

5/ PHP et la Programmation Orientée Objet (POO)

Pour faciliter l'appel d'une fonction, et surtout pour organiser correctement la structure d'une application PHP, on doit se servir des classes qui seront appelée au premier niveau. Sur l'exemple ci-dessous, la classe doit être stockée indépendamment, et sera chargée par la méthode : **require()** :

```
<?php

/**
 * Exemple d'une classe objet utilisateurs, ici la base de données est simulée par un tableau.
 */

class Page {

    /* Propriétés privée, accessibles uniquement dans cette classe : */
    private $_id;
    private $_quoi;
    private $_db=array(); //pseudo BD
    public $now; // propriété publique d'une variable, accessible en dehors de la classe

    /* Constructeur de la classe, Les éléments seront chargés par défaut */
    public function __construct() {
        echo "Bienvenue sur la page utilisateurs";

        /* $this correspond à la classe elle-même. ATTENTION La variable perd son $ */
        $this->_db['users']['id']['001'] = [ 'nom' => 'Leponge', 'prenom' => 'Bob', 'lic' => 'acquise', 'num' => md5(uniqid('',true)) ];

        $this->_db['users']['id']['002'] = [ 'nom' => 'Curry', 'prenom' => 'Marie', 'lic' => 'invalide', 'num' => md5(uniqid('',true)) ];

        $this->now = date("j/n/y à H:i:s", strtotime("now") );
    }

    /* Fonction publique, accessible à l'extérieur de cette classe */
    public function show($var1=null, $var2=null) {
        $this->_id = $var1;
        $this->_quoi = $var2;
        $this->_other($this->_id, $this->_quoi);
    }

    /* Fonction privée accessible à l'intérieur de cette classe avec : $this */
    private function _other($id, $quoi) {
        echo $this->_db['users']['id'][$id][$quoi];
    }
}

/**
 * Index de la page
 * En début de script, on charge la classe avec require()
 * require() arrête l'exécution du code en cas d'erreur, include() le continue.
 * require( _DIR_ . "/cette_classe.php" );
 */

echo "<h3>";
$maclasse = new Page(); //On instancie la classe
echo "</h3><li>Nom et prénom : <b>";
$maclasse->show("001","nom"); //On appelle la fonction publique nommée show
echo " ";
$maclasse->show("001","prenom");
echo "</b></li><li>Type de licence : <b>";
$maclasse->show("001","lic");
echo "</b></li>";
echo "</b></li><li>Numéro de licence : <b>";
$maclasse->show("001","num");
echo "</b></li><p></p><hr>";
echo "<i>Date de création du document : Le " . $maclasse->now . "</i>";

?>
```



EXERCICE 2

Nouveau fichier => **02-exo-classTest.php**

- Charger la classe depuis la cible : « **cls/users.php** ». Il y aura donc un dossier « **cls** » contenant un fichier nommé « **users.php** » qui contiendra uniquement la classe PHP. Celle-ci sera chargée avec **require()**
- Adapter et optimiser le code ci-dessus afin d'afficher les données des 2 utilisateurs (**001** et **002**). L'exemple proposé affiche seulement l'utilisateur 001.
- Ajouter un utilisateur avec l'id : **003**. Un email factice devra être ajouté dans la fausse base de données (*le tableau `$_db`*) pour chaque utilisateur.

POO : Comprendre la structure de l'objet

Une classe étant une définition, elle nous servira à créer des objets. Elle est composée d'attributs (variable d'instance ou propriété), et de **méthodes** (actions / opération au niveau de la classe identique à une fonction, mais encapsulé dans la classe) qui ont leur propre **visibilité** :

- **Privée** : accessible que dans l'objet.
- **Public** : accessible hors de l'objet.
- **Protected** : Accessible aux enfants (héritage), mais pas hors de la classe.

Il y a plusieurs types de **méthodes** :

- **Le constructeur** : méthode chargée par défaut, non obligatoire.
- **Le destructeur** : méthode déchargée par défaut, non obligatoire.
- **Les méthodes d'actions** : elles sont les fonctions.
- **Les méthodes accesseurs / mutateurs** (setter/getter) : charge ou décharge de l'information dans l'objet.

```
<?php

class Personne {

    // Attribut
    public $nom;
    public $prenom;
    private $salaire;
    public $nbEnfant;

    // Constructeur
    function __construct($prenom, $nom, $nbEnfant = 0) {
        $this->nom = $nom;
        $this->prenom = $prenom;
        $this->nbEnfant = $nbEnfant;
    }

    // Mutateur
    public function setSalaire($valeur) {
        $this->salaire = $valeur;
    }

    // Accesseur
    public function getSalaire() {
        return $this->salaire;
    }

    // Méthode
    public function identite() {
        return $this->nom . " " . $this->prenom;
    }
}
```

```
// Destructeur
function __destruct() {
    // Implémentation
}
?>
```

Dans l'exemple ci-dessus on instancie la classe **Personne** par :

```
$personne1 = new Personne("Bob", "Leponge", 0);
```

```
$personne2 = new Personne("Marco", "Polo", 4);
```

Et l'on peut ajouter ou afficher son salaire (setter/getter) :

```
$personne2->setSalaire(2000);
```

```
echo $personne2->prenom . " gagne " . $personne2->getSalaire() . ' euros/mois.';
```

Créer un objet c'est l'instancier (le charger en mémoire).



EXERCICE 3

Nouveau fichier => **03-exo-classSetterGetter.php**

- Créer un mutateur et un accesseur qui va charger ou afficher la date de naissance de la personne1 (\$personne1).
- Le retour doit s'afficher sous la forme d'un tableau avec l'identité de la personne (nom et prénom).

POO : L'héritage

L'héritage permet de généraliser le fonctionnement d'un objet. L'idée est de mettre dans un « objet parent » la logique de plusieurs objets qui fonctionnent de la même façon. Par exemple :

```
<?php

// Classe de base : Mammifère
class Mammifere {
    private $genre;

    public function __construct($genre) {
        $this->genre = $genre;
    }

    public function bonjour() {
        return "Je suis un mammifère, précisément un " . $this->genre . ", ";
    }
}

// Classe étendue : Carnivore
class Carnivore extends Mammifere {
    public function manger() {
        return "je mange de la viande.<br />";
    }
}

// Classe étendue : Omnivore
class Omnivore extends Mammifere {
    public function manger() {
        return "je mange de tout.<br />";
    }
}
```

```
// Classe étendue : Végétarien
class Vegetarien extends Mammifere {
    public function manger() {
        return "je suis végétarien.<br />";
    }
}

// Instances des objets étendus, La classe Mammifere est chargée par défaut :
$lion = new Carnivore('lion');
/* autres instances possibles */

// Utilisation des objets fléchés
echo $lion->bonjour();
echo $lion->manger();
/* autres utilisations possibles */
```



EXERCICE 4

Nouveau fichier => **04-exo-classHeritage.php**

A partir du code ci-dessus ajouter les autres instances possibles et leurs utilisations, notamment pour les "humains" et les "lapins" !

POO : Les classes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée. Elle permet de définir des comportements (méthodes) dont l'implémentation (le code dans la méthode) se fait dans les classes filles. Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par la classe mère abstraite.

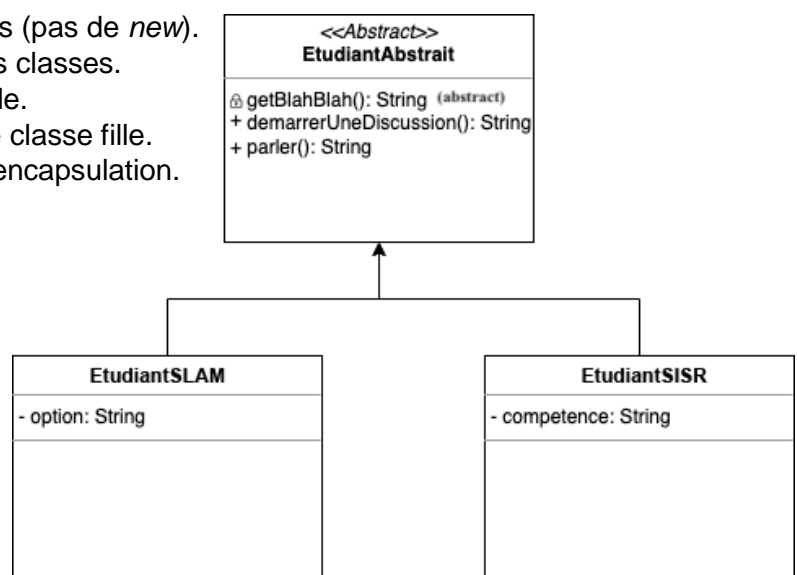
Nous aurons donc deux types de classes :

- **Des classes abstraites** (sans code, non instanciable).
- **Des classes concrètes** (avec du code, et instanciable).

Une classe abstraite doit posséder au moins une méthode abstraite (c'est-à-dire sans code). Si nécessaire, elle peut également avoir des méthodes concrètes (avec du code).

Les classes abstraites :

- Ne peuvent pas être instanciées (pas de *new*).
- Sont des modèles pour d'autres classes.
- Permettent de factoriser du code.
- Doivent être héritée depuis une classe fille.
- Apporte une sécurité grâce à l'encapsulation.



Un exemple issu du MCD précédant (modèle conceptuel des données) :

```
<?php

// Classe abstraite, non instanciable
abstract class EtudiantAbstrait {
    // Force les classes filles à définir cette méthode
    abstract protected function getBlahBlah();
    abstract public function demarrerUneDiscussion($sujet);

    // méthode commune
    public function parler() {
        return $this->getBlahBlah() . "<br />";
    }
}

// Classe fille, instanciable car concrète L'ensemble des méthodes possède du code
class EtudiantSLAM extends EtudiantAbstrait {
    private $option = "SLAM";

    //méthode protégée car issue de la classe abstraite, le nom doit être identique
    protected function getBlahBlah() {
        return "Sur le BTS SIO, mon option est : {$this->option}";
    }

    //méthode publique car issue de la classe abstraite, le nom doit être identique
    public function demarrerUneDiscussion($sujet) {
        return "Je vais vous parler de « {$sujet} »";
    }
}

// Classe fille, instanciable car concrète L'ensemble des méthodes possède du code
class EtudiantSISR extends EtudiantAbstrait {
    private $competence = "ADMINISTRER";

    //méthode protégée car issue de la classe abstraite, le nom doit être identique
    protected function getBlahBlah() {
        return "Sur le BTS SIO SISR, j'apprends à : {$this->competence}";
    }
}

// Instanciation
$class1 = new EtudiantSLAM();
echo $class1->parler(); // Sur le BTS SIO, mon option est : SLAM
// Je vais vous parler de « La POO avec PHP »
echo $class1->demarrerUneDiscussion('La POO avec PHP') . "<br />";
$class1 = new EtudiantAbstrait();
```

Remarque : Ici les accolades simplifient la syntaxe de la chaîne de caractère retournée :
 echo "Sur le BTS SIO, j'apprends à : {\$this->competence}";
 C'est une alternative à la concaténation.



EXERCICE 5

Nouveau fichier => **05-exo-classAbstraite.php**

Sur le code ci-dessus pourquoi obtient-on des erreurs ?
 Comment corriger ?

POO : La gestion des erreurs de sortie

PHP inclut deux méthodes pour la gestion des erreurs que l'on peut affecter dans une classe :

- `throw new Exception(string)`
→ Permet de personnaliser une erreur de sortie.
- `try {la_fonction_à_tester} catch {erreur et/ou string} finally {funct}`
→ Permet de tester une fonction et d'afficher une erreur si la fonction échoue. Le paramètre *finally* permet d'exécuter un élément quoiqu'il se passe (erreur ou pas).

```
<?php
class App {
    public function setup($param) {
        //nous avons besoin d'un paramètre :
        if(empty($param)) {
            throw new Exception("<b>Il faut au moins un paramètre !</b>");
        } //ce paramètre doit être un tableau :
        elseif(!is_array($param)) {
            throw new Exception("<b>Ce paramètre n'est pas un tableau !</b>");
        }
        //si tout va bien on l'affiche :
        print_r($param);
    }
}

//try AVANT la classe à charger :
try {
    $app = new App(); //classe instanciée
    $app->setup(); //on lance la fonction, est-ce correcte ?
} catch (Exception $e) { // $e représente l'erreur de sortie générée par PHP, ici personnalisée
    echo $e . "<br><b>Impossible de charger l'application.</b>";
}

?>
```



EXERCICE 6

06-exo-tryCatch.php

En testant le code ci-dessus, comment résoudre l'erreur de sortie ?

POO : le typage des méthodes depuis PHP 7 (et +)

Avec l'apparition de PHP7 la POO devient plus professionnelle et intègre le typage des méthodes. La valeur retournée sera alors forcément typée :

```
<?php
class AppUtil {
    private static $instance;
    private static $tab = [];

    private function __construct() {
        var_dump( self::alert("Bienvenue dans l'objet!") );
    }

    // Instance unique de la classe (singleton)
    public static function getInstance(): self {
        if (self::$instance === null){
            self::$instance = new self();
        }
        return self::$instance;
    }

    public static function alert(?string $message): string { // Le paramètre ?string est nullable, donc
        return $message; // ici soit une chaîne, soit null
    }

    public static function addNumbers(int $num1, int $num2): int {
        self::$tab['numbers'] = $num1 + $num2;
        return $num1 + $num2;
    }

    public static function getArr(): array {
        return self::$tab;
    }

    public static function getNum(): float {
        return self::$tab['numbers'];
    }

    public static function getUnionType(): int|string {
        return self::$tab['numbers'] . ' est un nombre.';
    }

    public static function resetNumbers(): void {
        self::$tab['numbers'] = 0;
    }

    public static function getFile( string $file ): bool {
        return file_exists($file);
    }

    public function __destruct() {
        var_dump( self::alert( get_class($this) . " est fermée." ) );
    }
}

echo '<pre>';
AppUtil::getInstance();
AppUtil::addNumbers(28,32);
var_dump(AppUtil::getArr());
var_dump(AppUtil::getNum());
var_dump(AppUtil::getFile(__DIR__ . '/test.html'));
var_dump(AppUtil::getUnionType());
AppUtil::resetNumbers();
var_dump(AppUtil::getNum());
?>
```

Remarque : cet exemple prend comme modèle une classe statique à instance unique (ou [singleton](#)). Par définition une classe [statique](#) ne possède pas d'instance, contrairement à un objet avec `$this->`



EXERCICE 7

07-exo-classType.php

En testant le code ci-dessus, commentez chacune des fonctions et leur type de retour, exemple :
Traduire le type en français !

```
/**
 * Retourne La version de PHP
 * @sort un entier (integer)
 */
```

POO : Les surcharges magiques des accesseurs et mutateurs

Une classe peut instancier des données reçues avec deux [surcharges magiques](#) : `__get` et `__set`. L'une se chargera de retourner une donnée, le "getter" (`__get`) et l'autre se chargera de la conserver, le "setter" (`__set`). Exemple avec `__get` seulement :

```
<?php

class Getter {

    public function __get($name) {
        if($name=="test1") return $this->test1();
        if($name=="test2") return $this->test2();
    }

    protected function test1() {
        return "Fonction numero 1 choisi...";
    }

    protected function test2() {
        return "Fonction numero 2 choisi...";
    }
}

$params = new Getter();
echo $params->test2;
echo "<br>";
echo $params->test1;

?>
```

Exemple simple utilisant les surcharges magiques `__get` et `__set` :

```
<?php

class GeoData {

    private $_vars = array();

    public function __get($name) {
        return $this->_vars[$name];
    }

    public function __set ($name, $value) {
        $this->_vars[$name] = $value;
    }
}

$geo = new GeoData();
$geo->continents = ['Afrique', 'Europe', 'Asie']; //on charge la classe en donnée (set)

echo "<pre>";
print_r($geo->continents); //on récupère cette donnée (get)

?>
```


Exemple d'une classe statique nommée « Config » qui se chargera d'écrire la donnée des paramètres pour autre classe, par exemple :

```
<?php

class Config {

    private $_vars = array();
    protected static $_instance;

    public static function getInstance() {
        if (!isset(self::$_instance)) {
            self::$_instance = new self();
        }
        return self::$_instance;
    }

    public function __get($name) {
        return $this->_vars[$name];
    }

    public function __set ($name, $value) {
        $this->_vars[$name] = $value;
    }
}

class GeoData {

    private $param;

    public function __construct() {
        // On déclare la propriété statique avec => ::
        $this->param = Config::getInstance();
    }

    public function set_my_continents() {
        //charge la donnée sur la class Config
        $this->param->continents = ['Afrique', 'Europe', 'Asie'];
    }

    public function get_my_continents() {
        //récupère cette donnée depuis la class Config
        return $this->param->continents;
    }
}

$geo = new GeoData();
$geo->set_my_continents();
$continents = $geo->get_my_continents();
echo "<pre>";
print_r( $continents );

?>
```

POO : Espace de nom : namespace

Afin d'organiser les classes et que l'on puisse facilement les retrouver le **namespace** (ou *espace de nom*) est indispensable. Cette pratique se retrouve dans la production d'une application PHP professionnelle destinée à être partagée en équipe ou à être publiée comme librairie. [Voir la préconisation côté PHP DOC](#)

Exemple simple d'un espace de nom **App**, avec un fichier nommé **cls.php** :

```
<?php

//Ici le nom associé à l'ensemble des éléments qui seront déclarées sur ce fichier
//jusqu'au prochain namespace (si on le décide)
namespace App;

//Une simple classe de calcul qui retournera un entier
class Calcul {

    public function additionner(int $a, int $b) : int {
        return $a + $b;
    }
}
```

On charge le fichier, et on instancie sa classe nommée. Ici le fichier est nommé **app.php** :

```
<?php

require('cls.php');

//FQCN : Fully Qualified Class Name (nom qualifié de la classe nommée)
//Attention : L'anti-slash sépare les éléments, pas le slash !
$instance = new App\Calcul; //on appelle le namespace et la classe nommée

//On exécute la fonction dans l'objet :
$resultat = $instance->additionner(4, 8);

var_dump($resultat);

?>
```

Renforcer ses apprentissages et la pratique structurée de la POO :

- ⇒ https://www.w3schools.com/php/php_oop_what_is.asp
- ⇒ <https://www.pierre-giraud.com/php-mysql-apprendre-coder-cours/introduction-programmation-orientee-objet/>

ANNEXE : POO et SINGLETON

Caractéristique	Objet	Classe statique	Singleton
Nature	Instance d'une classe	Ensemble de méthodes et propriétés accessibles statiquement	Instance unique d'une classe
Création	Instanciation avec le mot-clé <code>new</code>	N'est pas instanciée, existe dès le chargement de la classe	Instanciation contrôlée, généralement au premier appel à une méthode statique
Accès aux membres	Via l'instance (ex : <code>\$objet->propriete</code>)	Directement via le nom de la classe (ex : <code>Classe::methode()</code>)	Via une méthode statique de la classe (ex : <code>Singleton::getInstance()->methode()</code>)
Flexibilité	Très flexible, peut être personnalisé et étendu	Moins flexible, car ne peut pas être instanciée ou héritée	Plus flexible qu'une classe statique, mais moins qu'un objet classique
Utilisation typique	Modélisation d'entités, encapsulation de données et comportements	Méthodes utilitaires, constantes, configurations globales, helper classes	Gestionnaires de ressources, registres, objets de configuration globaux, points d'accès uniques
Héritage	Oui, peut hériter et être hérité	Non, ne peut pas être hérité	Peut être hérité, mais avec précautions
Instanciation multiple	Oui, autant d'instances que nécessaire	Non, pas d'instance	Non, une seule instance
Cycle de vie	Créé à l'instanciation, détruit par le garbage collector	Existe tout au long de l'exécution du programme	Créé généralement au premier appel à <code>getInstance()</code> , détruit à la fin du programme ou manuellement
Contexte	Modélisation d'entités, encapsulation de données et comportements	Méthodes utilitaires, constantes, configurations globales, helper classes	Gestionnaires de ressources, registres, objets de configuration globaux, points d'accès uniques

Sources :

- <https://refactoring.guru/fr/design-patterns/singleton>
- <https://www.ionos.fr/digitalguide/sites-internet/developpement-web/quest-ce-que-le-singleton-pattern/>