Programmation orientée objet PHP

Mickaël LE BRAS

Inspiré du cours de Pierre Giraud

https://www.pierre-giraud.com/php-mysql-apprendre-coder-cours/introduction-programmation-orientee-objet/

Introduction

Dans cette nouvelle partie, nous allons redécouvrir le PHP sous un nouvel angle avec la programmation orientée objet. La programmation orientée objet est une façon différente de coder qui va suivre des règles différentes et va amener une syntaxe différente, ce qui fait qu'elle peut être perçue comme difficile à comprendre pour des débutants.

Il convient de noter que de nombreux langages possèdent une écriture orientée objet car encore une fois cette façon de coder va s'avérer très puissante et centrale dans la carrière d'un développeur.

Définition

La programmation orientée objet (ou POO en abrégé) correspond à une autre manière d'imaginer, de construire et d'organiser son code.

Jusqu'à présent, nous avons codé de manière procédurale, c'est-à-dire en écrivant une suite de procédures et de fonctions dont le rôle était d'effectuer différentes opérations sur des données généralement contenues dans des variables et ceci dans leur ordre d'écriture dans le script.

La programmation orientée objet est une façon différente d'écrire et d'arranger son code autour de ce qu'on appelle des objets. Un objet est une entité qui va pouvoir contenir un ensemble de fonctions et de variables.

L'idée de la programmation orientée objet va donc être de grouper des parties de code qui servent à effectuer une tâche précise ensemble au sein d'objets afin d'obtenir une nouvelle organisation du code.

Les intérêts principaux de la programmation orientée objet vont être une structure générale du code plus claire, plus modulable et plus facile à maintenir et à déboguer.

Première approche

La programmation orientée objet se base sur un concept fondamental qui est que tout élément dans un script est un objet ou va pouvoir être considéré comme un objet. Pour comprendre ce qu'est précisément un objet, il faut avant tout comprendre ce qu'est une classe.

Une classe est un ensemble cohérent de code qui contient généralement à la fois des variables et des fonctions et qui va nous servir de plan pour créer des objets. Le but d'une classe va donc être de créer des objets que nous allons ensuite pouvoir manipuler.

Imaginons qu'on possède un site sur lequel les visiteurs peuvent s'enregistrer pour avoir accès à un espace personnel par exemple. Quand un visiteur s'enregistre pour la première fois, il devient un utilisateur du site.

Ici, on va essayer de comprendre comment faire pour créer le code qui permet cela. Pour information, ce genre de fonctionnalité est quasiment exclusivement réalisé en programmation orienté objet.

On veut « créer » un nouvel utilisateur à chaque fois qu'un visiteur s'enregistre à partir des informations qu'il nous a fournies. Un utilisateur va être défini par des attributs comme son nom d'utilisateur ou son mot de passe. Ces attributs vont être des variables. Ensuite, un utilisateur va pouvoir réaliser certaines actions spécifiques comme se connecter, se déconnecter, modifier son profil, etc. Ces actions vont être des fonctions.

Nous allons donc définir les attributs et actions / fonctions de notre utilisateur. Pour cela, on va créer un formulaire d'inscription sur notre site qui va demander un nom d'utilisateur et un mot de passe d'un côté, et allons devoir côté serveur récupérer ces informations et les associer à un utilisateur en précisant qu'il s'agit du nom d'utilisateur et du mot de passe. On va également définir les actions (fonctions) propres à nos utilisateurs : connexion, déconnexion, possibilité de commenter, etc.

On va donc créer un bloc de code qui va initialiser nos variables nom d'utilisateur et mot de passe par exemple et qui va définir les différentes actions que va pouvoir faire un utilisateur.

Ce bloc de code est le plan de base qui va nous servir à créer un nouvel utilisateur. On va également dire que c'est une classe. Dès qu'un visiteur s'inscrit, on va pouvoir créer un nouvel objet « utilisateur » à partir de cette classe et qui va disposer des variables et fonctions définies dans la classe. Lorsqu'on crée un nouvel objet, on dit également qu'on « instancie » ou qu'on crée une instance de notre classe.

Une classe est donc un bloc de code qui va contenir différentes variables, fonctions et éventuellement constantes et qui va servir de plan de création pour différents objets. Chaque objet créé à partir d'une même classe dispose des mêmes variables, fonctions et constantes définies dans la classe mais va pouvoir les implémenter différemment.

Classes et objets : Exemple de création

Reprenons notre exemple précédent et créons une première classe qu'on va appeler `Utilisateur`. Bien évidemment, nous n'allons pas créer tout un script de connexion utilisateur ici, mais simplement définir une première classe très simple.

En PHP, on crée une nouvelle classe avec le mot clé `class`. On peut donner n'importe quel nom à une nouvelle classe du moment qu'on n'utilise pas un mot réservé du PHP et que le premier caractère du nom de notre classe soit une lettre ou un underscore.

Par convention, on placera généralement chaque nouvelle classe créée dans un fichier à part et on placera également tous nos fichiers de classe dans un dossier qu'on pourra appeler `classes` ou `modeles` par exemple pour plus de simplicité (et de clarté du code).

On n'aura ensuite qu'à inclure les fichiers de classes nécessaires à l'exécution de notre script principal dans celui-ci grâce à une instruction `require` par exemple.

On va donc créer un nouveau fichier qu'on va appeler `Utilisateur.php`. Notez que par convention, le nom de fichier d'une classe a son premier caractère en majuscule.

Utilisateur.php

```php
<?php
class Utilisateur
{

}
```

Pour le moment, notre classe est vide. Vous pouvez remarquer que la syntaxe générale de déclaration d'une classe ressemble à ce qu'on a déjà vu avec les fonctions.

Nous allons également directement en profiter pour inclure notre classe dans notre fichier principal `index.php` avec une instruction `require`. Ici, mon fichier de classe est dans un sous-dossier «Modele» par rapport à mon fichier principal.

```php
<?php
require_once 'Modele/Utilisateur.php';
new Utilisateur();
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, init
    <title>Cours Orienté Objet PHP</title>
</head>
<body>
    <h1>Cours Orienté Objet PHP</h1>
</body>
</html>
```

Nous avons créé ce qu'on appelle une nouvelle instance de notre classe `Utilisateur`.

La syntaxe peut vous sembler particulière et c'est normal : rappelez-vous que je vous ai dit que le PHP orienté objet utilisait une syntaxe différente du PHP conventionnel.

Ici, le mot clef `new` est utilisé pour instancier une classe c'est-à-dire créer une nouvelle instance d'une classe.

Une instance correspond à la « copie » d'une classe. Le grand intérêt ici est qu'on va pouvoir effectuer des opérations sur chaque instance d'une classe sans affecter les autres instances.

Par exemple, imaginons que vous créiez deux nouveaux fichiers avec le logiciel Word. Chaque fichier va posséder les mêmes options : vous allez pouvoir changer la police, la taille d'écriture, etc. Cependant, le fait de personnaliser un fichier ne va pas affecter la mise en page du deuxième fichier.

A chaque fois qu'on instancie une classe, un **objet** est également automatiquement créé. Les termes « instance de classe » et « objet » ne désignent pas fondamentalement la même chose mais dans le cadre d'une utilisation pratique on pourra très souvent les confondre et c'est ce que nous allons faire dans ce cours.

Lorsqu'on instancie une classe, un objet est donc créé. Nous allons devoir capturer cet objet pour l'utiliser. Pour cela, nous allons généralement utiliser une variable qui deviendra alors une « variable objet » ou plus simplement un « objet ».

```
3   $Utilisateur = new Utilisateur();
```

Pour être tout à fait précis, notre variable en question ne va pas exactement contenir l'objet en soi mais plutôt une référence à l'objet. Nous reparlerons de ce point relativement complexe en fin de partie et allons pour le moment considérer que notre variable contient notre objet.

Classes et objets : L'essentiel à retenir

- Une classe est un « plan d'architecte » qui va nous permettre de créer des objets qui vont partager ses variables et ses fonctions. Chaque objet créé à partir d'une classe va disposer des mêmes variables et fonctions définies dans la classe mais va pouvoir implémenter ses propres valeurs.

- Pour l'instant, nous n'avons créé qu'une classe vide et donc il est possible que vous ne compreniez pas encore l'intérêt des classes. Dès le chapitre suivant, nous allons inclure des variables et des fonctions dans notre classe et plus le cours va avancer, plus vous allez comprendre les différents avantages de programmer en orienté objet et des classes. Je ne peux juste pas tout vous révéler d'un coup, il va falloir y aller brique après brique.

- Pour créer un objet, il faut instancier la classe en utilisant le mot clef `new`. Une instance est une « copie » de classe. On va stocker notre instance ou notre objet dans une variable pour pouvoir l'utiliser.

- Imaginez le plan de création de base d'un modèle de voiture. Ce plan va définir les différents éléments des voitures, c'est-à-dire de quoi va être composée chaque voiture ainsi que les actions de chaque voiture. Nous allons pouvoir créer des voitures à partir de ce plan qui va être l'équivalent d'une classe et chaque voiture créée à partir de celui-ci va posséder les éléments et actions définies dans le plan. Les voitures créées sont ici nos objets.

  Cependant, les éléments vont pouvoir avoir des apparences différentes : chaque voiture va posséder un moteur mais le moteur va pouvoir être différent pour chaque voiture (plus ou moins puissant). De même, chaque voiture va posséder une couleur mais la couleur de chaque voiture va pouvoir être différente.

  Chaque voiture créée à partir du plan va avoir accès au même nombre d'actions mais les différentes actions des voitures vont être différentes en fonction des éléments de chaque voiture : chaque voiture va pouvoir accélérer mais une voiture avec un moteur plus puissant va accélérer plus vite qu'une autre qui possède un moteur plus petit.

Les propriétés : définition et usage

Dans le chapitre précédent, nous avons créé une classe `Utilisateur` qui ne contenait pas de code. Nous avons également dit qu'une classe servait de plan pour créer des objets. Vous pouvez donc en déduire qu'une classe vide ne sert pas à grand-chose. Nous allons donc maintenant rendre notre classe plus fonctionnelle en lui ajoutant des éléments.

On va déjà pouvoir créer des variables à l'intérieur de nos classes. Les variables créées dans les classes sont appelées des propriétés, afin de bien les différencier des variables « classiques » créées en dehors des classes.

Une propriété, c'est donc tout simplement une variable définie dans une classe (ou éventuellement ajoutée à un objet après sa création).

Reprenons immédiatement notre classe `Utilisateur` et ajoutons lui deux propriétés qu'on va appeler `$username` et `$password` par exemple.

```php
<?php
class Utilisateur
{
    public $username;
    public $password;
}
```

Comme vous pouvez le voir, on déclare une propriété exactement de la même façon qu'une variable classique, en utilisant le signe `$`.

Le mot clé `public` signifie ici qu'on va pouvoir accès à nos propriétés depuis l'intérieur et l'extérieur de notre classe. Nous reparlerons de cela un peu plus tard, n'y prêtez pas attention pour le moment.

Ici, nous nous contentons de déclarer nos propriétés sans leur attribuer de valeur. Les valeurs des propriétés seront ici passées lors de la création d'un nouvel objet (lorsqu'un visiteur s'inscrira sur notre site).

Notez qu'il est tout-à-fait permis d'initialiser une propriété dans la classe, c'est-à-dire lui attribuer une valeur de référence à la condition que ce soit une valeur constante.

En situation réelle, ici, ce seront les visiteurs qui, lors de leur inscription, vont déclencher la création de nouveaux objets (qui vont être créés via notre classe `Utilisateur`.

Bien évidemment, réaliser le script complet qui va permettre cela est hors de question à ce niveau du cours. Nous allons donc nous contenter dans la suite de cette partie de créer de nouveaux objets manuellement pour pouvoir illustrer les différents concepts et leur fonctionnement.

Créons donc deux objets `$pierre` et `$mathilde` à partir de notre classe `Utilisateur` puis définissons ensuite des valeurs spécifiques pour les propriétés `$username` et `$password` pour chaque objet.

```php
<?php
require_once 'Modele/Utilisateur.php';
$Pierre = new Utilisateur();
$Mathilde = new Utilisateur();

$Pierre->username = "Pierre";
$Pierre->password = "abcdef";

$Mathilde->username = "Mathilde";
$Mathilde->password = "123456";
```

Ici, vous pouvez à nouveau observer une syntaxe que nous ne connaissons pas encore qui utilise le symbole ->.

Avant tout, souvenez-vous que les objets créés à partir d'une classe partagent ses propriétés et ses méthodes puisque chaque objet contient une « copie » de la classe.

Pour accéder aux propriétés définies originellement dans la classe depuis nos objets, on utilise l'opérateur -> qui est appelé opérateur objet. Notez qu'on ne précise pas de signe $ avant le nom de la propriété à laquelle on souhaite accéder dans ce cas.

Dans le cas présent, on va pouvoir accéder à notre propriété depuis notre objet (c'est-à-dire depuis l'extérieur de notre classe) car nous l'avons définie avec le mot clef `public`.

Notez cependant qu'il est généralement considéré comme une mauvaise pratique de laisser des propriétés de classes accessibles directement depuis l'extérieur de la classe et de mettre à jour leur valeur comme cela car ça peut poser des problèmes de sécurité dans le script.

Pour manipuler des propriétés depuis l'extérieur de la classe, nous allons plutôt créer des fonctions de classe dédiées afin que personne ne puisse directement manipuler nos propriétés et pour protéger notre script.

Les méthodes: définition et usage

Les fonctions définies à l'intérieur d'une classe sont appelées des méthodes. Là encore, nous utilisons un nom différent pour bien les différencier des fonctions créées en dehors des classes. On va pouvoir créer des méthodes dans nos classes dont le rôle va être d'obtenir ou de mettre à jour les valeurs de nos propriétés.

Dans notre classe `Utilisateur`, nous allons par exemple pouvoir créer trois méthodes :

- `getUsername(): Récupérer la valeur de $username`
- `setUsername(): Modifier la valeur de $username`
- `setPassword(): Modifier la valeur de $password`

Il y a beaucoup de nouvelles choses dans le code ci-contre que nous allons décortiquer ligne par ligne.

```php
<?php
class Utilisateur
{
    // On passe les propriétés en privées
    private $username;
    private $password;

    // On créé les méthodes en publique
    // On appelle cette méthode un "getter"
    public function getUsername()
    {
        // $this permet de faire référence à l'objet
        return $this->username;
    }

    // On appelle cette méthode un "setter"
    public function setUsername($newUsername)
    {
        $this->username = $newUsername;
    }

    public function setPassword($newPassword)
    {
        $this->password = $newPassword;
    }
}
```

Maintenant, voyons comment nous allons pouvoir utiliser ces méthodes dans notre index.php

```php
<?php
require_once 'Modele/Utilisateur.php';
$Pierre = new Utilisateur();
$Mathilde = new Utilisateur();

$Pierre->setUsername("Pierre");
$Pierre->setPassword("abcdef");

$Mathilde->setUsername("Mathilde");
$Mathilde->setPassword("123456");

?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
    <title>Cours Orienté Objet PHP</title>
</head>
<body>
    <h1>Cours Orienté Objet PHP</h1>
    <p>
        <?=$Pierre->getUsername();?>
    </p>
</body>
</html>
```

Ici, on inclut notre fichier de classe puis on instancie ensuite notre classe et on stocke cette instance dans un objet qu'on appelle `$Pierre`.

On va pouvoir accéder à nos méthodes `getUsername()`, `setUsername()` et `setPassword()` définies dans notre classe depuis notre objet puisqu'on a utilisé le mot clef `public` lorsqu'on les a déclarées.

En revanche, on ne va pas pouvoir accéder directement aux propriétés définies dans la classe puisqu'elles sont privées. On ne va donc pouvoir les manipuler que depuis l'intérieur de la classe via nos méthodes publiques.

On commence donc par utiliser nos méthodes pour définir une valeur à stocker dans nos propriétés.

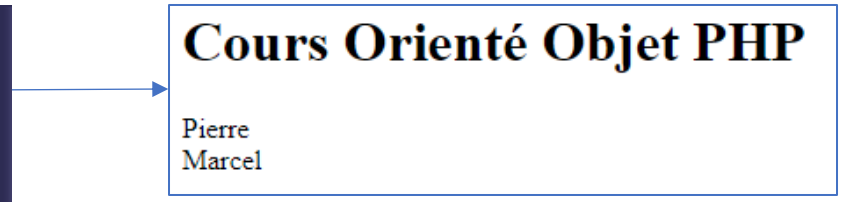
Faisons maintenant le test de modifier les propriétés de notre objet "Mathilde" et voyons ce qu'il se passe :

```php
<?php
require_once 'Modele/Utilisateur.php';
$Pierre = new Utilisateur();
$Mathilde = new Utilisateur();

$Pierre->setUsername("Pierre");
$Pierre->setPassword("abcdef");

$Mathilde->setUsername("Mathilde");
$Mathilde->setPassword("123456");

?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Cours Orienté Objet PHP</title>
</head>
<body>
    <h1>Cours Orienté Objet PHP</h1>
    <p>
        <?php
            $Mathilde->setUsername("Marcel");
            echo $Pierre->getUsername() . "<br>";
            echo $Mathilde->getUsername();
        ?>
    </p>
</body>
</html>
```



Comme vous pouvez le voir, on se sert ici de la même classe au départ qu'on va instancier plusieurs fois pour créer autant d'objets. Ces objets vont partager les propriétés et méthodes définies dans la classe qu'on va pouvoir utiliser de manière indépendante avec chaque objet.

La méthode constructeur: définition et usage

La méthode constructeur ou plus simplement le constructeur d'une classe est une méthode qui va être appelée (exécutée) automatiquement à chaque fois qu'on va instancier une classe.

Le constructeur va ainsi nous permettre d'initialiser des propriétés dès la création d'un objet, ce qui va pouvoir être très intéressant dans de nombreuses situations.

Pour illustrer l'intérêt du constructeur, reprenons notre class `Utilisateur` créée précédemment.

Lorsqu'on créé un nouvel objet à partir de cette classe, il faut ici ensuite appeler les méthodes `setUsername()` et `setPassword()` pour définir les valeurs de nos propriétés `$username` et `$password`, ce qui est en pratique loin d'être optimal.

Ici, on aimerait idéalement pouvoir définir immédiatement la valeur de nos deux propriétés lors de la création de l'objet (en récupérant en pratique les valeurs passées par l'utilisateur). Pour cela, on va pouvoir utiliser un constructeur.

```php
<?php
class Utilisateur
{
    // On passe les propriétés en privées
    private $username;
    private $password;

    public function __construct($un, $pw){
        $this->username = $un;
        $this->password = $pw;
    }

    // On créé les méthodes en publique
    // On appelle cette méthode un "getter"
    public function getUsername()
    {
        // $this permet de faire référence à l'objet
        return $this->username;
    }
}
```

On déclare un constructeur de classe en utilisant la syntaxe `function __construct()`. Il faut bien comprendre ici que le PHP va rechercher cette méthode lors de la création d'un nouvel objet et va automatiquement l'exécuter si elle est trouvée.

Nous allons utiliser notre constructeur pour initialiser certaines propriétés de nos objets dont nous pouvons avoir besoin immédiatement ou pour lesquelles il fait du sens de les initialiser immédiatement.

Dans notre cas, on veut stocker le nom d'utilisateur et le mot de passe choisi dans nos variables `$username` et `$password` dès la création d'un nouvel objet.

Pour cela, on va définir deux paramètres dans notre constructeur qu'on appelle ici `$un` (pour le **username**) et `$pw` (pour le **password**). Nous allons pouvoir passer les arguments à notre constructeur lors de l'instanciation de notre classe. On va ici passer un nom d'utilisateur et un mot de passe. Voici comment on va procéder en pratique :

```php
<?php
require_once 'Modele/Utilisateur.php';
$Pierre = new Utilisateur("Pierre", "abcdef");
$Mathilde = new Utilisateur("Mathilde", 123456);
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
    <title>Cours Orienté Objet PHP</title>
</head>
<body>
    <h1>Cours Orienté Objet PHP</h1>
    <p>
        <?php
            echo $Pierre->getUsername() . "<br>";
            echo $Mathilde->getUsername();
        ?>
    </p>
</body>
</html>
```

Lors de l'instanciation de notre classe `Utilisateur`, le PHP va automatiquement rechercher une méthode `__construct()` dans la classe à instancier et exécuter cette méthode si elle est trouvée. Les arguments passés lors de l'instanciation vont être utilisés dans notre constructeur et vont ici être stockés dans `$username` et `$password`.

Exercice : Instanciation d'un objet

Grâce aux connaissances que vous avez accumulés jusqu'à présent, vous allez devoir réaliser cet exercice :

- Modifiez le fichier "index.php" pour y include un formulaire d'inscription simple (input username, input password, et un bouton de submit). Ce formulaire devra envoyer les données en POST vers une page "inscription.php".

- La page "inscription.php" devra instancier l'objet utilisateur avec le username et le password qui auront été envoyés via le formulaire et afficher le message : "Inscription de l'utilisateur [username] effectuée" en utilisant l'objet Utilisateur (et pas la variable POST).

La méthode destructeur: définition et usage

La méthode destructeur va permettre de nettoyer les ressources avant que PHP ne libère l'objet de la mémoire.

Ici, vous devez bien comprendre que les variables-objets, comme n'importe quelle autre variable « classique », ne sont actives que durant le temps d'exécution du script puis sont ensuite détruites.

Cependant, dans certains cas, on voudra pouvoir effectuer certaines actions juste avant que nos objets ne soient détruits comme par exemple sauvegarder des valeurs de propriétés mises à jour ou fermer des connexions à une base de données ouvertes avec l'objet.

Dans ces cas-là, on va pouvoir effectuer ces opérations dans le destructeur puisque la méthode destructeur va être appelée automatiquement par le PHP juste avant qu'un objet ne soit détruit.

On va utiliser la syntaxe `function __destruct()` pour créer un destructeur. Notez qu'à la différence du constructeur, il est interdit de définir des paramètres dans un destructeur.

```php
<?php
class Utilisateur
{
    private $username;
    private $password;

    public function __construct($un, $pw){
        $this->username = $un;
        $this->password = $pw;
    }

    public function __destruct(){
        // code à executer
    }

    public function getUsername()
    {
        return $this->username;
    }
}
```

Le principe d'encapsulation

L'encapsulation désigne le principe de regroupement des données et du code qui les utilise au sein d'une même unité. On va très souvent utiliser le principe d'encapsulation afin de protéger certaines données des interférences extérieures en nous forçant à utiliser les méthodes définies pour manipuler les données. Dans le contexte de la programmation orientée objet en PHP, l'encapsulation correspond au groupement des données (propriétés, etc.) et des données permettant de les manipuler au sein d'une classe.

L'encapsulation va ici être très intéressante pour empêcher que certaines propriétés ne soient manipulées depuis l'extérieur de la classe. Pour définir qui va pouvoir accéder aux différentes propriétés, méthodes et constantes de nos classes, nous allons utiliser des limiteurs d'accès ou des niveaux de visibilité qui vont être représentés par les mots clefs `public`, `private` et `protected`.

Une bonne implémentation du principe d'encapsulation va nous permettre de créer des codes comportant de nombreux avantages. Parmi ceux-ci, le plus important est que l'encapsulation va nous permettre de garantir l'intégrité de la structure d'une classe en forçant l'utilisateur à passer par un chemin prédéfini pour modifier une donnée. Le principe d'encapsulation est l'un des piliers de la programmation orientée objet et l'un des concepts fondamentaux, avec l'héritage, le l'orienté objet en PHP.

Le principe d'encapsulation et la définition des niveaux de visibilité devra être au centre des préoccupations notamment lors de la création d'une interface modulable comme par exemple la création d'un site auquel d'autres développeurs vont pouvoir ajouter des fonctionnalités comme WordPress ou PrestaShop (avec les modules) ou lors de la création d'un module pour une interface modulable.

L'immense majorité de ces structures sont construits en orienté objet car c'est la façon de coder qui présente la plus grande modularité et qui permet la maintenance la plus facile car on va éclater notre code selon différentes classes. Il faudra néanmoins bien réfléchir à qui peut avoir accès à tel ou tel élément de telle classe afin de garantir l'intégrité de la structure et éviter des conflits entre des éléments de classes (propriétés, méthodes, etc.).

Les niveaux de visibilité des propriétés et des méthodes

On va pouvoir définir trois niveaux de visibilité ou d'accessibilité différents pour nos propriétés, méthodes et constantes grâce aux mots clés :

- Public : Les propriétés et méthodes seront accessibles partout (dans l'intérieur et dans l'extérieur de la classe)

- Private : Les propriétés et méthodes seront accessibles uniquement à l'intérieur de la classe qui les a définies.

- Protected : Les propriétés et méthodes vont être accessibles uniquement depuis l'intérieur de la classe qui les a définit ainsi que depuis les classe qui en hériteront. Nous reparlerons du concept d'héritage très bientôt dans ce cours.

Lors de la définition de propriétés dans une classe, il faudra obligatoirement définir un niveau de visibilité pour chaque propriété. Dans le cas contraire, une erreur sera renvoyée.

Pour les méthodes, en revanche, nous ne sommes pas obligés de définir un niveau de visibilité même si je vous recommande fortement de la faire à chaque fois. Les méthodes pour lesquelles nous n'avons défini aucun niveau de visibilité de manière explicite seront définies automatiquement comme publiques.

L’idée à retenir est qu’on ne peut pas accéder directement à nos propriétés définies comme privées depuis notre script principal c’est-à-dire depuis l’extérieur de la classe. Il faut qu’on passe par nos fonctions publiques qui vont elles pouvoir les manipuler depuis l’intérieur de la classe.

Si vous essayez par exemple d’afficher directement le contenu de la propriété `$username` en écrivant `echo $Pierre->username` dans notre script principal, par exemple, l’accès va être refusé et une erreur va être renvoyée. En revanche, si on définit notre propriété comme publique, ce code fonctionnera normalement.

Comment bien choisir le niveau de visibilité?

Vous l'aurez compris, la vraie difficulté ici va être de déterminer le « bon » niveau de visibilité des différents éléments de notre classe.

Cette problématique est relativement complexe et d'autant plus pour un débutant car il n'y a pas de directive absolue. De manière générale, on essaiera toujours de protéger un maximum notre code de l'extérieur et donc de définir le niveau d'accessibilité minimum possible.

Ensuite, il va falloir s'interroger sur le niveau de sensibilité de chaque élément et sur les impacts que peuvent avoir chaque niveau d'accès à un élément sur le reste d'une classe tout en identifiant les différents autres éléments qui vont avoir besoin d'accéder à cet élément pour fonctionner.

Ici, il n'y a vraiment pas de recette magique : il faut avoir une bonne expérience du PHP et réfléchir un maximum avant d'écrire son code pour construire le code le plus cohérent et le plus sécurisé possible. Encore une fois, cela ne s'acquière qu'avec la pratique.

Pour le moment, vous pouvez retenir le principe suivant qui fonctionnera dans la majorité des cas (mais qui n'est pas un principe absolu, attention) : on définira généralement nos méthodes avec le mot clef `public` et nos propriétés avec les mots clefs `protected` ou `private`.

Le principe d'Héritage : Intruduction

Vous devriez maintenant normalement commencer à comprendre la syntaxe générale utilisée en POO PHP.

Un des grands intérêts de la POO est qu'on va pouvoir rendre notre code très modulable, ce qui va être très utile pour gérer un gros projet ou si on souhaite le distribuer à d'autres développeurs. Cette modularité va être permise par le principe de séparation des classes qui est à la base même du PHP et par la réutilisation de certaines classes ou par l'implémentation de nouvelles classes en plus de classes de base déjà existantes.

Sur ce dernier point, justement, il va être possible plutôt que de créer des classes complètement nouvelles d'étendre (les possibilités) de classes existantes, c'est-à-dire de créer de nouvelles classes qui vont hériter des méthodes et propriétés de la classe qu'elles étendent (sous réserve d'y avoir accès) tout en définissant de nouvelles propriétés et méthodes qui leur sont propres.

Certains développeurs vont ainsi pouvoir proposer de nouvelles fonctionnalités sans casser la structure originale de notre code et de nos scripts. C'est d'ailleurs tout le principe de la solution e-commerce PrestaShop (c'est un CMS basé sur le langage PHP qui permet de réaliser des boutiques en ligne sans écrire une ligne de code).

Le principe d'Héritage : Comment faire ?

Nous allons pouvoir étendre une classe grâce au mot clef `extends`. En utilisant ce mot clef, on va créer une classe « fille » qui va **hériter** de toutes les propriétés et méthodes de son parent par défaut et qui va pouvoir les manipuler de la même façon (à condition de pouvoir y accéder).

Illustrons immédiatement cela en créant une nouvelle classe `Admin` qui va étendre notre classe `Utilisateur` définie dans les leçons précédentes par exemple.

Nous allons créer cette classe dans un nouveau fichier en utilisant le mot clef `extends` comme cela :

```php
<?php
class Administrateur extends Utilisateur {

}
```

Notre classe `Administrateur` étend la classe `Utilisateur` . Elle hérite et va pouvoir accéder à toutes les méthodes et aux propriétés de notre classe `Utilisateur` qui n'ont pas été définies avec le mot clef `private`.

Nous allons désormais pouvoir créer un objet à partir de notre classe `Administrateur` et utiliser les méthodes publiques définies dans notre classe `Utilisateur` et dont hérite `Administrateur`.

Attention cependant : afin d'être utilisées, les classes doivent déjà être connues et la classe mère doit être définie avant l'écriture d'un héritage. Il faudra donc bien penser à inclure les classes mère et fille dans le fichier de script principal en commençant par la mère.

Le principe d'Héritage : Les classes étendues et la visibilité

Dans le cas présent, notre classe mère `Utilisateur` possède deux propriétés avec un niveau de visibilité défini sur `private` et trois méthodes dont le niveau de visibilité est `public`.

Ce qu'il faut bien comprendre ici, c'est qu'on ne va pas pouvoir accéder aux propriétés de la classe `Utilisateur` depuis la classe étendue `Administrateur` : comme ces propriétés sont définies comme privées, elles n'existent que dans la classe `Utilisateur`.

Comme les méthodes de notre classe mère sont définies comme publiques, cependant, notre classe fille va en hériter et les objets créés à partir de la classe étendue vont donc pouvoir utiliser ces méthodes pour manipuler les propriétés de la classe mère.

Notez par ailleurs ici que si une classe fille ne définit pas de constructeur ni de destructeur, ce sont les constructeur et destructeur du parent qui vont être utilisés.

```php
<?php
require_once 'Modele/Utilisateur.php';
require_once 'Modele/Administrateur.php';
$Pierre = new Utilisateur("Pierre", "abcdef");
$Mathilde = new Administrateur("Mathilde", 123456);
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
    <title>Cours Orienté Objet PHP</title>
</head>
<body>
    <h1>Cours Orienté Objet PHP</h1>
    <p>
        <?php
            echo $Pierre->getUsername() . "<br>";
            echo $Mathilde->getUsername();
        ?>
    </p>
</body>
</html>
```

Ici, on crée deux objets `$Pierre` et `$Mathilde`. Notre objet `$Mathilde` est créé en instanciant la classe étendue `Administrateur`.

Notre classe `Administrateur` est pour le moment vide. Lors de l'instanciation, on va donc utiliser le constructeur de la classe parent `Utilisateur` pour initialiser les propriétés de notre objet. Cela fonctionne bien ici puisqu'encore une fois le constructeur est défini à l'intérieur de la classe parent et peut donc accéder aux propriétés de cette classe, tout comme la fonction `getUsername()`.

Cependant, si on essaie maintenant de manipuler les propriétés de notre classe parent depuis la classe `Administrateur`, cela ne fonctionnera pas car les propriétés sont définies comme privées dans la classe mère et ne vont donc exister que dans cette classe.

Si on définit une nouvelle méthode dans la classe `Administrateur` dont le rôle est de renvoyer la valeur de `$username` par exemple, le PHP va chercher une propriété `$username` dans la classe `Administrateur` et ne va pas la trouver.

Il va se passer la même chose si on réécrit une méthode de notre classe parent dans notre classe parent et qu'on tente de manipuler une propriété privée de la classe parent dedans, alors le PHP renverra une erreur.

Note : lorsqu'on redéfinit une méthode (non privée) ou une propriété (non privée) dans une classe fille, on dit qu'on « surcharge » celle de la classe mère. Cela signifie que les objets créés à partir de la classe fille utiliseront les définitions de la classe fille plutôt que celles de la classe mère.

```php
<?php
class Administrateur extends Utilisateur {

    // On tente d'afficher le $username
    public function getUsername2(){
        return $this->username;
    }

    /*
    On surcharge la méthode getUsername() de Utilisateur
    Ici, on conserve le même code dans la méthode, mais
    c'est cette méthode qui sera appelée depuis un objet de
    cette classe
    */
    public function getUsername(){
        return $this->username;
    }
}
```

```php
<body>
    <h1>Cours Orienté Objet PHP</h1>
    <p>
        <?php
            echo $Pierre->getUsername() . "<br>";
            echo $Mathilde->getUsername2() . "<br>";
            echo $Mathilde->getUsername() . "<br>";
        ?>
    </p>
</body>
```

Cours Orienté Objet PHP

Pierre

Notice: Undefined property: Administrateur::$username in [chemin vers le fichier] on line 6

Call Stack

#	Time	Memory	Function	Location
1	0.0008	409752	{main}( )	...\index.php:0
2	0.0014	412152	Administrateur->getUsername2( )	...\index.php:20

Ici, la valeur de `$username` de l'objet `$Mathilde` n'est jamais renvoyée puisqu'on essaie de la manipuler depuis la classe étendue `Administrateur`, ce qui est impossible.

Si on souhaite que des classes étendues puissent manipuler les propriétés d'une classe mère, alors il faudra définir le niveau de visibilité de ces propriétés comme `protected` dans la classe mère.

En définissant nos propriétés `$username` et `$password` comme `protected` dans la classe `Utilisateur`, notre classe fille peut tout à fait les manipuler et les méthodes des classes étendues utilisant ces propriétés vont fonctionner normalement.

```php
<?php
class Utilisateur
{
    protected $username;
    protected $password;

    public function __construct($un, $pw){
        $this->username = $un;
        $this->password = $pw;
    }

    public function __destruct(){
        // code à executer
    }

    public function getUsername()
    {
        return $this->username;
    }
}
```

Le principe d'Héritage : Définitions de nouveaux éléments

L’intérêt principal d’étendre des classes plutôt que d’en définir de nouvelles se trouve dans la notion d’héritage des propriétés et des méthodes : chaque classe étendue va hériter des propriétés et des méthodes (non privées) de la classe mère.

Cela permet donc une meilleure maintenance du code (puisqu’en cas de changement il suffit de modifier le code de la classe mère) et fait gagner beaucoup de temps dans l’écriture du code.

Cependant, créer des classes filles qui sont des « copies » d’une classe mère n’est pas très utile. Heureusement, bien évidemment, nous allons également pouvoir définir de nouvelles propriétés et méthodes dans nos classes filles et ainsi pouvoir « étendre » les possibilités de notre classe de départ.

Ici, nous pouvons par exemple définir de nouvelles propriétés et méthodes spécifiques à notre classe `Administrateur` . On pourrait par exemple permettre aux objets de la classe `Administrateur` de bannir un utilisateur ou d’obtenir la liste des utilisateurs bannis.

Pour cela, on peut rajouter une propriété `$bannis` qui va contenir la liste des utilisateurs bannis ainsi que deux méthodes `setBannis()` et `getBannis()`. Nous n'allons évidemment ici pas véritablement créer ce script mais simplement créer le code pour ajouter un nouveau prénom dans `$bannis` et pour afficher le contenu de la propriété.

```php
<?php
class Administrateur extends Utilisateur {

    protected $bannis;

    public function setBannis($username){
        $this->bannis[] = $username;
    }

    public function getBannis(){
        return $this->bannis;
    }
}
```

```php
<body>
    <h1>Cours Orienté Objet PHP</h1>
    <p>
        <?php
            echo $Pierre->getUsername() . "<br>";
            echo $Mathilde->getUsername() . "<br><hr>";

            $Mathilde->setBannis("Sardoche");
            $Mathilde->setBannis("Chap");

            echo "<b>Utilisateurs bannis :</b> <br>";

            foreach($Mathilde->getBannis() as $banni){
                echo $banni."<br>";
            }
        ?>
    </p>
</body>
```

En plus de définir de nouvelles propriétés et méthodes dans nos classes étendues, nous allons également pouvoir surcharger, c'est-à-dire redéfinir certaines propriétés ou méthodes de notre classe mère dans nos classes filles. Pour cela, il va nous suffire de déclarer à nouveau la propriété ou la méthode en question en utilisant le même nom et en lui attribuant une valeur ou un code différent.

Dans ce cas-là, il va cependant falloir respecter quelques règles notamment au niveau de la définition de la visibilité qui ne devra jamais être plus réduite dans la définition surchargée par rapport à la définition de base. Nous reparlerons de la surcharge dans la prochaine leçon et je vais donc laisser ce sujet de côté pour le moment.

Finalement, notez que rien ne nous empêche d'étendre à nouveau une classe étendue. Ici, par exemple, on pourrait tout à fait étendre notre classe `Administrateur` avec une autre classe `SuperAdministrateur`.

L'héritage va alors traverser les générations : les classes filles de `Administrateur` hériteront des méthodes et propriétés non privées de `Administrateur` mais également de celles de leur grand parent `Utilisateur`.