# LA PROGRAMMATION ORIENTÉE OBJET EN PHP

Jean-Claude AZIAHA

# SOMMAIRE

**PARTIE A**: Introduction

1- Qu’est-ce que la POO ?

2- Quelle est son utilité ?

3- Quel environnement de travail pour coder en POO ?

# SOMMAIRE

**PARTIE B** : Les bases de la poo

# SOMMAIRE

**PARTIE B**: Les bases de la poo

# INTRODUCTION

# 1- Qu'est-ce que la programmation orientée objet

La programmation orientée objet n'est pas un langage de programmation.

C'est un paradigme de programmation.

C’est-à-dire que c'est une manière de programmer dont le point central est l'utilisation des objets.

# 2- Quelle est son utilité

- La programmation orientée objet va nous permettre de mieux organiser notre code.
- Nous pourrons grâce aux objets, regrouper les informations, en plusieurs blocs solides et de manière sécurisée.
- Très utile lors du développement de gros projets car qui dit gros projet dit beaucoup de lignes de code.
- Il faut donc une bonne architecture et une bonne organisation.

# 3- Quel environnement de travail pour coder en POO ?

- L’environnement de travail demeure le même que lors de l’écriture du code PHP en mode procédural.
- Tous les éléments de bases requis pour programmer en orienté objet sont donc déjà présents.
- C’est-à-dire un serveur(Apache), PHP, MySQL, un éditeur de texte.

# LES BASES DE LA POO

# 1- Qu'est-ce donc qu'un objet ?

Un objet en programmation est une variable que l’on obtient grâce à l’instanciation d’une classe.

Cet objet peut contenir à la fois des propriétés(attributs) et des méthodes.

# 1- Qu'est-ce donc qu'un objet ?

- Les propriétés représentent ses attributs.
  Elles peuvent être de type scalaire (int, string…) et/ou de type composé (tableau, objet).

- Les méthodes sont en réalité des fonctions qui représentent les actions que peut effectuer cet objet.

# 1- Qu'est-ce donc qu'un objet ?

- Nous venons de dire que l’on obtient un objet à partir de l’instanciation d’une classe.
- Prenons donc le temps de comprendre ce que c’est qu’une classe,
- Puis, nous verrons comment créer des objets à partir de cette classe.

# 2- Qu'est-ce qu'une classe ?

- La classe, c'est comme une maquette sur laquelle l’on se base pour créer le nombre d’objets que l’on souhaite.

- Elle détermine donc ce qu’il sera possible de faire avec l’objet.

- Pour déclarer une classe en PHP, il faut utiliser le mot clé **class** suivi du nom de la classe (Le nom de la classe commence par une lettre majuscule, par convention).

# 2- Qu'est-ce qu'une classe ?

```php
<?php
    class Robot
    {

    }
?>
```

# 3- Comment instancier une classe

Instancier une classe revient à créer un objet.

La formule est de mettre le mot clé **new** suivi du nom de classe.

Exemple : new Robot;

Afin de le manipuler plus facilement dans la suite du code, nous allons stocker cet objet dans une variable. La syntaxe devient donc

$robot0 = new Robot;

# 3- Comment instancier une classe

```php
<?php
    class Robot
    {

    }

    $robot1 = new Robot;
?>
```

# 4- Que peut contenir une classe ?

Une classe contient :

- des **attributs** qui sont des variables ou des constantes
- des **méthodes** qui sont des fonctions

Il faut savoir que pour déclarer un attribut ou une méthode dans une classe, il faut obligatoirement lui **préciser le mode de visibilité**.

La définition de ce mode de visibilité est communément appelée : **l'encapsulation**.

# 5- L'encapsulation

L'**encapsulation est** un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en empêchant l'accès aux données .

L' **encapsulation**, comme dit précédemment, c'est le mode de visibilité d'un attribut et/ou d'une méthode appartenant à une classe.

# 5- L'encapsulation

Il existe 3 modes:

- **public**: C’est le mode le plus ouvert. Les attributs et/ou les méthodes qui sont déclarés en mode public dans une classe sont accessibles partout. Que cela soit dans la classe elle-même ou en dehors.

# 5- L'encapsulation

**protected**: C’est le mode moyennement restrictif.

C’est-à-dire qu’en déclarant des attributs ou des méthodes en mode protected, ces derniers sont accessibles uniquement:

- Dans la classe elle-même
- Ses classes filles (C’est-à-dire les classes qui vont hériter de la classe parente).

# 5- L'encapsulation

- **private**: C’est le mode le plus restrictif. Comme le dit son nom, c’est privé! Cela veut donc dire que les attributs et/ou les méthodes qui sont déclarés en mode private dans une classe sont accessibles uniquement dans la classe qui les a définis.

# 5- L'encapsulation

```php
<?php

class Vehicule
{
    public $marque = "Audi";
    protected $couleur = "bleue";
    private $nitro = true;
}

$vehicule0 = new Vehicule;
echo $vehicule0->marque . "</br/>"; // Audi
echo $vehicule0->couleur . "</br/>";
// Fatal error: Uncaught Error: Cannot access protected property
```

# 5- L'encapsulation

```php
<?php
    class Vehicule
    {
        public $marque = "Audi";
        protected $couleur = "bleue";
        private $nitro = true;

        public function demarrer()
        {
            echo "Vroum!!! Elle démarre";
        }

        protected function rouler() // or private function
        {
            echo "Elle roule";
        }
    }

    $vehicule0 = new Vehicule;
    echo $vehicule0->demarrer() . "</br/>"; // Vroum!!! Elle démarre
    echo $vehicule0->rouler() . "</br/>";
    // Fatal error: Uncaught Error: Call to protected method Vehicule::rouler()
```

# 5- L'encapsulation

Au final, faut-il déclarer les attributs d’une classe en public, protected ou private ?

**Cela dépendra du besoin.**

Toutefois, de manière générale, il est plutôt conseillé de déclarer les **attributs** d’une classe en **privé** pour des **raisons de sécurité**.

**Comment** fait-on alors pour y **accéder** en dehors de la classe puisque les **attributs** sont **privés**?

# 6- Les getters et setters

C’est là qu’entre en jeu les **getters** et les **setters**.

Les **getters** sont donc des méthodes qui sont déclarées en public dans une classe.

C’est grâce aux **getters** que nous allons récupérer les attributs déclarés en **protected** ou **private**

Prenons un exemple pour mieux éclaircir tout ça.

# 5- L'encapsulation

index.php > ...

```php
<?php
    class Vehicule
    {
        public $marque = "Audi";
        protected $couleur = "bleue";

        public function getMarque()
        {
            return $this->marque;
        }

        public function getCouleur()
        {
            return $this->couleur;
        }
    }

    $vehicule0 = new Vehicule;
    echo $vehicule0->getMarque() . "\n";  // Audi
    echo $vehicule0->getCouleur() . "\n"; //bleue

```

# 6- Les getters et setters

Les **setters** sont des méthodes qui seront déclarées en public dans une classe.

C’est grâce aux setters que nous allons modifier la valeur d'une propriété de la classe.

Prenons un exemple pour mieux éclaircir tout ça.

# 6- Les getters et setters

index.php > ...

```php
<?php
    class Vehicule
    {
        public $marque = "Audi";
        protected $couleur = "bleue";

        public function setMarque($marque)
        {
            $this->marque = $marque;
        }

        public function setCouleur($couleur)
        {
            $this->couleur = $couleur;
        }

        public function getMarque()
        {
            return $this->marque;
        }

        public function getCouleur()
        {
            return $this->couleur;
        }
    }
```

# 6- Les getters et setters

```
27
28      $vehicule0 = new Vehicule;
29      $vehicule0->setMarque("Tesla") . "\n";
30      $vehicule0->setCouleur("blanche") . "\n";
31
32      echo $vehicule0->getMarque() . "\n"; //Tesla
33      echo $vehicule0->getCouleur() . "\n"; //blanche
34
```

# 6- Les getters et setters

Dans cet exemple, nous avons modifié la valeur des attributs $marque et $couleur de notre classe **Vehicule**.

Bien que les attributs soient déclarés en privé, nous avons pu réaliser cette opération grâce aux setters **setMarque**(){} et **setCouleur**(){}.

Afin d'afficher la nouvelle valeur que contient $marque et $couleur, nous avons utilisé les getters **getMarque**(){} et **getCouleur**(){}

# 7- $this

Nous avons commencé depuis le chapitre sur les getters et les setters à utiliser **$this**.

**$this** lorsqu'il est utilisé dans une classe **désigne l'objet courant**.
C'est-à-dire l'objet qui sera généré à partir de cette classe.

**$this** est utilisé pour désigner un attribut et/ou une méthode au sein d'une classe, en faisant référence à l'objet issu de cette classe.

# 7- $this

index.php > ...

```php
<?php
    class Vehicule
    {
        public $marque = "Audi";
        protected $couleur = "bleue";

        public function getMarque()
        {
            return $this->marque;
        }

        public function getCouleur()
        {
            return $this->couleur;
        }

        public function getPhrase()
        {
            echo "La voiture est de marque: " . $this->marque . " et sa couleur est " . $this->couleur . ".\n";
        }
    }
    $vehicule0 = new Vehicule();
    $vehicule0->getPhrase() . "\n"; //La voiture est de marque: Audi et sa couleur est bleue.
```

# 8- Le constructeur

Jusqu’ici, dans notre classe Vehicule, les attributs ont toujours contenu des valeurs par défaut.

Pour être plus clair, à chaque fois qu'on créé un objet sur la base de la classe Vehicule, cet objet aura toujours par défaut une marque « Audi » et une couleur « bleue ».

# 8- Le constructeur

Si notre classe est censée être une maquette sur laquelle on peut se baser pour créer plusieurs véhicules, il serait plus intéressant d'avoir la possibilité de définir la marque, ainsi que la couleur nous-même lors de la création de cet objet.

C'est dans ce sens qu'intervient la fonction magique **__construct()**

Encore appelée, le constructeur d'une classe.

# 8- Le constructeur

Le constructeur est une fonction magique dans le sens où c'est elle qui est appelée en premier à chaque fois que l'on instancie une classe.

Cette fonction existe par défaut dans une classe.

Elle est donc toujours là, que l’on la déclare nous-même ou non.

La particularité des fonctions magiques est que leur nom est précédé de __

Cette fonction peut prendre des paramètres ou non.

Commençons par la créer sans paramètres.

# 8- Le constructeur

index.php > ...

```php
<?php
    class Vehicule
    {
        public $marque;
        protected $couleur;

        public function __construct()
        {

        }
        public function getMarque(){return $this->marque;}
        public function getCouleur(){return $this->couleur;}

        public function setMarque($marque){$this->marque = $marque;}
        public function setCouleur($couleur){$this->couleur = $couleur;}
    }

    $vehicule0 = new Vehicule();
    $vehicule0->setMarque("Renault");
    echo $vehicule0->getMarque() . "\n";

```

# 8- Le constructeur

Dans cet exemple, le constructeur n’a pas de paramètres.

$vehicule0 = new Vehicule;

Ou

$vehicule0 = new Vehicule();

Ces 2 déclarations veulent dire la même chose mais on préferera la seconde.

Dans l’exemple qui va suivre,
nous allons maintenant ajouter des paramètres.

# 8- Le constructeur

index.php > ...

```php
<?php
    class Vehicule
    {
        private $marque;
        private $couleur;

        public function __construct($marque, $couleur)
        {
            $this->marque = $marque;
            $this->couleur = $couleur;
        }
        public function getMarque(){return $this->marque;}
        public function getCouleur(){return $this->couleur;}
    }

    $vehicule0 = new Vehicule("Renault", "grise");
    echo $vehicule0->getMarque() . "\n"; //Renault
    echo $vehicule0->getCouleur() . "\n"; //grise

```

# 8- Le constructeur

Le constructeur de la classe s’attend donc à prendre en paramètres et respectivement, la marque et la couleur du véhicule.

Lors de la création d’un objet sur la base de la classe Vehicule, il faut donc obligatoirement lui passer en paramètres ces 2 éléments et dans le bon ordre.

Ensuite, on récupère les valeurs passées en paramètres et on les sauvegarde dans les attributs concernés.

# 8- Le constructeur

index.php > ...

```php
<?php
    class Vehicule
    {
        private $marque;
        private $couleur;

        public function __construct($marque, $couleur)
        {
            $this->marque = $marque;
            $this->couleur = $couleur;
        }
        public function getMarque(){return $this->marque;}
        public function getCouleur(){return $this->couleur;}
    }

    $vehicule0 = new Vehicule("Renault", "grise");
    $vehicule1 = new Vehicule("Peugeot", "bleue");
    $vehicule2 = new Vehicule("Ferrari", "rouge");

    echo $vehicule0->getMarque() . "\n"; //Renault
    echo $vehicule1->getMarque() . "\n"; //Peugeot
    echo $vehicule2->getMarque() . "\n"; //Ferrari
```

# 9- Les méthodes et attributs statiques

Ce sont des méthodes qui agissent sur la classe et non sur l'objet.

On ne retrouvera donc **jamais** l'opérateur **$this** dans ce type de méthodes.

Pour appeler une méthode statique, l'opérateur utilisé est le double deux points **::**

Il est également possible d'appeler cette méthode sur un objet.

# 9- Les méthodes et attributs statiques

```php
<?php
    class Vehicule
    {
        private $marque;
        private $couleur;
        private $vitesse;
        private $nb_roues;
        private $proprietaire;

        public static function rouler()
        {
            echo "Ce vehicule peut rouler \n";
        }
    }
    Vehicule::rouler(); //Ce vehicule peut rouler

    $vehicule1 = new Vehicule;
    $vehicule1->rouler(); //Ce vehicule peut rouler
?>
```

# 9- Les méthodes et attributs statiques

index.php > ...

```php
<?php
    class Vehicule
    {
        private $marque;
        private $couleur;
        private static $nbre_instances = 0;

        public function __construct($marque, $couleur)
        {
            $this->marque = $marque;
            $this->couleur = $couleur;
            self::$nbre_instances++;
        }
        public function getMarque(){return $this->marque;}
        public function getCouleur(){return $this->couleur;}
        public static function getNbreInstances(){return self::$nbre_instances;}
    }

    $vehicule0 = new Vehicule("Renault", "grise");
    $vehicule1 = new Vehicule("Peugeot", "bleue");
    $vehicule2 = new Vehicule("Ferrari", "rouge");
    echo Vehicule::getNbreInstances(); //3
    $vehicule0->getNbreInstances();
```

# 10- L'héritage

En poo, l’héritage consiste à créer une nouvelle classe (enfant) à partir d’une classe existante (parente). Ce qui permet de réutiliser les attributs et les méthodes de la classe parente.

On dit que cette nouvelle classe **hérite** de la classe mère.

Une classe mère peut avoir plusieurs filles mais une classe fille ne peut avoir qu’une classe mère en PHP.

# 10- L'héritage

index.php > ...

```php
<?php

    class Vehicule
    {
        private $marque = "Ferrari";

        public function getMarque()
        {
            return $this->marque;
        }
    }

    class Voiture extends Vehicule
    {

    }

    $voiture0 = new Voiture;
    echo $voiture0->getMarque(); // Ferrari

```

# 10- L'héritage

index.php > ...

```php
<?php
    class Vehicule
    {
        private $marque = "Ferrari";

        public function getMarque()
        {
            return $this->marque;
        }
    }

    class Voiture extends Vehicule{}

    class Moto extends Vehicule{}

    $voiture0 = new Voiture;
    echo $voiture0->getMarque() . "\n"; // Ferrari

    $moto0 = new Moto;
    echo $moto0->getMarque() . "\n"; // Ferrari

```

# 10- L'héritage

Dans cet exemple, Voiture et Moto sont des classes enfants et qui ont pour parent la classe Vehicule. Elles ont donc hérité de tous les attributs et méth odes de la classe Véhicule.

L’héritage a été réalisé grâce au mot clé **extends**.

Voiture et Moto possèdent désormais :

* l’attribut $marque ainsi que sa valeur
* La méthode getMarque();

# 10- L'héritage

Le concept d’héritage ne s’arrête pas à la duplication du contenu d'une classe.

Là où cela devient intéressant, c’est que la classe fille peut ajouter ses propres attributs et méthodes (en plus de celles dont elle a hérité).

# 10- L'héritage

```php
<?php
    class Vehicule
    {
        private $marque = "BMW";
        private $couleur;
        private $nb_roues;

        public function getMarque()
        {
            return $this->marque;
        }
    }

    class Voiture extends Vehicule{
        private $position_volant = "gauche";
        public function getPositionVolant()
        {
            return $this->position_volant;
        }
    }

    class Moto extends Vehicule{
    }

    $voiture1 = new Voiture;
    $moto1 = new Moto;

    echo $voiture1->getPositionVolant(); //gauche
    echo $moto1->getPositionVolant(); //Error: Call to undefined method Moto::getPositionVolant()
```

# 11- Redéfinition d'une méthode ou d'un attribut dans une classe

- La classe **parente A** possède une méthode **m().**
- La classe **fille A1** qui **hérite de**
  **A** possède automatiquement la même méthode **m().**

La classe fille **A1** peut modifier le contenu de sa méthode **m().**

On parle alors de redéfinition ou de surcharge.

# 11- Redéfinition d'une méthode ou d'un attribut dans une classe

```php
<?php
    class Vehicule{
        private $marque;
        private $couleur;
        private $nb_roues;

        public function afficher_Message()
        {
            echo "Ceci est un véhicule. ";
        }
    }

    class Voiture extends Vehicule
    {
        public function __construct($marque_param)
        {
            $this->marque = $marque_param;
        }

        public function afficher_Message()
        {
            parent::afficher_Message();
            echo "Plus précsement, une voiture de marque: " . $this->marque . ".";
        }
    }

    $voiture1 = new Voiture("Fiat");
    $voiture1->afficher_Message();
    //Ceci est un véhicule. Plus précsement, une voiture de marque: Fiat.
?>
```

# 12- Classes et méthodes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée.

C’est à dire que **l’on ne peut pas faire**: $obj = **new** ClasseAbstraite();

On ne peut donc pas créer des objets à partir d’une classe.

Une classe abstraite peut avoir des méthodes abstraites et des méthodes non abstraites.

Il n’est pas possible de restreindre le mode de visibilité d'une méthode abstraite dans sa classe fille.

# 12- Classes et méthodes abstraites

```php
<?php
    abstract class Vehicule
    {
        abstract function typeVehicule();
    }

    class Voiture extends Vehicule
    {
        public function typeVehicule()
        {
            echo "Ceci est un vehicule de type Voiture";
        }
    }

    $voiture1 = new Voiture();
    echo $voiture1->typeVehicule(); //Ceci est une voiture et de marque: BMW
```

# 13- Classes finales

Comme son nom l’indique, une classe finale ne peut pas avoir de classes filles.

Les méthodes finales ne peuvent pas être redéfinies ou surchargées.

Au niveau de la syntaxe, il faut simplement ajouter le mot clé **final** devant les classes ou les fonctions désirées finales.

# 13- Classes finales

```php
<?php

    class Vehicule
    {

    }

    final class Moto extends Vehicule
    {

    }

    class MotoEnfant extends Moto
    {

    }
    //Fatal error: Class MotoEnfant may not inherit from final class (Moto)
```

# 14- Les interfaces

Les interfaces peuvent être considérées comme des classes totalement abstraites.

Elles ne peuvent donc pas être instanciées.

Toutes les méthodes d’une interface doivent être publiques et redéfinies au sein des classes dans lesquelles l’interface est implémentée.

Les méthodes que contient une interface ne peuvent être ni abstraites ni finales.

# 11- Les interfaces

```php
<?php
interface Actions
{
    public function enAutomatique();
}

class Vehicule
{
    protected $marque;
    protected $couleur;
    protected $proprietaire;
}

class Voiture extends Vehicule implements Actions
{
    public function enAutomatique()
    {
        echo "Boite de vitesse automatique.";
    }
}

$voiture1 = new Voiture();
echo $voiture1->enAutomatique(); //Boite de vitesse automatique.
```

# 11- Les traits

Les traits représentent un mécanisme qui permet **d'intégrer** du **code** d'un fichier **indépendant dans** une ou plusieurs **autres classes**.

C'est un peu comme faire le "**include**" d'un fichier dans un autre.

Les traits ressemblent énormément aux interfaces mais il n'est pas possible de les instancier; Seulement de les utiliser.
Ils regroupent généralement des méthodes réutilisables dans plusieurs classes différentes qui ont des points communs.

# 11- Les traits

```php
<?php
    trait Auteur
    {
        protected $nom_auteur;
        public function getAuteur(){return $this->nom_auteur;}
    }

    class Ecrivain
    {
        use Auteur;
        public function __construct($nom)
        {
            $this->nom_auteur = $nom;
        }
    }

    class Peintre
    {
        use Auteur;
        public function __construct($nom)
        {
            $this->nom_auteur = $nom;
        }
    }
```

# 11- Les traits

```php
<?php

    $ecrivain = new Ecrivain("J. K. Rowling");
    $peintre  = new Peintre("Pierre Soulages");

    echo $ecrivain->getAuteur() . "<br/>";
    echo $peintre->getAuteur()  . "<br/>";
```