Cours très bien rédigé, provenant de https://www.pierre-giraud.com/php-mysql-apprendre-coder-cours/oriente-objet-classe-etendue-heritage/

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.

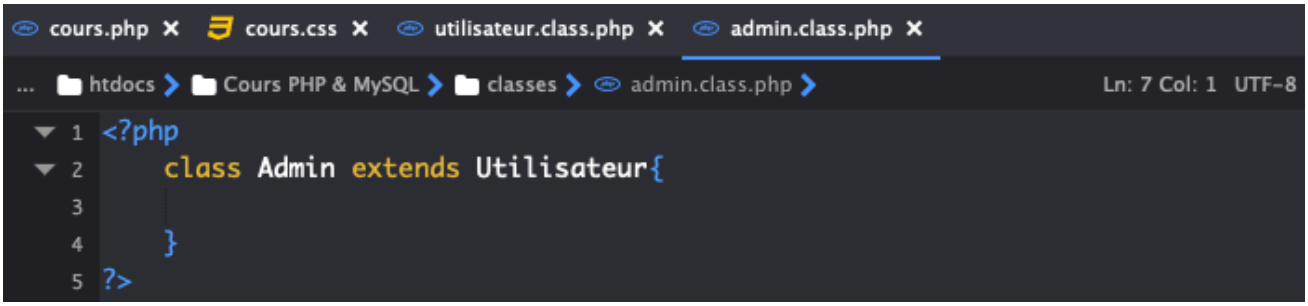
Certains développeurs vont ainsi pouvoir proposer de nouvelles fonctionnalités sans casser la structure originale de notre code et de nos scripts. C'est d'ailleurs tout le principe de la solution e-commerce PrestaShop (nous reparlerons de cela en fin de chapitre).

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 :



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">
    <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
<?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;
  }
}
?>
```

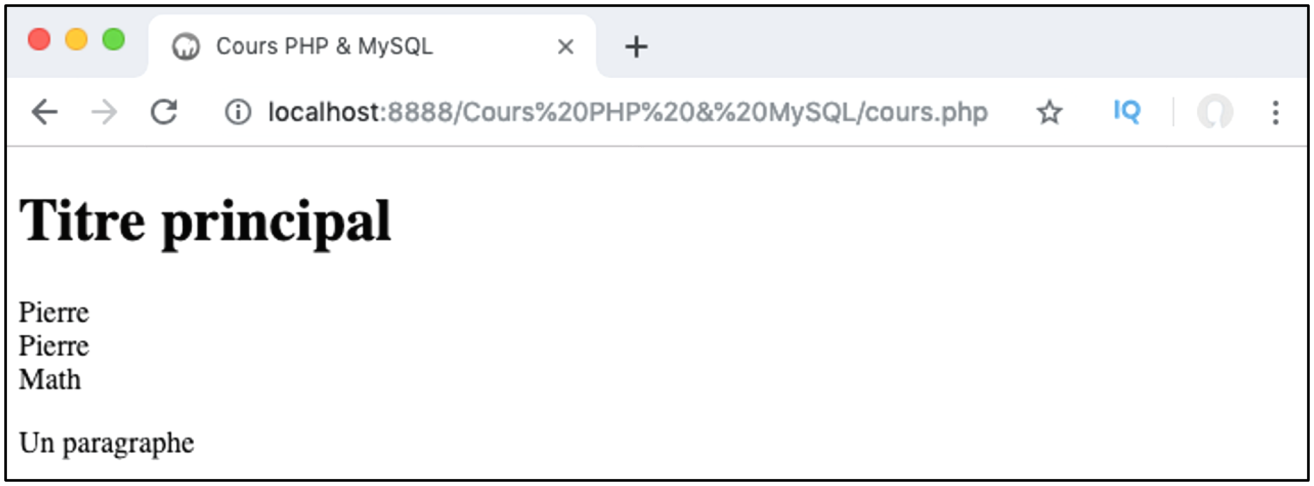
```php
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
  </head>
  <body>
    <h1>Titre principal</h1>
<?php
require 'classes/utilisateur.class.php';
require 'classes/admin.class.php';
$pierre = new Admin('Pierre', '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
<?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 (puisqu'en cas de changement il suffit de modifier le code de la classe mère) et fait gagner beaucoup de temps dans l'écriture du code.

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

Ici, nous pouvons par exemple définir de nouvelles propriétés et méthodes spécifiques à notre classe `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
<?php
class Admin extends Utilisateur{
  protected $ban;
  public function setBan($b){
    $this->ban[] .= $b;
  }
```

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

```php
<!DOCTYPE html>
<html>
<head>
<title>Cours PHP & MySQL</title>
<meta charset="utf-8">
<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 orienté objet PHP.

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 modifiiez 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).