PHP: Programmation orientée objet(POO)

Pr. Abdelali El Gourari

Plan

- *Introduction génerale*
- *Classes et Objets*
- *Création d'un objet*
- *Encapsulation*
- *Manipulation d'objets*
- *Manipulation de classes*
- *Héritage*
- *Abstraction*
- *Finalisation*
- *Méthodes magiques*
- *Comparaison*
- *Interfaces*
- *Traits*
- *Design Pattern*
- *Namespace*
- *Exceptions*
- *PHP Data Objects*
- *Methodes Pratiques*

Introduction

- *Historique*
- *Une autre technique de développement, Pourquoi ?*
- *A quels besoins répond la programmation orientée objet ?*
- *Avantages de la programmation orientée objet*
- *Inconvénients de la programmation orientée objet*
- *Unified Modeling Language*
- *Langage php et programmation orientée objet*

Classes et Objets

- *Qu'est-ce qu'un objet ?*
- *Qu'est-ce qu'une classe*
- *Différence entre une classe et un objet*
- *Application concrète de la classe sur l'objet*

Création d'un objet

- *Introduction*
- *Instanciation*
- *Inférence*
- *Transformation*
- *Classe par défaut*

Encapsulation

- *Définition*
- *Pourquoi ?*
- *Concevoir pour réutiliser…comment faire ?*
- *Intérêt de l'encapsulation*
- *Niveau de visibilité*

Manipulation d'objets

- *La pseudo-variable $this*
- *Accesseur (getter) / Mutateurs (setter)*

Manipulation de classes

- *Constantes*
- *Eléments statique*
- *Appartenance à la classe ou a l'objet ?*
- *Opérateur de résolution de portée*
- *Différence entre self:: et $this->*

Héritage

- *Concept*
- *Fonctionnement*
- *Relation et notion d'héritage*
- *Effectuer un héritage, dans quels cas ?*
- *Surcharger une méthode*

Abstraction

- *Définition*
- *Principe et utilisation*
- *Classes abstraites*
- *Methodes abstraites*
- *Informations*

Finalisation

- *Définition*
- *Classes finales*
- *Méthodes finales*

Méthodes magiques

- *Introduction*
- *__construct*
- *__destruct*
- *__set*
- *__get*
- *__call*
- *__tostring*
- *__isset*
- *__sleep*
- *__wakeup*
- *__unset*
- *__invoke*
- *__setstate*
- *__clone*

Comparaison, Traits et Interfaces

Comparaison

- *Les opérateurs*

Trait

- *Introduction*
- *Utilisation*
- *Fonctionnement*

Interfaces

- *Introduction*
- *Principe et utilisation*
- *Différence héritage et implémentation*

HIISTORIIQUE

- *La notion d'objet a été introduite avec le langage de programmation Simula, créé à Oslo entre 1962 et 1967 dans le but de faciliter la programmation de logiciels de simulation.*
- *Avec ce langage de programmation, les caractéristiques et les comportements des objets à simuler sont décrits dans le code source.*
- *Le langage de programmation orientée objet Smalltalk a été créé par le centre de recherche Xerox en 1972.*
- *La programmation orientée objet est devenue populaire en 1983 avec la sortie du langage de programmation C++, un langage orienté objet, dont l'utilisation ressemble volontairement au populaire langage C.*

A QUELS BESOIINS REPOND LA PROGRAMMATIION ORIIENTEE OBJJET ?

Le problème qui se pose quand nous avons besoin d'élaborer des programmes complexes, c'est que nous sommes très vite confrontés aux problèmes suivants :

- *Comment comprendre et réutiliser les programmes faits par d'autres ?*
- *Comment réutiliser les programmes que vous avez écrits il y a plusieurs mois et dont vous avez oublié le fonctionnement ?*

A QUELS BESOIINS REPOND LA PROGRAMMATIION ORIIENTEE OBJJET ?

- *Comment "cloner" rapidement des programmes déjà faits pour des applications légèrement différentes ?*
- *Comment programmer simplement des actions simples sur des éléments variés et complexes ?*
- *Comment ajouter des fonctions sans tout réécrire et tout retester ?*

A QUELS BESOIINS REPOND LA PROGRAMMATIION ORIIENTEE OBJJET ?

Avec une approche fonctionnelle, tout changement de spécification peut avoir des conséquences catastrophiques… Il faut parfois tout refaire de A à Z…

Pour résoudre ces problèmes, il faut développer trois stratégies :

- *Modéliser différemment*
- *Modulariser*
- *encapsuler*

Un bon modèle doit réunir deux qualités :

- *Faciliter la compréhension du programme étudié, en réduisant la complexité*
- *Permettre de simuler le comportement du programme*

Une classe

Une classe est définie en utilisant le mot-clé **class** , suivi du nom de la classe et d'une paire d'accolades (**{}**). Toutes ses propriétés et méthodes vont à l'intérieur des accolades :

```php
<?php
class Fruit {
  // code goes here...
}
?>
```

Ci-dessous, nous déclarons une classe nommée Fruit composée de deux propriétés ($name et $color) et de deux méthodes set_name() et get_name() pour définir et obtenir la propriété $name :

Une classe

```php
<?php
class Fruit {
  // Properties
  public $name;
  public $color;
  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}
?>
```

Remarque : Dans une classe, les variables sont appelées propriétés et les fonctions sont appelées méthodes !

Un objet

Nous pouvons créer plusieurs objets à partir d'une classe. Chaque objet a toutes les propriétés et méthodes définies dans la classe, mais elles auront des valeurs de propriété différentes.

Les objets d'une classe sont créés à l'aide du mot-clé new.

Dans l'exemple ci-contre, $apple et $banana sont des instances de la classe Fruit :

Un objet

```php
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}
```

```php
$apple = new Fruit();
$banana = new Fruit();
$apple->set_name('Apple');
$banana->set_name('Banana');

echo $apple->get_name();
echo "<br>";
echo $banana->get_name();
?>
```

Un objet

Dans l'exemple ci-dessous, nous ajoutons deux autres méthodes à la classe Fruit, pour définir et obtenir la propriété $color :

```php
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
  function set_color($color)
{
    $this->color = $color;
  }
  function get_color() {
    return $this->color;
  }
}
```

Un objet

```php
$apple = new Fruit();
$apple->set_name('Apple');
$apple->set_color('Red');
echo "Name: " . $apple->get_name();
echo "<br>";
echo "Color: " . $apple->get_color();
?>
```

Niveau de visibilité

Publique (public) : est la visibilité la plus large (accessible directement en dehors de la classe).

- Un attribut ou une méthode déclarée comme publique les rendent accessibles de partout, autant depuis l'intérieur de l'objet mais aussi depuis l'extérieur.

Privée (private): est la plus restrictive (accessible uniquement depuis la classe courante).

- Seule la classe elle-même peut accéder à une méthode ou à un attribut.

Protégée (protected) : est intermédiaire.

- Les méthodes et attributs seront visibles par la classe courante, toutes les classes filles, mais inaccessibles en dehors des classes mères/filles.

Le mot-clé **$this**

Le mot-clé $this fait référence à l'objet courant et n'est disponible qu'à l'intérieur des méthodes.

```php
<?php
class car {
  public $name;
}
$BMW = new car();
$BMW->name = “BMW";

echo $BMW->name;
?>
```

Le mot-clé **$this**

Alors, où pouvons-nous changer la valeur de la propriété $name ? Il existe deux façons :

À l'intérieur de la classe (en ajoutant une méthode set_name() et en utilisant $this) :

```php
<?php
class car {
  public $name;
  function set_name($Xname) {
    $this->name = $Xname;
  }
}
```

```php
$BMW = new car();
$BMW->set_name("BMW");

echo $BMW->name;
?>
```

En dehors de la classe (en modifiant directement la valeur de la propriété) :

L'encapsulation

- L'encapsulation peut être utilisée si les propriétés de l'objet sont privées et mises à jour via des méthodes publiques.
- L'encapsulation en PHP peut être réalisée en utilisant l'implémentation de spécificateurs d'accès.
- Il est très prudent quant au concept d'héritage des OOP, car l'héritage peut souvent saper le concept d'encapsulation.
- L'héritage expose certains détails d'une classe parente, rompant efficacement l'encapsulation.

Avantages de l'encapsulation

- Masquage et abstraction des données : les détails inutiles, la représentation interne et la mise en œuvre sont cachés aux utilisateurs finaux pour la protection de la structure des données et de la classe.
- L'accès aux données est interdit aux membres des autres classes en créant des méthodes privées. Il protège l'état interne de tout objet en gardant les variables membres privées et en empêchant tout état incohérent. C'est l'inclusion de données et d'opérations connexes dans cet objet.

Avantages de l'encapsulation

- Sécurité des données : l'encapsulation contribue à rendre les données très robustes et sécurisées, car les données et les fonctions membres sont regroupées pour former un objet. Toutes les tâches sont effectuées à l'intérieur sans aucun souci extérieur et cela rend également la vie très facile.
- Réduit la complexité : l'encapsulation aide à réduire la complexité du développement du logiciel en masquant les détails de la mise en œuvre et en exposant les méthodes ou les opérations.

Avantages de l'encapsulation

- Réutilisabilité : il existe des instances, vous n'avez pas à réécrire les mêmes fonctionnalités que vous avez héritées de la classe parent.
- Fiabilité : vous pouvez rendre la classe en lecture seule ou en écriture seule en écrivant des méthodes SET ou GET.

Avantages de l'encapsulation

- Test plus facile du code : le code PHP encapsulé est facile à tester car les fonctions utilisées pour tester la classe enfant garantissent également le test des fonctions de la classe parent.
- Flexibilité accrue : les variables de classe sont accessibles par les méthodes GET ou SET, ce qui augmente la flexibilité. Il est facile à maintenir car l'implémentation interne peut être modifiée sans changer le code.

Exemple_1

- Les Getters et les setters sont des méthodes utilisées pour définir ou récupérer les valeurs de variables, généralement privées.
- Comme son nom l'indique, une méthode getter est une technique qui permet d'obtenir ou de récupérer la valeur d'un objet. De même, une méthode setter est une technique qui définit la valeur d'un objet.

```php
<?php
    class Person{
        private $name;
        public function setName($name){
            $this->name = $name;
        }
        public function getName(){
            return $this->name;
        }
    }
    $person = new Person();
    echo $person->name;
?>
```

Output: PHP Error Cannot access private property Person::$name

Explication

Dans notre classe Personne ci-dessus, nous avons une propriété privée appelée $name. Comme il s'agit d'une propriété privée, nous ne pouvons pas y accéder directement comme ci-dessus et cela produira une erreur fatale.

```php
<?php
    class Person{
        private $name;
        public function setName($name){
            $this->name = $name;
        }
        public function getName(){
            return $this->name;
        }
    }
    $person = new Person();
    echo $person->name;
?>
```

Exemle_2

Ici, pour donner accès à nos propriétés privées, nous avons créé une fonction "getter" appelée getData, toujours parce que la visibilité des propriétés est réglée sur private, vous ne pourrez pas non plus changer ou modifier leurs valeurs. Vous devez donc utiliser l'une des fonctions "setter" que nous avons créées : setName. Après cela, nous avons instancié notre objet Personne.

```php
<?php
    class Person{
        private $name;
        public function setName($name){
            $this->name = $name;
        }
        public function getName(){
            return 'welocme'. $this->name;
        }
    }
    $person = new Person();
    $person->setName('Alex');
    $name = $person->getName();
    echo $name;
?>
```

Output: welcomeAlex

Conclusion

- La programmation orientée objet en PHP est réalisée en utilisant le concept d'encapsulation qui est utilisé pour cacher des informations. Cela réduit la facilité d'accès aux attributs de la classe actuelle. Les méthodes Getter et Setter sont utilisées pour éviter les accès indésirables externes. Cela aide également à valider les nouvelles valeurs attribuées aux propriétés.
- En bref, l'encapsulation en PHP est le processus de masquage de tous les détails secrets d'un objet qui ne contribuent en fait pas beaucoup aux caractéristiques cruciales de la classe.

Constantes de classe

- ✓ Les **constantes** ne peuvent pas être modifiées une fois déclarées.
- ✓ Les **constantes** de classe peuvent être utiles si vous avez besoin de définir des données constantes dans une classe.
- ✓ Une **constante** de classe est déclarée à l'intérieur d'une classe avec le mot-clé **`const`**.
- ✓ Les **constantes** de classe sont sensibles à la casse. Cependant, il est recommandé de nommer les constantes en toutes lettres majuscules.

Nous pouvons accéder à une constante depuis l'extérieur de la classe en utilisant le nom de la classe suivi de l'opérateur de résolution de portée (`::`) suivi du nom de la constante:

Constantes de classe

```php
<?php
class Constante {

  const Message = "My name is constante";
}

echo Constante:: Message;

?>
```

Constantes de classe

Ou, nous pouvons accéder à une constante depuis l'intérieur de la classe en utilisant le mot-clé **self** suivi de l'opérateur de résolution de portée (::) suivi du nom de la constante:

```php
<?php

class  Constante {
  const Message = " My name is constante ";
  public function test() {
    echo self::Message;
  }
}

$ Constante = new  Constante();
$ Constante ->test();

?>
```

Classes abstraites

- Les classes et méthodes abstraites sont lorsque la classe **Father** a une méthode nommée, mais a besoin de sa ou ses classes **son** pour remplir les tâches.
  - ✓ Une **classe abstraite** est une classe qui contient au moins une méthode abstraite.
  - ✓ Une **méthode abstraite** est une méthode déclarée mais non implémentée dans le code.

Classes abstraites

- Une **classe** ou une **méthode abstraite** est définie avec le mot-clé **abstract** :

```php
<?php
abstract class FatherClass {
  abstract public function test1();
  abstract public function test2($X, $Y);
  abstract public function test3() : string;
}
?>
```

Classes abstraites

- Lors de l'héritage d'une **classe abstraite**, la méthode de la classe **son** doit être définie avec le même nom et le même modificateur d'accès ou un modificateur d'accès moins restreint. S
- Si la méthode abstraite est définie comme protégée, la méthode de la classe **son** doit être définie comme protégée ou publique, mais pas privée.
- De plus, le type et le nombre d'arguments requis doivent être identiques.
- Cependant, les classes **son** peuvent avoir en plus des arguments facultatifs.

Classes abstraites

```php
<?php
// Father class
abstract class Car {
  public $name;
  public function __construct($name) {
    $this->name = $name;
  }
  abstract public function test() : string;
}
```

Classes abstraites

```php
// Child classes
class Audi extends Car {
  public function test() : string {
    return "I'm an $this->name!";
  }
}

class Volvo extends Car {
  public function test() : string {
    return "I'm a $this->name!";
  }
}

class Citroen extends Car {
  public function test() : string {
    return "I'm a $this->name!";
  }
}
```

Classes abstraites

```php
// Créer des objets à partir des classes enfantines
$audi = new audi("Audi");
echo $audi->test();
echo "<br>";

$volvo = new volvo("Volvo");
echo $volvo->test();
echo "<br>";

$citroen = new citroen("Citroen");
echo $citroen->test();
?>
```

Classes abstraites

Les classes **Audi**, **Volvo** et **Citroën** sont héritées de la classe **Car**. Cela signifie que les classes **Audi**, **Volvo** et **Citroën** peuvent utiliser la propriété publique **$name** ainsi que la méthode publique **__construct()** de la classe **Car** en raison de l'héritage.

Mais, **test()** est une **méthode abstraite** qui doit être définie dans toutes les classes enfants et elles doivent renvoyer une chaîne.

Classes abstraites

```php
<?php
abstract class FatherClass {
  // Méthode abstraite avec un  argument
abstract protected function Name($name);
}
class ChildClass extends  FatherClass {
  public function Name($name) {
    if ($name == “Abdelali") {
      $prefix = “Handsome";
    } elseif ($name != “Abdelali") {
      $prefix = “You will be handsome do not worry";
    } else {
      $prefix = "";
    }
    return "{$prefix} {$name}";
  }
}
```

Classes abstraites

```
$class = new ChildClass;
echo $class->Name(“Mehdi");
echo "<br>";
echo $class->Name(“Abdelali");
?>
```

Regardons un autre exemple où la méthode abstraite a un argument, et la classe enfant a deux arguments facultatifs qui ne sont pas définis dans la méthode abstraite du parent :

Classes abstraites

```php
<?php
abstract class FatherClass {
  // Méthode abstraite avec un  argument
  abstract protected function Name($name);
}
class ChildClass extends  FatherClass {
  // La classe enfant peut définir des arguments facultatifs qui ne figurent pas dans la méthode abstraite du parent

  public function Name($name, $separator = ".", $greet = "") {
    if ($name == “Abdelali") {
      $prefix = " Handsome";
    } elseif ($name == “walid") {
      $prefix = " You will be handsome do not worry ";
    } else {
      $prefix = "";
    }
    return "{$greet} {$prefix}{$separator} {$name}";
```

Classes abstraites

```php
$class = new ChildClass;
echo $class->Name(“Walid");
echo "<br>";
echo $class->Name(“Mehdi");
?>
```

Interfaces

Les interfaces vous permettent de spécifier les méthodes qu'une classe doit implémenter.

Les interfaces facilitent l'utilisation d'un grand nombre de classes différentes de la même manière. Lorsqu'une ou plusieurs classes utilisent la même interface, on parle de "**polymorphisme**".

Les interfaces sont déclarées à l'aide du mot-clé **interface** :

Interfaces

```php
<?php
interface InterfaceName {
  public function test1();
  public function test2($name, $color);
  public function test3() : string;
}
?>
```

Interfaces et classes abstraites

Les interfaces sont similaires aux classes abstraites. Les différences entre les interfaces et les classes abstraites sont les suivantes :

- Les **interfaces** ne peuvent pas avoir de **propriétés**, alors que les classes **abstraites** peuvent en avoir.
- Toutes les **méthodes** des interfaces doivent être publiques, alors que les **méthodes** des classes **abstraites** sont publiques ou protégées.
- Toutes les **méthodes** d'une interface sont **abstraites**, elles ne peuvent donc pas être implémentées dans le code et le mot-clé **abstract** n'est pas nécessaire.
- Les classes peuvent **implémenter** une interface tout en **héritant** d'une autre classe.

Utilisation des interfaces

Pour implémenter une interface, une classe doit utiliser le mot-clé **implements**.
Une classe qui implémente une interface doit implémenter toutes les méthodes de l'interface.

```php
<?php
interface Animal {
  public function makeSound();
}

class Cat implements Animal {
  public function makeSound() {
    echo "Meow";
  }
}

$animal = new Cat();
$animal->makeSound();
?>
```

Explication de l'exemple

Dans l'exemple précédent, disons que nous aimerions écrire un logiciel qui gère un groupe d'animaux. Tous les animaux peuvent effectuer certaines actions, mais chacun d'entre eux le fait à sa manière.

En utilisant des interfaces, nous pouvons écrire un code qui fonctionne pour tous les animaux, même si chacun d'entre eux se comporte différemment :

Exemple

```php
<?php
// Interface definition
interface Animal {
  public function makeSound();
}

// Class definitions
class Cat implements Animal {
  public function makeSound() {
    echo " Meow ";
  }
}

class Dog implements Animal {
  public function makeSound() {
    echo " Bark ";
  }
}
```

```php
class Mouse implements Animal {
  public function makeSound() {
    echo " Squeak ";
  }
}
// Create a list of animals
$cat = new Cat();
$dog = new Dog();
$mouse = new Mouse();
$animals = array($cat, $dog, $mouse);

// Tell the animals to make a sound
foreach($animals as $animal) {
  $animal->makeSound();
}
//  The forEach() method executes a
provided function once for each array
element.
?>
```

Explication de l'exemple

Cat, Dog et Mouse sont toutes des classes qui **implémentent** l'interface Animal, ce qui signifie qu'elles sont toutes capables d'émettre un son à l'aide de la méthode **makeSound()**. De ce fait, nous pouvons parcourir en boucle tous les animaux et leur demander d'émettre un son, même si nous ne savons pas de quel type d'animal il s'agit.

Comme l'interface n'indique pas aux classes comment implémenter la méthode, chaque animal peut émettre un son à sa manière.

Caractéristiques

PHP ne prend en charge que l'héritage unique : une classe enfant ne peut hériter que d'un seul parent.

Alors, que se passe-t-il si une classe doit hériter de plusieurs comportements ? Les traits POO résolvent ce problème.

Les traits sont utilisés pour déclarer des méthodes qui peuvent être utilisées dans plusieurs classes.
Les traits peuvent avoir des méthodes abstraites qui peuvent être utilisées dans plusieurs classes, et les méthodes peuvent avoir n'importe quel modificateur d'accès (public, privé ou protégé).

Syntaxe

Les traits sont déclarés avec le mot-clé `trait` :

```php
<?php
trait TraitName {
  // some code...
}
?>
```

Syntaxe

Pour utiliser un trait dans une classe, utilisez le mot-clé use:

```php
<?php
class MyClass {
  use TraitName;
}
?>
```

Exemple

```php
<?php
trait message1 {
public function msg1() {
    echo "OOP is fun! ";
  }
}

class Welcome {
  use message1;
}

$obj = new Welcome();
$obj->msg1();
?>
```

Explication de l'exemple

Ici, nous déclarons un trait : message1. Ensuite, nous créons une classe : Welcome. La classe utilise le trait et toutes les méthodes du trait seront disponibles dans la classe.

Si d'autres classes ont besoin d'utiliser la fonction msg1(), utilisez simplement le trait message1 dans ces classes. Cela réduit la duplication de code, car il n'est pas nécessaire de redéclarer la même méthode encore et encore.

Exemple

```php
<?php
trait message1 {
  public function msg1() {
    echo "OOP is fun! ";
  }
}

trait message2 {
  public function msg2() {
    echo "OOP reduces code
duplication!";
  }
}
```

```php
class Welcome {
  use message1;
}

class Welcome2 {
  use message1, message2;
}

$obj = new Welcome();
$obj->msg1();
echo "<br>";

$obj2 = new Welcome2();
$obj2->msg1();
$obj2->msg2();
?>
```

Explication de l'exemple

Ici, nous déclarons deux traits : message1 et message2. Ensuite, nous créons deux classes : Welcome et Welcome2. La première classe (Welcome) utilise le trait message1 et la deuxième classe (Welcome2) utilise à la fois les traits message1 et message2 (plusieurs traits sont séparés par une virgule).