I. Programmation Orientée Objet

La POO est une façon de structurer et d'organiser notre code en utilisant des objets qui représentent des entités du monde réel. Cela nous permet d'écrire du code plus organisé, réutilisable et facile à maintenir.

La POO permet de regrouper les variables et les fonctions sous un même "toit", que l'on appelle une classe.

Pourquoi utiliser la POO ?

Imaginons que vous deviez créer un logiciel pour gérer une bibliothèque. Chaque livre dans la bibliothèque a des caractéristiques comme un titre, un auteur et un nombre de pages et des actions comme "emprunter", "retourner", etc. Grâce à la POO, vous allez pouvoir créer un objet Livre qui regroupe toutes ces informations et comportements. Cela permet de simplifier le code et d'éviter de répéter les mêmes choses encore et encore.

1. Les concepts fondamentaux de la POO

1.1. Classe et Objet

Classe : Imaginez qu'une classe est un plan de construction ou un modèle pour créer quelque chose. C'est une sorte de moule qui définit ce que l'objet doit avoir comme caractéristiques et ce qu'il peut faire comme actions. Par exemple, une classe Voiture pourrait définir des caractéristiques comme la couleur ou la marque, et des actions comme démarrer ou freiner.

Objet : Un objet, quant à lui, est une instance de cette classe. Lorsque vous créez un objet, vous prenez ce modèle (la classe) et vous le "transformez" en une version concrète. Si la classe est le plan, l'objet est la voiture réelle qui a une couleur spécifique, une marque précise et peut démarrer ou freiner. Par exemple, vous pouvez créer un objet appelé maVoiture basé sur la classe Voiture et lui donner une couleur rouge et une marque Peugeot.

1.1.1. Créer une classe

Pour créer une classe en PHP, nous utilisons le mot-clé class, suivi du nom de la classe qui commence toujours par une majuscule. A l'intérieur, on définit des propriétés (caractéristiques) et des méthodes (actions). Les propriétés sont simplement des variables PHP et les méthodes des fonctions.

Prenons l'exemple de la classe Voiture :

```php
class Voiture {
   // Propriétés (caractéristiques de la voiture)
   public $couleur;
   public $marque;
   public $vitesseMax;
```

```
    // Méthodes (actions que la voiture peut effectuer)
    public function demarrer() {
        echo "La voiture démarre.<br>";
    }

    public function freiner() {
        echo "La voiture freine.<br>";
    }
}
```

Propriétés : Ce sont des variables qui définissent les caractéristiques de l'objet. Dans notre exemple, couleur, marque et vitesseMax sont des propriétés qui détermineront la couleur, la marque et la vitesse maximale de la voiture.

Méthodes : Ce sont des fonctions qui définissent ce que l'objet peut faire. Ici, nous avons deux méthodes : demarrer() et freiner(), qui simulent les actions que la voiture peut effectuer.

Une fois que la classe est définie, vous pouvez maintenant créer un objet basé sur cette classe.

1.1.2. Instancier un objet

Instancier un objet, c'est comme fabriquer une voiture à partir du plan que vous avez créé avec la classe, instancier signifie créer. En PHP, pour instancier un objet, on utilise le mot-clé new, suivi du nom de la classe. On parle alors d'instance de la classe.

```
// Création d'un objet Voiture
$maVoiture = new Voiture();

// Assignation des valeurs aux propriétés de l'objet
$maVoiture->couleur = "Rouge";
$maVoiture->marque = "Peugeot";
$maVoiture->vitesseMax = 180;

// Utilisation des méthodes de l'objet
echo "Ma voiture est une $maVoiture->marque de couleur $maVoiture->couleur.<br>";
$maVoiture->demarrer();  // Affiche "La voiture démarre."
$maVoiture->freiner();   // Affiche "La voiture freine."
```

Nous avons créé un objet maVoiture à partir de la classe Voiture en utilisant new Voiture(). maVoiture est une instance de la classe Voiture.

Pour affecter des valeurs à l'objet, on utilise l'opérateur ->. Par exemple, $maVoiture->couleur = "Rouge"; permet de définir la couleur de notre voiture.

Pour faire appel aux actions définies dans la classe, on utilise également ->. Par exemple, $maVoiture->demarrer(); fait démarrer la voiture.

1.1.3. Le Constructeur en PHP

Un constructeur est une méthode spéciale qui est automatiquement appelée lorsque vous créez un objet à partir d'une classe. Son rôle principal est d'initialiser les propriétés de l'objet avec des valeurs spécifiques dès sa création. Cela vous permet de préparer l'objet avec les bonnes valeurs sans avoir à les assigner manuellement après.

Prenons l'exemple de la classe Voiture pour mieux comprendre :

```
class Voiture {
    public $marque;
    public $couleur;
    public $vitesseMax;

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

    public function accelerer() {
        echo "La voiture accélère !<br>";
    }

    public function freiner() {
        echo "La voiture freine.<br>";
    }
}
```

Le constructeur est une méthode spéciale appelée __construct(). Il prend trois paramètres : $marque, $couleur, et $vitesseMax, qui correspondent aux valeurs que nous souhaitons attribuer aux propriétés de l'objet qui sera instancié. Ces valeurs sont ensuite utilisées pour initialiser les propriétés de l'objet.

Ensuite, pour créer l'objet, on utilise toujours le mot-clé new suivi du nom de la classe, mais cette fois-ci, nous passons en paramètre les valeurs de l'objet que nous créons :

```
$voiture1 = new Voiture("Peugeot", "Rouge", 190);
```

Le constructeur permet d'initialiser un objet de manière propre et claire dès sa création. Cela évite de devoir assigner des valeurs à chaque propriété après avoir créé l'objet. C'est

particulièrement utile lorsque vous avez plusieurs propriétés à initialiser. Cela rend votre code plus compact et facile à maintenir.

1.2. Encapsulation et Visibilité des Propriétés et Méthodes

L'encapsulation est un concept clé en programmation orientée objet. Elle consiste à regrouper les données (les propriétés) et les comportements (les méthodes) au sein d'une même unité, qui est la classe. L'objectif est de protéger les données internes d'un objet en les rendant accessibles uniquement de manière contrôlée. Cela permet de sécuriser l'intégrité de l'objet et d'éviter toute manipulation directe des données depuis l'extérieur de la classe.

L'encapsulation est rendue possible grâce au concept de visibilité des propriétés et des méthodes. Grâce à la visibilité nous pouvons contrôler qui peut accéder à ces éléments à l'intérieur ou à l'extérieur de la classe.

1.2.1. Les niveaux de visibilité

public
Les propriétés et méthodes publiques sont accessibles de partout. Vous pouvez y accéder depuis l'intérieur de la classe, dans les classes dérivées (héritée), et même depuis l'extérieur de la classe.

private
Les propriétés et méthodes privées sont accessibles uniquement dans la classe qui les déclare. Cela signifie qu'elles sont inaccessibles depuis une instance de la classe ou depuis une classe dérivée. Cela permet de bien protéger les données internes de la classe.

protected
Les propriétés et méthodes protégées sont accessibles à la fois dans la classe qui les déclare et dans les classes dérivées, mais elles sont inaccessibles à l'extérieur. Cela permet de donner un accès limité aux classes filles, tout en protégeant les données externes.

Pourquoi utiliser la visibilité ?

L'utilisation des différents niveaux de visibilité permet de protéger les données sensibles des objets. Par exemple, si une propriété est déclarée comme privée, elle ne pourra pas être modifiée directement depuis l'extérieur de la classe. Cela garantit que les données sont manipulées de manière contrôlée et sécurisée, en évitant toute modification non souhaitée ou erronée. Cela permet à la classe de prendre en charge et de gérer ses propres données de façon fiable.

Lorsqu'une instance, une autre classe, ou tout autre programme doit accéder aux données d'une classe, nous utilisons des getters et des setters. Ce sont des méthodes publiques spéciales qui permettent de lire et modifier les propriétés de la classe tout en maintenant un contrôle strict sur les accès.

Ainsi, les accès aux propriétés ne se font pas directement. Au lieu de cela, ce sont ces méthodes qui permettent l'interaction avec les données, et la classe se charge de vérifier et de valider les modifications avant qu'elles ne soient appliquées.

1.2.2. Exemple

Reprenons l'exemple de la classe Voiture où nous voulons protéger la propriété de la vitesse maximale. Nous allons déclarer cette propriété comme privée et utiliser des méthodes publiques pour accéder et modifier cette propriété de manière contrôlée.

```php
class Voiture {
   // Propriété privée : elle ne peut être accédée directement de l'extérieur
   private $vitesseMax;

   // Constructeur pour initialiser la vitesse maximale
   public function __construct($vitesseMax) {
       $this->vitesseMax = $vitesseMax;
   }

   // Méthode publique pour obtenir la vitesse maximale
   public function getVitesseMax() {
       return $this->vitesseMax;
   }

   // Méthode publique pour modifier la vitesse maximale de manière contrôlée
   public function setVitesseMax($vitesse) {
       // On vérifie que la nouvelle vitesse est positive avant de la modifier
       if ($vitesse > 0) {
           $this->vitesseMax = $vitesse;
       } else {
           echo "La vitesse ne peut pas être inférieure ou égale à zéro.<br>";
       }
   }
}

$voiture1 = new Voiture(190);

// Accès à la propriété via la méthode publique
echo "La vitesse maximale est : {$voiture1->getVitesseMax()} km/h.<br>";

// Modification de la propriété via la méthode publique
$voiture1->setVitesseMax(200);
echo "Modification : la vitesse maximale est : {$voiture1->getVitesseMax()} km/h.<br>";

// Tentative de modification avec une valeur invalide
$voiture1->setVitesseMax(-50);  // Affiche un message d'erreur
```

La propriété $vitesseMax est déclarée comme privée en utilisant le mot-clé private. Cela signifie que vous ne pouvez pas y accéder directement depuis l'extérieur de la classe. Pour interagir avec cette propriété, vous devez utiliser les méthodes publiques.

Les méthodes getVitesseMax() et setVitesseMax() sont publiques, ce qui signifie qu'elles peuvent être appelées de l'extérieur de la classe. Ces méthodes servent à obtenir et modifier la vitesse maximale de manière contrôlée.

Dans la méthode setVitesseMax(), nous avons ajouté une vérification pour nous assurer que la nouvelle valeur de la vitesse est positive avant de la modifier. Si l'utilisateur essaie de définir une vitesse négative ou nulle, un message d'erreur est affiché. Cela montre comment l'encapsulation non seulement protège les données, mais aussi permet de valider les modifications avant qu'elles ne soient appliquées.

2. TP

Vous allez créer un jeu de rôle dans lequel le joueur peut créer différents personnages (Guerriers, Orques) et les faire combattre. Le but est de pratiquer la création de classes, d'objets, et l'utilisation du constructeur et de l'encapsulation (visibilité, getters, setters).

2.1. Étapes à suivre

1. **Créer la classe Personnage** :
   - Déclarez les propriétés privées suivantes :
     - nom : Le nom du personnage.
     - sante : Le niveau de santé (de 0 à 100).
     - force : La force d'attaque (de 10 à 100).
     - defense : La défense (de 0 à 100).
     - type : Le type de personnage (Guerrier ou Orc).
   - Ajoutez un constructeur pour initialiser ces propriétés lors de la création d'un personnage.

2. **Protéger les propriétés avec l'encapsulation** :
   - Déclarez toutes les propriétés comme privées.
   - Créez des getters pour accéder aux propriétés.
   - Créez des setters pour modifier les propriétés avec des contrôles appropriés :
     - La santé ne peut pas dépasser 100 ni être inférieure à 0.
     - La force doit être comprise entre 10 et 100.
     - La défense doit être comprise entre 0 et 100.

3. **Ajouter la méthode attaquer()** :
   - Ajoutez une méthode publique attaquer($adversaire) qui permet à un personnage d'attaquer un autre :
     - Les dégâts infligés sont calculés comme : force - defense de l'adversaire.
     - Réduisez la santé de l'adversaire en fonction des dégâts.
     - Si la santé devient inférieure ou égale à 0, affichez un message indiquant que l'adversaire est mort.

2.2. Scénarios

1. **Création des personnages** :
   - Créez deux personnages :
     - Un Guerrier (nom, santé, force, défense).
     - Un Orc (nom, santé, force, défense).
2. **Simulation de combat** :
   - Faites en sorte que le Guerrier attaque l'Orc. Affichez la santé des deux personnages avant et après chaque attaque.
   - Répétez jusqu'à ce qu'un personnage meure (sa santé atteint zéro ou moins).