

PROGRAMMATION ORIENTÉE OBJET : POO

<https://www.php.net/manual/fr/oop5.intro.php>



Introduction à la P00

■ Rappels de la programmation :

➤ Tout programme contient 2 parties fondamentales :

- **Les données** : dans un programme, les données sont véhiculées par des variables.
Le stockage des informations se fait via une base de données.
- **Les traitements** : ce sont les lignes d'instructions qui permettent d'utiliser les données (ajouter / modification / lecture)

Introduction à la P00

■ Deux logiques de programmation

➤ La programmation séquentielle / procédurale :

- Cette méthode de programmation dissocie fortement les données et les traitements et chaque instruction se réalise indépendamment des données. Il n'y a pas de corrélation directe entre les deux.
- Les fonctions créées permettent de structurer le code et de le rendre réutilisable (concept de factorisation)
- La logique de programmation est basée sur l'enchaînement d'instructions

Introduction à la P00

■ Deux logiques de programmation

➤ La programmation Orientée Objet :

- Cette méthode permet de regrouper des informations et des traitements en un « **bloc** »
- La logique est basée sur la création d'entités et non sur l'enchaînement d'instructions

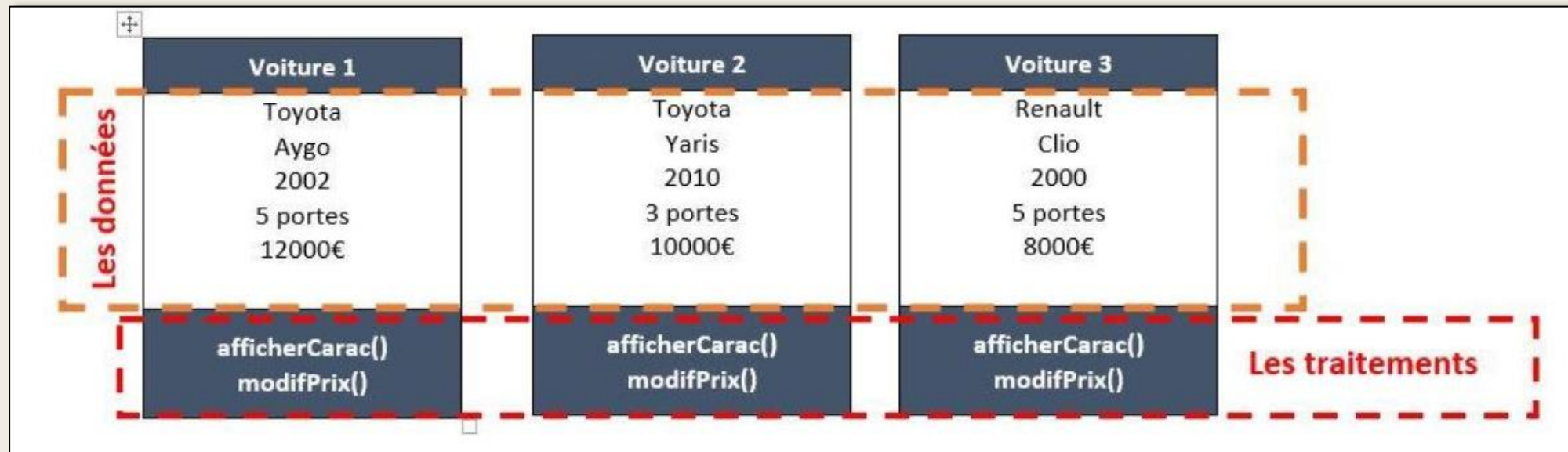
La logique de la P00

- En P00, on va se poser des questions sur les éléments dans le « **bloc** ».
- **Exemple :** Vous souhaitez créer une application qui gère une concession automobile et va permettre aux clients d'acheter des voitures.
 - *En P00, vous allez commencer par vous poser la question : quelles sont les entités qui auront un « **bloc** » dans l'application ?*
 - Les voitures
 - Les clients
 - Le parc automobile
 - Quelles sont les informations dont j'aurai besoin sur chacune de mes entités?
 - Voitures : modèle, marque, plaque d'immatriculation...
 - Client : nom, prénom, adresse, téléphone.....
 - Parc automobile : adresse, nom, téléphone....

La logique de la P00

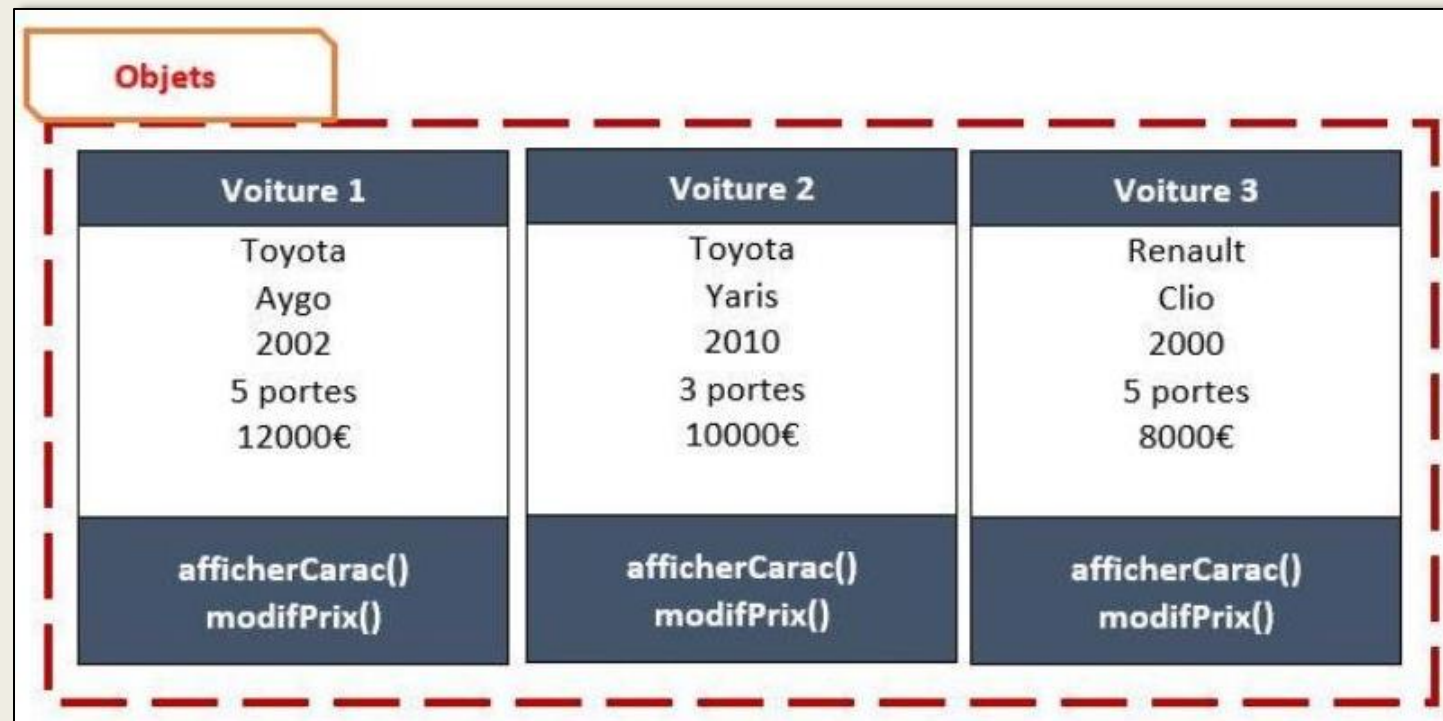
- Quelles sont les traitements associés à ces entités ?
 - Voitures : Afficher les caractéristiques, modifier l'état du véhicule, modifier le prix....
 - Client : réserver, modifier les informations personnelles, commander....
 - Parc automobile: lister les voitures, ajouter une voiture, supprimer une voiture.....
- On va créer des blocs logiques qui donneront des « objets »

Exemple pour les voitures :



La logique de la P00

- De cette logique objet, on va pouvoir créer des « moules » permettant de grouper la structure des informations (données + traitements)
- Cette structure sera appelée une **classe**



Des variables aux objets

- En programmation procédurale nous utilisons des variables pour véhiculer des informations.
- Par exemple pour une voiture, on va décrire ses informations comme :
 - *Marque*
 - *Modèle*
 - *Plaque d'immatriculation*
 -

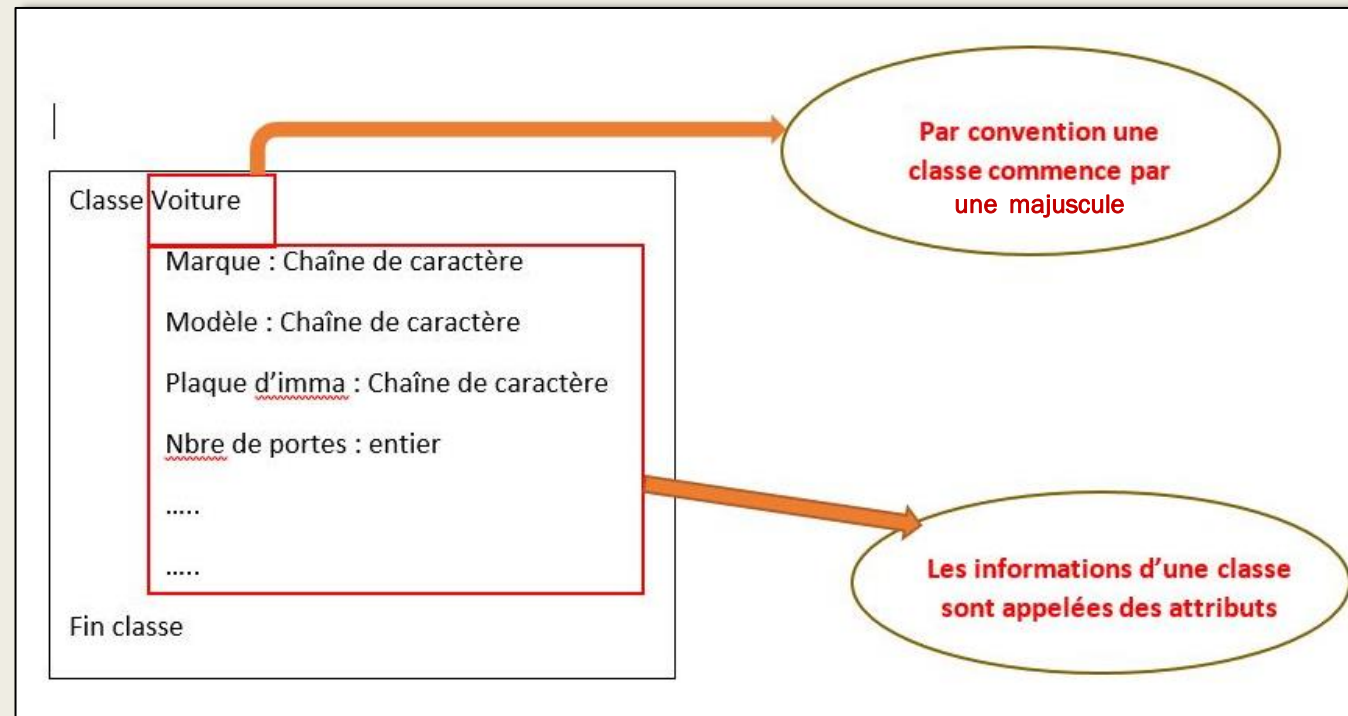


Des variables aux objets

- Si nous ne disposons que d'un seul personnage il est « simple » à gérer, puisque nous n'avons à stocker qu'une variable par information.
- Mais si nous avons 3 personnages disposant des même types d'informations, il va devenir très compliqué à gérer et on pourrait créer des incohérences ou des erreurs dans nos programmes.
- Pour gérer l'ensemble de ces informations, il nous faut donc disposer de (nbre de) variables multipliés par le nombre de personnages que nous avons.

Des variables aux objets

- Pour simplifier l'utilisation de toutes ces informations, on va utiliser une « super variable » qui s'appelle objet
- Et pour créer un objet, on va devoir créer une structure, cette structure s'appelle **une classe**



Propriétés et Méthodes

- **Propriétés (attributs)** : Les propriétés sont des variables définies dans une classe qui représentent les caractéristiques d'un objet.
- **Méthodes** : Les méthodes sont des fonctions définies dans une classe qui décrivent les comportements d'un objet.

Encapsulation

- L'encapsulation est le concept de restreindre l'accès direct à certaines composantes d'un objet et de les protéger contre toute modification non autorisée. Cela se fait en utilisant des modificateurs de visibilité :
 - **Public** : La propriété ou méthode est accessible de partout.
 - **Protected** : La propriété ou méthode est accessible dans la classe où elle est définie et dans les classes héritées.
 - **Private** : La propriété ou méthode est accessible uniquement dans la classe où elle est définie.

Création d'objet : Constructeur

- Pour créer des objets à partir d'une classe, il nous faut définir un constructeur
- Un constructeur est une fonction intégré comme l'élément d'une classe
- L'instanciation est la création et l'initialisation d'un objet. On parle donc « **d'instanciation** » de la classe Personnage.

Création d'objet : Constructeur

```
Plante.class.php
1  <?php
2
3  class Plante {
4      public $nom;
5      public $type;
6      public $hauteur;
7      public $dureeDeVie;
8      public $famille;
9
10     public function __construct($nom, $type, $hauteur, $dureeDeVie, $famille) {
11         $this->nom = $nom;
12         $this->type = $type;
13         $this->hauteur = $hauteur;
14         $this->dureeDeVie = $dureeDeVie;
15         $this->famille = $famille;
16     }
17
18     public function afficherDetails() {
19         echo "Nom: $this->nom\n" . "<br>";
20         echo "Type: $this->type\n" . "<br>";
21         echo "Hauteur: $this->hauteur\n" . "<br>";
22         echo "Durée de vie: $this->dureeDeVie\n" . "<br>";
23         echo "Famille: $this->famille\n" . "<br>";
24         echo "*****<br>";
25     }
26 }
27
```

```
index.php x  Plante.class.php
index.php
1  <?php
2  require_once "Plante.class.php";
3
4  $chene = new Plante("Le chêne", "arbre", "20 mètres", "100 ans", "Fagacées");
5  $rose = new Plante("La rose", "fleur", "1 mètre", "2 ans", "Rosacées");
6  $tournesol = new Plante("Le tournesol", "fleur", "3 mètres", "1 an", "Astéracées");
7
8  $chene->afficherDetails();
9  $rose->afficherDetails();
10 $tournesol->afficherDetails();
11
```

Visibilité des informations : Public/Private & Getter / Setter

- Donc maintenant la déclaration des attributs ne sera plus « public » mais « private »
- On ne pourra plus accéder aux attributs de la classe

```
3  class Plante {  
4      private $nom;  
5      private $type;  
6      private $hauteur;  
7      private $dureeDeVie;  
8      private $famille;  
9  }
```

- Pour pallier à ce problème, on va utiliser des « **getter** » et « **setter** »

Visibilité des informations : Public/Private & Getter / Setter

- « **Getter** » : Un attribut « **private** » n'est accessible ni en lecture, ni en écriture. Nous allons donc utiliser des « **getter** », ce sont des méthodes qui nous permettent d'accéder aux données des ces attributs.

```
3  class Plante {
4      private $nom;
5      private $type;
6      private $hauteur;
7      private $dureeDeVie;
8      private $famille;
9
10     public function __construct($nom, $type, $hauteur, $dureeDeVie, $famille) {
11         $this->nom = $nom;
12         $this->type = $type;
13         $this->hauteur = $hauteur;
14         $this->dureeDeVie = $dureeDeVie;
15         $this->famille = $famille;
16     }
17
18     public function getNom() {return $this->nom; }
19     public function getType() {return $this->type;}
20     public function getHauteur() { return $this->hauteur;}
21     public function getDureeDeVie() {return $this->dureeDeVie;}
22     public function getFamille() {return $this->famille;}
23
24 }
```


Visibilité des informations : Public/Private & Getter / Setter

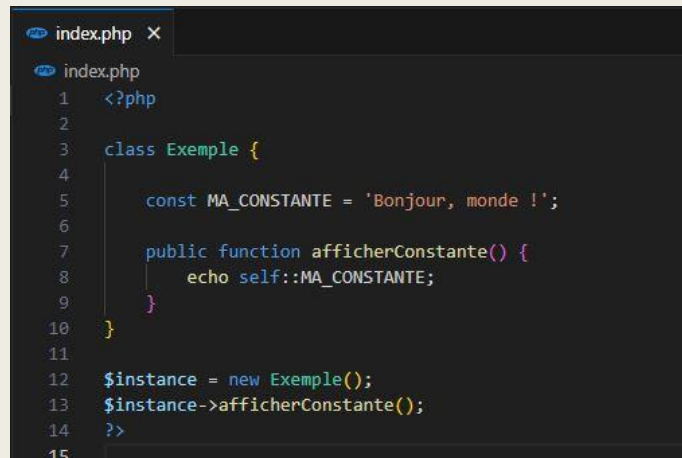
- « **Setter** » : Ce sont des méthodes qui ne servent qu'à une chose, changer la valeur d'un des attributs de la classe.
- En fait quand on met un attribut en « **private** », on n'y a plus accès depuis l'extérieur de la classe. Par contre, il reste accessible à l'intérieur de la classe. Un « **setter** » permet alors de retourner l'objet lui-même avec un « **return \$this** ».

```
3 class Plante {
4     private $nom;
5     private $type;
6     private $hauteur;
7     private $dureeDeVie;
8     private $famille;
9
10    public function __construct($nom, $type, $hauteur, $dureeDeVie, $famille) {
11        $this->nom = $nom;
12        $this->type = $type;
13        $this->hauteur = $hauteur;
14        $this->dureeDeVie = $dureeDeVie;
15        $this->famille = $famille;
16    }
17
18    //getter
19    public function getNom() {return $this->nom; }
20    public function getType() {return $this->type;}
21    public function getHauteur() { return $this->hauteur;}
22    public function getDureeDeVie() {return $this->dureeDeVie;}
23    public function getFamille() {return $this->famille;}
24
25    //setter
26    public function setNom($nom) {
27        $this->nom = $nom;
28    }
29    public function setType($type) {
30        $this->type = $type;
31    }
32    public function setHauteur($hauteur) {
33        $this->hauteur = $hauteur;
34    }
35    public function setDureeDeVie($dureeDeVie) {
36        $this->dureeDeVie = $dureeDeVie;
37    }
38    public function setFamille($famille) {
39        $this->famille = $famille;
40    }
41}
```

```
index.php
1 <?php
2 require_once "Plante2.class.php";
3
4 $chene = new Plante("Le chêne", "arbre", "20 mètres", "100 ans", "Fagacées");
5 $rose = new Plante("La rose", "fleur", "1 mètre", "2 ans", "Rosacées");
6 $tournesol = new Plante("Le tournesol", "fleur", "3 mètres", "1 an", "Astéracées");
7
8 $chene->afficherDetails();
9 $rose->afficherDetails();
10 $tournesol->afficherDetails();
11
12 $chene->setHauteur("50 mètres");
13 $chene->afficherDetails();
14 ?>
15 |
```

Les constantes de classe:

- Pour définir une constante de classe, on va utiliser le mot clé **const** suivi du nom de la constante en majuscules. On ne va pas utiliser ici de signe \$ comme avec les variables.
- Par défaut (si rien n'est précisé), une constante sera considérée comme publique et on pourra donc y accéder depuis l'extérieur de la classe dans laquelle elle a été définie.
- Depuis la version 7.1 de PHP, on peut définir une visibilité pour nos constantes (**public**, **private** et **protected**)

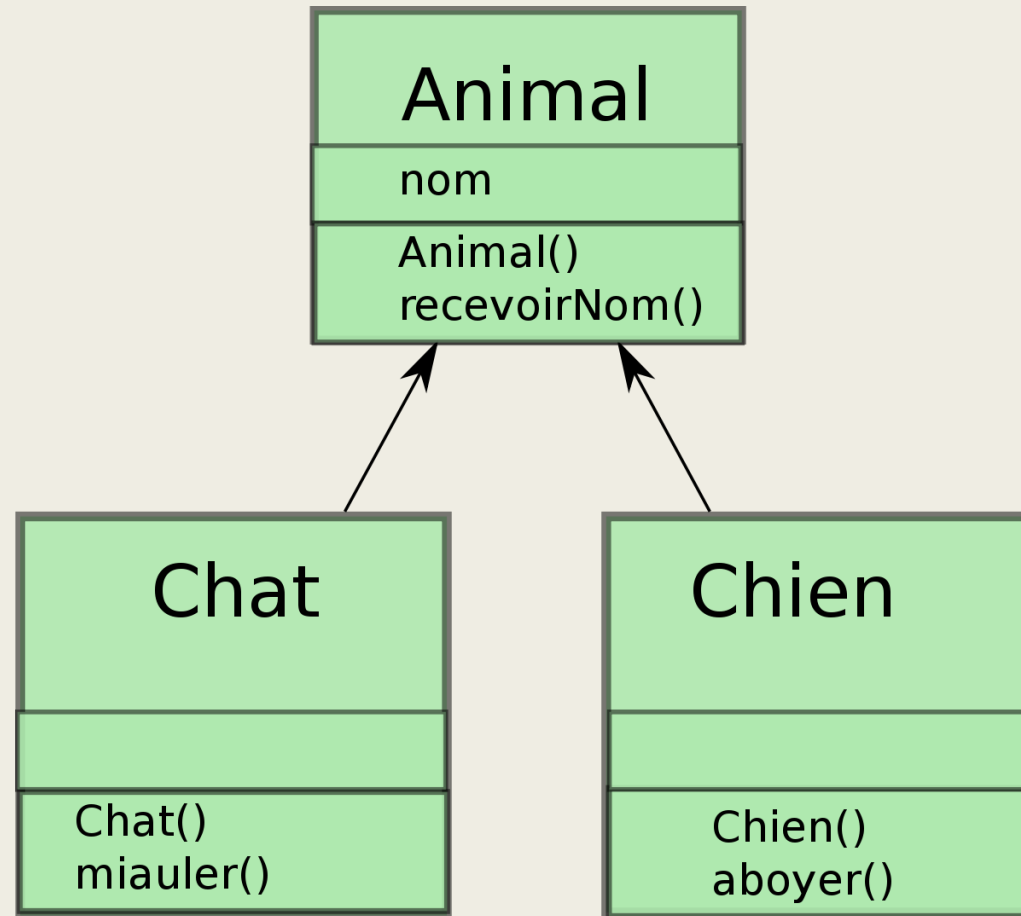


```
index.php x
index.php
1  <?php
2
3  class Exemple {
4
5      const MA_CONSTANTE = 'Bonjour, monde !';
6
7      public function afficherConstante() {
8          echo self::MA_CONSTANTE;
9      }
10 }
11
12 $instance = new Exemple();
13 $instance->afficherConstante();
14 ?>
15
```

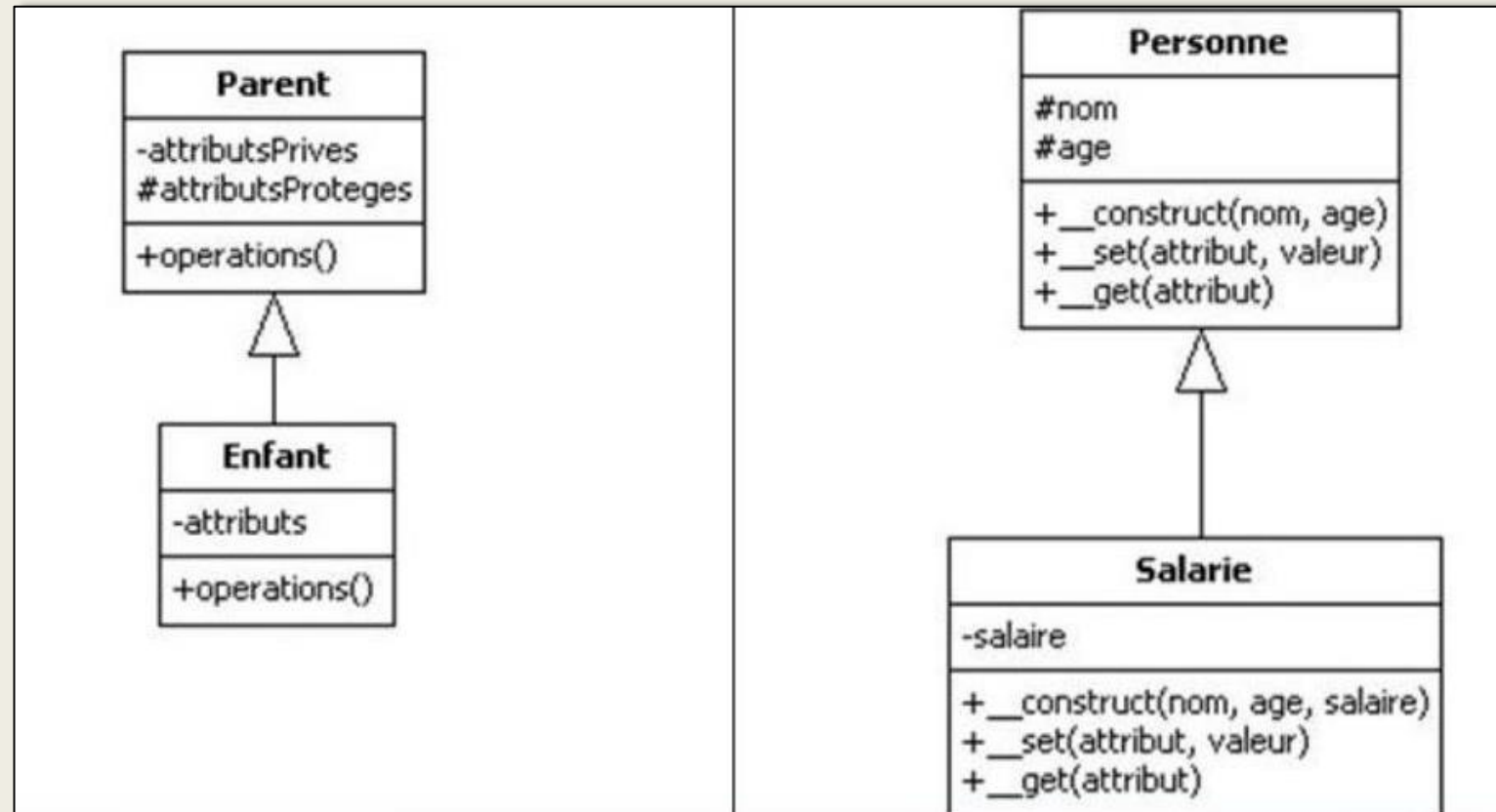
Héritage

- L'héritage est cette possibilité pour une classe d'hériter des attributs et méthodes d'une classe parent .
- L'héritage est une spécialisation/généralisation parce qu'il permet à une classe dérivée(ou enfant) d'hériter des propriétés et méthodes d'une classe de base (parent), tout en ajoutant ou en modifiant des fonctionnalités spécifiques
- La classe enfant hérite donc des attributs et méthodes du parent (mais seuls les attributs public et protected sont accessibles directement à partir des descendants) et possède elle-même ses propres attributs et méthodes

Héritage : Exemple



Héritage : Exemple



Classes abstraites, interfaces

- Une **classe abstraite** en PHP est une classe qui ne peut pas être instanciée directement. Elle est utilisée comme modèle pour d'autres classes. Une classe abstraite peut contenir des méthodes abstraites et des méthodes concrètes. Les méthodes abstraites sont des méthodes déclarées dans la classe abstraite sans implémentation, et les sous-classes doivent les implémenter.

Classes abstraites, interfaces

- Les caractéristiques d'une classe abstraite :
 1. **Non instanciable** : Vous ne pouvez pas créer une instance d'une classe abstraite directement.
 2. **Peut contenir des méthodes abstraites** : Une méthode abstraite est une méthode déclarée sans implémentation.
 3. **Peut contenir des méthodes concrètes** : Les classes abstraites peuvent aussi avoir des méthodes avec une implémentation que les sous-classes peuvent utiliser ou redéfinir.
 4. **Doit être héritée** : Une classe abstraite est destinée à être étendue par d'autres classes qui implémentent les méthodes abstraites.

Classes abstraites, interfaces

- Une interface ne contient que des déclarations de méthodes publiques sans implémentation.
- Les classes qui implémentent une interface doivent fournir des implémentations pour toutes les méthodes définies par l'interface.
- Elles ne peuvent pas contenir de méthodes avec des implémentations, y compris les constructeurs.