Documentation v1

Introduction à la Programmation Orientée Objet en PHP

1. Definition de OOP:

La **programmation orientée objet (POO)** en PHP est un paradigme de programmation qui repose sur des concepts tels que les **objets**, les **classes**, et leurs interactions. Elle permet de structurer le code de manière modulaire, réutilisable et facile à maintenir.

2. principaux fondements de la POO en PHP:

a. Une **classe** est un plan ou un modèle à partir duquel des objets sont créés. Elle définit les **propriétés** (variables) et les **méthodes** (fonctions) que l'objet peut avoir.
b. Un **objet** est une instance d'une classe. C'est une représentation concrète de la classe avec ses propres données.
c. **Propriétés** : Ce sont des variables définies dans une classe et associées à ses objets. Elles stockent des données spécifiques à chaque instance.
d. **Méthodes** : Ce sont des fonctions définies dans une classe qui effectuent des actions ou manipulent les propriétés.
e. La visibilité des membres d'une classe (propriétés et méthodes) contrôle l'accès à ces derniers :
    - **+ public** : Accessible depuis n'importe où (dans la classe, dans ses instances, ou en dehors).
    - **- private** : Accessible uniquement dans la classe où il est défini.
    - **# protected** : Accessible uniquement dans la classe où il est défini et dans ses classes dérivées.
f. **Constructeur** : Une méthode spéciale appelée automatiquement lors de la création d'un objet. Elle est utilisée pour initialiser les propriétés.

g. **Destructeur** : Une méthode appelée automatiquement lorsqu'un objet est détruit ou en fin de script.
h. Une **interface** définit un ensemble de méthodes que les classes qui implémentent cette interface doivent obligatoirement définir.
i. Une **classe abstraite** peut contenir des méthodes abstraites (sans implémentation) et des méthodes concrètes.

3. La création des classes et des objets

Une **classe** est une structure qui regroupe des propriétés (variables) et des méthodes (fonctions) qui définissent le comportement et l'état des objets créés à partir de cette classe.
Ex:

```php
<?php
class Fruit {

  // Properties
  public $name;
  public $color;

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

Un **objet** est une instance d'une classe. Une fois qu'une classe est définie, vous pouvez créer des objets qui en héritent les propriétés et les méthodes.

```php
$apple = new Fruit();
$banana = new Fruit();
```

Un **constructeur** est une méthode spéciale appelée automatiquement lors de la création d'un objet.

```php
class Fruit {
  public $name;
  public $color;

  function __construct($name,$color) {
     $this->name = $name;
     $this->color = $color;
  }
}

$apple = new Fruit("Apple",”red”);
```

Encapsulation et Modificateurs d'Accès :

1. Définition:

consiste à regrouper les données (**propriétés**) et les méthodes qui opèrent sur ces données dans une classe tout en contrôlant l'accès à ces éléments depuis l'extérieur de la classe. Ce mécanisme améliore la sécurité et la modularité du code.

2. Les Modificateurs d'Accès

Les modificateurs d'accès déterminent la visibilité des **propriétés** et **méthodes** d'une classe

a) public

- Accessible depuis **n'importe où** : à l'intérieur de la classe, depuis une classe dérivée, ou à l'extérieur de la classe.

- C'est le modificateur par défaut si aucun autre modificateur n'est précisé.

b) private

- Accessible uniquement à l'intérieur de la classe où la propriété ou méthode est définie.
- Les classes dérivées (héritage) ou les instances extérieures ne peuvent pas y accéder directement.
- Utilisé pour protéger les données sensibles ou des comportements internes.

c) protected

- Accessible à l'intérieur de la classe et dans les classes qui en héritent.

3. Encapsulation avec des Getters et Setters

Pour permettre un accès contrôlé aux propriétés privées ou protégées, on utilise des **méthodes d'accès** : les **getters** (lecture) et **setters** (modification)

```
class Banque {
    private $solde = 0; // Propriété privée

    public function getSolde() {
        return $this->solde;
    }

    public function setSolde($montant) {
        if ($montant >= 0) {
             $this->solde = $montant;
        } else {
            echo "Le montant doit être positif.";
        }
    }
}
$compte = new Banque();
$compte->setSolde(1000);
echo $compte->getSolde();
```

Héritage et Polymorphisme

1. Héritage

L'héritage permet à une classe (**classe enfant**) de **hériter** des propriétés et méthodes d'une autre classe (**classe parente**). Cela favorise la réutilisation du code et évite les duplications.

- La classe parente (ou base) contient des fonctionnalités communes.
- La classe enfant peut utiliser ou étendre ces fonctionnalités.

```php
<?php
     class Vehicule {
         public $marque;
         public $modele;

         public function demarrer() {
             return "Le véhicule démarre !";
         }

     }

     class Voiture extends Vehicule {
     public $typeCarburant;
     public function afficherDetails() {
          return "Marque : $this->marque,Modèle :
     $this->modele, Carburant : $this->typeCarburant"; }
     }

     $maVoiture = new Voiture();
     $maVoiture->marque = "Toyota";
     $maVoiture->modele = "Corolla";
     $maVoiture->typeCarburant = "Essence";

     echo $maVoiture->demarrer();

     echo $maVoiture->afficherDetails();
?>
```

2. Polymorphisme

Le polymorphisme permet à une méthode d'avoir un comportement différent en fonction de la classe dans laquelle elle est définie ou utilisée. Cela peut être mis en œuvre grâce :

- Au remplacement de méthode : Une classe enfant peut redéfinir une méthode héritée pour personnaliser son comportement.
- À l'utilisation d'interfaces ou de classes abstraites.

1. *Redéfinition de méthode (Overriding)*

la classe enfant redéfinit une méthode de la classe parente pour modifier son comportement.

```php
<?php
    class Vehicule {
        public function demarrer() {
            return "Le véhicule démarre !";
        }
    }

    class Moto extends Vehicule {
        public function demarrer() {
            return "La moto démarre avec un bruit distinctif
    !";
        }
    }

    class Voiture extends Vehicule {
        public function demarrer() {
            return "La voiture démarre silencieusement !";
        }
    }


    $moto = new Moto();
    $voiture = new Voiture();

    echo $moto->demarrer(); // Affiche : La moto démarre
    avec un bruit distinctif !
    echo $voiture->demarrer(); // Affiche : La voiture
    démarre silencieusement !
?>
```

2. *Classes abstraites*

Une classe abstraite est une classe qui ne peut pas être instanciée directement. Elle sert de modèle pour les classes enfants et peut contenir des méthodes abstraites (non définies) que les classes enfants doivent implémenter.

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

      class Voiture extends Vehicule {
          public function demarrer() {
              return "La voiture démarre avec un moteur à combustion.";
          }
      }

      class Moto extends Vehicule {
          public function demarrer() {
              return "La moto démarre avec un coup de kick.";
          }
      }


      $voiture = new Voiture();
      $moto = new Moto();

      echo $voiture->demarrer();
       // Affiche : La voiture démarre avec un moteur à combustion.
      echo $moto->demarrer();
       // Affiche : La moto démarre avec un coup de kick.
?>
```

3. *Interfaces*

Une interface est similaire à une classe abstraite, mais elle ne peut contenir que des méthodes abstraites. Une classe peut implémenter plusieurs interfaces, ce qui permet d'ajouter plusieurs comportements.

```php
<?php

    interface Vehicule {
        public function demarrer();
        public function arreter();
    }

    class Voiture implements Vehicule {
        public function demarrer() {
            return "La voiture démarre.";
        }

        public function arreter() {
            return "La voiture s'arrête.";
        }
    }

    class Moto implements Vehicule {
        public function demarrer() {
            return "La moto démarre.";
        }

        public function arreter() {
            return "La moto s'arrête.";
        }
    }

    $voiture = new Voiture();
    $moto = new Moto();

    echo $voiture->demarrer();
     // Affiche : La voiture démarre.
    echo $moto->arreter();
    // Affiche : La moto s'arrête.
?>
```