TP : POO en PHP moderne
Classes, Héritage, Polymorphisme, POO avancée + Mini MVC

Objectifs du TP

- Comprendre les concepts de base de la POO : classe, objet, attribut, méthode, constructeur.
- Utiliser l'encapsulation, l'héritage et le polymorphisme.
- Découvrir des notions modernes : classes abstraites, interfaces, traits, enums, attributs statiques.
- Mettre en pratique avec des exercices progressifs en PHP.
- Voir un mini exemple d'architecture MVC.

1 Notes de cours : rappel POO

1.1 Classe, objet, attributs, méthodes

Une **classe** est un modèle, un plan de construction. Un **objet** est une instance d'une classe.

Les **attributs** sont les données de l'objet. Les **méthodes** sont les fonctions qui agissent sur ces données.

1.2 Constructeur

Le **constructeur** (`__construct`) est une méthode spéciale appelée automatiquement lors de la création d'un objet avec `new`. On l'utilise pour initialiser les attributs.

```php
<?php

class Personne {
    private string $prenom;
    private string $nom;

    public function __construct(string $prenom, string $nom) {
        $this->prenom = $prenom;
        $this->nom    = $nom;
    }

    public function identite(): string {
        return "$this->prenom $this->nom";
    }
}

$p = new Personne("Paul", "Langevin");
echo $p->identite();
```

1.3 Encapsulation

L'encapsulation sert à contrôler l'accès aux données.

- `public` : accessible partout.
- `private` : accessible uniquement dans la classe.
- `protected` : accessible dans la classe et ses classes enfants.

2 Héritage

L'**héritage** permet de créer une classe *enfant* à partir d'une classe *parent*. On écrit : `class Enfant extends Parent`.

La classe enfant hérite des attributs et des méthodes du parent et peut :

- ajouter de nouveaux attributs / méthodes ;
- redéfinir (surcharger) une méthode existante.

Exemple : Personne → Etudiant

```php
<?php

class Personne {
    protected string $prenom;
    protected string $nom;

    public function __construct(string $prenom, string $nom) {
        $this->prenom = $prenom;
        $this->nom    = $nom;
    }

    public function identite(): string {
        return "$this->prenom $this->nom";
    }
}

class Etudiant extends Personne {
    private string $programme;

    public function __construct(string $prenom, string $nom, string $programme) {
        parent::__construct($prenom, $nom);
        $this->programme = $programme;
    }

    public function identite(): string {
        return parent::identite() . " - Étudiant en $this->programme";
    }
}

$e = new Etudiant("Sarah", "Gagnon", "Informatique");
echo $e->identite();
```

3 Polymorphisme

Le **polymorphisme** signifie "plusieurs formes". En POO, plusieurs classes différentes peuvent avoir une méthode du *même nom* et on peut les utiliser de manière uniforme.

Exemple : tous les animaux ont une méthode `parler()`, mais le résultat dépend du type concret.

```php
<?php

class Animal {
    public function parler(): string {
        return "???";
    }
}

class Chien extends Animal {
    public function parler(): string {
        return "Wouf !";
    }
}

class Chat extends Animal {
    public function parler(): string {
        return "Miaou !";
    }
}

$animaux = [ new Chien(), new Chat(), new Chien() ];

foreach ($animaux as $a) {
    echo $a->parler() . "<br>";
}
```

4 POO moderne : notions avancées

4.1 Classes abstraites

Une **classe abstraite** ne peut pas être instanciée directement. Elle sert de modèle et peut contenir des méthodes abstraites (sans corps) que les classes enfants doivent implémenter.

```php
<?php

abstract class Forme {
    abstract public function aire(): float;
}

class Rectangle extends Forme {
    public function __construct(
        private float $largeur,
        private float $hauteur
    ) {}

    public function aire(): float {
        return $this->largeur * $this->hauteur;
    }
}
```

4.2 Interfaces

Une **interface** définit une liste de méthodes à implémenter. Toutes les classes qui implémentent une interface doivent fournir le code de ces méthodes.

```php
<?php

interface Affichable {
    public function afficher(): string;
}

class Article implements Affichable {
    public function __construct(
        private string $titre,
        private string $contenu
    ) {}

    public function afficher(): string {
        return "Article: {$this->titre}";
```

```
    }
}
```

4.3 Traits

Les **traits** permettent de réutiliser du code dans plusieurs classes sans passer par l'héritage.

```php
<?php

trait Logger {
    public function log(string $message): void {
        echo "[LOG] {$message}<br>";
    }
}

class ServiceA {
    use Logger;

    public function executer(): void {
        $this->log("ServiceA::executer() appelé");
    }
}
```

4.4 Enums

Les **énumérations** (`enum`) permettent de définir un ensemble fini de valeurs possibles.

```php
<?php

enum RoleUtilisateur: string {
    case ADMIN  = 'admin';
    case CLIENT = 'client';
}

class Utilisateur {
    public function __construct(
        private string $nom,
        private RoleUtilisateur $role
    ) {}

    public function estAdmin(): bool {
        return $this->role === RoleUtilisateur::ADMIN;
    }
```

```
}
```

4.5 Attributs statiques

Un **attribut statique** appartient à la classe et non à une instance particulière.

```php
<?php

class Compteur {
    private static int $nbInstances = 0;

    public function __construct() {
        self::$nbInstances++;
    }

    public static function getNbInstances(): int {
        return self::$nbInstances;
    }
}
```

5 Exemple : Sans POO vs avec POO

Sans POO

```php
<?php

$prenom1 = "Paul";
$nom1    = "Langevin";
$age1    = 48;

$prenom2 = "Marie";
$nom2    = "Dubois";
$age2    = 34;

function identite($prenom, $nom, $age) {
    return "$prenom $nom, $age ans";
}

echo identite($prenom1, $nom1, $age1);
echo "<br>";
echo identite($prenom2, $nom2, $age2);
```

Avec POO

```php
<?php

class Personne {
    private string $prenom;
    private string $nom;
    private int $age;

    public function __construct(string $prenom, string $nom, int $age) {
        $this->prenom = $prenom;
        $this->nom    = $nom;
        $this->age    = $age;
    }

    public function identite(): string {
        return "$this->prenom $this->nom, $this->age ans";
    }
}

$p1 = new Personne("Paul", "Langevin", 48);
$p2 = new Personne("Marie", "Dubois", 34);
```

```php
echo $p1->identite();
echo "<br>";
echo $p2->identite();
```

6 Mini MVC pédagogique

On sépare le code en trois parties :

- **Model** : représente les données (classe).
- **Controller** : crée l'objet, prépare les données.
- **View** : affiche le résultat.

Arborescence proposée

```
mini_mvc/
  model/Livre.php
  controller/LivreController.php
  view/vueLivre.php
```

Model : Livre.php

```php
<?php

class Livre {
    public string $titre;
    public string $auteur;

    public function __construct(string $titre, string $auteur) {
        $this->titre  = $titre;
        $this->auteur = $auteur;
    }

    public function getDescription(): string {
        return $this->titre " - " . $this->auteur;
    }
}
```

Controller : LivreController.php

```php
<?php
require_once "../model/Livre.php";

$livre = new Livre("Harry Potter à l'école des sorciers", "J.K. Rowling");
```

```php
require "../view/vueLivre.php";
```

View : vueLivre.php

```php
<?php
// vueLivre.php
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Livre</title>
</head>
<body>
    <h1>Informations sur le livre</h1>

    <p><?php echo $livre->getDescription(); ?></p>
</body>
</html>
```

7 Exercices

Exercice 1 — Classe Étudiant (base POO)

Créer une classe `Etudiant` avec :

- `prenom`, `nom`, `programme` (attributs privés) ;
- un constructeur pour initialiser ces attributs ;
- une méthode `presentation()` qui retourne Prénom Nom — Programme.

Créer au moins deux objets `Etudiant` et afficher leur présentation.

Exercice 2 — Produit et total

Créer une classe `Produit` avec :

- `nom`, `prix`, `quantite` ;
- une méthode `total()` qui retourne `prix * quantite`.

Exercice 3 — Compte et encapsulation

Créer une classe `Compte` avec :

- un attribut privé `solde` (float) ;
- une méthode `depot($montant)` ;
- une méthode `retrait($montant)` qui refuse un retrait si les fonds sont insuffisants ;
- une méthode `getSolde()`.

Exercice 4 — Héritage Personne / Enseignant

Créer :

- une classe `Personne` avec `prenom`, `nom` et une méthode `identite()` ;
- une classe `Enseignant` qui hérite de `Personne` et ajoute `matiere` ;
- redéfinir `identite()` dans `Enseignant`.

Exercice 5 — Polymorphisme Vehicule

Créer :

- une classe `Vehicule` avec une méthode `afficher()` ;
- deux classes `Voiture` et `Moto` qui héritent de `Vehicule` et redéfinissent `afficher()` ;
- un tableau de véhicules, puis parcourir et appeler `afficher()`.

Exercice 6 — Classe abstraite Forme

Créer une classe abstraite `Forme` avec une méthode abstraite `aire()`. Créer `Rectangle` et `Cercle` qui étendent `Forme` et implémentent `aire()`. Créer un tableau de formes et afficher leur aire.

Exercice 7 — Interface Affichable

Créer une interface `Affichable` avec une méthode `afficher()`. Créer `Article` et `Video` qui implémentent `Affichable`. Créer un tableau d'objets et afficher le résultat de `afficher()` pour chacun.

Exercice 8 — Trait Logger

Créer un trait `Logger` avec une méthode `log($message)`. Créer deux classes `ServiceA` et `ServiceB` qui utilisent ce trait et l'appellent dans leurs méthodes.

Exercice 9 — Enum RoleUtilisateur

Créer un `enum RoleUtilisateur` avec les valeurs `ADMIN` et `CLIENT`. Créer une classe `Utilisateur` qui possède un rôle de type `RoleUtilisateur` et une méthode `estAdmin()`.

Exercice 10 — Static Compteur

Créer une classe `Compteur` avec un attribut statique comptant le nombre d'instances créées. Afficher le nombre total d'instances après avoir créé plusieurs objets.