

# **PARTIE X**

**Programmation  
orientée objet :  
concepts de base**

# Introduction à la programmation orientée objet en PHP

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.

Je vais essayer de vous présenter le PHP orienté objet étape par étape et vous conseille fortement de ne pas faire l'impasse sur cette partie car cette nouvelle façon d'écrire son code va posséder des avantages indéniables qu'on va illustrer ensemble et est l'une des grandes forces du PHP.

Par ailleurs, il convient de noter que de nombreux langages serveurs possèdent une écriture orientée objet car encore une fois cette façon de coder va s'avérer très puissante.

## Qu'est-ce que la programmation orientée objet (POO) ?

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 classes et d'objets qu'on va créer à partir de ces classes. Une classe est une entité qui va pouvoir contenir un ensemble de fonctions et de variables.

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.

La programmation orientée objet va introduire des syntaxes différentes de ce qu'on a pu voir jusqu'à présent et c'est l'une des raisons principales pour lesquelles le POO en PHP est vu comme une chose obscure et compliquée par les débutants.

Au final, si vous arrivez à comprendre cette nouvelle syntaxe et si vous faites l'effort de comprendre les nouvelles notions qui vont être amenées, vous allez vous rendre compte que la POO n'est pas si complexe : ce n'est qu'une façon différente de coder qui va amener de nombreux avantages.

Pour vous donner un aperçu des avantages concrets de la POO, rappelez-vous du moment où on a découvert les fonctions prêtes à l'emploi en PHP. Aujourd'hui, on les utilise constamment car celles-ci sont très pratiques : elles vont effectuer une tâche précise sans qu'on ait à imaginer ni à écrire tout le code qui les fait fonctionner.

Maintenant, imaginez qu'on dispose de la même chose avec les objets : ce ne sont plus des fonctions mais des ensembles de fonctions et de variables enfermées dans des objets

et qui vont effectuer une tâche complexe qu'on va pouvoir utiliser directement pour commencer à créer des scripts complexes et complets !

## Classes, objets et instance : 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 similaires que nous allons ensuite pouvoir manipuler.

Il est généralement coutume d'illustrer ce que sont les objets et les classes en faisant des parallèles avec la vie de tous les jours. De manière personnelle, j'ai tendance à penser que ces parallèles embrouillent plus qu'ils n'aident à comprendre et je préfère donc vous fournir ici un exemple plus concret qui reste dans le cadre de la programmation.

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 en pratique quasiment exclusivement réalisé en programmation orienté objet.

Qu'essaie-t-on de réaliser ici ? 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. En programmation, ces attributs vont être traduits par des variables.

Ensuite, un utilisateur va pouvoir réaliser certaines actions spécifiques comme se connecter, se déconnecter, modifier son profil, etc. En programmation, ces actions sont représentées par des fonctions.

A chaque fois qu'un visiteur s'inscrit et devient utilisateur, on va donc devoir créer des variables « nom d'utilisateur », « mot de passe », etc. et définir leurs valeurs et donner les permissions à l'utilisateur d'utiliser les fonctions connexion, déconnexion, etc.

Pour cela, on va créer un formulaire d'inscription sur notre site qui va demander un nom d'utilisateur et un mot de passe, etc. On va également définir les actions (fonctions) propres à nos utilisateurs : connexion, déconnexion, possibilité de commenter, etc.

Sur notre site, on s'attend à avoir régulièrement de nouveaux visiteurs qui s'inscrivent et donc de nouveaux utilisateurs. Il est donc hors de question de définir toutes ces choses manuellement à chaque fois.

A la place, on va plutôt 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 des objets similaires. 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 clef **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** par exemple pour plus de simplicité.

Comme je vous l'ai dit plus haut, l'un des grands avantages de la POO se situe dans la clarté du code produit et cette clarté est notamment le résultat d'une bonne séparation 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 en plus de notre fichier principal **cours.php** qu'on va appeler **utilisateur.class.php**. Notez qu'on appellera généralement nos fichiers de classe « maClasse.class.php » afin de bien les différencier des autres et par convention une nouvelle fois.

Dans ce fichier de classe, nous allons donc pour le moment tout simplement créer une classe **Utilisateur** avec le mot clef **class**.

```
cours.php x cours.css x utilisateur.class.php x
... htdocs > Cours PHP & MySQL > classes > utilisateur.class.php > Ln: 8 Col: 1 UTF-8
1 <?php
2     class Utilisateur{
3
4     }
5 ?>
```

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 `cours.php` avec une instruction `require`. Ici, mon fichier de classe est dans un sous-dossier « classes » par rapport à mon fichier principal.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```

Ensuite, nous allons rajouter la ligne suivante dans notre script principal :

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

      new Utilisateur();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```

Ci-dessus, 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.

Pour information, la grande différence est que chaque instance de classe est unique et peut donc être identifiée de manière unique ce qui n'est pas le cas pour les objets d'une même classe.

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 ».

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

      $pierre = new Utilisateur();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```

Pour être tout à fait précis, notre variable en question ne va 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, instances 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.

# Propriétés et méthodes en PHP orienté objet

Dans cette nouvelle leçon, nous allons définir ce que sont les propriétés et les méthodes et allons ajouter des propriétés et des méthodes à notre classe `Utilisateur` créée dans la leçon précédente afin de tendre vers une classe fonctionnelle.

## 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 `$user_name` et `$user_pass` par exemple (pour « nom d'utilisateur » et « mot de passe utilisateur »).

```
<?php
class Utilisateur{
    public $user_name;
    public $user_pass;
}
?>
```

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

Le mot clef `public` signifie ici qu'on va pouvoir accéder 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 (elle doit pouvoir être évaluée pendant la compilation du code et ne doit pas dépendre d'autres facteurs déterminés lors de l'exécution du code).



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 `$user_name` et `$user_pass` pour chaque objet.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

      $pierre = new Utilisateur();
      $mathilde = new Utilisateur();

      $pierre->user_name = 'Pierre';
      $pierre->user_pass = 'abcdef';

      $mathilde->user_name = 'Math';
      $mathilde->user_pass = 123456;
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```

Ici, vous pouvez à nouveau observer une syntaxe que nous ne connaissons pas encore qui utilise le symbole `->`. Expliquons ce qu'il se passe ici.

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.

Cet opérateur sert à indiquer au PHP qu'on souhaite accéder à un élément défini dans notre classe via un objet créé à partir de cette classe. 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

---

Nous allons également pouvoir déclarer des fonctions à l'intérieur de nos classes.

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.

Une méthode est donc tout simplement une fonction déclarée dans une classe. 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 qu'on va appeler `getNom()`, `setNom()` et `setPass()`.

Le rôle de `getNom()` va être de récupérer la valeur contenue dans la propriété `$user_name`.

Les rôles de `setNom()` et de `setPass()` vont être respectivement de définir ou de modifier la valeur contenue dans les propriétés `$user_name` et `$user_pass` relativement à l'objet courant (la valeur de la propriété ne sera modifiée que pour l'objet couramment utilisé).

Profitez-en pour noter que les méthodes qui servent à définir / modifier / mettre à jour une valeur sont appelées des `setters`. Généralement, on fera commencer leur nom par `set` afin de bien les identifier comme on l'a fait pour nos méthodes `setNom()` et `setPass()`.

De même, les méthodes qui servent à récupérer des valeurs sont appelées des `getters` et on fera commencer leur nom par `get`. Ces notations sont des conventions qui ont pour but de clarifier les scripts et de simplifier la vie des développeurs.

Rajoutons nos méthodes à l'intérieur de notre classe :

```

<?php
class Utilisateur{
    private $user_name;
    private $user_pass;

    public function getNom(){
        return $this->user_name;
    }

    public function setNom($new_user_name){
        $this->user_name = $new_user_name;
    }

    public function setPasse($new_user_pass){
        $this->user_pass = $new_user_pass;
    }
}
?>

```

Il y a beaucoup de nouvelles choses dans ce code que nous allons décortiquer ligne par ligne.

Tout d'abord, vous pouvez commencer par noter qu'on a modifié le niveau d'accessibilité (c'est-à-dire la portée) de nos propriétés `$user_name` et `$user_pass` dans la définition de notre classe en modifiant le mot clef devant la déclaration de nos propriétés.

En effet, nous avons utilisé le mot clef `private` à la place de `public` qui signifie que nos propriétés ne sont désormais plus accessibles que depuis l'intérieur de la classe.

En revanche, nous avons utilisé le mot clef `public` devant nos deux méthodes afin de pouvoir les utiliser depuis l'extérieur de la classe.

Ensuite, vous devriez également avoir remarqué qu'on utilise un nouveau mot clef dans ces deux méthodes : le mot clef `$this`. Ce mot clef est appelé pseudo-variable et sert à faire référence à l'objet couramment utilisé.

Cela signifie que lorsqu'on va appeler une méthode de classe depuis un objet, la pseudo-variable `$this` va être remplacée (substituée) par l'objet qui utilise la méthode actuellement.

Prenons un exemple concret afin que vous compreniez bien ce point important. Pour cela, retournons dans notre fichier de script principal et modifions quelques lignes comme cela :

```

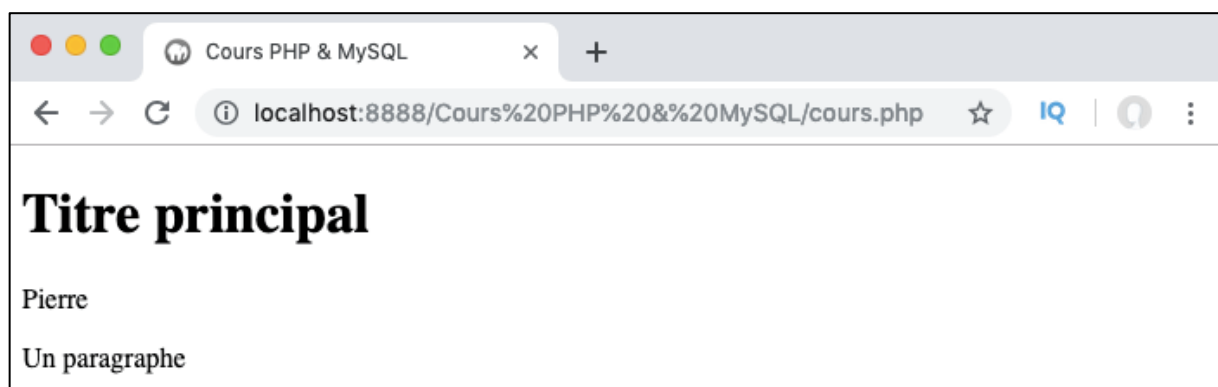
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

      $pierre = new Utilisateur();
      $mathilde = new Utilisateur();

      $pierre->setNom('Pierre');
      $pierre->setPasse('abcdef');
      echo $pierre->getNom();
    ?>
    <p>Un paragraphe</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 **setNom()**, **getNom()** et **setPass()** 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 **setNom()** et **setPass()** pour définir une valeur à stocker dans nos propriétés **\$user\_name** et **\$user\_pass**.

Pour cela, on utilise à nouveau l'opérateur objet `->` pour exécuter notre méthode depuis notre objet.

Les méthodes `setNom()` et `setPass()` vont chacune avoir besoin qu'on leur passe un argument pour fonctionner normalement. Ces arguments vont correspondre au nom d'utilisateur et au mot de passe choisis par l'utilisateur qu'on va stocker dans les propriétés `$user_name` et `$user_pass` relatives à notre instance.

Note : encore une fois, en pratique, nous recevrons ces données de l'utilisateur lui-même. Il faudra donc bien faire attention à les vérifier et à les traiter (comme n'importe quelle autre donnée envoyée par l'utilisateur) avant d'effectuer toute opération afin d'être sûr que les données envoyées sont conformes au format attendu et ne sont pas dangereuses. Pour le moment, nous nous passons de cette étape.

Comme je vous l'ai dit plus haut, la pseudo-variable `$this` dans le code de notre méthode sert à faire référence à l'objet couramment utilisé.

Dans le cas présent, la ligne `$this->user_name = $new_user_name` signifie littéralement que l'on souhaite stocker dans la propriété `$user_name` définie dans notre classe le contenu de `$new_user_name` (c'est-à-dire l'argument qui va être passé) POUR un objet en particulier et en l'occurrence ici pour `$pierre`. La pseudo-variable `$this` fait référence dans ce cas à `$pierre`.

De la même façon, on retourne le contenu de `$user_name` relatif à notre objet `$pierre` grâce à `getNom()`.

Créons immédiatement un deuxième objet afin de bien illustrer une nouvelle fois le fait que `$this` sert de référence à l'objet couramment utilisé.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

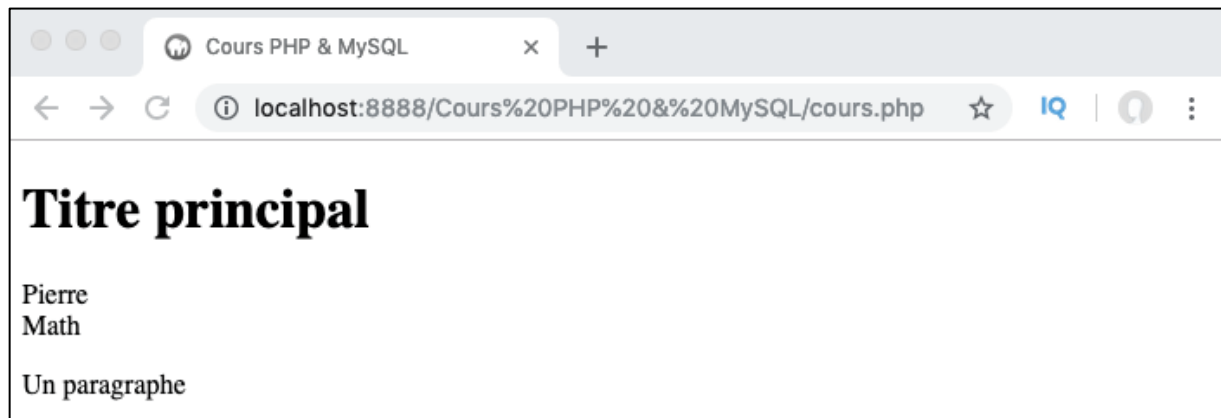
      $pierre = new Utilisateur();
      $mathilde = new Utilisateur();

      $pierre->setNom('Pierre');
      $pierre->setPasse('abcdef');

      $mathilde->setNom('Math');
      $mathilde->setPasse(123456);

      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';
    ?>
    <p>Un paragraphe</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.

# Constructeur et destructeur d'objets

Dans cette nouvelle leçon, nous allons nous intéresser à deux méthodes spéciales qui vont pouvoir être définies dans nos classes qui sont les méthodes constructeur et destructeur.

## 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.

```
<?php
class Utilisateur{
    private $user_name;
    private $user_pass;

    public function getNom(){
        return $this->user_name;
    }

    public function setNom($new_user_name){
        $this->user_name = $new_user_name;
    }

    public function setPasse($new_user_pass){
        $this->user_pass = $new_user_pass;
    }
}
?>
```

Notre classe possède deux propriétés `$user_name` et `$user_pass` et trois propriétés `getNom()`, `setNom()` et `setPasse()` dont les rôles respectifs sont de retourner le nom d'utilisateur de l'objet courant, de définir le nom d'utilisateur de l'objet courant et de définir le mot de passe de l'objet courant.

Lorsqu'on crée un nouvel objet à partir de cette classe, il faut ici ensuite appeler les méthodes `setNom()` et `setPasse()` pour définir les valeurs de nos propriétés `$user_name` et `$user_pass`, 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
class Utilisateur{
    private $user_name;
    private $user_pass;

    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function getNom(){
        return $this->user_name;
    }
}
?>

```

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 `$user_name` et `$user_pass` dès la création d'un nouvel objet.

Pour cela, on va définir deux paramètres dans notre constructeur qu'on appelle ici `$n` et `$p`. 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 :



```

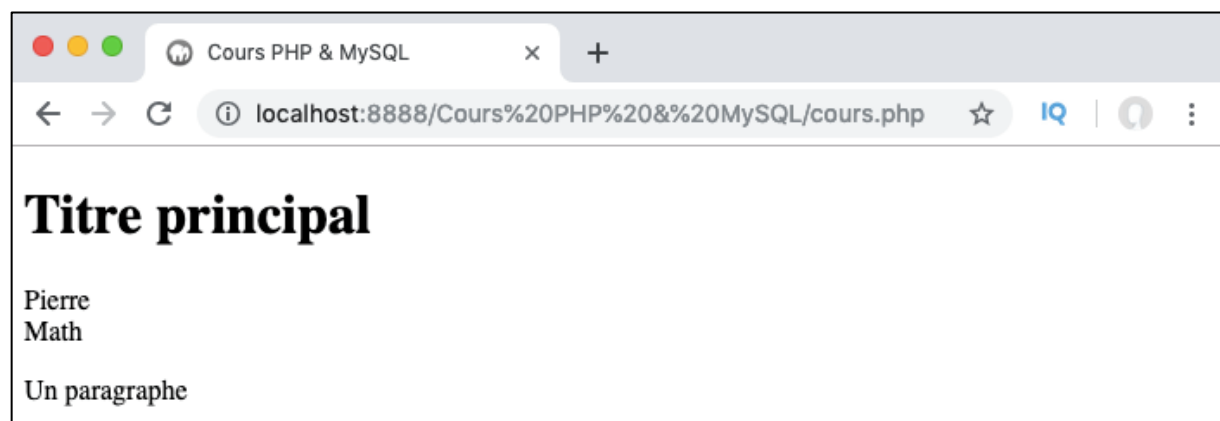
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

      $pierre = new Utilisateur('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';
    ?>
    <p>Un paragraphe</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 `$user_name` et `$user_pass`.

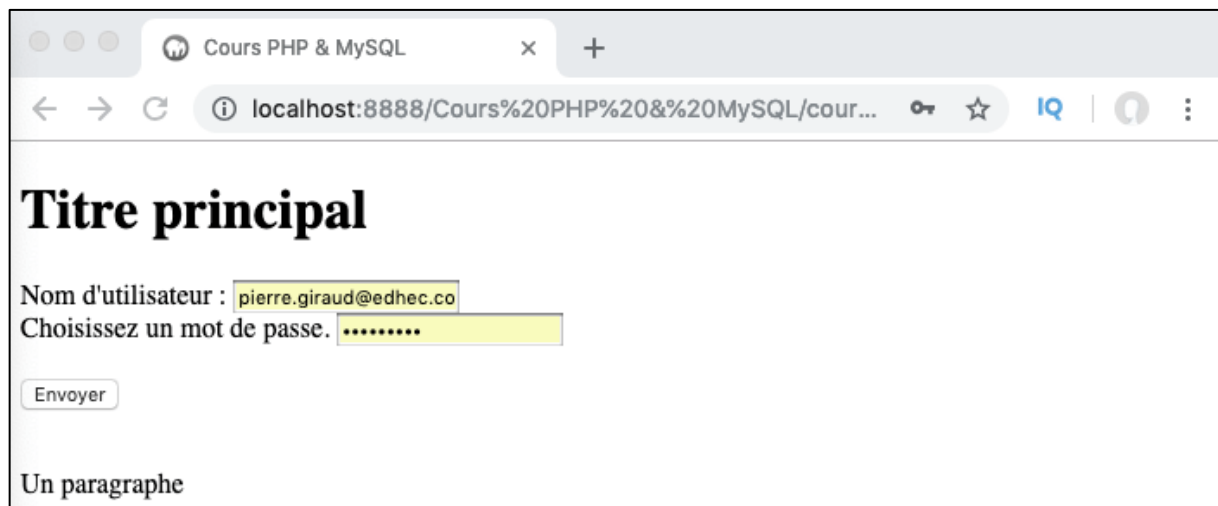
Ici, on peut déjà avoir un premier aperçu d'un script « utile » en pratique si vous le désirez afin de bien comprendre à quoi vont servir les notions étudiées jusqu'à maintenant « en vrai » :

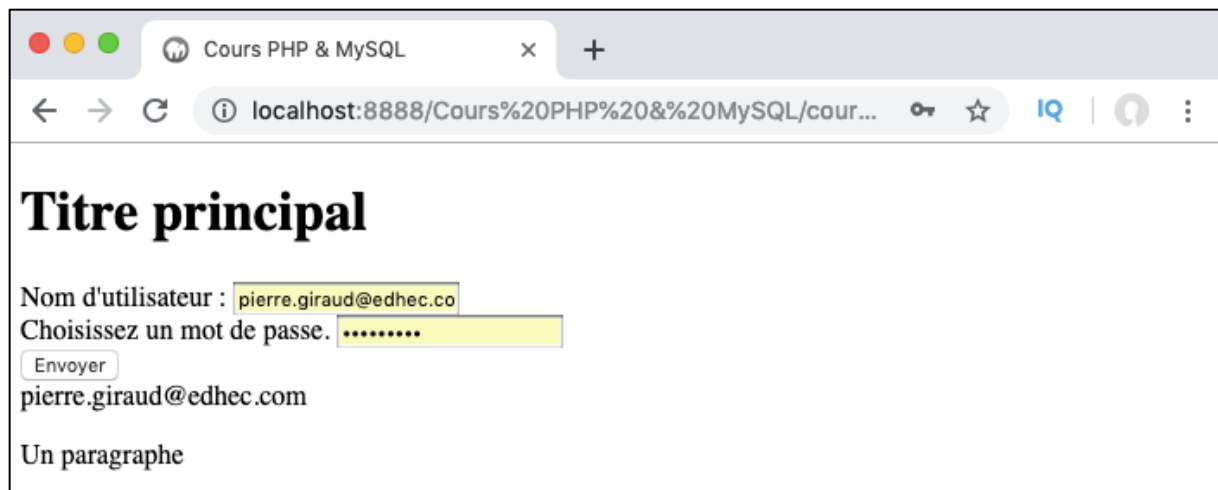
```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <form action='cours.php' method='post'>
      <label for='nom'>Nom d'utilisateur : </label>
      <input type='text' name='nom' id='nom'><br>
      <label for='pass'>Choisissez un mot de passe.</label>
      <input type='password' name='pass' id='pass'><br>
      <input type='submit' value='Envoyer'>
    </form>
    <?php
      require 'classes/utilisateur.class.php';
      //+ Vérification des données reçues (regex + filtres)
      //+ Stockage des données (base de données)
      $pierre = new Utilisateur($_POST['nom'], $_POST['pass']);
      echo $pierre->getNom(). '<br>';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```





**Titre principal**

Nom d'utilisateur : pierre.giraud@edhec.co

Choisissez un mot de passe. \*\*\*\*\*

Envoyer

pierre.giraud@edhec.com

Un paragraphe

On crée ici un formulaire en HTML qui va demander un nom d'utilisateur et un mot de passe à nos visiteurs. Vous pouvez imaginer que ce formulaire est un formulaire d'inscription. Ensuite, en PHP, on récupère les informations envoyées et on les utilise pour créer un nouvel objet. L'intérêt ici est que notre objet va avoir accès aux méthodes définies dans notre classe.

Bien évidemment, ici, notre script n'est pas complet puisqu'en pratique il faudrait analyser la cohérence des données envoyées et vérifier qu'elles ne sont pas dangereuses et car nous stockerions également les informations liées à un nouvel utilisateur en base de données pour pouvoir par la suite les réutiliser lorsque l'utilisateur revient sur le site et souhaite s'identifier.

## La méthode destructeur

---

De la même façon, on va également pouvoir définir une méthode destructeur ou plus simplement un destructeur de classe.

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.

Il est difficile d'expliquer concrètement l'intérêt d'un destructeur ici à des personnes qui n'ont pas une connaissance poussée du PHP. Pas d'inquiétude donc si vous ne

comprenez pas immédiatement l'intérêt d'une telle méthode, on pourra illustrer cela de manière plus concrète lorsqu'on parlera des bases de données.

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
class Utilisateur{
    private $user_name;
    private $user_pass;

    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        return $this->user_name;
    }
}
?>
```

# Encapsulation et visibilité des propriétés et méthodes

Dans cette nouvelle leçon, nous allons présenter le principe d'encapsulation et comprendre ses enjeux et intérêt et allons voir comment implémenter ce principe en pratique via les niveaux de visibilité des propriétés, méthodes et des constantes de classe.

Nous étudierons plus en détail les constantes dans une prochaine leçon, je n'en parlerai donc pas véritablement ici.

## 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 forçant l'utilisateur 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, méthodes et constantes en POO PHP

---

On va pouvoir définir trois niveaux de visibilité ou d'accessibilité différents pour nos propriétés, méthodes et constantes (depuis PHP 7.1.0) grâce aux mots clefs **public**, **private** et **protected**.

Les propriétés, méthodes ou constantes définies avec le mot clef **public** vont être accessibles partout, c'est-à-dire depuis l'intérieur ou l'extérieur de la classe.

Les propriétés, méthodes ou constantes définies avec le mot clef **private** ne vont être accessibles que depuis l'intérieur de la classe qui les a définies.

Les propriétés, méthodes ou constantes définies avec le mot clef **protected** ne vont être accessibles que depuis l'intérieur de la classe qui les a définies ainsi que depuis les classes qui en héritent ou la classe parente. Nous reparlerons du concept d'héritage dans la prochaine leçon.

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 et constantes, 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 et constantes pour lesquelles nous n'avons défini aucun niveau de visibilité de manière explicite seront définies automatiquement comme publiques.

Ici, lorsqu'on parle « d'accès », on se réfère à l'endroit où les propriétés et méthodes sont utilisées. Reprenons l'exemple de notre classe utilisateur pour bien comprendre :

```
<?php
class Utilisateur{
    private $user_name;
    private $user_pass;

    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        return $this->user_name;
    }
}
?>
```

Notre classe possède deux propriétés définies comme **private** et trois méthodes définies comme **public**. « L'intérieur » de la classe correspond au code ci-dessus.

Ici, on s'aperçoit par exemple que notre constructeur manipule nos propriétés `$user_name` et `$user_pass`. Il en a le droit puisque le constructeur est également défini dans la classe, tout comme la méthode `getNom()`.

Nos méthodes sont ici définies comme publiques, ce qui signifie qu'on va pouvoir les exécuter depuis l'extérieur de la classe. Lorsqu'on crée un nouvel objet dans notre script principal à partir de notre classe, par exemple, on appelle (implicitement) le constructeur depuis l'extérieur de la classe.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

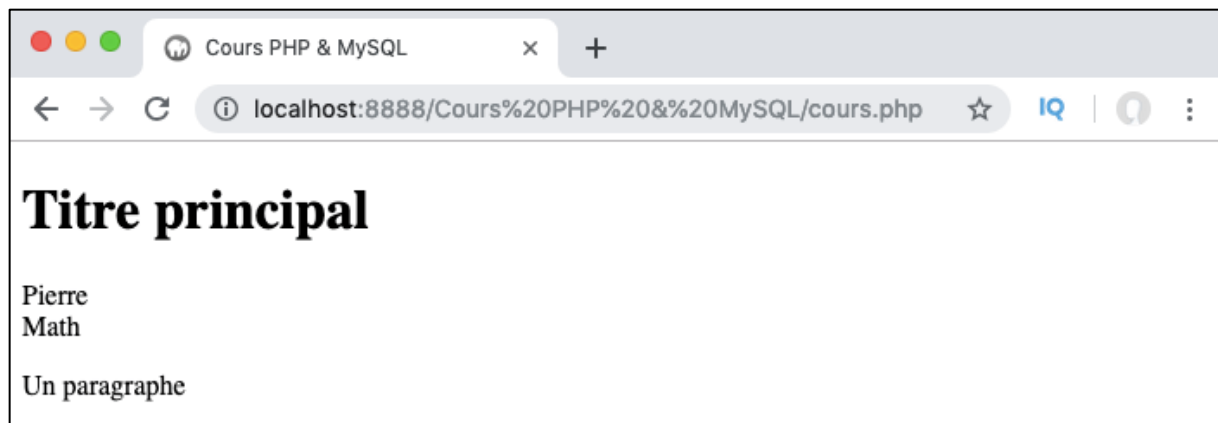
  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';

      $pierre = new Utilisateur('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```

On a le droit de la faire puisque notre constructeur est défini comme public. Ensuite, notre constructeur va modifier la valeur de nos propriétés depuis l'intérieur de la classe et c'est cela qu'il faut bien comprendre : on peut ici modifier la valeur de nos propriétés indirectement car c'est bien notre constructeur défini dans la classe qui les modifie.

La même chose se passe avec la méthode `getNom()` qui affiche la valeur de la propriété `$user_name`. Cette méthode est définie comme publique, ce qui signifie qu'on peut l'appeler depuis l'extérieur de la classe. Ensuite, notre méthode va récupérer la valeur de `$user_name` depuis l'intérieur de la classe puisque c'est là qu'elle a été définie.



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 pouvoir les manipuler depuis l'intérieur de la classe.

Si vous essayez par exemple d'afficher directement le contenu de la propriété `$user_name` en écrivant `echo $pierre->user_name` 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 des différents éléments d'une classe

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`.



# Classes étendues et héritage

Dans cette nouvelle leçon, nous allons voir comment étendre une classe et comprendre les intérêts qu'il va y avoir à faire cela. Nous expliquerons également comment fonctionne l'héritage dans le cadre de la programmation orientée objet en PHP.

## Étendre une classe : principe et utilité

---

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 (nous reparlerons de cela en fin de chapitre).

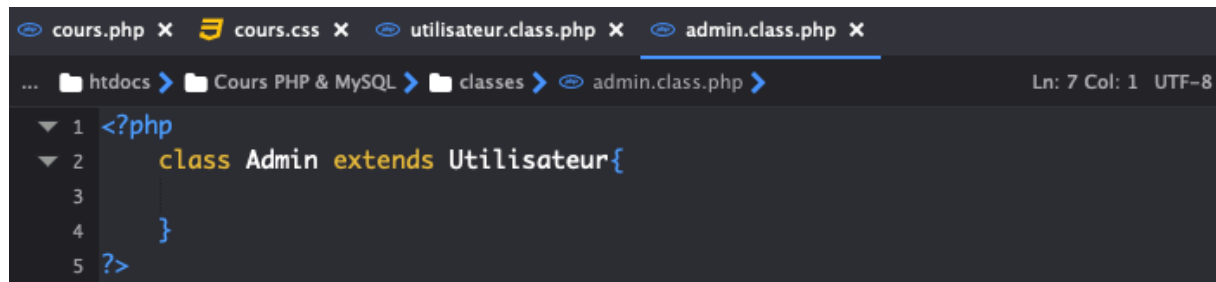
## Comment étendre une classe en pratique

---

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 :



```
1 <?php
2 class Admin extends Utilisateur{
3
4 }
5 ?>
```

Notre classe `Admin` 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 `Admin` et utiliser les méthodes publiques définies dans notre classe `Utilisateur` et dont hérite `Admin`.

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.

## 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 `Admin` : 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.

```

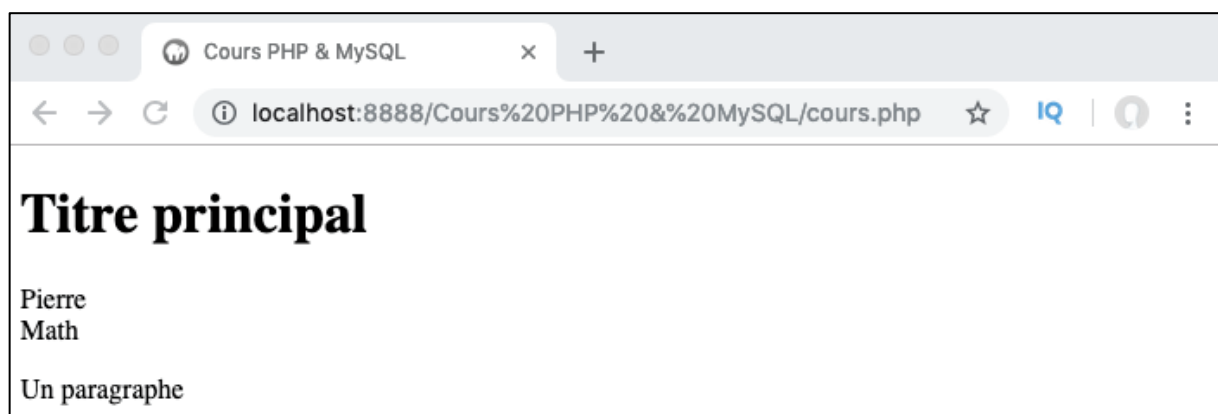
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on crée deux objets `$pierre` et `$mathilde`. Notre objet `$pierre` est créé en instanciant la classe étendue `Admin`.

Notre classe `Admin` 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 `getNom()`.

Cependant, si on essaie maintenant de manipuler les propriétés de notre classe parent depuis la classe `Admin`, 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 `Admin` dont le rôle est de renvoyer la valeur de `$user_name` par exemple, le PHP va chercher une propriété `$user_name` dans la classe `Admin` 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.

Regardez plutôt l'exemple ci-dessous :

```
<?php
class Admin extends Utilisateur{
    ///On tente d'afficher $user_name qui n'existe pas dans Admin
    public function getNom2(){
        return $this->user_name;
    }

    /*On surcharge la méthode getNom() de Utilisateur. Ici, on conserve
    *le même code dans la méthode mais c'est cette méthode qui sera
    *utilisée par $pierre*/
    public function getNom(){
        return $this->user_name;
    }
}
?>
```

```

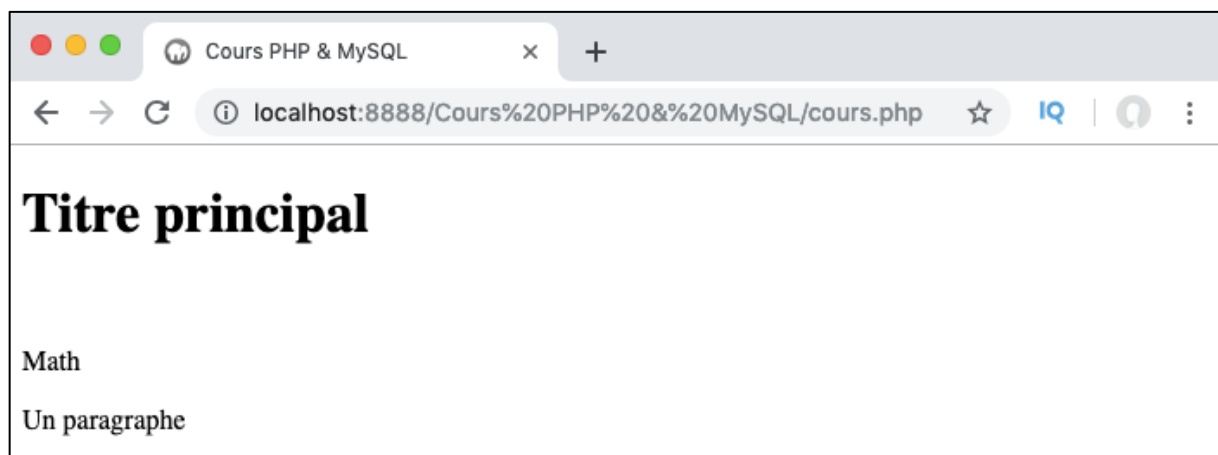
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      echo $pierre->getNom2(). '<br>';
      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, la valeur de `$user_name` de l'objet `$pierre` n'est jamais renvoyée puisqu'on essaie de la manipuler depuis la classe étendue `Admin`, 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.

```

<?php
class Utilisateur{
    protected $user_name;
    protected $user_pass;

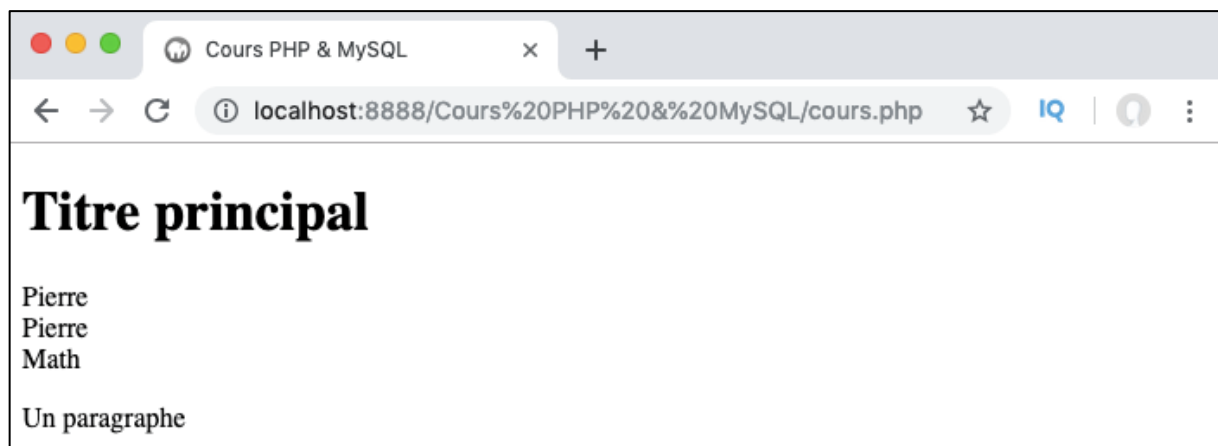
    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        return $this->user_name;
    }
}
?>

```

En définissant nos propriétés `$user_name` et `$user_pass` 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.



## Définition de nouvelles propriétés et méthodes dans une classe étendue et surcharge

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 (puisque 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 `Admin`. On pourrait par exemple permettre aux objets de la classe `Admin` de bannir un utilisateur ou d'obtenir la liste des utilisateurs bannis.

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

```
<?php
class Admin extends Utilisateur{
    protected $ban;

    public function setBan($b){
        $this->ban[] .= $b;
    }
    public function getBan(){
        echo 'Utilisateurs bannis par ' . $this->user_name . ' : ';
        foreach($this->ban as $valeur){
            echo $valeur . ', ';
        }
    }
}
?>
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

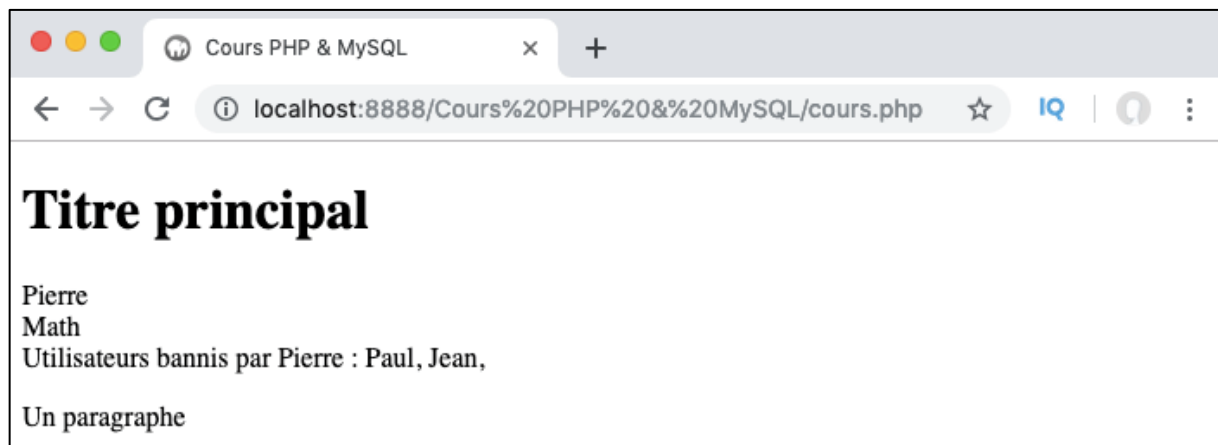
  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';

      $pierre->setBan('Paul');
      $pierre->setBan('Jean');
      echo $pierre->getBan();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



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 `Admin` avec une autre classe `SuperAdmin`.

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

## Comprendre la puissance et les risques liés aux classes étendues à travers l'exemple de la solution e-commerce PrestaShop

---

L'architecture du célèbre logiciel e-commerce PrestaShop a été créée en PHP orienté objet.

Cela rend PrestaShop modulable à l'infini et permet à des développeurs externes de développer de nouvelles fonctionnalités pour la solution.

En effet, le logiciel PrestaShop de base contient déjà de nombreuses classes et certaines vont pouvoir être étendues par des développeurs externes tandis que d'autres, plus sensibles ou essentielles au fonctionnement de la solution (ce qu'on appelle des classes « cœurs ») ne vont offrir qu'un accès limité.

Le fait d'avoir créé PrestaShop de cette manière est une formidable idée puisque ça permet aux développeurs de développer de nouveaux modules qui vont s'intégrer parfaitement à la solution en prenant appui sur des classes déjà existantes dans PrestaShop.

Cependant, c'est également le point principal de risque et l'ambiguïté majeure par rapport à la qualité de cette solution pour deux raisons.

Le premier problème qui peut survenir est que certains développeurs peuvent par mégarde ou tout simplement par manque d'application surcharger (c'est-à-dire réécrire ou encore substituer) certaines méthodes ou propriétés de classes lorsqu'ils créent leurs modules et le faire d'une façon qui va amener des bugs et des problèmes de sécurité sur les boutiques en cas d'installation du module en question.

A priori, l'équipe de validation des modules de PrestaShop est là pour éviter que ce genre de modules passent et se retrouvent sur la place de marché officielle.

Le deuxième problème est beaucoup plus insidieux et malheureusement quasiment impossible à éviter : imaginons que vous installiez plusieurs modules de développeurs différents sur votre solution de PrestaShop de base.

Si vous êtes malchanceux, il est possible que certains d'entre eux tentent d'étendre une même classe et donc de surcharger les mêmes méthodes ou propriétés, ou encore utilisent un même nom en créant une nouvelle classe ou en étendant une classe existante.

Dans ce cas-là, il y aura bien entendu un conflit dans le code et selon la gravité de celui-ci cela peut faire totalement planter votre boutique. Le problème étant ici que vous n'avez aucun moyen d'anticiper cela à priori lorsque vous êtes simple marchand et non un développeur aguerri.

Finalement, notez que si vous créez une architecture en POO PHP et que vous laissez la possibilité à des développeurs externes de modifier ou d'étendre cette architecture, vous devrez toujours faire bien attention à proposer une rétrocompatibilité de votre code à chaque mise à jour importante.

En effet, imaginons que vous modifiez une classe de votre architecture : vous devrez toujours faire en sorte que les codes d'autres développeurs utilisant cette classe avant la mise à jour restent valides pour ne pas que tout leur code plante lorsqu'ils vont eux-mêmes mettre la solution à jour (ou tout au moins les prévenir avant de mettre la mise à jour en production pour qu'ils puissent adapter leur code).

# Surcharge et opérateur de résolution de portée

Dans cette nouvelle leçon, nous allons voir précisément ce qu'est la surcharge d'éléments dans le cadre du PHP orienté objet ainsi que les règles liées à la surcharge.

Nous allons également en profiter pour introduire l'opérateur de résolution de portée, un opérateur qu'il convient de connaître car il va nous servir à accéder à divers éléments dans nos classes et notamment aux éléments surchargés, aux constantes et aux éléments statiques qu'on étudiera dans les prochaines leçons.

## La surcharge de propriétés et de méthodes en PHP orienté objet

---

En PHP, on dit qu'on « surcharge » une propriété ou une méthode d'une classe mère lorsqu'on la redéfinit dans une classe fille.

Pour surcharger une propriété ou une méthode, il va falloir la redéclarer en utilisant le même nom. Par ailleurs, si on souhaite surcharger une méthode, il faudra également que la nouvelle définition possède le même nombre de paramètres.

De plus, notez qu'on ne va pouvoir surcharger que des méthodes et propriétés définies avec des niveaux de visibilité **public** ou **protected** mais qu'il va être impossible de surcharger des éléments définis comme **private** puisque ces éléments n'existent / ne sont accessibles que depuis la classe qui les déclare.

Finalement, notez que lorsqu'on surcharge une propriété ou une méthode, la nouvelle définition doit obligatoirement posséder un niveau de restriction de visibilité plus faible ou égal, mais ne doit en aucun cas avoir une visibilité plus restreinte que la définition de base. Par exemple, si on surcharge une propriété définie comme **protected**, la nouvelle définition de la propriété ne pourra être définie qu'avec **public** ou **protected** mais pas avec **private** qui correspond à un niveau de visibilité plus restreint.

Notez qu'il va être relativement rare d'avoir à surcharger des propriétés. Généralement, nous surchargerons plutôt les méthodes d'une classe mère depuis une classe fille. Prenons immédiatement un exemple concret en surchargeant la méthode `getNom()` de notre classe parent **Utilisateur** dans notre classe étendue **Admin**.

```

<?php
class Utilisateur{
    protected $user_name;
    protected $user_pass;

    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        return $this->user_name;
    }
}
?>

```

```

<?php
class Admin extends Utilisateur{
    protected $ban;

    public function getNom(){
        return strtoupper($this->user_name);
    }

    public function setBan($b){
        $this->ban[] .= $b;
    }

    public function getBan(){
        echo 'Utilisateurs bannis par '.$this->user_name. ' : ' ;
        foreach($this->ban as $valeur){
            echo $valeur .', ' ;
        }
    }
}
?>

```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

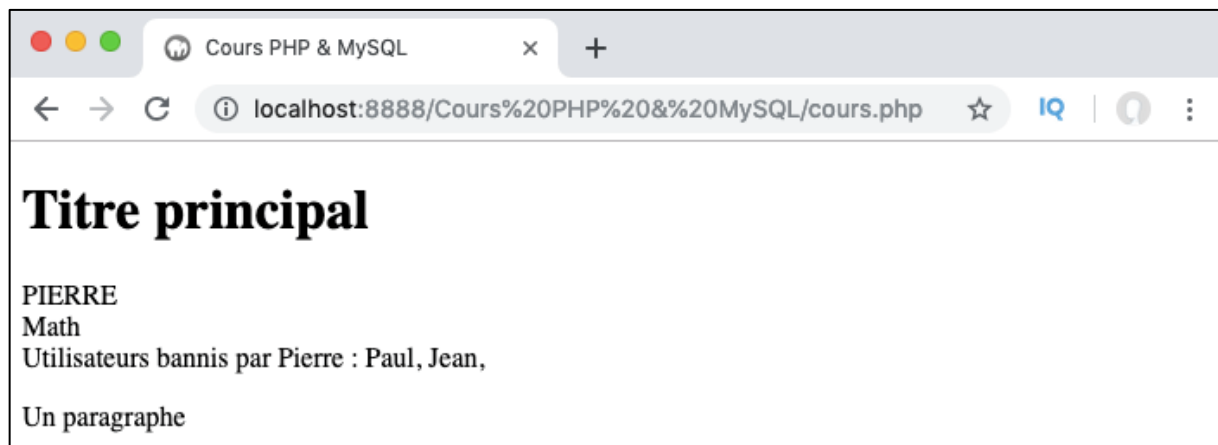
  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      echo $pierre->getNom(). '<br>';
      echo $mathilde->getNom(). '<br>';

      $pierre->setBan('Paul');
      $pierre->setBan('Jean');
      echo $pierre->getBan();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on modifie le code de notre méthode `getNom()` en transformant le résultat renvoyé en majuscules grâce à la fonction `strtoupper()` (« string to upper » ou « chaîne en majuscules »).

Lorsqu'on appelle notre méthode depuis notre objet `$pierre` qui est un objet de la classe `Admin`, on voit bien que c'est la nouvelle définition de la méthode qui est exécutée.

On va de la même façon pouvoir surcharger le constructeur de la classe parent en définissant un nouveau constructeur dans la classe étendue. Dans le cas présent, on

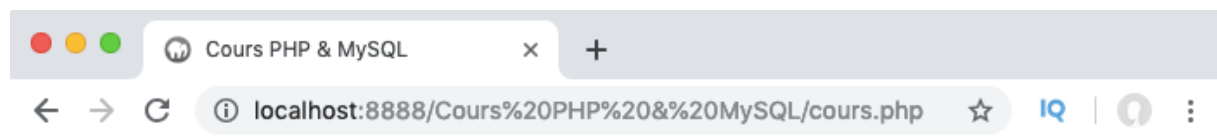
pourrait définir le nom directement en majuscules pour les objets de la classe `Admin` afin qu'il s'affiche toujours en majuscules.

```
<?php
class Admin extends Utilisateur{
    protected $ban;

    public function __construct($n, $p){
        $this->user_name = strtoupper($n);
        $this->user_pass = $p;
    }

    public function setBan($b){
        $this->ban[] .= $b;
    }

    public function getBan(){
        echo 'Utilisateurs bannis par '.$this->user_name.' : ';
        foreach($this->ban as $valeur){
            echo $valeur .', ';
        }
    }
}
?>
```



## Titre principal

PIERRE

Math

Utilisateurs bannis par PIERRE : Paul, Jean,

Un paragraphe

Notez que le PHP a une définition particulière de la surcharge par rapport à de nombreux autres langages. Pour de nombreux langages, « surcharger » une méthode signifie écrire différentes versions d'une même méthode avec un nombre différents de paramètres.

## Accéder à une méthode ou une propriété surchargée grâce à l'opérateur de résolution de portée

Parfois, il va pouvoir être intéressant d'accéder à la définition de base d'une propriété ou d'une méthode surchargée. Pour faire cela, on va pouvoir utiliser l'opérateur de résolution de portée qui est symbolisé par le signe `::` (double deux points).

Nous allons également devoir utiliser cet opérateur pour accéder aux constantes et aux méthodes et propriétés définies comme statiques dans une classe (nous allons étudier tous ces concepts dans les prochains chapitres).

Pour le moment, concentrons-nous sur l'opérateur de résolution de portée et illustrons son fonctionnement dans le cas d'une méthode ou d'une propriété surchargée.

Nous allons pouvoir utiliser trois mots clefs pour accéder à différents éléments d'une classe avec l'opérateur de résolution de portée : les mots clefs `parent`, `self` et `static`.

Dans le cas où on souhaite accéder à une propriété ou à une méthode surchargée, le seul mot clef qui va nous intéresser est le mot clef `parent` qui va nous servir à indiquer qu'on souhaite accéder à la définition de la propriété ou de la méthode faite dans la classe mère.

Pour illustrer cela, nous allons modifier la méthode `getNom()` de notre classe mère `Utilisateur` afin qu'elle `echo` la nom de l'objet l'appelant plutôt qu'utiliser une instruction `return` (qui empêcherait l'exécution de tout code après l'instruction).

```
<?php
class Utilisateur{
    protected $user_name;
    protected $user_pass;

    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        echo $this->user_name;
    }
}
?>
```

Ensuite, on va surcharger notre méthode `getNom()` dans notre classe étendue `Admin`. La méthode de notre classe fille va reprendre le code de celle de la classe parent et ajouter de nouvelles instructions. Ici, plutôt que de réécrire le code de la méthode de base, on peut utiliser `parent::getNom()` pour appeler la méthode parent depuis notre méthode dérivée. Ensuite, on choisit d'`echo` un texte.

```

<?php
class Admin extends Utilisateur{
    protected $ban;

    public function __construct($n, $p){
        $this->user_name = strtoupper($n);
        $this->user_pass = $p;
    }

    public function getNom(){
        parent::getNom();
        echo ' (depuis la classe étendue)<br>';
    }

    public function setBan($b){
        $this->ban[] .= $b;
    }
    public function getBan(){
        echo 'Utilisateurs bannis par '.$this->user_name.' : ';
        foreach($this->ban as $valeur){
            echo $valeur .', ';
        }
    }
}
?>

```

Lorsqu'un objet de `Admin` appelle `getNom()`, la méthode `getNom()` de `Admin` va être utilisée. Cette méthode appelle elle-même la méthode de la classe parent qu'elle surcharge et ajoute un texte au résultat de la méthode surchargée.

Le mot clef `parent` fait ici référence à la classe parent de la classe dans laquelle la méthode appelante est définie. Dans notre cas, la méthode appelante se trouve dans `Admin`. Le code `parent::getNom()` va donc chercher une méthode nommée `getNom()` dans la classe parent de `Admin`, c'est-à-dire `Utilisateur`, et l'exécuter.



```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef');
      $mathilde = new Utilisateur('Math', 123456);

      $pierre->getNom();
      $mathilde->getNom();

      echo '<br>';

      $pierre->setBan('Paul');
      $pierre->setBan('Jean');
      echo $pierre->getBan();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



# Constantes de classe

Nous n'en avons pas véritablement parlé jusqu'à présent mais nous allons également pouvoir définir des constantes dans une classe. Nous allons dans cette leçon apprendre à le faire et apprendre à accéder à la valeur d'une constante avec l'opérateur de résolution de portée.

## Rappel sur les constantes et définition de constantes dans une classe

---

Une constante est un conteneur qui ne va pouvoir stocker qu'une seule et unique valeur durant la durée de l'exécution d'un script. On ne va donc pas pouvoir modifier la valeur stockée dans une constante.

Pour définir une constante de classe, on va utiliser le mot clef `const` suivi du nom de la constante en majuscules. On ne va pas utiliser ici de signe `$` comme avec les variables. Le nom d'une constante va respecter les règles communes de nommage PHP.

Depuis la version 7.1 de PHP, on peut définir une visibilité pour nos constantes (`public`, `protected` ou `private`).

Par défaut (si rien n'est précisé), une constante sera considérée comme publique et on pourra donc y accéder depuis l'intérieur et depuis l'extérieur de la classe dans laquelle elle a été définie.

Par ailleurs, notez également que les constantes sont allouées une fois par classe, et non pour chaque instance de classe. Cela signifie qu'une constante appartient intrinsèquement à la classe et non pas à un objet en particulier et que tous les objets d'une classe vont donc partager cette même constante de classe.

Ajoutons immédiatement une constante à notre classe mère `Utilisateur` :

```

<?php
class Utilisateur{
    protected $user_name;
    protected $user_pass;
    public const ABONNEMENT = 15;

    public function __construct($n, $p){
        $this->user_name = $n;
        $this->user_pass = $p;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        echo $this->user_name;
    }
}
?>

```

Ici, on crée une constante **ABONNEMENT** qui stocke la valeur « 15 ». On peut imaginer que cette constante représente le prix d'un abonnement mensuel pour accéder au contenu de notre site.

Faites bien attention : si vous définissez un niveau de visibilité pour une constante, assurez-vous de travailler avec une version de PHP 7.1 ou ultérieure. Dans le cas où vous travaillez avec une version antérieure, la déclaration de la visibilité d'une constante sera illégale et le code ne fonctionnera pas.

## Accéder à une constante avec l'opérateur de résolution de portée

---

Pour accéder à une constante, nous allons à nouveau devoir utiliser l'opérateur de résolution de portée. La façon d'accéder à une constante va légèrement varier selon qu'on essaie d'y accéder depuis l'intérieur de la classe qui la définit (ou d'une classe étendue) ou depuis l'extérieur de la classe.

Dans le cas où on tente d'accéder à la valeur d'une constante depuis l'intérieur d'une classe, il faudra utiliser l'un des deux mots clefs **self** ou **parent** qui vont permettre d'indiquer qu'on souhaite accéder à une constante définie dans la classe à partir de laquelle on souhaite y accéder (**self**) à qu'on souhaite accéder à une constante définie dans une classe mère (**parent**).

Essayons déjà de manipuler notre constante depuis la définition de la classe :

```

<?php
class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;

    /*Attention: si vous utilisez une version PHP < PHP 7.1, ce code ne
    *fonctionnera pas*/
    public const ABONNEMENT = 15;

    public function __construct($n, $p, $r){
        $this->user_name = $n;
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function __destruct(){
        //Du code à exécuter
    }

    public function getNom(){
        echo $this->user_name;
    }

    public function setPrixAbo(){
        /*On peut imaginer qu'on calcule un prix d'abonnement différent
        *selon les profils des utilisateurs*/
        if($this->user_region === 'Sud'){
            return $this->prix_abo = self::ABONNEMENT / 2;
        }else{
            return $this->prix_abo = self::ABONNEMENT;
        }
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }
}
?>

```

L'idée ici va être de définir un tarif d'abonnement différent en fonction des différents profils des utilisateurs et en se basant sur notre constante de classe **ABONNEMENT**.

On va vouloir définir un tarif préférentiel pour les utilisateurs qui viennent du Sud (car c'est mon site et je fais ce que je veux !). On va ici rajouter deux propriétés dans notre classe : **\$user\_region** qui va contenir la région de l'utilisateur et **\$prix\_abo** qui va contenir le prix de l'abonnement après calcul.

On va commencer par modifier notre constructeur pour qu'on puisse initialiser la valeur de **\$user\_region** dès la création d'un objet.

Ensuite, on crée une méthode `setPrixAbo()` qui va définir le prix de l'abonnement en fonction de la région passée. Dans le cas où l'utilisateur indique venir du « Sud », le prix de l'abonnement sera égal à la moitié de la valeur de la constante `ABONNEMENT`.

Vous pouvez remarquer qu'on utilise `self::ABONNEMENT` pour accéder au contenu de notre constante ici. En effet, la constante a été définie dans la même classe que la méthode qui l'utilise.

On crée enfin une méthode `getPrixAbo()` qui renvoie la valeur contenue dans la propriété `$prix_abo` pour un objet.

On va maintenant se rendre dans notre classe étendue `Admin` :

```
<?php
class Admin extends Utilisateur{
    protected $ban;
    public const ABONNEMENT = 5;

    public function __construct($n, $p, $r){
        $this->user_name = strtoupper($n);
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function setBan($b){
        $this->ban[] .= $b;
    }
    public function getBan(){
        echo 'Utilisateurs bannis par ' . $this->user_name . ' : ';
        foreach($this->ban as $valeur){
            echo $valeur . ', ';
        }
    }

    public function setPrixAbo(){
        if($this->user_region === 'Sud'){
            return $this->prix_abo = self::ABONNEMENT;
        }else{
            return $this->prix_abo = parent::ABONNEMENT / 2;
        }
    }
}
?>
```

On surcharge ici la constante `ABONNEMENT` de la classe parente en lui attribuant une nouvelle valeur. On a tout à fait le droit puisqu'il s'agit ici de surcharge et non pas de changement dynamique de valeur d'une même constante (ce qui est interdit).

Pour les objets de la classe `Admin`, on définit le prix de l'abonnement de base à 5. Dans notre classe, on modifie le constructeur pour y intégrer un paramètre « région » et on supprime également la méthode `getNom()` créée précédemment.

Enfin, on surcharge la méthode `setPrixAbo()` : si l'administrateur indique venir du Sud, le prix de son abonnement va être égal à la valeur de la constante `ABONNEMENT` définie dans la classe courante (c'est-à-dire dans la classe `Admin`).

Dans les autres cas, le prix de l'abonnement va être égal à la valeur de la constante `ABONNEMENT` définie dans la classe parent (c'est-à-dire la classe `Utilisateur`) divisée par 2.

Ici, `self` et `parent` nous servent à indiquer à quelle définition de la constante `ABONNEMENT` on souhaite se référer.

Il ne nous reste plus qu'à exécuter nos méthodes et à afficher les différents prix de l'abonnement pour différents types d'utilisateurs. On va également vouloir renvoyer le contenu de la constante `ABONNEMENT` telle que définie dans notre classe `Utilisateur` et dans la classe étendue `Admin`.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Utilisateur('Flo', 'flotri', 'Est');

      $pierre->setPrixAbo();
      $mathilde->setPrixAbo();
      $florian->setPrixAbo();

      $u = 'Utilisateur';
      echo 'Valeur de ABONNEMENT dans Utilisateur : ' . $u::ABONNEMENT . '<br>';
      echo 'Valeur de ABONNEMENT dans Admin : ' . Admin::ABONNEMENT . '<br>';

      echo 'Prix de l\'abonnement pour ';
      $pierre->getNom();
      echo ' : ';
      $pierre->getPrixAbo();
      echo '<br>Prix de l\'abonnement pour ';
      $mathilde->getNom();
      echo ' : ';
      $mathilde->getPrixAbo();
      echo '<br>Prix de l\'abonnement pour ';
      $florian->getNom();
      echo ' : ';
      $florian->getPrixAbo();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

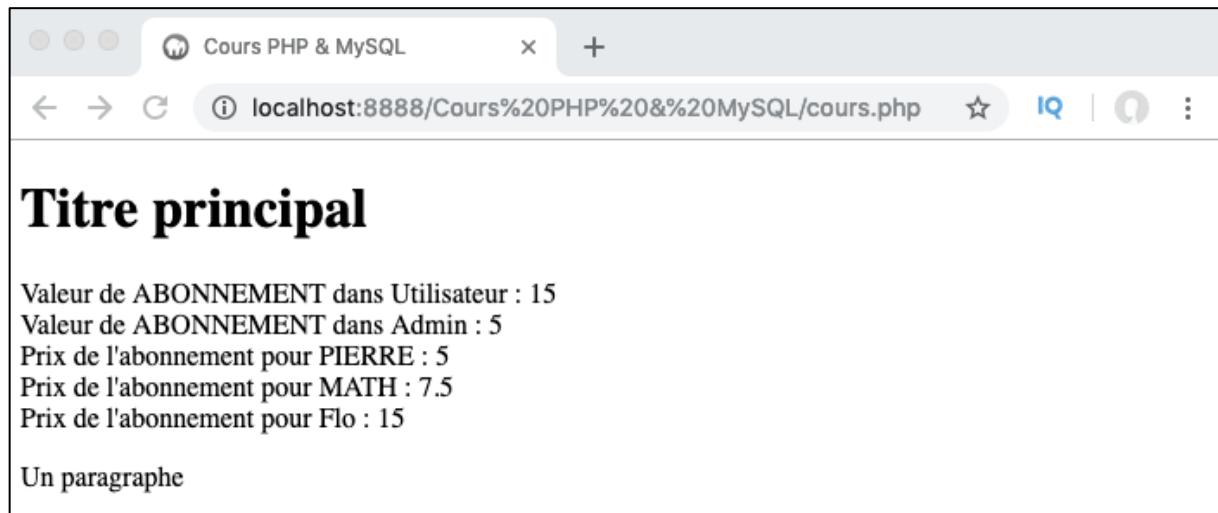
```

Ici, on tente d'accéder à notre constante depuis l'extérieur de la classe. On va utiliser une syntaxe différente qui va réutiliser le nom de la classe plutôt que les mots clefs `self` ou `parent` qui n'auraient aucun sens dans le cas présent.

On est obligés de préciser la classe ici car je vous rappelle qu'une constante n'appartient pas à une instance ou à un objet en particulier mais est définie pour la classe en soi.

Vous pouvez également remarquer que j'ai défini une variable `$u` dans le code ci-dessus. En effet, vous devez savoir qu'il est possible en PHP de référencer une classe en utilisant une variable, c'est-à-dire d'utiliser une variable pour faire référence à une classe. Pour faire cela, il suffit de stocker le nom exact de la classe dans la variable et on pourra ensuite l'utiliser comme référence à notre classe.

Dans mon code, l'écriture `$u::ABONNEMENT` est donc strictement équivalente à `UTILISATEUR::ABONNEMENT`.





# Propriétés et méthodes statiques

Dans cette nouvelle leçon, nous allons découvrir ce que sont les propriétés et méthodes statiques, leur intérêt et comment créer et utiliser des propriétés et méthodes statiques.

## Définition des propriétés et méthodes statiques

---

Une propriété ou une méthode statique est une propriété ou une méthode qui ne va pas appartenir à une instance de classe ou à un objet en particulier mais qui va plutôt appartenir à la classe dans laquelle elle a été définie.

Les méthodes et propriétés statiques vont donc avoir la même définition et la même valeur pour toutes les instances d'une classe et nous allons pouvoir accéder à ces éléments sans avoir besoin d'instancier la classe.

Pour être tout à fait précis, nous n'allons pas pouvoir accéder à une propriété statique depuis un objet. En revanche, cela va être possible dans le cas d'une méthode statique.

Attention à ne pas confondre propriétés statiques et constantes de classe : une propriété statique peut tout à fait changer de valeur au cours du temps à la différence d'une constante dont la valeur est fixée. Simplement, la valeur d'une propriété statique sera partagée par tous les objets issus de la classe dans laquelle elle a été définie.

De manière générale, nous n'utiliserons quasiment jamais de méthode statique car il n'y aura que très rarement d'intérêt à en utiliser. En revanche, les propriétés statiques vont s'avérer utiles dans de nombreux cas.

## Définir et accéder à des propriétés et à des méthodes statiques

---

On va pouvoir définir une propriété ou une méthode statique à l'aide du mot clef `static`.

Prenons immédiatement un premier exemple afin que vous compreniez bien l'intérêt et le fonctionnement des propriétés et méthodes statiques.

Pour cela, retournons dans notre classe étendue `Admin`. Cette classe possède une propriété `$ban` qui contient la liste des utilisateurs bannis un l'objet courant de `Admin` ainsi qu'une méthode `getBan()` qui renvoie le contenu de `$ban`.

Imaginons maintenant que l'on souhaite stocker la liste complète des utilisateurs bannis par tous les objets de `Admin`. Nous allons ici devoir définir une propriété dont la valeur va pouvoir être modifiée et qui va être partagée par tous les objets de notre classe c'est-à-dire une propriété qui ne va pas appartenir à un objet de la classe en particulier mais à la classe en soi.

Pour faire cela, on va commencer par déclarer notre propriété `$ban` comme statique et modifier le code de nos méthodes `getBan()` et `setBan()`.

En effet, vous devez bien comprendre ici qu'on ne peut pas accéder à une propriété statique depuis un objet, et qu'on ne va donc pas pouvoir utiliser l'opérateur objet `->` pour accéder à notre propriété statique.

Pour accéder à une propriété statique, nous allons une fois de plus devoir utiliser l'opérateur de résolution de portée `::`.

Regardez plutôt le code ci-dessous :

```
<?php
class Admin extends Utilisateur{
    protected static $ban;
    public const ABONNEMENT = 5;

    public function __construct($n, $p, $r){
        $this->user_name = strtoupper($n);
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function setBan(...$b){
        foreach($b as $banned){
            self::$ban[] .= $banned;
        }
    }

    public function getBan(){
        echo 'Utilisateurs bannis : ';
        foreach(self::$ban as $valeur){
            echo $valeur .', ';
        }
    }

    public function setPrixAbo(){
        if($this->user_region === 'Sud'){
            return $this->prix_abo = self::ABONNEMENT;
        }else{
            return $this->prix_abo = parent::ABONNEMENT / 2;
        }
    }
}
?>
```

Ici, on commence par déclarer notre propriété `$ban` comme statique avec la syntaxe `protected static $ban`. La propriété va donc appartenir à la classe et sa valeur va être partagée par tous les objets de la classe.

Ensuite, on modifie notre fonction `setBan()` pour utiliser notre propriété statique. Ici, vous pouvez déjà noter que j'ai ajouté `...` devant la liste des paramètres de notre méthode.

Nous avons déjà vu cette écriture lors de la partie sur les fonctions : elle permet à une fonction d'accepter un nombre variable d'arguments. On utilise cette écriture ici pour permettre à nos objets de bannir une ou plusieurs personnes d'un coup.

Dans notre méthode, on remplace `$this->ban` par `self::$ban` puisque notre propriété `$ban` est désormais statique et appartient à la classe et non pas à un objet en particulier. Il faut donc utiliser l'opérateur de résolution de portée pour y accéder.

La boucle `foreach` nous permet simplement d'ajouter les différentes valeurs passées en argument dans notre propriété statique `$ban` qu'on définit comme un tableau.

De même, on modifie le code de la méthode `getBan()` afin d'accéder à notre propriété statique et de pouvoir afficher son contenu en utilisant à nouveau la syntaxe `self::$ban`.

Chaque objet de la classe `Admin` va ainsi pouvoir bannir des utilisateurs en utilisant `setBan()` et chaque nouvel utilisateur banni va être stocké dans la propriété `$ban`. De même, chaque objet va pouvoir afficher la liste complète des personnes bannies en utilisant `getBan()`.

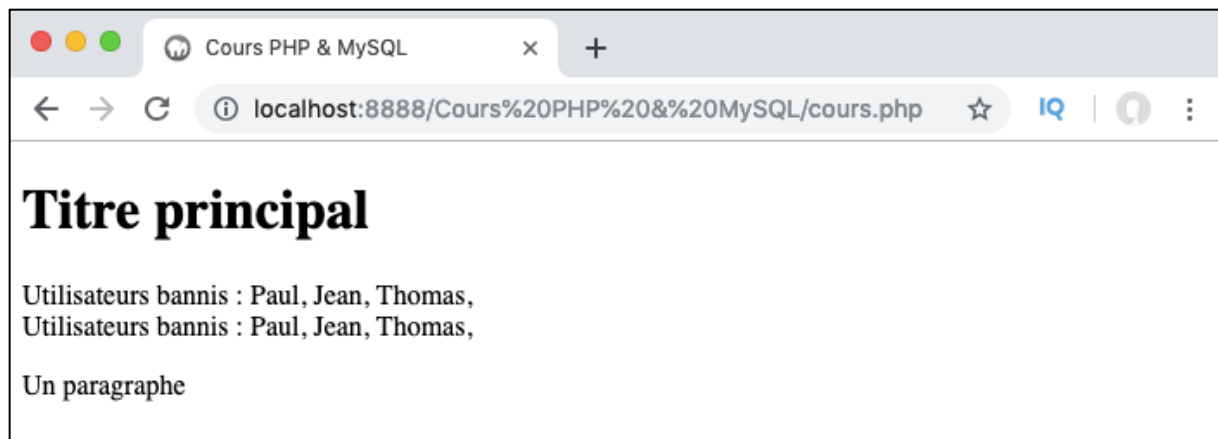
```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Utilisateur('Flo', 'flotri', 'Est');

      $pierre->setBan('Paul', 'Jean');
      $mathilde->setBan('Thomas');

      $pierre->getBan();
      echo '<br>';
      $mathilde->getBan();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```



# Méthodes et classes abstraites

En PHP orienté objet, nous allons pouvoir définir des classes et des méthodes dites « abstraites ». Nous allons présenter dans cette leçon les intérêts des classes et méthodes abstraites et voir comment déclarer des classes et des méthodes comme abstraites en pratique.

## Les classes et méthodes abstraites : définition et intérêt

---

Une classe abstraite est une classe qui ne va pas pouvoir être instanciée directement, c'est-à-dire qu'on ne va pas pouvoir manipuler directement.

Une méthode abstraite est une méthode dont seule la signature (c'est-à-dire le nom et les paramètres) va pouvoir être déclarée mais pour laquelle on ne va pas pouvoir déclarer d'implémentation (c'est-à-dire le code dans la fonction ou ce que fait la fonction).

Dès qu'une classe possède une méthode abstraite, il va falloir la déclarer comme classes abstraite.

Pour comprendre les intérêts des classes et des méthodes abstraites, il faut bien penser que lorsqu'on code en PHP, on code généralement pour quelqu'un ou alors on fait partie d'une équipe. D'autres développeurs vont ainsi généralement pouvoir / devoir travailler à partir de notre code, le modifier, ajouter des fonctionnalités, etc.

L'intérêt principal de définir une classe comme abstraite va être justement de fournir un cadre plus strict lorsqu'ils vont utiliser notre code en les forçant à définir certaines méthodes et etc.

En effet, une classe abstraite ne peut pas être instanciée directement et contient généralement des méthodes abstraites. L'idée ici va donc être de définir des classes mères abstraites et de pousser les développeurs à étendre ces classes.

Lors de l'héritage d'une classe abstraite, les méthodes déclarées comme abstraites dans la classe parent doivent obligatoirement être définies dans la classe enfant avec des signatures (nom et paramètres) correspondantes.

Cette façon de faire va être très utile pour fournir un rail, c'est-à-dire une ligne directrice dans le cas de développements futurs.

En effet, en créant un plan « protégé » (puisque une classe abstraite ne peut pas être instanciée directement) on force les développeurs à étendre cette classe et on les force également à définir les méthodes abstraites.

Cela nous permet de nous assurer que certains éléments figurent bien dans la classe étendue et permet d'éviter certains problèmes de compatibilité en nous assurant que les classes étendues possèdent une structure de base commune.

# Définir des classes et des méthodes abstraites en pratique

---

Pour définir une classe ou une méthode comme abstraite, nous allons utiliser le mot clef `abstract`.

Vous pouvez déjà noter ici qu'une classe abstraite n'est pas structurellement différente d'une classe classique (à la différence de la présence potentielle de méthodes abstraites) et qu'on va donc tout à fait pouvoir ajouter des constantes, des propriétés et des méthodes classiques dans une classe abstraite.

Les seules différences entre les classes abstraites et classiques sont encore une fois qu'une classe abstraite peut contenir des méthodes abstraites et doit obligatoirement être étendue pour utiliser ses fonctionnalités.

Reprenons nos classes `Utilisateur` et `Admin` créées précédemment pour illustrer de manière pratique l'intérêt des classes et méthodes abstraites.

Précédemment, nous avons créé une méthode `setPrixAbo()` qui calculait le prix de l'abonnement pour un utilisateur classique dans notre classe `Utilisateur` et on avait surchargé le code de cette fonction dans `Admin` pour calculer un prix d'abonnement différent pour les admin.

Ici, cela rend notre code conceptuellement étrange car cela signifie que `Utilisateur` définit des choses pour un type d'utilisateur qui sont les utilisateurs « de base » tandis que `Admin` les définit pour un autre type d'utilisateur qui sont les « admin ». Le souci que j'ai avec ce code est que chacune de nos deux classes s'adresse à un type différent d'utilisateur mais que nos deux classes ne sont pas au même niveau puisque `Admin` est un enfant de `Utilisateur`.

Normalement, si notre code est bien construit, on devrait voir une hiérarchie claire entre ce que représentent nos classes mères et nos classes enfants. Dans le cas présent, j'aimerais que ma classe mère définisse des choses pour TOUS les types d'utilisateurs et que les classes étendues s'occupent chacune de définir des spécificités pour UN type d'utilisateur en particulier.

Encore une fois, ici, on touche à des notions qui sont plus de design de conception que des notions de code en soi mais lorsqu'on code la façon dont on crée et organise le code est au moins aussi importante que le code en soi. Il faut donc toujours essayer d'avoir la structure globale la plus claire et la plus pertinente possible.

Ici, nous allons donc partir du principe que nous avons deux grands types d'utilisateurs : les utilisateurs classiques et les administrateurs. On va donc transformer notre classe `Utilisateur` afin qu'elle ne définisse que les choses communes à tous les utilisateurs et allons définir les spécificités de chaque type utilisateur dans des classes étendues `Admin` et `Abonne`.

```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    public const ABONNEMENT = 15;

    public function __destruct(){
        //Du code à exécuter
    }

    abstract public function setPrixAbo();

    public function getNom(){
        echo $this->user_name;
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }
}
?>

```

On commence déjà par modifier notre classe parent `Utilisateur` en la définissant comme abstraite avec le mot clef `abstract`. On supprime le constructeur qui va être défini dans les classes étendues et on déclare également la méthode `setPrixAbo()` comme abstraite.

Ici, définir la méthode `setPrixAbo()` comme abstraite fait beaucoup de sens puisque chaque type d'utilisateur va avoir un prix d'abonnement calculé différent (c'est-à-dire une implémentation de la méthode différente) et car on souhaite que le prix de l'abonnement soit calculé.

En définissant `setPrixAbo()` comme abstraite, on force ainsi les classes étendues à l'implémenter.

Les propriétés vont être partagées par tous les types d'utilisateurs et les méthodes comme `getNom()` vont avoir une implémentation identique pour chaque utilisateur. Cela fait donc du sens de les définir dans la classe abstraite.

## Étendre des classes abstraites et implémenter des méthodes abstraites

Maintenant qu'on a défini notre classe `Utilisateur` comme abstraite, il va falloir l'étendre et également implémenter les méthodes abstraites.

On va commencer par aller dans notre classe étendue `Admin` et supprimer la constante `ABONNEMENT` puisque nous allons désormais utiliser celle de la classe abstraite. On va donc également modifier le code de notre méthode `setPrixAbo()`.

```

<?php
class Admin extends Utilisateur{
    protected static $ban;

    public function __construct($n, $p, $r){
        $this->user_name = strtoupper($n);
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function setBan(...$b){
        foreach($b as $banned){
            self::$ban[] .= $banned;
        }
    }

    public function getBan(){
        echo 'Utilisateurs bannis : ';
        foreach(self::$ban as $valeur){
            echo $valeur .', ';
        }
    }

    public function setPrixAbo(){
        if($this->user_region === 'Sud'){
            return $this->prix_abo = parent::ABONNEMENT / 6;
        }else{
            return $this->prix_abo = parent::ABONNEMENT / 3;
        }
    }
}
?>

```

Ici, lors de l'implémentation d'une méthode déclarée comme abstraite, on ne doit pas réécrire **abstract** puisque justement on implémente la méthode abstraite. Dans le code, on change **self::** par **parent::** ainsi que le calcul.

On va ensuite créer un nouveau fichier de classe qu'on va appeler **abonne.class.php**.

Dans ce fichier, nous allons définir un constructeur pour nos abonnés qui représentent nos utilisateurs de base et allons à nouveau implémenter la méthode **setPrixAbo()**.



```

<?php
class Abonne extends Utilisateur{
    public function __construct($n, $p, $r){
        $this->user_name = $n;
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function setPrixAbo(){
        if($this->user_region === 'Sud'){
            return $this->prix_abo = parent::ABONNEMENT / 2;
        }else{
            return $this->prix_abo = parent::ABONNEMENT;
        }
    }
}
?>

```

On peut maintenant retourner sur notre script principal et créer des objets à partir de nos classes étendues et voir comment ils se comportent :

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

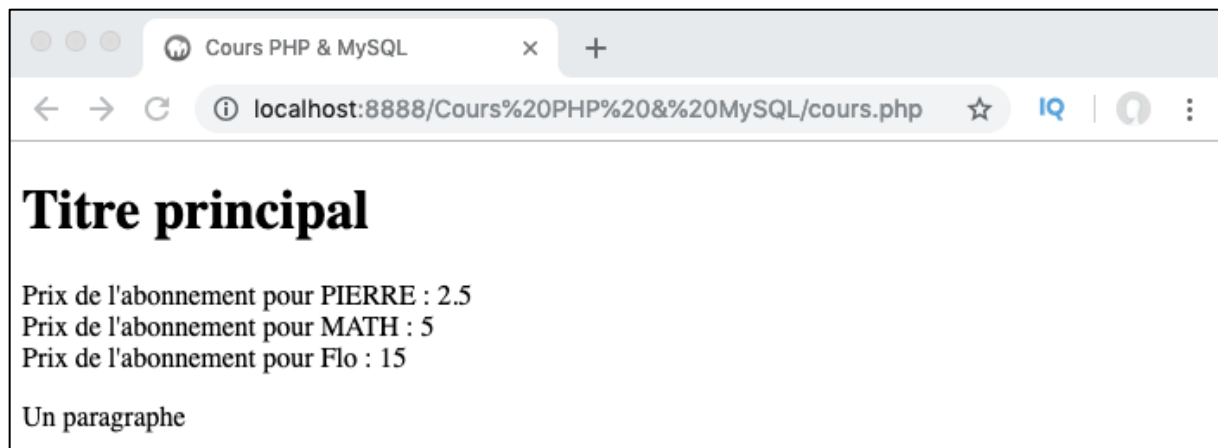
  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';
      require 'classes/abonne.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');

      $pierre->setPrixAbo();
      $mathilde->setPrixAbo();
      $florian->setPrixAbo();

      echo 'Prix de l\'abonnement pour ';
      $pierre->getNom();
      echo ' : ';
      $pierre->getPrixAbo();
      echo '<br>Prix de l\'abonnement pour ';
      $mathilde->getNom();
      echo ' : ';
      $mathilde->getPrixAbo();
      echo '<br>Prix de l\'abonnement pour ';
      $florian->getNom();
      echo ' : ';
      $florian->getPrixAbo();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



# Interfaces

Dans cette nouvelle leçon, nous allons découvrir le concept d'interfaces en PHP orienté objet. Nous allons ici particulièrement insister la compréhension de ce qu'est une interface, la différence entre une interface et une classe abstraite ainsi que sur les cas où il va être intéressant de définir des interfaces.

## Définition des interfaces en PHP objet et différence avec les classes abstraites

---

Les interfaces vont avoir un but similaire aux classes abstraites puisque l'un des intérêts principaux liés à la définition d'une interface va être de fournir un plan général pour les développeurs qui vont implémenter l'interface et de les forcer à suivre le plan donné par l'interface.

De la même manière que pour les classes abstraites, nous n'allons pas directement pouvoir instancier une interface mais devoir l'implémenter, c'est-à-dire créer des classes dérivées à partir de celle-ci pour pouvoir utiliser ses éléments.

Les deux différences majeures entre les interfaces et les classes abstraites sont les suivantes :

1. Une interface ne peut contenir que les signatures des méthodes ainsi qu'éventuellement des constantes mais pas de propriétés. Cela est dû au fait qu'aucune implémentation n'est faite dans une interface : une interface n'est véritablement qu'un plan ;
2. Une classe ne peut pas étendre plusieurs autres classes à cause des problèmes d'héritage. En revanche, une classe peut tout à fait implémenter plusieurs interfaces.

Je pense qu'il est ici intéressant de bien illustrer ces deux points et notamment d'expliquer pourquoi une classe n'a pas l'autorisation d'étendre plusieurs autres classes.

Pour cela, imaginons qu'on ait une première classe **A** qui définit la signature d'une méthode `diamond()` sans l'implémenter.

Nous créons ensuite deux classes **B** et **C** qui étendent la classe **A** et qui implémentent chacune d'une manière différente la méthode `diamond()`.

Finalement, on crée une classe **D** qui étend les classes **B** et **C** et qui ne redéfinit pas la méthode `diamond()`. Dans ce cas-là, on est face au problème suivant : la classe **D** doit-elle utiliser l'implémentation de `diamond()` faite par la classe **B** ou celle faite par la classe **C** ?

Ce problème est connu sous le nom du « problème du diamant » et est la raison principale pour laquelle la plupart des langages de programmation orientés objets (dont le PHP) ne permettent pas à une classe d'étendre deux autres classes.

En revanche, il ne va y avoir aucun problème par rapport à l'implémentation par une classe de plusieurs interfaces puisque les interfaces, par définition, ne peuvent que définir la signature d'une méthode et non pas son implémentation.

Profitez-en ici pour noter que les méthodes déclarées dans une classe doivent obligatoirement être publiques (puisque'elles devront être implémentées en dehors de l'interface) et que les constantes d'interface ne pourront pas être écrasées par une classe (ou par une autre interface) qui vont en hériter.

## Définir et implémenter une interface en pratique

---

On va pouvoir définir une interface de la même manière qu'une classe mais en utilisant cette fois-ci le mot clef **interface** à la place de **class**. Nous nommerons généralement nos fichiers d'interface en utilisant « interface » à la place de « classe ». Par exemple, si on crée une interface nommée **Utilisateur**, on enregistrera le fichier d'interface sous le nom **utilisateur.interface.php** par convention.

```
<?php
interface Utilisateur{
    public const ABONNEMENT = 15;
    public function getNom();
    public function setPrixAbo();
    public function getPrixAbo();
}
?>
```

On va ensuite pouvoir réutiliser les définitions de notre interface dans des classes. Pour cela, on va implémenter notre interface. On va pouvoir faire cela de la même manière que lors de la création de classes étendues mais on va cette fois-ci utiliser le mot clef **implements** à la place de **extends**.

```

<?php
class Abonne implements Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    private $user_pass;

    public function __construct($n, $p, $r){
        $this->user_name = $n;
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function getNom(){
        echo $this->user_name;
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }

    public function setPrixAbo(){
        if($this->user_region === 'Sud'){
            return $this->prix_abo = Utilisateur::ABONNEMENT / 2;
        }else{
            return $this->prix_abo = Utilisateur::ABONNEMENT;
        }
    }
}
?>

```

```

<?php
class Admin implements Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    private $user_pass;
    protected static $ban;

    public function __construct($n, $p, $r){
        $this->user_name = strtoupper($n);
        $this->user_pass = $p;
        $this->user_region = $r;
    }

    public function getNom(){
        echo $this->user_name;
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }

    public function setBan(...$b){
        foreach($b as $banned){
            self::$ban[] .= $banned;
        }
    }

    public function getBan(){
        echo 'Utilisateurs bannis : ';
        foreach(self::$ban as $valeur){
            echo $valeur .', ';
        }
    }

    public function setPrixAbo(){
        if($this->user_region === 'Sud'){
            return $this->prix_abo = Utilisateur::ABONNEMENT / 6;
        }else{
            return $this->prix_abo = Utilisateur::ABONNEMENT / 3;
        }
    }
}
?>

```

Ici, on utilise une interface `Utilisateur` plutôt que notre classe abstraite `Utilisateur` qu'on laisse de côté pour le moment.

Notez bien ici que toutes les méthodes déclarées dans une interface doivent obligatoirement être implémentées dans une classe qui implémente une interface.

Vous pouvez également observer que pour accéder à une constante d'interface, il va falloir préciser le nom de l'interface devant l'opérateur de résolution de portée.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.interface.php';
      require 'classes/admin.class.php';
      require 'classes/abonne.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');

      $pierre->setPrixAbo();
      $mathilde->setPrixAbo();
      $florian->setPrixAbo();

      echo 'Prix de l\'abonnement pour ';
      $pierre->getNom();
      echo ' : ';
      $pierre->getPrixAbo();
      echo '<br>Prix de l\'abonnement pour ';
      $mathilde->getNom();
      echo ' : ';
      $mathilde->getPrixAbo();
      echo '<br>Prix de l\'abonnement pour ';
      $florian->getNom();
      echo ' : ';
      $florian->getPrixAbo();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```





Par ailleurs, notez également qu'on va aussi pouvoir étendre une interface en utilisant le mot clef **extends**. Dans ce cas-là, on va créer des interfaces étendues qui devront être implémentées par des classes de la même manière que pour une interface classique.

## Interface ou classe abstraite : comment choisir ?

Les interfaces et les classes abstraites semblent à priori servir un objectif similaire qui est de créer des plans ou le design de futures classes.

Cependant, avec un peu de pratique, on va vite s'apercevoir qu'il est parfois plus intéressant d'utiliser une classe abstraite ou au contraire une interface en fonction de la situation.

Les interfaces ne permettent aucune implémentation mais permet simplement de définir des signatures de méthodes et des constantes tandis qu'on va tout à fait pouvoir définir et implémenter des méthodes dans une classe abstraite.

Ainsi, lorsque plusieurs classes possèdent des similarités, on aura plutôt tendance à créer une classe mère abstraite dans laquelle on va pouvoir implémenter les méthodes communes puis d'étendre cette classe.

En revanche, si nous avons des classes qui vont pouvoir posséder les mêmes méthodes mais qui vont les implémenter de manière différente, alors il sera certainement plus adapté de créer une interface pour définir les signatures des méthodes communes et d'implémenter cette interface pour laisser le soin à chaque classe fille d'implémenter les méthodes comme elles le souhaitent.

Dans notre cas, par exemple, nos types d'utilisateurs partagent de nombreuses choses. Il est donc ici plus intéressant de créer une classe abstraite plutôt qu'une interface.

Encore une fois, on touche ici plus à des problèmes de structure et de logique du code qu'à des problèmes de code en soi et différents développeurs pourront parvenir au même résultat de différentes façons.

J'essaie ici simplement de vous présenter comment il est recommandé de travailler idéalement pour avoir la structure de code la plus propre et la plus facilement maintenable.

Une nouvelle fois, je ne saurais trop vous répéter l'importance de la réflexion face à un projet complexe avant l'étape de codage : il est selon moi essentiel de créer un premier plan sur papier décrivant ce à quoi on souhaite arriver et comment on va structurer notre code pour y arriver.

## Les interfaces prédéfinies

---

De la même manière qu'il existe des classes prédéfinies, c'est-à-dire des classes natives ou encore prêtes à l'emploi en PHP, nous allons pouvoir utiliser des interfaces prédéfinies.

Parmi celles-ci, nous pouvons notamment noter :

- L'interface **Traversable** dont le but est d'être l'interface de base de toutes les classes permettant de parcourir des objets ;
- L'interface **Iterator** qui définit des signatures de méthodes pour les itérateurs ;
- L'interface **IteratorAggregate** qui est une interface qu'on va pouvoir utiliser pour créer un itérateur externe ;
- L'interface **Throwable** qui est l'interface de base pour la gestion des erreurs et des exceptions ;
- L'interface **ArrayAccess** qui permet d'accéder aux objets de la même façon que pour les tableaux ;
- L'interface **Serializable** qui permet de personnaliser la linéarisation d'objets.

Nous n'allons pas étudier ces interfaces en détail ici. Cependant, nous allons en utiliser certaines dans les parties à venir et notamment nous servir de **Throwable** et des classes dérivées prédéfinies **Error** et **Exception** dans la partie liée à la gestion des erreurs et des exceptions.

# Méthodes magiques

Dans cette nouvelle leçon, nous allons discuter de méthodes spéciales en PHP objet : les méthodes magiques qui sont des méthodes qui vont être appelées automatiquement suite à un événement déclencheur propre à chacune d'entre elles.

Nous allons établir la liste complète de ces méthodes magiques en PHP7 et illustrer le rôle de chacune d'entre elles.

## Définition et liste des méthodes magiques PHP

---

Les méthodes magiques sont des méthodes qui vont être appelées automatiquement dans le cas d'un événement particulier.

La méthode `__construct()`, par exemple, est une méthode magique. En effet, cette méthode s'exécute automatiquement dès que l'on instancie une classe dans laquelle on a défini un constructeur.

Les méthodes magiques reconnaissables en PHP au fait que leur nom commence par un double underscore `__`. En voici la liste :

- `__construct()` ;
- `__destruct()` ;
- `__call()` ;
- `__callStatic()` ;
- `__get()` ;
- `__set()` ;
- `__isset()` ;
- `__unset()` ;
- `__toString()` ;
- `__clone()` ;
- `__sleep()` ;
- `__wakeup()` ;
- `__invoke()` ;
- `__set_state()` ;
- `__debugInfo()`.

Nous allons dans cette leçon comprendre le rôle de chacune de ces méthodes et voir comment les utiliser intelligemment.

## Les méthodes magiques `__construct()` et `__destruct()`

---

Nous connaissons déjà les deux méthodes magiques `__construct()` et `__destruct()` encore appelées méthodes « constructeur » et « destructeur ».

La méthode `__construct()` va être appelée dès qu'on va instancier une classe possédant un constructeur.

Cette méthode est très utile pour initialiser les valeurs dont un objet a besoin dès sa création et avant toute utilisation de celui-ci.

La méthode magique `__destruct()` est la méthode « contraire » de la fonction constructeur et va être appelée dès qu'il n'y a plus de référence sur un objet donné ou dès qu'un objet n'existe plus dans le contexte d'une certaine application. Bien évidemment, pour que le destructeur soit appelé, vous devrez le définir dans la définition de la classe tout comme on a l'habitude de le faire avec le constructeur.

Le destructeur va nous servir à « nettoyer » le code en détruisant un objet une fois qu'on a fini de l'utiliser. Définir un destructeur va être particulièrement utile pour effectuer des opérations juste avant que l'objet soit détruit.

En effet, en utilisant un destructeur nous avons la maîtrise sur le moment exact où l'objet va être détruit. Ainsi, on va pouvoir exécuter au sein de notre destructeur différentes commandes comme sauvegarder des dernières valeurs de l'objet dans une base de données, fermer la connexion à une base de données, etc. juste avant que celui-ci soit détruit.

## Les méthodes magiques `__call()` et `__callStatic()`

---

La méthode magique `__call()` va être appelée dès qu'on essaie d'utiliser une méthode qui n'existe pas (ou qui est inaccessible) à partir d'un objet. On va passer deux arguments à cette méthode : le nom de la méthode qu'on essaie d'exécuter ainsi que les arguments relatifs à celle-ci (si elle en a). Ces arguments vont être convertis en un tableau.

La méthode magique `__callStatic()` s'exécute dès qu'on essaie d'utiliser une méthode qui n'existe pas (ou qui est inaccessible) dans un contexte statique cette fois-ci. Cette méthode accepte les mêmes arguments que `__call()`.

Ces deux méthodes vont donc s'avérer très utiles pour contrôler un script et pour éviter des erreurs fatales qui l'empêcheraient de s'exécuter convenablement ou même pour prévenir des failles de sécurité.

```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    public const ABONNEMENT = 15;

    public function __destruct(){
        //Du code à exécuter
    }

    abstract public function setPrixAbo();

    public function getNom(){
        echo $this->user_name;
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }

    public function __call($methode, $arg){
        echo 'Méthode ' . $methode. ' inaccessible depuis un contexte objet.';
        <br>Arguments passés : ' . implode(', ', $arg). '<br>';
    }

    public static function __callStatic($methode, $arg){
        echo 'Méthode ' . $methode. ' inaccessible depuis un contexte statique.';
        <br>Arguments passés : ' . implode(', ', $arg). '<br>';
    }
}
?>

```

On définit ici deux méthodes magiques dans notre classe `Utilisateur` créée précédemment.

Si une méthode appelée depuis un contexte objet est inaccessible, `__call()` va être appelée automatiquement.

Si une méthode appelée depuis un contexte statique est inaccessible, `__callStatic()` va être appelée automatiquement.

Nos méthodes vont ici simplement renvoyer un texte avec le nom de la méthode inaccessible et les arguments passés.

Encore une fois, il faut bien comprendre que ce qui définit une méthode magique est simplement le fait que ces méthodes vont être appelées automatiquement dans un certain contexte.

Cependant, les méthodes magiques ne sont pas des méthodes prédéfinies et il va donc déjà falloir les définir quelque part pour qu'elles soient appelées et également leur fournir un code à exécuter.

Pour tester le fonctionnement de ces deux méthodes, il suffit d'essayer d'exécuter des méthodes qui n'existent pas dans notre script principal :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';
      require 'classes/abonne.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');

      $pierre->test1('argument1');
      echo '<br>';
      Admin::test2('argument2', 'argument3');
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```



Les méthodes magiques `__get()`, `__set()`, `__isset()` et `__unset()`

---

La méthode magique `__get()` va s'exécuter si on tente d'accéder à une propriété inaccessible (ou qui n'existe pas) dans une classe. Elle va prendre en argument le nom de la propriété à laquelle on souhaite accéder.

La méthode magique `__set()` s'exécute dès qu'on tente de créer ou de mettre à jour une propriété inaccessible (ou qui n'existe pas) dans une classe. Cette méthode va prendre comme arguments le nom et la valeur de la propriété qu'on tente de créer ou de mettre à jour.

```
<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    public const ABONNEMENT = 15;

    public function __destruct(){
        //Du code à exécuter
    }

    abstract public function setPrixAbo();

    public function getNom(){
        echo $this->user_name;
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }

    public function __get($prop){
        echo 'Propriété ' . $prop. ' inaccessible.<br>';
    }
    public function __set($prop, $valeur){
        echo 'Impossible de mettre à jour la valeur de ' . $prop. ' avec "'
            . $valeur. '" (propriété inaccessible)';
    }
}
?>
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';
      require 'classes/abonne.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');

      $pierre->prixAbo; //Inaccessible depuis un objet car protected
      echo '<br>';
      $pierre->prixAbo = 20;
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



De manière similaire, la méthode magique `__isset()` va s'exécuter lorsque les fonctions `isset()` ou `empty()` sont appelées sur des propriétés inaccessibles.

La fonction `isset()` va servir à déterminer si une variable est définie et si elle est différente de `NULL` tandis que la fonction `empty()` permet de déterminer si une variable est vide.

Finalement, la méthode magique `__unset()` va s'exécuter lorsque la fonction `unset()` est appelée sur des propriétés inaccessibles.

La fonction `unset()` sert à détruire une variable.



Notez par ailleurs que les 4 méthodes magiques `__get()`, `__set()`, `__isset()` et `__empty()` ne vont pas pouvoir fonctionner dans un contexte statique mais uniquement dans un contexte objet.

## La méthode magique `__toString()`

---

La méthode magique `__toString()` va être appelée dès que l'on va traiter un objet comme une chaîne de caractères (par exemple lorsqu'on tente d'`echo` un objet).

Attention, cette méthode doit obligatoirement retourner une chaîne ou une erreur sera levée par le PHP.

```
<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    public const ABONNEMENT = 15;

    public function __destruct(){
        //Du code à exécuter
    }

    abstract public function setPrixAbo();

    public function getNom(){
        echo $this->user_name;
    }

    public function getPrixAbo(){
        echo $this->prix_abo;
    }

    public function __toString(){
        return 'Nom d\'utilisateur : ' . $this->user_name. '<br>
        Prix de l\'abonnement : ' . $this->prix_abo. '<br><br>';
    }
}
?>
```

```

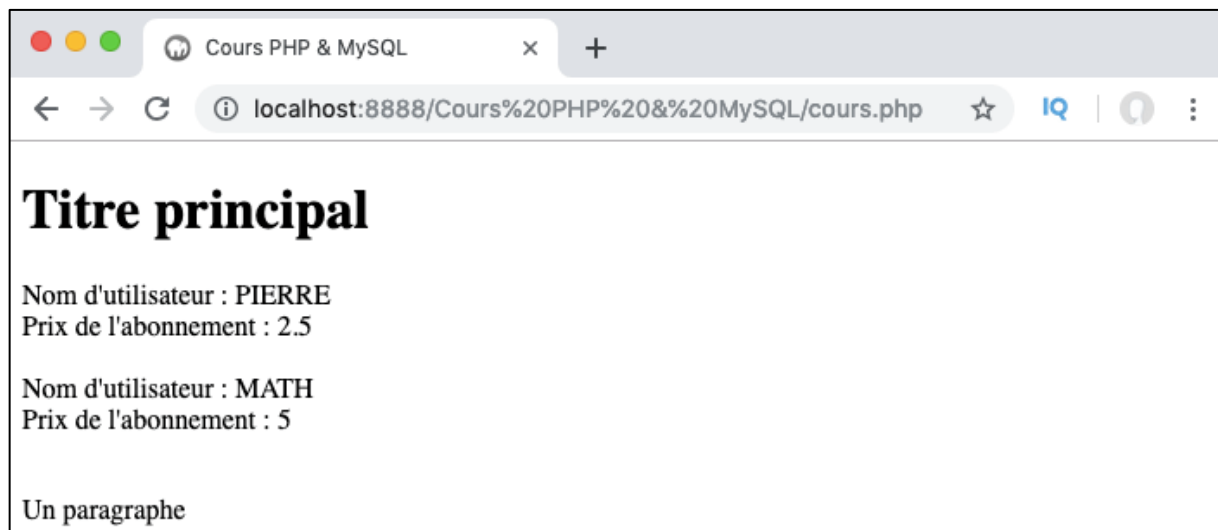
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';
      require 'classes/abonne.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');

      $pierre->setPrixAbo();
      $mathilde->setPrixAbo();
      echo $pierre;
      echo $mathilde;
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



## La méthode magique `__invoke()`

La méthode magique `__invoke()` va être appelée dès qu'on tente d'appeler un objet comme une fonction.

```

<?php
    abstract class Utilisateur{
        protected $user_name;
        protected $user_region;
        protected $prix_abo;
        protected $user_pass;
        public const ABONNEMENT = 15;

        public function __destruct(){
            //Du code à exécuter
        }

        abstract public function setPrixAbo();

        public function getNom(){
            echo $this->user_name;
        }

        public function getPrixAbo(){
            echo $this->prix_abo;
        }

        public function __invoke($arg){
            echo 'Un objet a été utilisé comme une fonction.
            <br>Argument passé : ' . $arg. ' <br><br>';
        }
    }
?>

```

```

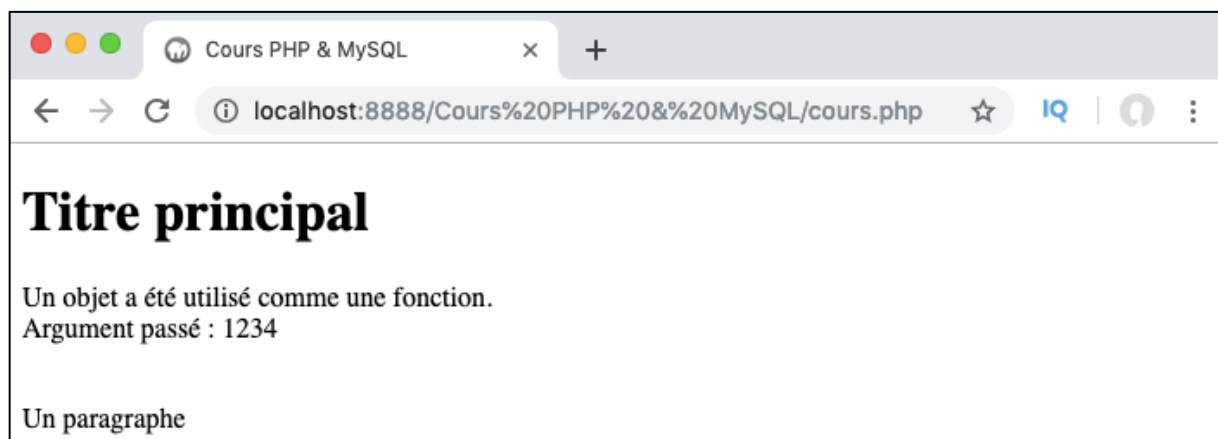
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/utilisateur.class.php';
      require 'classes/admin.class.php';
      require 'classes/abonne.class.php';

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');

      $pierre(1234);
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



## La méthode magique `__clone()`

La méthode magique `__clone()` s'exécute dès que l'on crée un clone d'un objet pour le nouvel objet créé.

Cette méthode va nous permettre notamment de modifier les propriétés qui doivent l'être après avoir créé un clone pour le clone en question.

Nous parlerons du clonage d'objet dans un chapitre ultérieur, je ne vais donc pas m'attarder sur cette méthode magique pour le moment.

## Les méthodes magiques `__sleep()`, `__wakeup()`, `__set_state()` et `debugInfo()`

---

Nous n'étudierons pas ici les méthodes magiques `__sleep()`, `__wakeup()`, `__set_state()` et `debugInfo()` simplement car elles répondent à des besoins très précis et ne vont avoir d'intérêt que dans un contexte et dans un environnement spécifique.