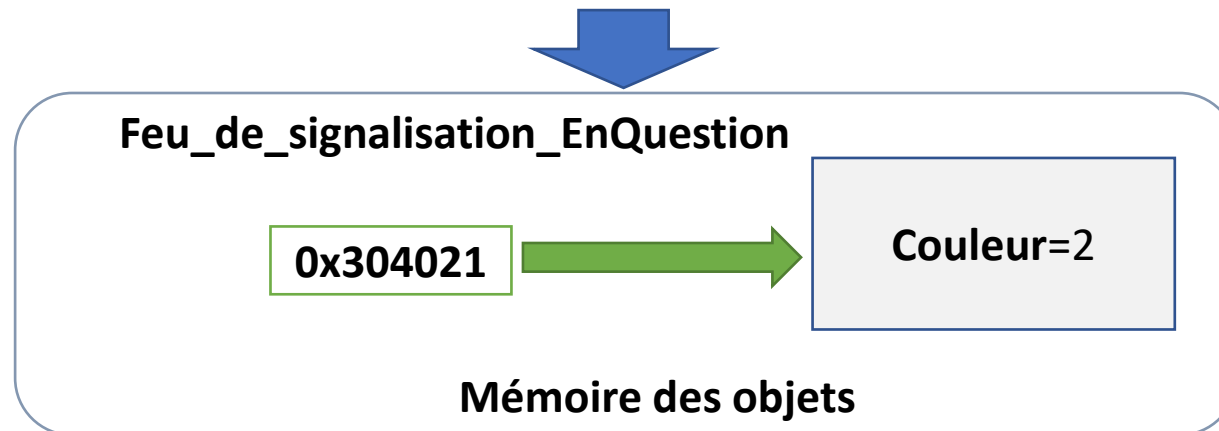
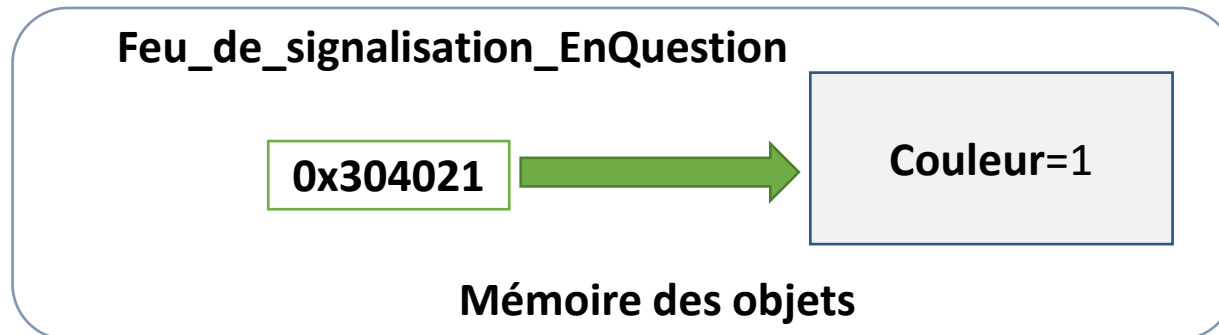
Objet: instance d'une classe



Appel d'une méthode d'une classe

`instanceDeClasse.méthode()`

Exemple: Soit la classe Feu_De_Signalisation



Classe Feu de Signalisation

Feu_de_Signalisation
Couleur
Change()

`Feu_de_signalisation_EnQuestion.change()`
Application de la méthode `change()`
définie dans la classe `Feu_de_Signalisation`

Membre de classe vs membre d'instance



Un membre est un attribut ou une méthode. Il existe 2 types de membres (de classe/d'instance)

- **Membre de classe**

- Un Attribut de classe est associé à sa classe et non à une instance de cette classe.
- Méthode de classe est associée à une classe et non à un objet.

Remarque:

- Une méthode de classe ne peut exploiter que des membres de classe
- Une méthode d'instance peut utiliser n'importe quel type de membre (classe ou instance)

- **Membre d'instance**

Un attribut d'instance est associée à une instance de la classe.

Chaque objet possède donc sa propre copie de la propriété

.Une méthode d'instance est associée à une instance d'une classe.

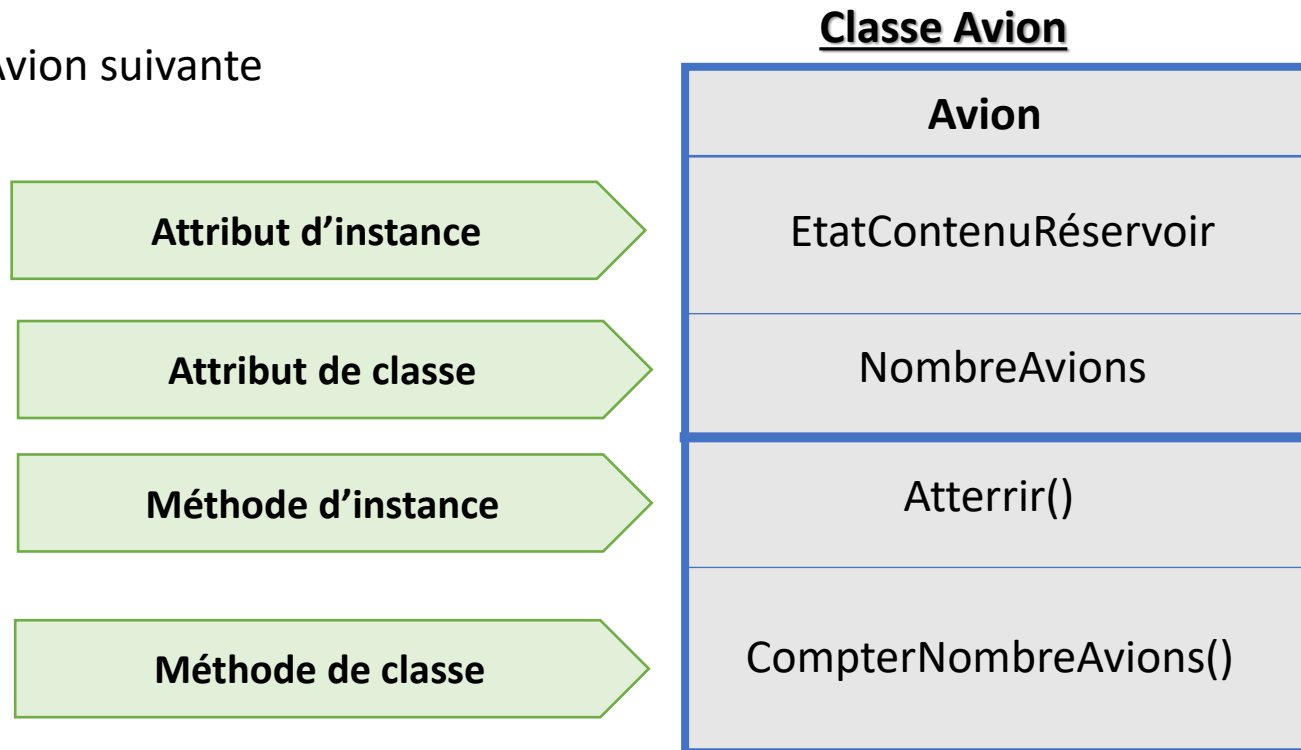
Chaque objet possède donc sa propre copie de la méthode.

Membre de classe vs membre d'instance



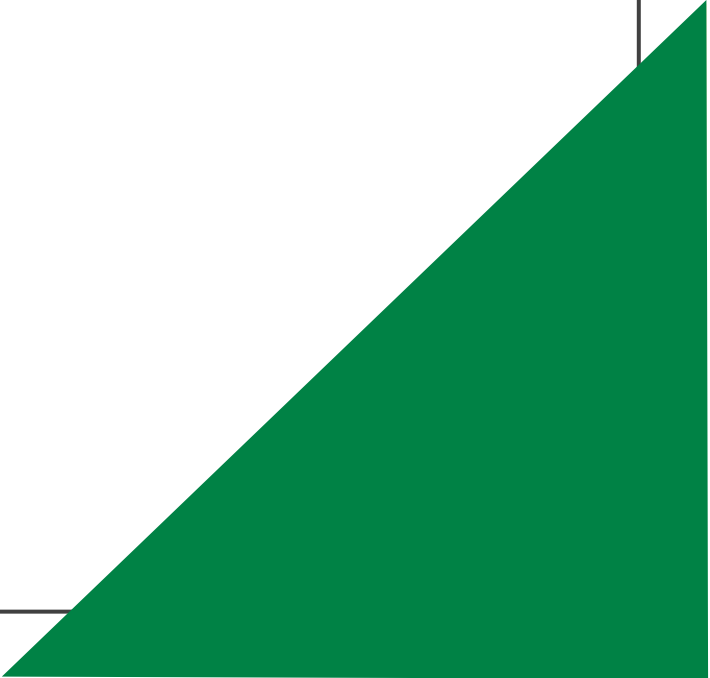
Exemple:

la classe Avion suivante



CHAPITRE 2

DÉFINIR UN OBJET

- 1 - Notion d'objet
 - 2 - Notion de classe
 - 3 - Constructeur/Destructeur**
 - 4 - Objets en interaction
- 

Constructeur



- Le constructeur d'une classe est une méthode appelée une seule fois par objet et **ce au moment de sa création.**
- Cette méthode sera appelée lors de la création de l'objet.
- Le constructeur peut disposer d'un nombre quelconque de paramètres, éventuellement aucun.
- Le rôle du constructeur est d'initialiser les attributs de l'objet

Destructeur

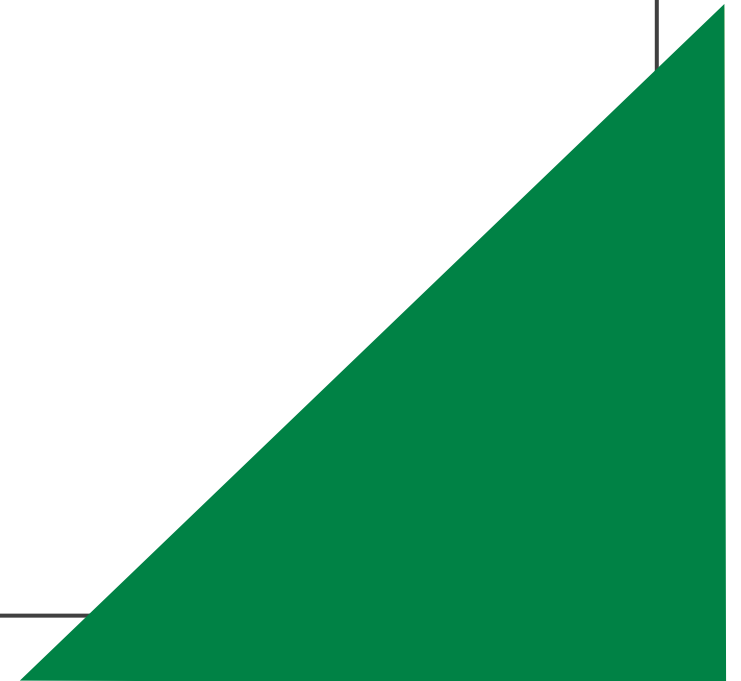


- Le destructeur est une méthode qui permet **la destruction d'un objet non référencé**.
- Un destructeurs permet:
 - Gérer les erreurs
 - Libérer les ressources utilisées de manière certaine
 - Assurer la fermeture de certaines parties du code.
- Les langages qui utilisent des ramasse-miettes (exemple Python) n'offrent pas le mécanisme des destructeurs puisque le programmeur ne gère pas la mémoire lui-même
 - Un ramasse-miettes est programme de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.

CHAPITRE 2

DÉFINIR UN OBJET

- 1 - Notion d'objet
- 2 - Notion de classe
- 3 - Constructeur/Destructeur
- 4 - Objets en interaction**



Objets en interaction

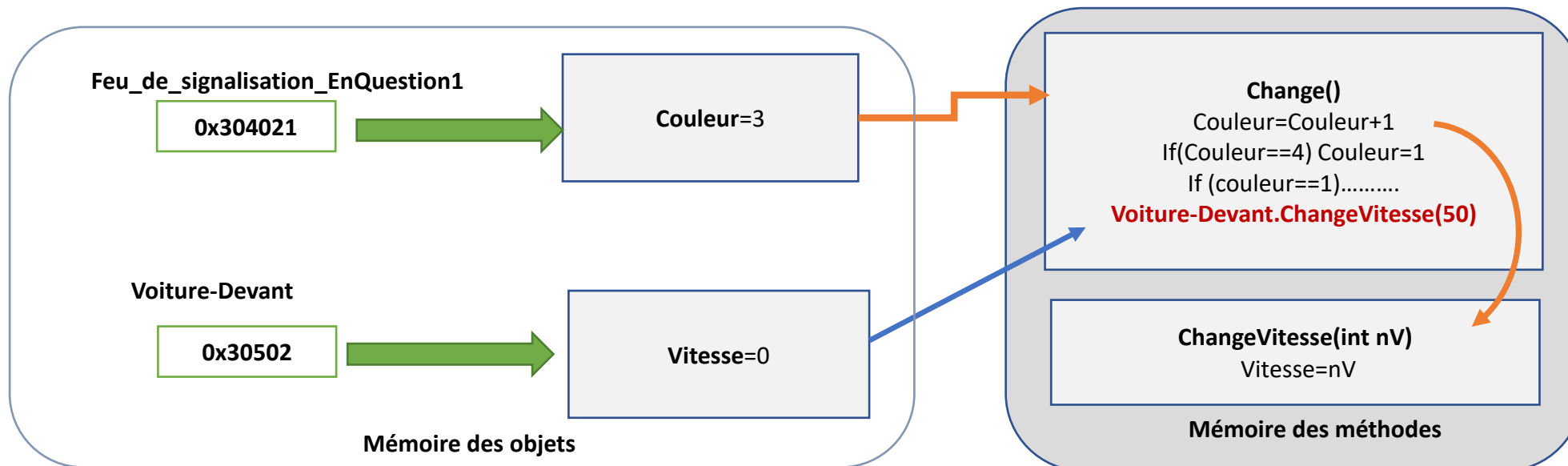


- La communication entre les objets est la base de la POO
- Les objets communiquent par envois de messages ceci:
 - Quand un objet demande à un autre d'exécuter une méthode qui est propre à cet autre.

Objets en interaction



- Supposons que le feu « **Feu_de_signalisation_EnQuestion1** » fasse ralentir ou accélérer la voiture « **Voiture-Devant** »
- Objet Feu_de_signalisation_EnQuestion est l'**expéditeur** de message
- Objet Voiture-Devant est le **destinataire** de message



PARTIE 2

CONNAÎTRE LES PRINCIPAUX PILIERS DE LA POO



CHAPITRE 1

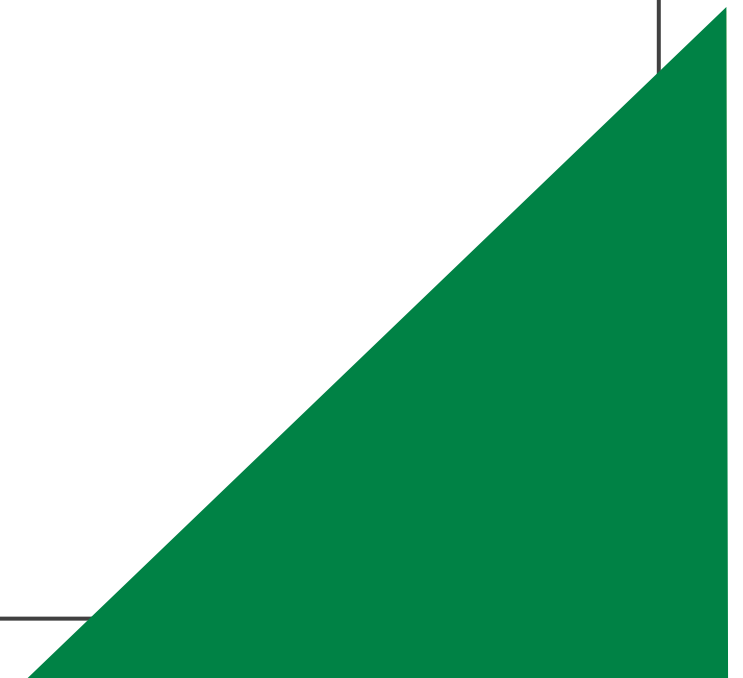
DÉFINIR L'HÉRITAGE ET LE POLYMORPHISME

1 - Principe et intérêt de l'héritage

2 - Types de l'héritage

3 - Redéfinition des méthodes

4 - Principe du polymorphisme



Principe de l'héritage



- Le concept de l'héritage spécifie une relation de **spécialisation/généralisation** entre les classes (Document : livre, revu, .../Personne : étudiant, employé...)
- Lorsqu'une classe D hérite d'une classe B :
 - D possède toutes les caractéristiques de B et aussi, d'autres caractéristiques qui sont spécifiques à D
 - D est une spécialisation de B (un cas particulier)
 - B est une généralisation de D (cas général)
 - D est appelée classe dérivée (fille)
 - B est appelée classe de base (mère ou super-classe)
- **Tout objet instancié de D est considéré, aussi, comme un objet de type B**
- **Un objet instancié de B n'est pas forcément un objet de type D**

Principe de l'héritage



Exemple:

Considérons la définition des 3 classes Personne, Etudiant et Employé suivantes:

Personne
Nom
CIN
anneeNaiss
age()

Etudiant
Nom
CIN
anneeNaiss
note1, note2
age()
moyenne()

Employe
Nom
CIN
anneeNaiss
prixHeure
nbreHeure
age()
Salaire()

Problème:

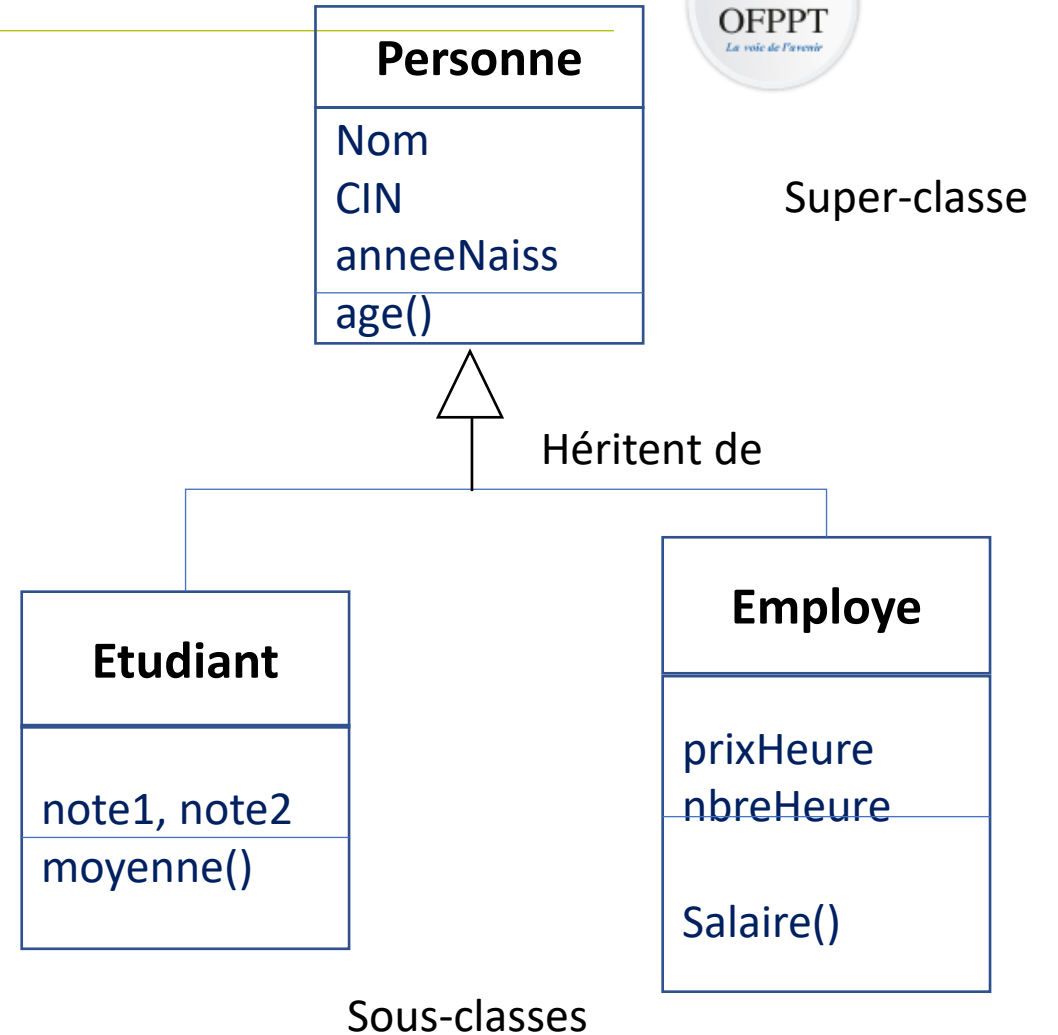
- Duplication du code
- Une modification faite sur un attribut ou méthode doit être refaite sur les autres classes

Principe de l'héritage



Solution

- Placer dans la **classe mère** toutes les informations communes à toutes les classes.
- Les classes filles ne comportent que les attributs ou méthodes **plus spécifiques**.
- Les classes filles **héritent** automatiquement les attributs (et les méthodes) qui n'ont pas besoin d'être ré-écrits.



Intérêt de l'héritage



- L'héritage minimise l'écriture du code en **regroupant les caractéristiques communes** entre classes au sein d'une seule (la classe de base) sans duplication.
- **Localisation facile**, en un point unique, des sections de code.
- La rectification du code se fait dans des endroits uniques grâce à la **non redondance de description**
- L'extension ou l'ajout de nouvelles classes est favorisée surtout en cas d'une hiérarchie de classes bien conçue
- Définition des informations et des comportements aux niveaux opportuns.
- Rapprochement de la modélisation des systèmes d'informations aux cas réels

CHAPITRE 1

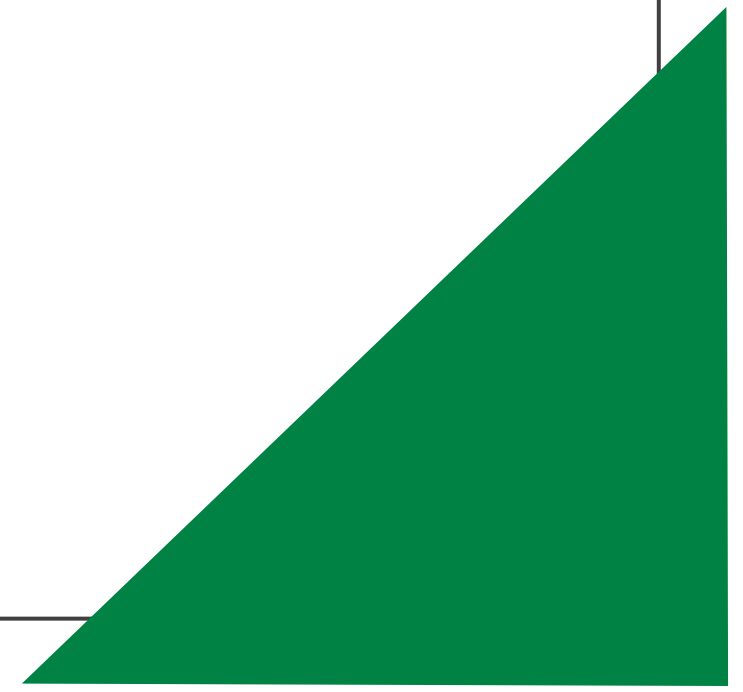
DÉFINIR L'HÉRITAGE ET LE POLYMORPHISME

1 - Principe et intérêt de l'héritage

2 - Types de l'héritage

3 - Redéfinition des méthodes

4 - Principe du polymorphisme



Héritage multiple

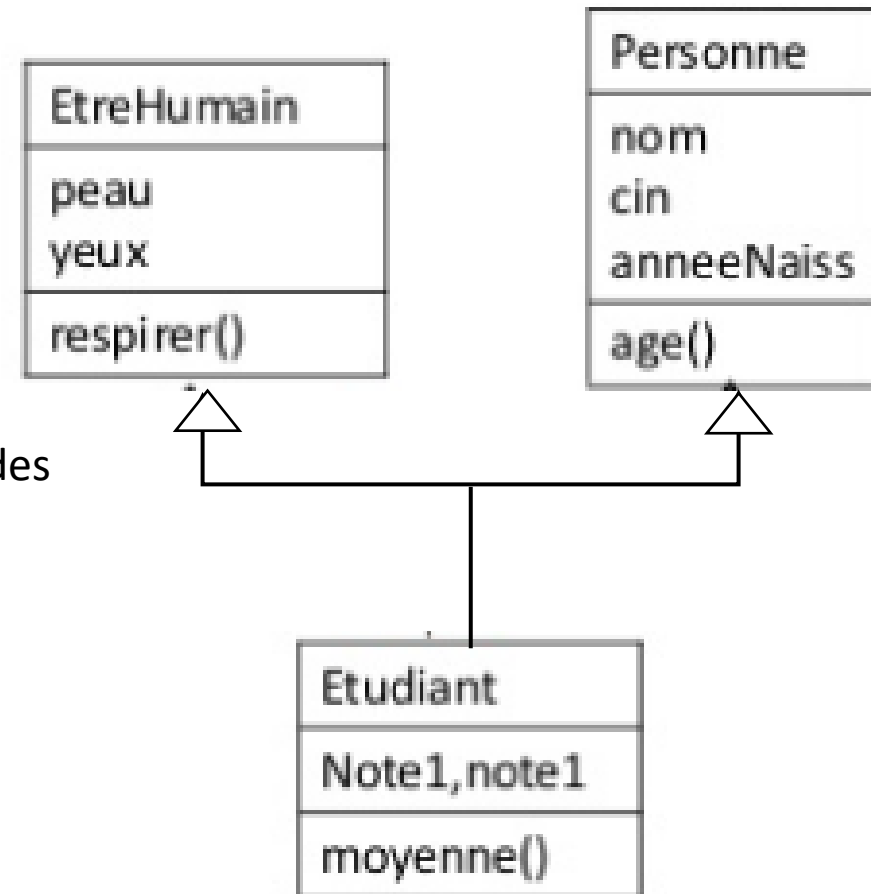


- Une classe peut hériter de plusieurs classes

Exemple:

Un Etudiant est à la fois une Personne et un EtreHumain

- La classe Etudiant hérite les attributs et les méthodes des deux classes → Il deviennent ses propres attributs et méthodes



Héritage en cascade

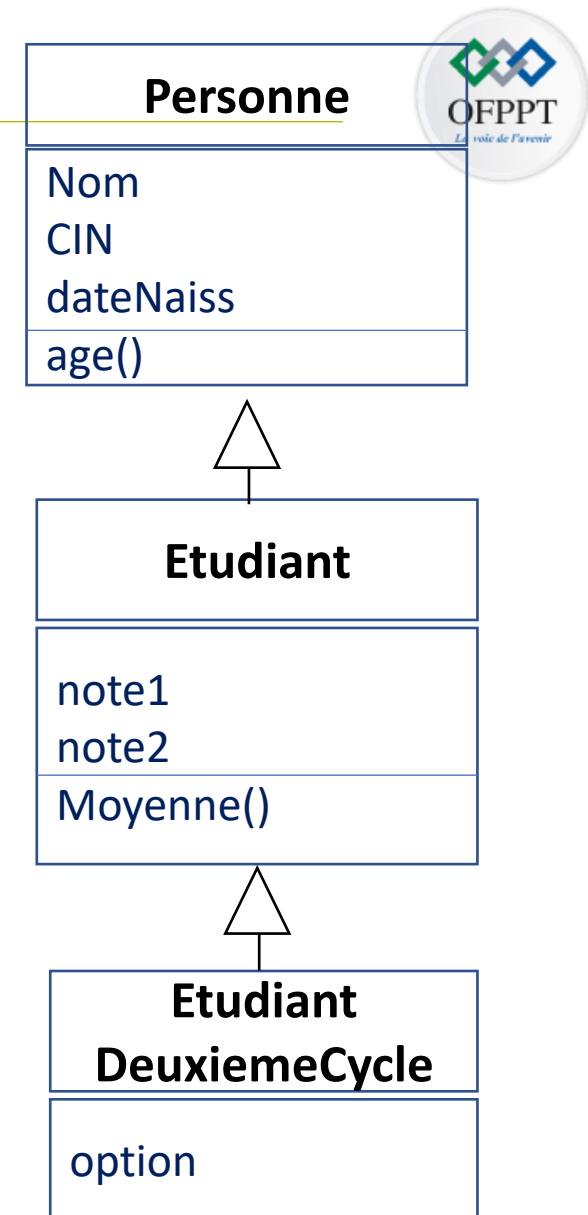
- Une classe sous-classe peut être elle-même une super-classe

Exemple:

Etudiant hérite de Personne

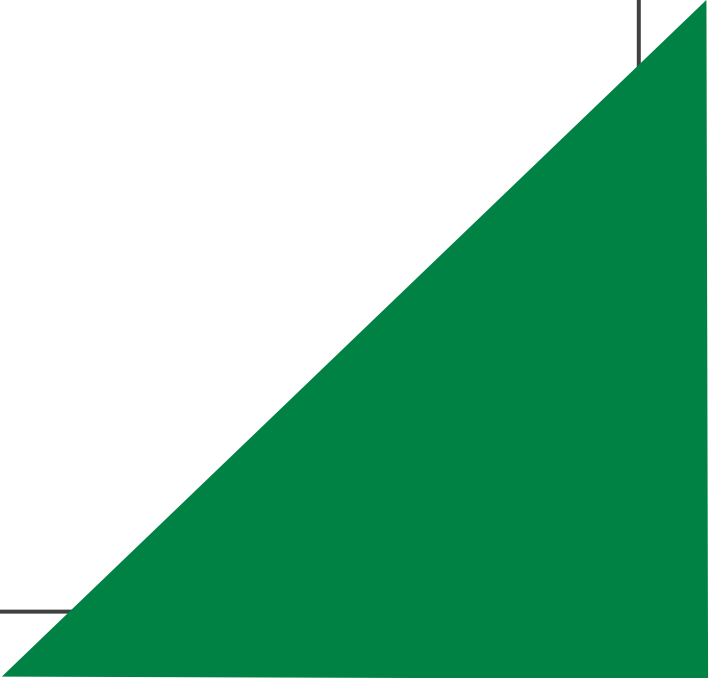
EtudiantDeuxièmeCycle hérite de Etudiant

→ EtudiantDeuxièmeCycle hérite de Personne



CHAPITRE 1

DÉFINIR L'HÉRITAGE ET LE POLYMORPHISME

- 1 - Principe et intérêt de l'héritage
 - 2 - Types de l'héritage
 - 3 - Redéfinition des méthodes**
 - 4 - Principe du polymorphisme
- 

Redéfinition des méthodes héritées



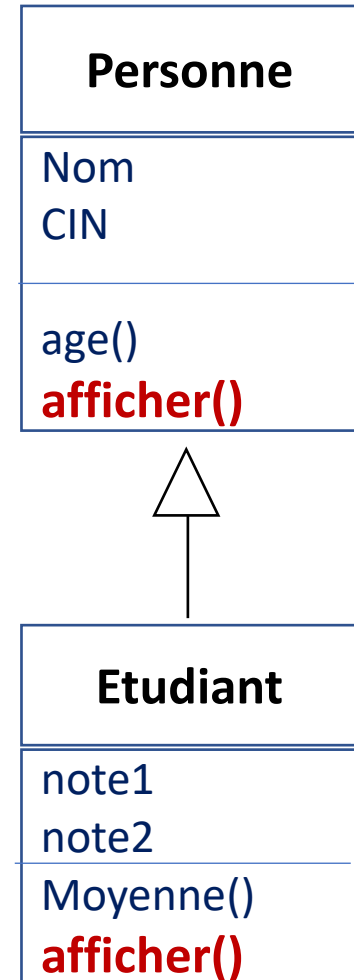
- Une méthode héritée peut être redéfinie si sa version initiale n'est pas satisfaisante pour la classe dérivée
- La redéfinition consiste à **conserver l'entête de la méthode** et à **proposer un code différent**
- Si une méthode héritée est redéfinie, c'est uniquement **la nouvelle version qui fait parti de la description de la classe dérivée**
- Si la méthode définie au niveau de la classe dérivée est de type différent, ou de paramètres différents, alors il s'agit d'une nouvelle méthode qui s'ajoute à celle héritée de la classe de base

Redéfinition des méthodes héritées



Exemple:

- Soit la classe Etudiant qui hérite de la classe Personne
- La méthode afficher() de la classe Personne affiche les attribut Nom, CIN d'une personne
- La classe Etudiant hérite la méthode afficher() de la classe Personne et la **redéfinit**, elle propose un nouveau code (la classe Etudiant **ajoute** l'affichage des attributs notes1 et note2 de l'étudiant)



Mécanisme de la liaison retardée



Soit **C** la classe réelle d'un objet **o** à qui on envoie un message « **o.m()** »

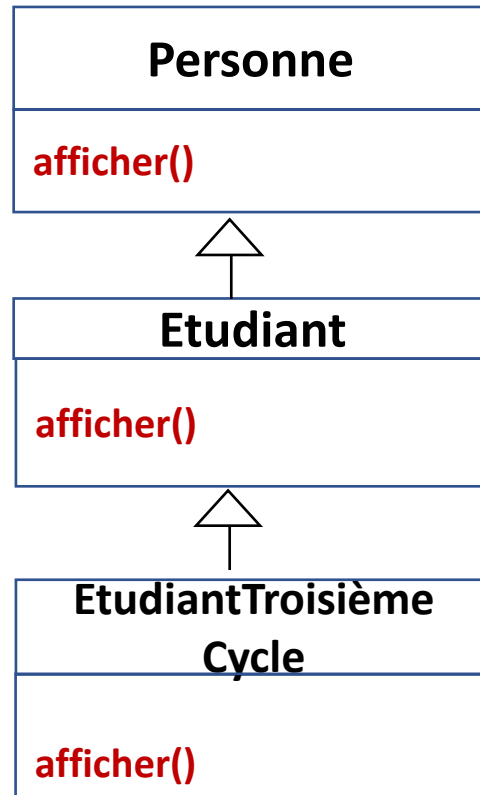
- **o.m()** peut appeler la méthode **m()** de **C** ou de n'importe quelle sous-classe de **C**
- Si le code de la classe **C** contient la définition (ou la redéfinition) d'une méthode **m()**, c'est cette méthode qui sera exécutée
- Sinon, la recherche de la méthode **m()** se poursuit dans la classe mère de **C**, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode **m()** qui est alors exécutée

Mécanisme de la liaison retardée



Exemple: Soit etc un objet de type EtudiantTroisièmeCycle

1^{er} cas



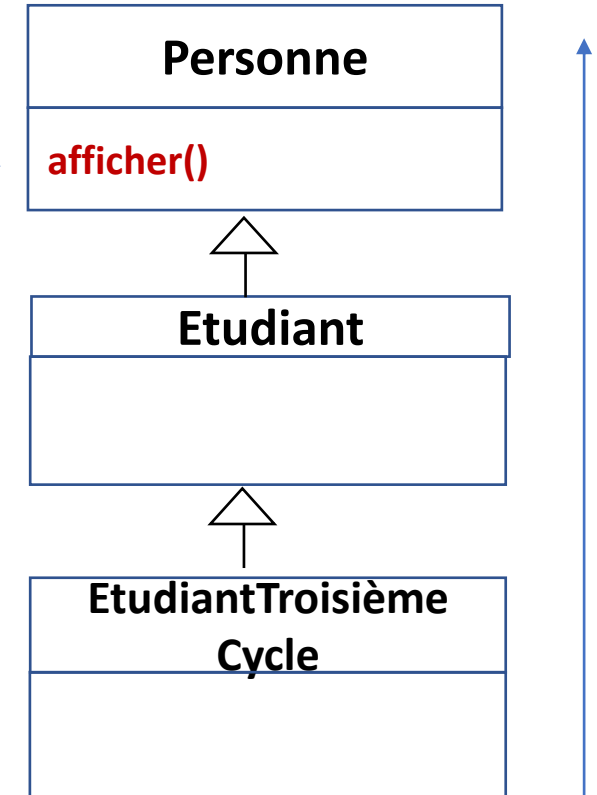
Afficher() de la classe EtudiantTroisièmeCycle est exécutée

etc.afficher()

2^{ème} cas

etc.afficher()

Afficher() de la classe Personne est exécutée



CHAPITRE 1

DÉFINIR L'HÉRITAGE ET LE POLYMORPHISME

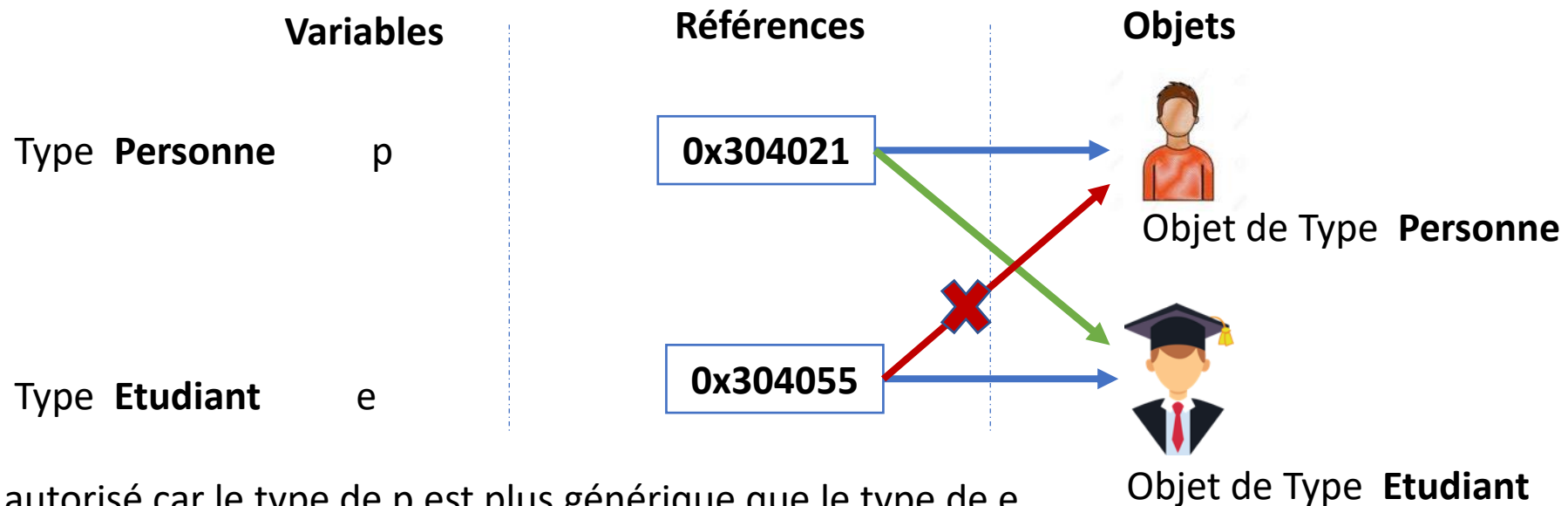
- 1 - Principe et intérêt de l'héritage
 - 2 - Types de l'héritage
 - 3 - Redéfinition des méthodes
 - 4 - Principe du polymorphisme**
- 

Principe de polymorphisme



- Le polymorphisme désigne un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence

Exemple: Une instance de Etudiant peut « être vue comme » une instance de Personne (**Pas l'inverse!!**)

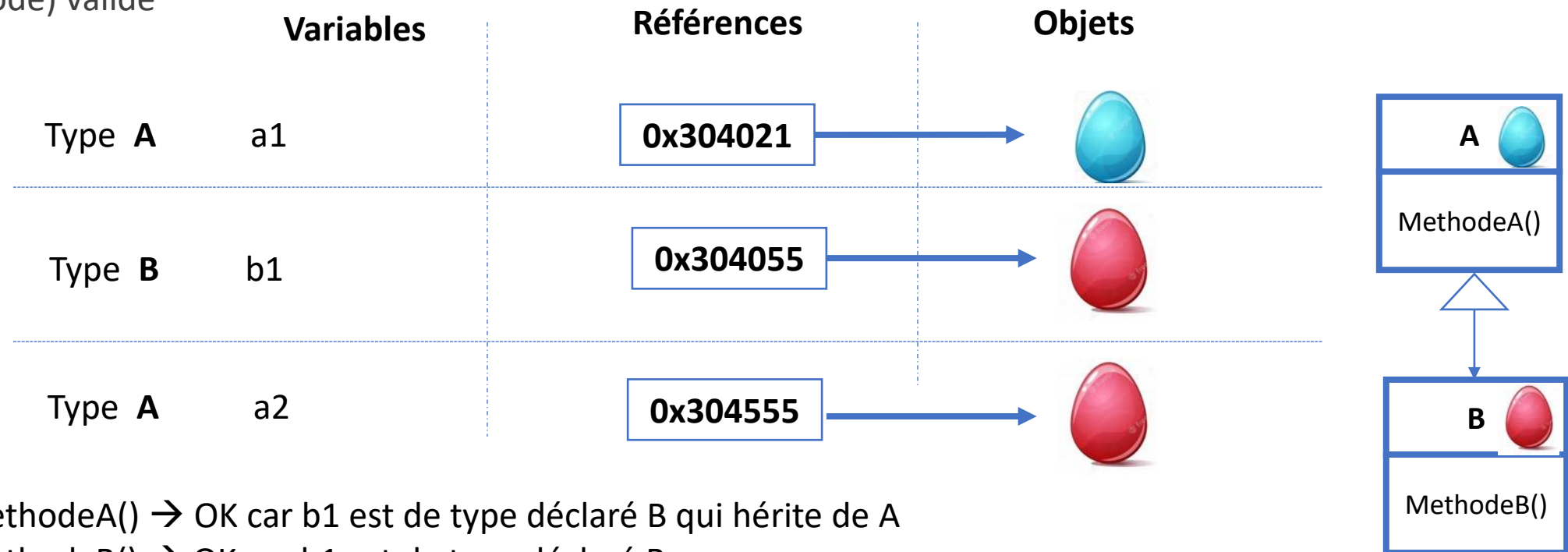


- $p=e \rightarrow$ autorisé car le type de p est plus générique que le type de e
- $e=p \rightarrow$ non autorisé car le type de e est plus spécifique que celui de p

Principe de polymorphisme



- le type de la variable est utilisé par le compilateur pour déterminer si on accède à un membre (attribut ou méthode) valide



b1.MethodeA() → OK car b1 est de type déclaré B qui hérite de A

b1.MethodeB() → OK car b1 est de type déclaré B

a2.MethodeA() → OK car a2 est de type déclaré A

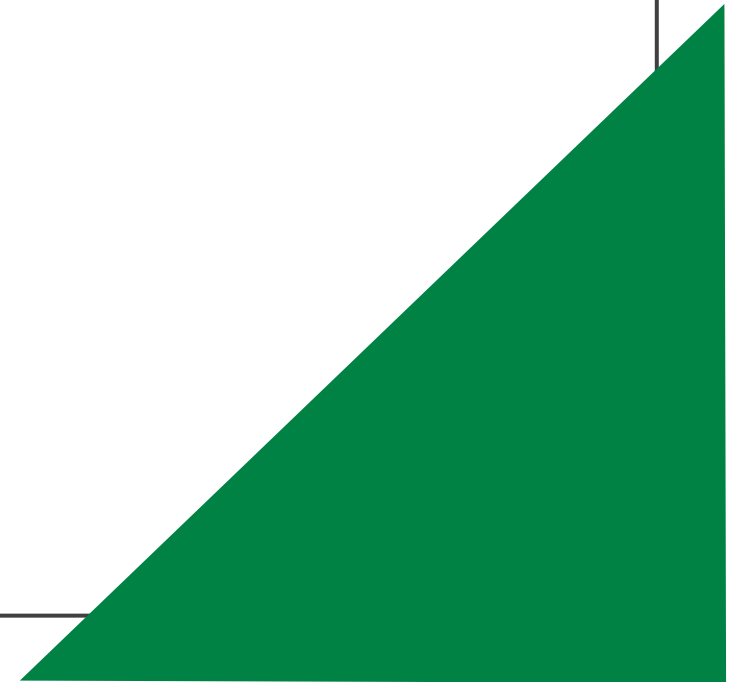
a2.MethodeB() → ERREUR car a2 est de type A (même si le type l'objet référencé est B)

CHAPITRE 2

CONNAITRE L'ENCAPSULATION

1 - Principe de l'encapsulation

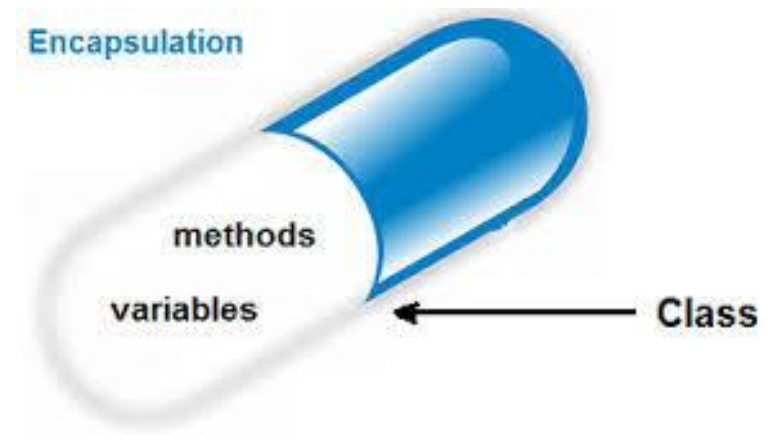
2 - Niveaux de visibilité



Principe de l'encapsulation



- L'encapsulation est le fait de réunir à l'intérieur d'une même entité (objet) le code (**méthodes**) + données (**attributs**).
- L'encapsulation consiste à protéger l'information contenue dans un objet.
- **Il est donc possible de masquer les informations d'un objet aux autres objets**



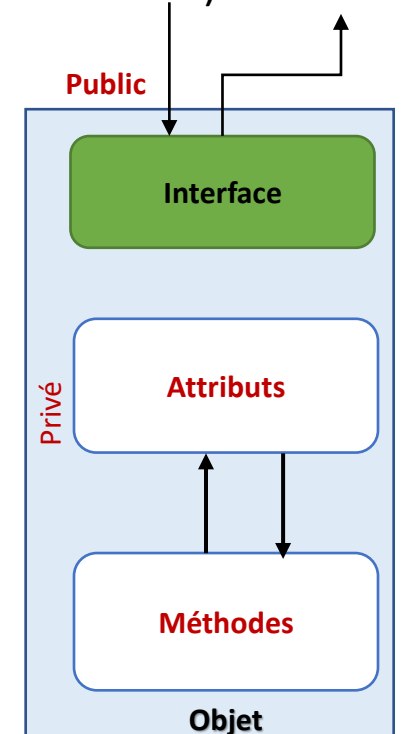
Principe de l'encapsulation



- L'encapsulation consiste donc à masquer les détails d'implémentation d'un objet, en définissant une **interface**.
- L'interface est la **vue externe d'un objet**, elle définit les services **accessibles** (Attributs et méthodes) aux utilisateurs de l'objet.

interface = liste des signatures des méthodes accessibles

Interface = Carte de visite de l'objet



Intérêt de l'encapsulation



- Les objets restreignent leur accès qu'aux méthodes de leur classe

→ Ils protègent leurs attributs

- Cela permet d'avoir un **contrôle sur tous les accès**.

Exemple:

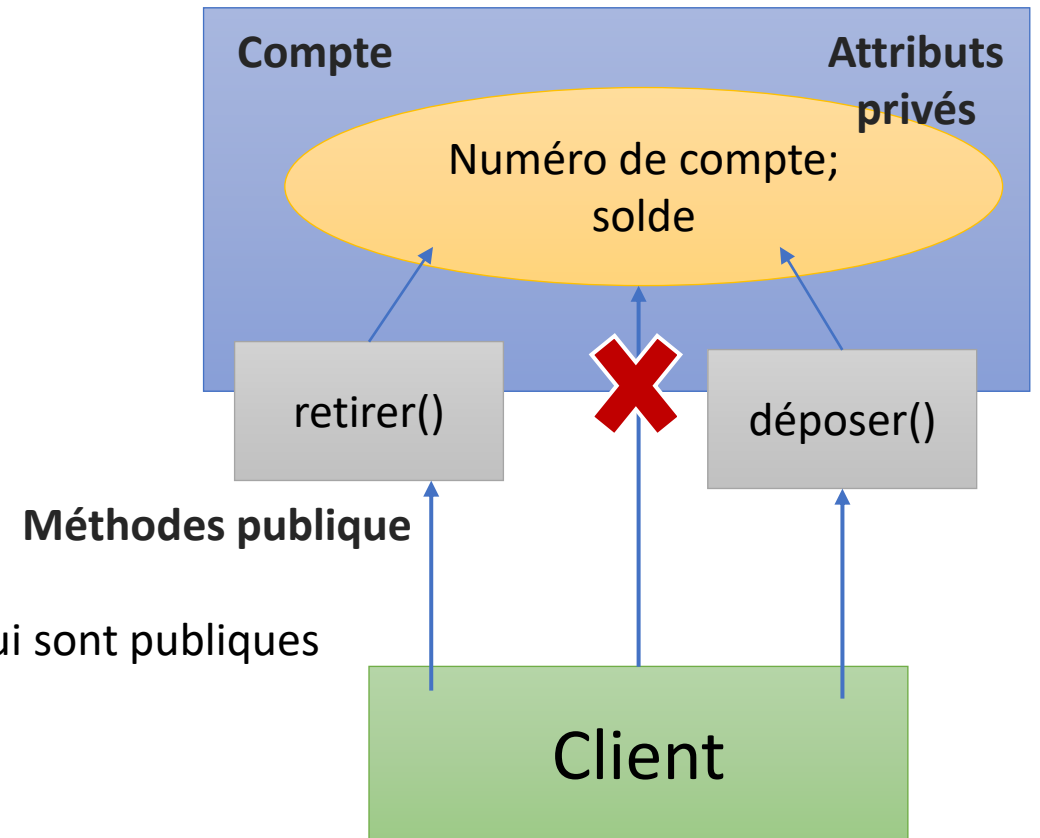
Les attributs de la classe compte sont privés

Ainsi le solde d'un compte n'est pas accessible

par un client qu'à travers les méthodes retirer() et déposer() qui sont publiques

→ Un client ne peut modifier son solde qu' on effectuant une

Opération de dépôt ou de retrait

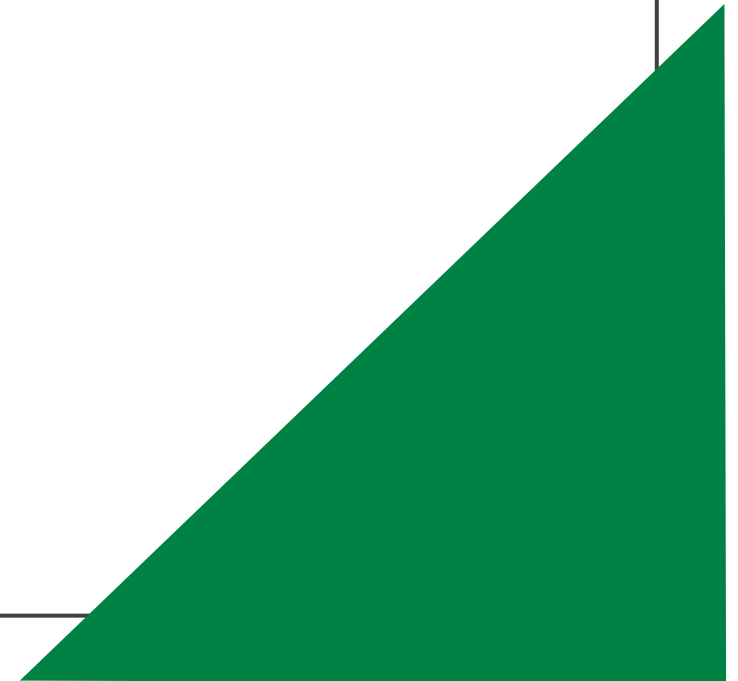


CHAPITRE 2

CONNAITRE L'ENCAPSULATION

1 - Principe de l'encapsulation

2 - Niveaux de visibilité

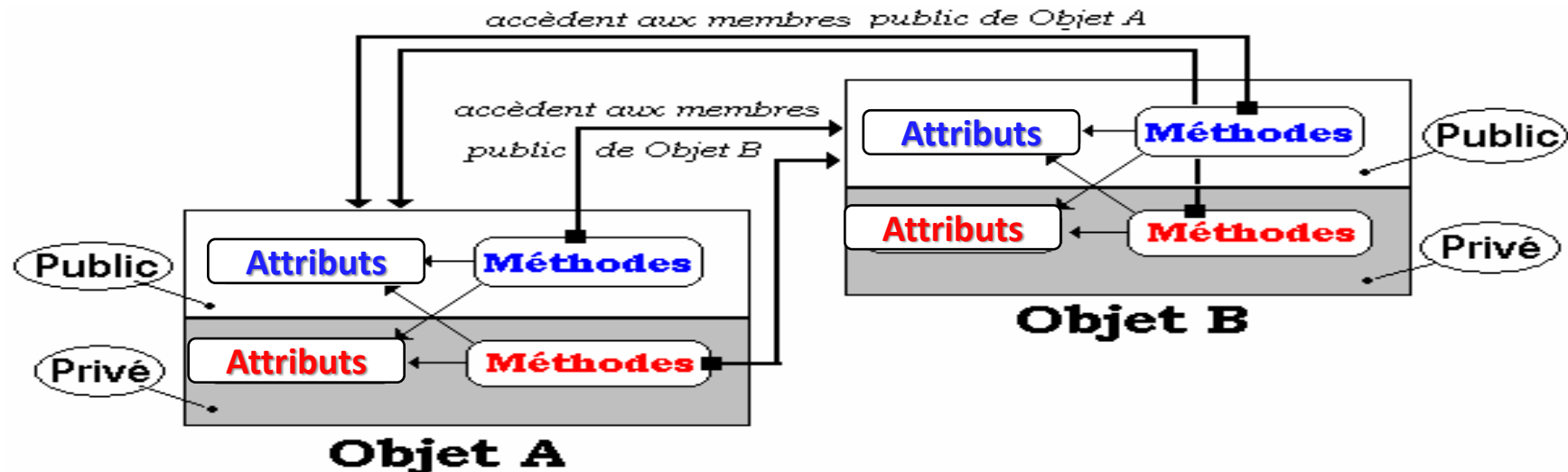


Niveaux de visibilité des membres



Il existe 2 niveaux de visibilité des membres d'une classe:

- **Public**: les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité publique → **Il s'agit du plus bas niveau de protection des données**
- **Privé**: l'accès aux données est limité aux méthodes de la classe elle-même → **Il s'agit du niveau de protection des données le plus élevé**



Niveaux de visibilité des membres



Les niveaux intermédiaires d'encapsulation

- **Classes Amies**

une classe pourrait décider de se rendre entièrement accessible à quelques autres classes, privilégiées, qu'elle déclarera comme faisant partie de ses « amies »

- **Une classe dans une autre**

Une classe rend accessible ses attributs et ses méthodes déclarés privés à une autre classe lorsque cette seconde classe est créée à l'intérieur de la première. **La classe englobée aura un accès privilégié à tout ce qui constitue la classe englobante.**

Niveaux de visibilité des membres



Les niveaux intermédiaires d'encapsulation

- **Mécanisme d'héritage**

Consiste à ne permettre qu'aux seuls enfants de la classe (ses héritiers) un accès aux attributs et méthodes *protected* du parent

- **Utilisation des paquetages**

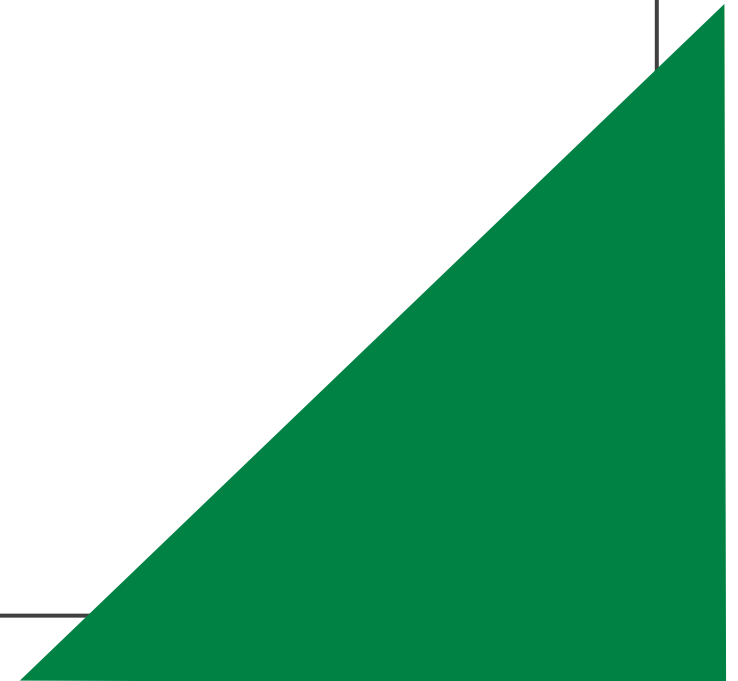
Consiste à libérer l'accès, uniquement aux classes faisant partie d'un même paquetage (un même répertoire)

CHAPITRE 3

CARACTÉRISER
L'ABSTRACTION

1 - Classes et méthodes abstraites

2 - Interface



Classes et méthodes abstraites



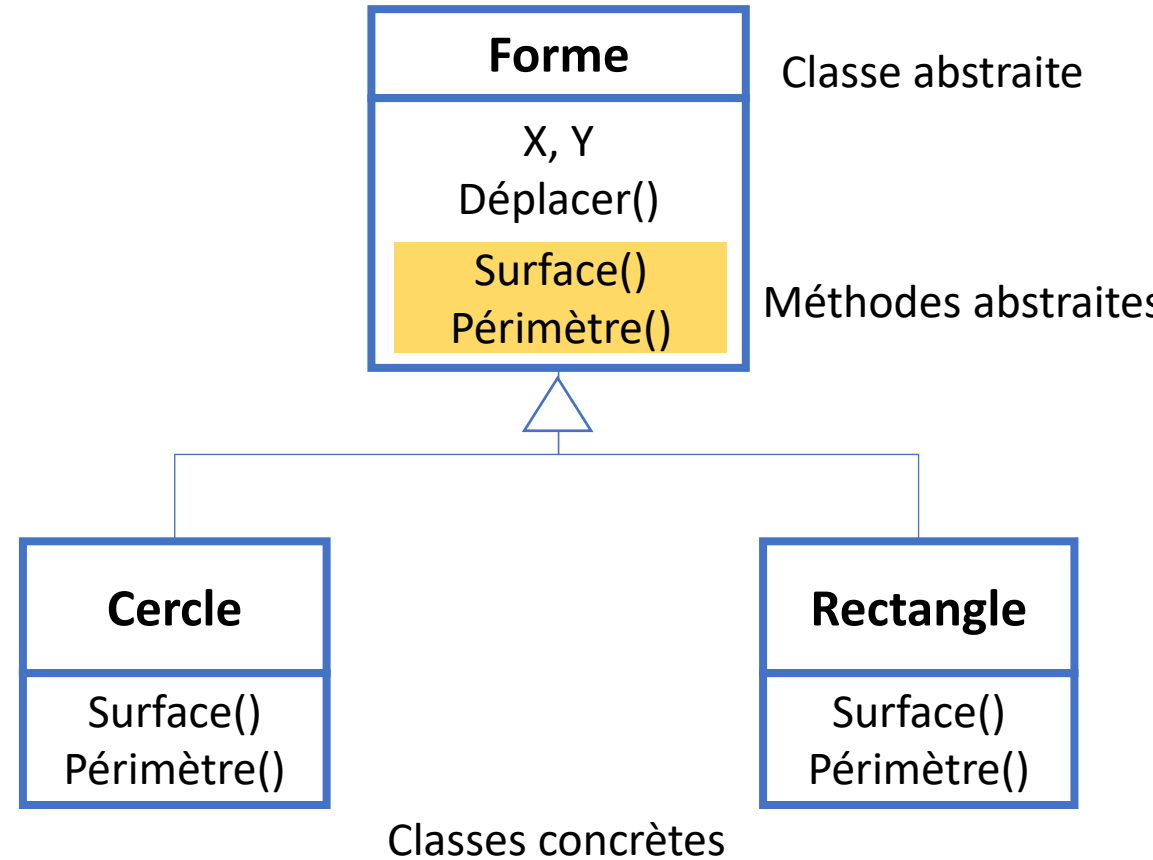
- Une **classe abstraite** est une classe qui **contient une ou plusieurs méthodes abstraites**. Elle peut malgré tout contenir d'autres méthodes habituelles (appelées parfois méthodes concrètes).
- Une **méthode abstraite** est une méthode qui **ne contient pas de corps**. Elle possède simplement une signature de définition (pas de bloc d'instructions)
- Une classe possédant une ou plusieurs méthodes abstraites devient obligatoirement une classe abstraite.
- Une classe abstraite peut ne pas comporter de méthodes abstraites

Classes et méthodes abstraites



Exemple:

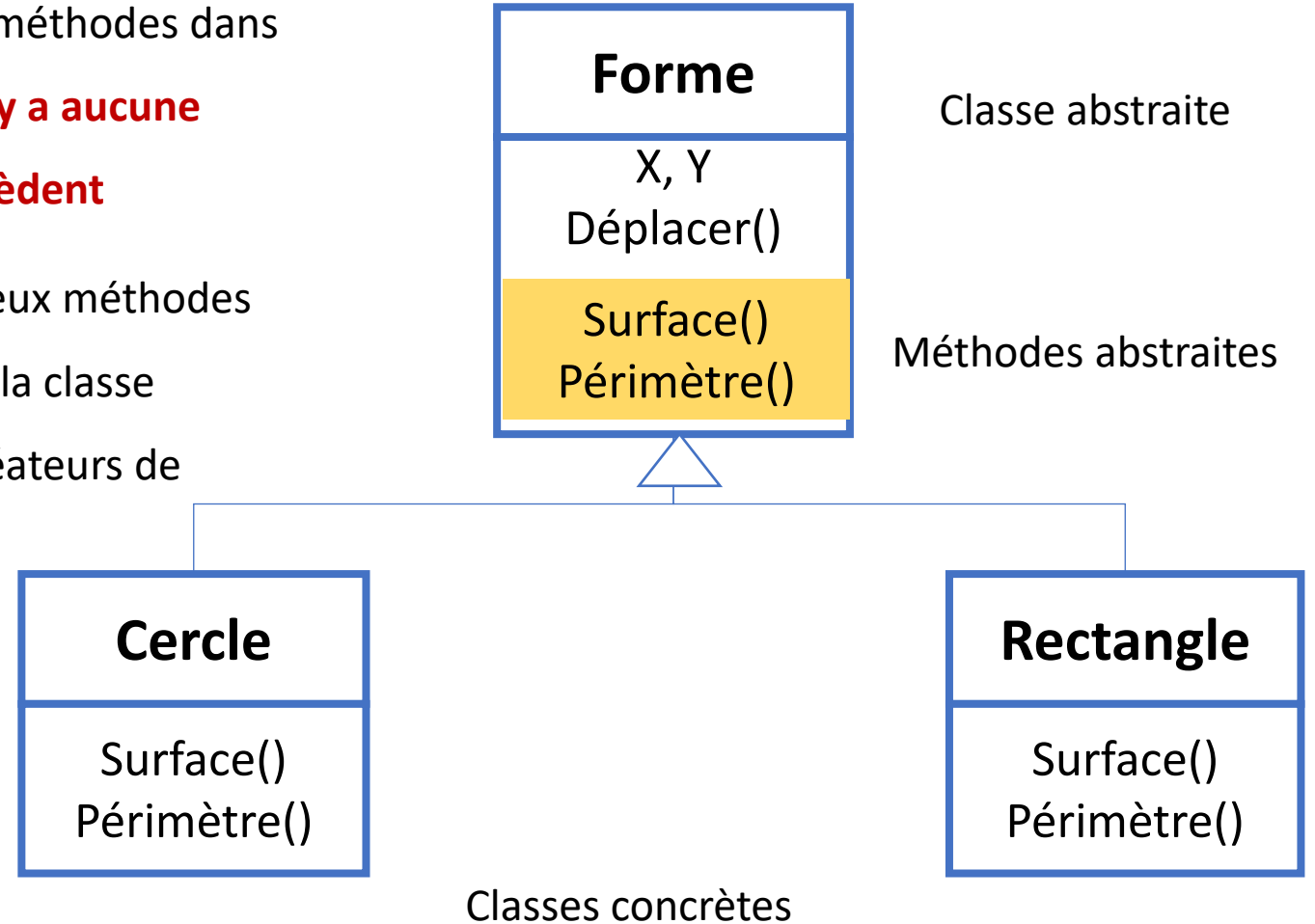
- Sachant que toutes les formes possèdent les propriétés périmètre et surface, il est judicieux de placer les méthodes définissant ces propriétés (périmètre() et surface()) dans la classe qui est à la racine de l'arborescence (Forme)
- La classe Forme ne peut cependant pas implémenter ces deux méthodes car on ne peut pas calculer le périmètre et la surface sans connaître le détail de la forme



Classes et méthodes abstraites



- On pourrait naturellement implémenter ces méthodes dans chacune des sous-classes de `Forme` mais **il n'y a aucune garantie que toutes les sous-classes les possèdent**
- **Solution:** Déclaration dans la classe `Forme` deux méthodes abstraites `périmètre()` et `surface()` ce qui rend la classe `Forme` elle-même abstraite et impose aux créateurs de sous-classes de les implémenter



Classes et méthodes abstraites



Les règles suivantes s'appliquent aux classes abstraites:

- Une classe abstraite **ne peut pas être instanciée**
- Une sous-classe d'une classe abstraite ne peut être instanciée que si elle redéfinit chaque méthode abstraite de sa classe parente et qu'elle fournit une implémentation (un corps) pour chacune des méthodes abstraites.
- Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, cette sous-classe est elle-même abstraite (et ne peut donc pas être instanciée)

Utilité des classes abstraites



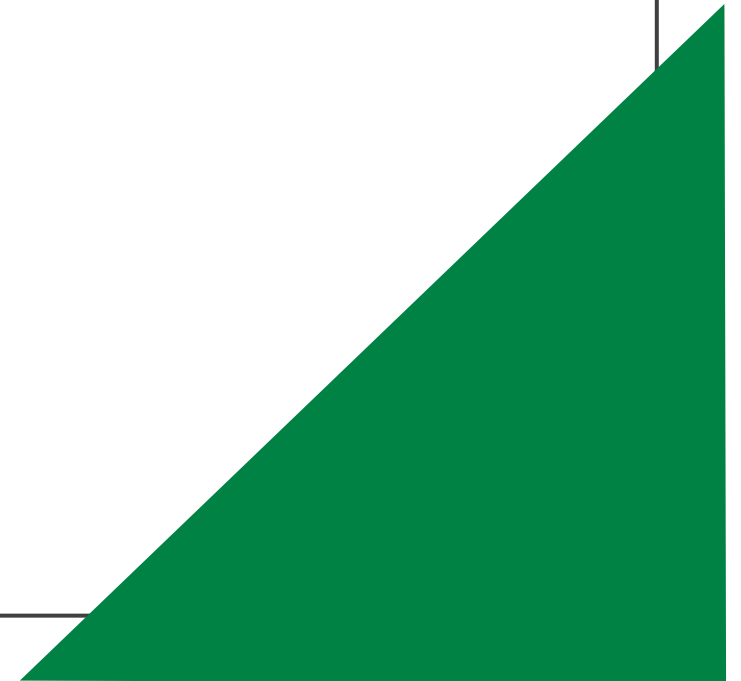
- les classes abstraites permettent de définir des fonctionnalités (des comportements) que les **sous-classes** devront **impérativement implémenter**
- Les utilisateurs des sous-classes d'une classe abstraite sont donc assurés de trouver toutes les méthodes définies dans la classe abstraite dans chacune des sous-classes concrètes
- Les classes abstraites constituent donc une sorte de contrat (spécification contraignante) qui garantit que certaines méthodes seront disponibles dans les sous-classes et qui oblige les programmeurs à les implémenter dans toutes les sous-classes concrètes.

CHAPITRE 3

CARACTÉRISER L'ABSTRACTION

1 - Classes et méthodes abstraites

2 - Interface



Principe d'interface



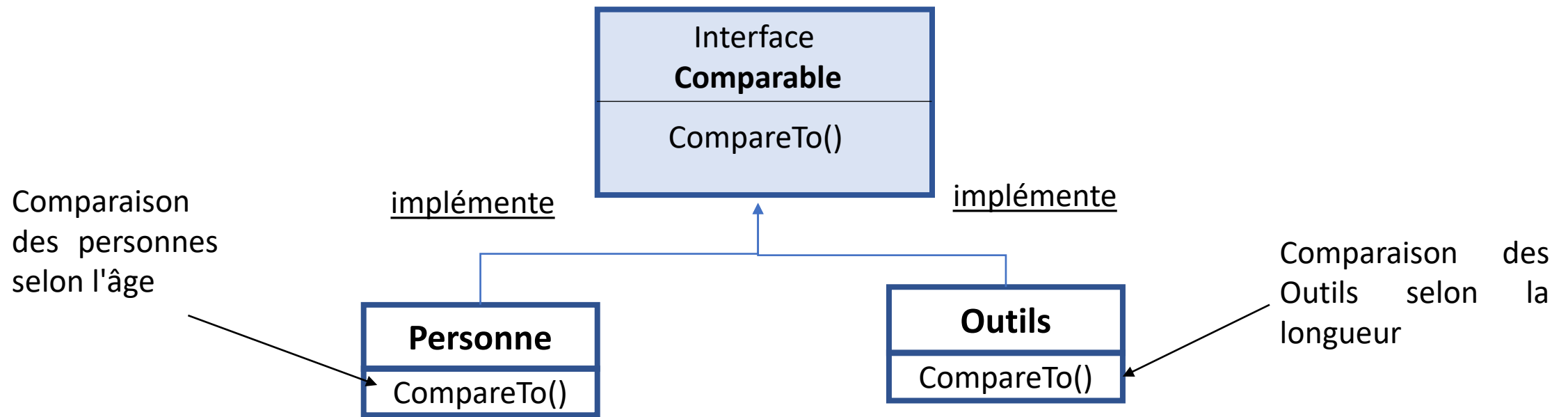
- Comme une classe et une classe abstraite, une interface permet de définir un nouveau type (référence).
- Une interface est une forme particulière de classe où **toutes les méthodes sont abstraites**
- Lorsqu'une classe implémente une interface, elle indique ainsi qu'elle s'engage à fournir une implémentation (c'est-à-dire un corps) pour chacune des méthodes abstraites de cette interface.
- Si une classe implémente plus d'une interface, elle doit implémenter toutes les méthodes abstraites de chacune des interfaces.

Principe d'interface



Exemple1: Si l'on souhaite caractériser la fonctionnalité de comparaison qui est commune à tous les objets qui ont une relation d'ordre (plus petit, égal, plus grand), on peut définir l'interface Comparable.

- Les classe `Personne` et `Outils` qui implémentent l'interface `Comparable` **doivent présenter une implémentation de la méthode `CompareTo()` sinon elles seront abstraites**



Principe d'interface



Exemple2:

Supposons que nous voulions que les classes dérivées de la classe `Forme` disposent toutes d'une méthode `imprimer()` permettant d'imprimer les formes géométriques.

Solution 1: Ajouter une méthode abstraite `Imprimer()` à la classe `Forme` et ainsi chacune des sous-classes concrètes devrait implémenter cette méthode

→ si d'autres classes (qui n'héritent pas de `Forme` souhaitent également disposer des fonctions d'impression, elles devraient à nouveau déclarer des méthodes abstraites d'impression dans leur arborescence.

Solution 2: la classe `forme` implémente l'interface `imprimable` ayant la méthode `Imprimer()`

