

PARTIE XI

Programmation orientée objet : notions avancées

Chainage de méthodes

Dans cette nouvelle leçon, nous allons découvrir ce qu'est le chainage de méthodes en PHP et apprendre à chainer des méthodes en pratique.

Principe et intérêt du chainage de méthodes en POO PHP

Chainer des méthodes nous permet d'exécuter plusieurs méthodes d'affilée de façon simple et plus rapide, en les écrivant à la suite les unes des autres, « en chaine ».

En pratique, il va suffire d'utiliser l'opérateur d'objet pour chainer différentes méthodes. On écrira quelque chose de la forme `$objet->methode1()->methode2()`.

Cependant, pour pouvoir utiliser le chainage de méthodes, il va falloir que nos méthodes chaînées retournent notre objet afin de pouvoir exécuter la méthode suivante. Dans le cas contraire, une erreur sera renvoyée.

Le chaînage de méthodes en pratique

Prenons immédiatement un exemple afin de bien comprendre comment fonctionne le chainage de méthodes.

Pour cela, nous allons nous appuyer sur notre classe `Utilisateur` créée dans la partie précédente et à laquelle nous allons ajouter deux méthodes.

```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    protected $x = 0;
    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 plusUn(){
        $this->x++;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
    public function moinsUn(){
        $this->x--;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
}
?>

```

Ici, on commence par initialiser une propriété `$x` à zéro puis on crée deux nouvelles méthodes de classe `plusUn()` et `moinsUn()` dont le rôle est d'ajouter ou d'enlever un à la valeur de `$x` puis d'afficher un message avec la nouvelle position de notre objet.

```

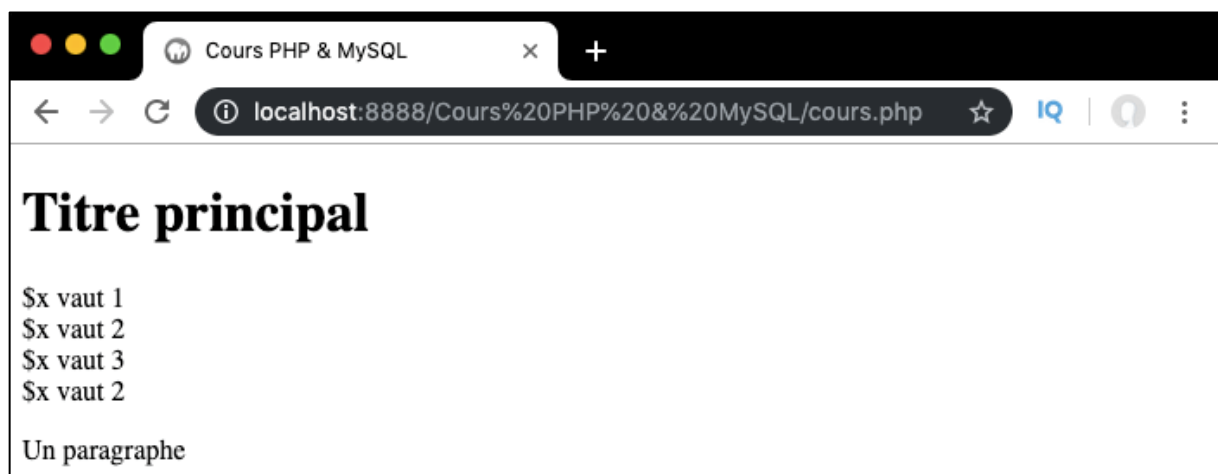
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
  </head>

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

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

      $pierre->plusUn()->plusUn()->plusUn()->moinsUn();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ensuite, on utilise l'opérateur objet pour exécuter toutes les méthodes à la suite, d'un seul coup. C'est précisément ce qu'on appelle le chaînage de méthodes.

Notez bien l'instruction `return $this` à la fin du code de chacune de nos deux méthodes. Cette instruction est ici obligatoire. En effet, comme je l'ai précisé plus haut, vous devez impérativement retourner l'objet en soi pour pouvoir utiliser le chaînage de méthodes. Si vous omettez le `return $this` vous allez avoir une erreur.

La grande limitation des méthodes chaînées est donc qu'on doit retourner l'objet afin que la méthode suivante s'exécute. On ne peut donc pas utiliser nos méthodes pour retourner une quelconque autre valeur puisqu'on ne peut retourner qu'une chose en PHP.

Closures et classes anonymes

Dans cette nouvelle leçon, nous allons aborder la notion de classes anonymes. Pour bien comprendre comment vont fonctionner les classes anonymes, nous allons en profiter pour présenter les fonctions anonymes ou « closures ».

Découverte des fonctions anonymes et de la classe Closure

Les fonctions anonymes, qu'on appelle également des closures, sont des fonctions qui ne possèdent pas de nom.

On va créer une fonction anonyme de la même façon que l'on crée une fonction normale à la différence qu'on ne va ici pas préciser de nom.

```
<!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
      function(){
        echo 'Fonction anonyme bien exécutée';
      }
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```

A ce stade, vous devriez déjà vous poser deux questions : quel est l'intérêt de créer de telles fonctions et comment peut-on les exécuter ou les appeler si elles ne possèdent pas de nom ?

Les closures en PHP ont été de manière historique principalement utilisées en tant que fonction de rappel car les fonctions de rappel. Une fonction de rappel est une fonction qui va être appelée par une autre fonction.

Notez que les fonctions anonymes sont implémentées en utilisant la classe prédéfinie **Closure**.

Appeler une fonction anonyme

Depuis PHP 7, il existe trois grands moyens simples d'appeler une fonction anonyme :

- En les auto-invoquant de manière similaire au langage JavaScript ;
- En les utilisant comme fonctions de rappel ;
- En les utilisant comme valeurs de variables.

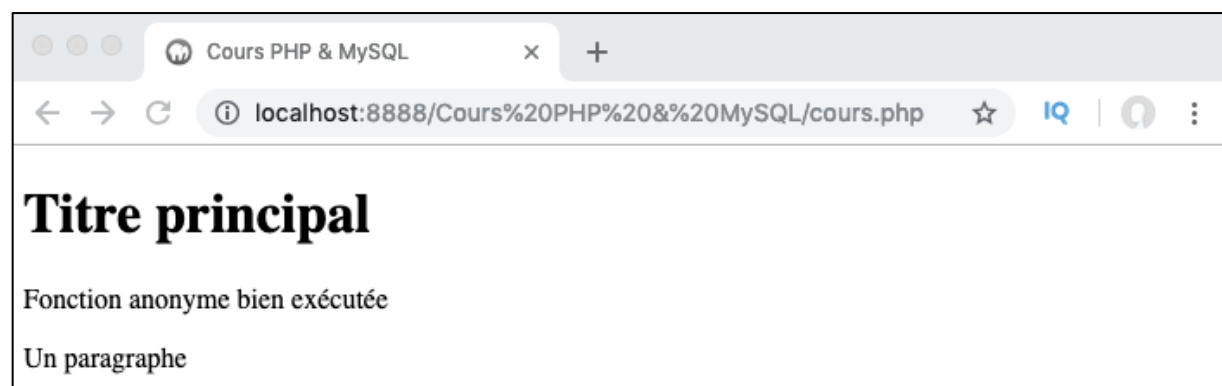
Auto-invoquer une fonction anonyme

Depuis PHP 7, on peut auto-invoquer nos fonctions anonymes, c'est-à-dire faire en sorte qu'elles s'appellent elles-mêmes de manière automatique à la manière du JavaScript.

Pour cela, il va suffire d'entourer notre fonction anonyme d'un premier couple de parenthèses et d'ajouter un autre couple de parenthèses à la suite du premier couple 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
      (function(){
        echo 'Fonction anonyme bien exécutée';
      })();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```



Créer des fonctions anonymes auto-invoquées va être très intéressant lorsqu'on ne va vouloir effectuer une tâche qu'une seule fois. Dans ce cas-là, en effet, il n'y a aucun intérêt de créer une fonction classique.

Par ailleurs, dans certains codes, nous n'allons pas pouvoir appeler une fonction manuellement mais allons vouloir que la fonction s'exécute automatiquement. Dans ces cas-là, utiliser la syntaxe ci-dessus va être très pertinent.

Utiliser une fonction anonyme comme fonction de rappel

Une fonction de rappel est une fonction qui va être appelée par une autre fonction. Pour cela, nous allons passer notre fonction de rappel en argument de la fonction appelante.

Les fonctions de rappel peuvent être de simples fonctions nommées, des fonctions anonymes ou encore des méthodes d'objets ou des méthodes statiques.

Dans le cas d'une fonction de rappel nommée, nous passerons le nom de la fonction de rappel en argument de la fonction appelante. Dans le cas d'une fonction de rappel anonyme, nous enfermerons la fonction dans une variable qu'on passera en argument de la fonction qui va l'appeler.

Bien évidemment, toutes les fonctions n'acceptent pas des fonctions de rappel en arguments mais seulement certaines comme la fonction `usort()` par exemple qui va servir à trier un tableau en utilisant une fonction de comparaison ou encore la fonction `array_map()` qui est la fonction généralement utilisée pour illustrer l'intérêt des closures.

La fonction `array_map()` va appliquer une fonction sur des éléments d'un tableau et retourner un nouveau tableau. On va donc devoir passer deux arguments à celle-ci : une fonction qui va dans notre cas être une closure et un tableau. Prenons immédiatement un exemple.

```

<!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>

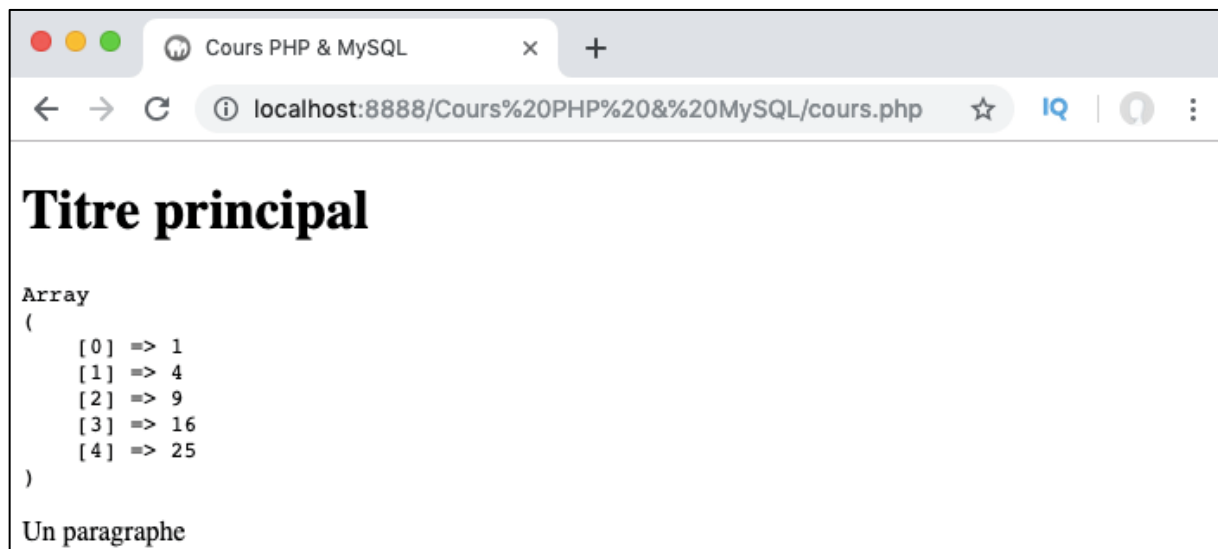
  <body>
    <h1>Titre principal</h1>
    <?php
      /*La closure ci-dessous accepte un nombre en argument et
      *retourne son carré*/
      $squ = function(float $x){
        return $x**2;
      };

      //Définition d'un tableau
      $tb = [1, 2, 3, 4, 5];

      //array_map() exécute notre closure sur chaque élément du tableau
      $tb_squ = array_map($squ, $tb);

      echo '<pre>';
      print_r($tb_squ);
      echo '</pre>';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Dans cet exemple, on commence par créer une closure dont le rôle est de calculer et de renvoyer le carré d'un nombre. On affecte notre closure à la variable `$squ` puis on crée ensuite une variable tableau stockant cinq chiffres.

Finalement, on utilise notre fonction `array_map()` en lui passant notre variable contenant notre fonction de rappel ainsi que notre tableau afin qu'elle renvoie un nouveau tableau et appliquant notre fonction de rappel à chaque élément du tableau passé.

On stocke le résultat renvoyé par `array_map()` dans une nouvelle variable `$tb_squ` qui est également une variable tableau et on affiche son contenu avec `print_r()`.

Appeler des fonctions anonymes en utilisant des variables

Lorsqu'on assigne une fonction anonyme en valeur de variable, notre variable va automatiquement devenir un objet de la classe prédéfinie `Closure`.

La classe `Closure` possède des méthodes qui vont nous permettre de contrôler une closure après sa création. Cette classe possède également une méthode magique `__invoke()` qui va ici s'avérer très utile puisqu'on va donc pouvoir exécuter nos closures simplement.

Je vous rappelle ici que la méthode magique `__invoke()` va s'exécuter dès qu'on se sert d'un objet comme d'une fonction.

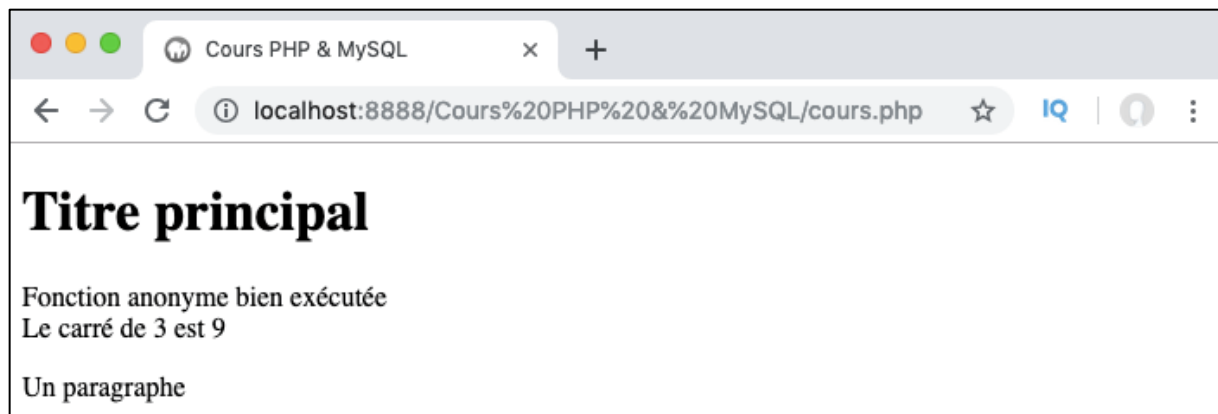
Cela va nous permettre d'utiliser la syntaxe suivante pour appeler nos fonctions anonymes :

```
<!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
      $txt = function(){
        echo 'Fonction anonyme bien exécutée';
      };

      $squ = function(float $x){
        return 'Le carré de ' . $x . ' est ' . $x**2;
      };

      $txt();
      echo '<br>';
      echo $squ(3);
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```



Comme vous pouvez le constater, nos variables objets `$txt` et `$squ` sont utilisées comme fonctions pour exécuter les closures contenues à l'intérieur et les résultats sont bien renvoyés.

Définition et intérêt des classes anonymes

Les classes anonymes ont été implémentées récemment en PHP, puisque leur support n'a été ajouté qu'avec le PHP 7.

Les classes anonymes, tout comme les fonctions anonymes, sont des classes qui ne possèdent pas de nom.

Les classes anonymes vont être utiles dans le cas où des objets simples et uniques ont besoin d'être créés à la volée.

Créer des classes anonymes va donc principalement nous faire gagner du temps et améliorer in-fine la clarté de notre code.

On va pouvoir passer des arguments aux classes anonymes via la méthode constructeur et celles-ci vont pouvoir étendre d'autres classes ou encore implémenter des interfaces et utiliser des traits comme le ferait une classe ordinaire.

Notez qu'on va également pouvoir imbriquer une classe anonyme à l'intérieur d'une autre classe. Toutefois, on n'aura dans ce cas pas accès aux méthodes ou propriétés privées ou protégées de la classe contenante.

Pour utiliser des méthodes ou propriétés protégées de la classe contenante, la classe anonyme doit étendre celle-ci. Pour utiliser les propriétés privées de la classe contenant dans la classe anonyme, il faudra les passer via le constructeur.

Créer et utiliser des classes anonymes

Voyons immédiatement comment créer et manipuler des classes anonymes à travers différents exemples.

```

<!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
      //On crée une classe anonyme qu'on stocke dans une variable (objet)
      $anonyme = new class{
        public $user_name;
        public const BONJOUR = 'Bonjour ';

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

      $anonyme->setNom('Pierre');
      echo $anonyme::BONJOUR;
      echo $anonyme->getNom();
      echo '<br><br>';

      //Affiche les infos de $anonyme
      var_dump($anonyme);
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on commence avec un exemple simple en se contentant de déclarer une classe anonyme qu'on assigne à une variable qui devient de fait un objet.

Notre classe anonyme contient des propriétés, méthodes et constantes tout comme une classe classique.

Ensuite, on effectue différentes opérations simples : récupération et affichage des valeurs des propriétés, exécution des méthodes de notre classe anonyme, etc. afin que vous puissiez observer les différentes opérations que l'on peut réaliser.

Notez bien une nouvelle fois que le support des classes anonymes par le PHP est relativement récent : il est donc tout à fait possible que votre éditeur ne reconnaisse pas cette écriture (ce qui n'est pas grave en soi) ou que votre WAMP, MAMP, etc. n'arrive pas à l'exécuter si vous ne possédez une version PHP postérieure à la version 7.

On peut encore assigner une classe anonyme à une variable en passant par une fonction.

Dans ce cas-là, on pourra écrire quelque chose 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
      //On crée une classe anonyme qu'on stocke dans une variable (objet)
      function anonyme(){
        return new class{
          public $user_name;
          public const BONJOUR = 'Bonjour ';

          public function setNom($n){
            $this->user_name = $n;
          }
          public function getNom(){
            return $this->user_name;
          }
        };
      }
      $anonyme = anonyme();

      $anonyme->setNom('Pierre');
      echo $anonyme::BONJOUR;
      echo $anonyme->getNom();
      echo '<br><br>';

      //Affiche les infos de $anonyme
      var_dump($anonyme);
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```

Ce code est tout à fait comparable au précédent à la différence qu'on crée une fonction qui va retourner la définition de notre classe anonyme tout simplement.

Finalement, on peut également utiliser un constructeur pour passer des arguments à une classe anonyme lors de sa création.

```

<!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
      //On crée une fonction qui retourne une classe anonyme
      function anonyme($n){
        return new class($n){
          public $user_name;
          public const BONJOUR = 'Bonjour ';

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

      //On stocke le résultat de la fonction dans une variable objet
      $anonyme = anonyme('Pierre');
      echo $anonyme::BONJOUR;
      echo $anonyme->getNom();
      echo '<br><br>';

      //Affiche les infos de $anonyme
      var_dump($anonyme);
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```

Ici, on définit le même paramètre `$n` lors de la définition de notre fonction et de notre classe anonyme. On passe ensuite l'argument `Pierre` lors de l'affectation du résultat de notre fonction dans la variable `$anonyme`. Cet argument va être stocké dans la propriété `$user_name` de notre classe.

Finalement, retenir que dans le cas où une classe anonyme est imbriquée dans une autre classe, la classe anonyme doit l'étendre afin de pouvoir utiliser ses propriétés et méthodes protégées. Pour utiliser ses méthodes et propriétés privées, alors il faudra également les passer via le constructeur.

Regardez plutôt l'exemple suivant :

```

<!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
      class Externe{
        private $age = 29;
        protected $nom = 'Pierre';

        public function anonyme(){
          return new class($this->age) extends Externe{
            private $a;
            private $n;

            public function __construct($age){
              $this->a = $age;
            }
            public function getNomAge(){
              return 'Nom : ' . $this->nom. ', âge : ' . $this->a;
            }
          };
        }
      }

      $obj = new Externe;
      echo $obj->anonyme()->getNomAge();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on déclare une première classe nommée **Externe** qui contient une propriété privée **\$age** et une propriété protégée **\$nom**.

Notre classe `Externe` contient également une méthode qui retourne une classe anonyme. On va vouloir utiliser les propriétés de code `>Externe` dans la classe anonyme. Pour cela, on passe notre variable protégée dans la définition de la classe et dans le constructeur.

Cette leçon et ce dernier exemple en particulier doivent vous sembler plus difficile à appréhender que le reste jusqu'ici. Pas d'inquiétude, c'est tout à fait normal car on commence à toucher à des notions vraiment avancées et il n'est pas simple d'en montrer l'intérêt à travers des exemples simples.

En pratique, je vous rassure, vous n'aurez que très rarement à faire ce genre de choses ou même à utiliser les classes anonymes.

Cependant, utiliser des classes anonymes peut s'avérer très pratique dans certaines situations et je dois donc vous présenter ce qu'il est possible de faire avec elles aujourd'hui en POO PHP.

L'auto chargement des classes

Il est considéré comme une bonne pratique en PHP orienté objet de créer un fichier par classe. Ceci a principalement pour but de conserver une meilleure clarté dans l'architecture générale d'un site et de simplifier la maintenabilité du code en séparant bien les différents éléments.

L'un des inconvénients de cette façon de procéder, cependant, est qu'on va possiblement avoir à écrire de longues séries d'inclusion de classes (une inclusion par classe) dans nos scripts lorsque ceux-ci ont besoin de plusieurs classes.

Pour éviter de rallonger le code inutilement et de nous faire perdre du temps, nous avons un moyen en PHP de charger (c'est-à-dire d'inclure) automatiquement nos classes d'un seul coup dans un fichier.

Pour cela, nous allons pouvoir utiliser la fonction `spl_autoload_register()`. Cette fonction nous permet d'enregistrer une ou plusieurs fonctions qui vont être mises dans une file d'attente et que le PHP va appeler automatiquement dès qu'on va essayer d'instancier une classe.

L'idée ici va donc être de passer une fonction qui permet de n'inclure que les classes dont on a besoin dans un script et de la passer à `spl_autoload_register()` afin qu'elle soit appelée dès que cela est nécessaire.

On va pouvoir ici soit utiliser une fonction nommée, soit idéalement créer une fonction anonyme :

```

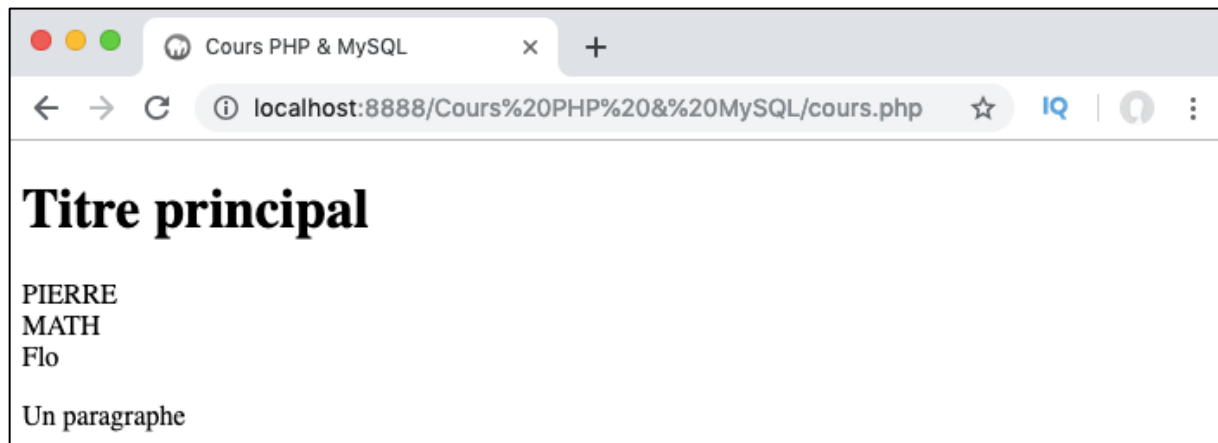
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

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

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

```



Dans ce script, on tente d'instancier nos classes `Admin` et `Abonne` créées précédemment. Pour rappel, ces deux classes étendent la classe parent `Utilisateur`.

On utilise ici la fonction `spl_autoload_register()` en lui passant une fonction anonyme en argument donc le rôle est d'inclure des fichiers de classe.

En résultat, la fonction `spl_autoload_register()` sera appelée dès qu'on va instancier une classe et va tenter d'inclure la classe demandée en exécutant la fonction anonyme. Notez

que cette fonction va également tenter de charger les éventuelles classes parents en commençant par les parents.

Ici, la fonction `spl_autoload_register()` va donc tenter d'inclure les fichiers `utilisateur.class.php`, `admin.class.php` et `abonne.class.php` situés dans un dossier « classes ».

Vous comprenez ici j'espère tout l'intérêt de placer tous nos fichiers de classes dans un même dossier et de respecter une norme d'écriture lorsqu'on nomme nos fichiers de classe puisque cela nous permet de pouvoir écrire des instructions formatées comme le `require` de notre fonction `spl_autoload_register()`.

Tenter d'auto-charger une classe inaccessible

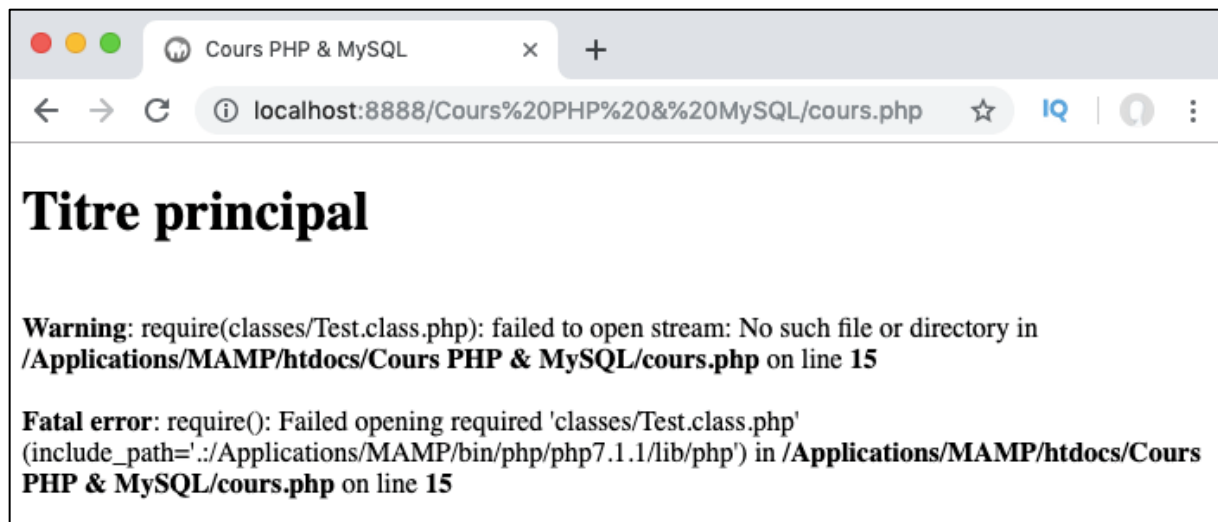
Notez que si vous tentez d'inclure une classe qui est introuvable ou inaccessible le PHP renverra une erreur fatale.

```
<!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
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      $pierre = new Admin('Pierre', 'abcdef', 'Sud');
      $mathilde = new Admin('Math', 123456, 'Nord');
      $florian = new Abonne('Flo', 'flotri', 'Est');
      $test = new Test(); //La classe Test n'existe pas

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



On va pouvoir prendre en charge les erreurs et les exceptions en particulier en utilisant la classe **Exception**. La prise en charge des erreurs et des exceptions est cependant un sujet relativement complexe qui justifie une partie de cours en soi. Nous verrons comment cela fonctionne en détail dans la prochaine partie.

Le mot clef final en PHP objet

Depuis la version 5 de PHP, on peut empêcher les classes filles de surcharger une méthode en précisant le mot clef `final` avant la définition de celle-ci.

Si la classe elle-même est définie avec le mot clef `final` alors celle-ci ne pourra tout simplement pas être étendue.

Cela peut être utile si vous souhaitez empêcher explicitement certains développeurs de surcharger certaines méthodes ou d'étendre certaines classes dans le cas d'un projet Open Source par exemple.

Définir une méthode finale

Illustrons cela avec quelques exemples, en commençant avec la définition d'une méthode finale.

Pour cela, on peut reprendre nos classes `Utilisateur`, `Abonne` et `Admin` et par exemple déjà surcharger la méthode `getNom()` définie dans la classe parent `Utilisateur` depuis notre classe étendue `Admin` :

```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    protected $x = 0;
    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 plusUn(){
        $this->x++;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
    public function moinsUn(){
        $this->x--;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
}
?>

```

```

<?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 getNom(){
        echo $this->user_name. '(Admin)';
    }

    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, lorsqu'on tente d'appeler notre méthode `getNom()` depuis un objet de la classe `Admin`, la définition de la méthode mère est bien surchargée et c'est la définition de la classe fille qui est utilisée.

```

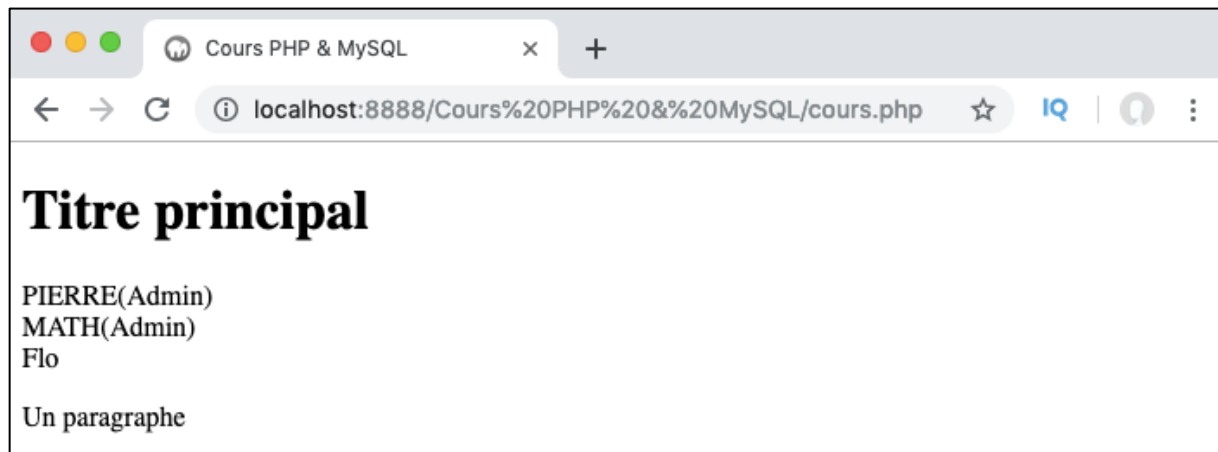
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

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

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

```



Essayons maintenant de définir notre méthode `getNom()` comme finale dans la classe `Utilisateur`.


```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    protected $x = 0;
    public const ABONNEMENT = 15;

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

    abstract public function setPrixAbo();

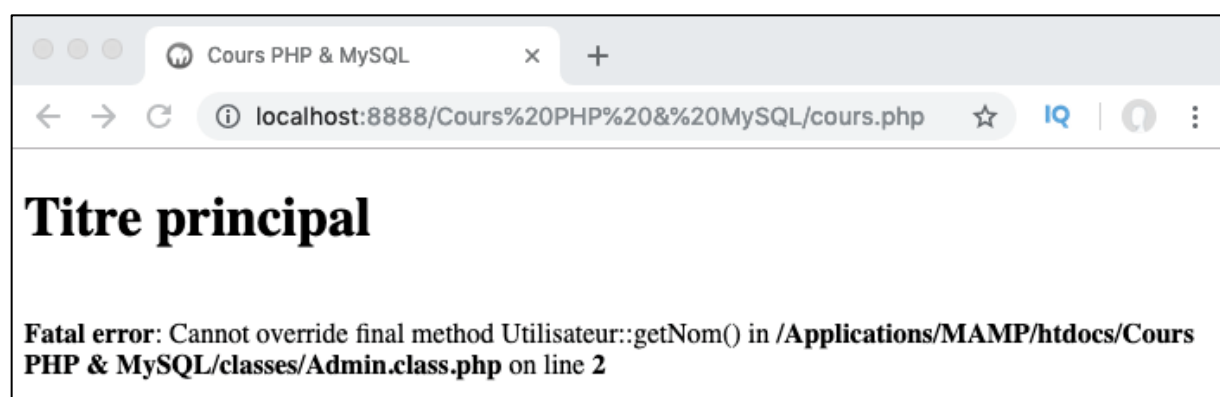
    final public function getNom(){
        echo $this->user_name;
    }

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

    public function plusUn(){
        $this->x++;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
    public function moinsUn(){
        $this->x--;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
}
?>

```

Comme notre méthode est définie avec le mot clef **final**, on n'a plus le droit de la surcharger dans une classe étendue. Si on tente de faire cela, une erreur fatale sera levée par le PHP :



Définir une classe finale

Si on définit une classe avec le mot clef `final`, on indique que la classe ne peut pas être étendue. Là encore, si on tente tout de même d'étendre la classe, le PHP renverra une erreur fatale.

```
<?php
final class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    protected $x = 0;
    public const ABONNEMENT = 15;

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

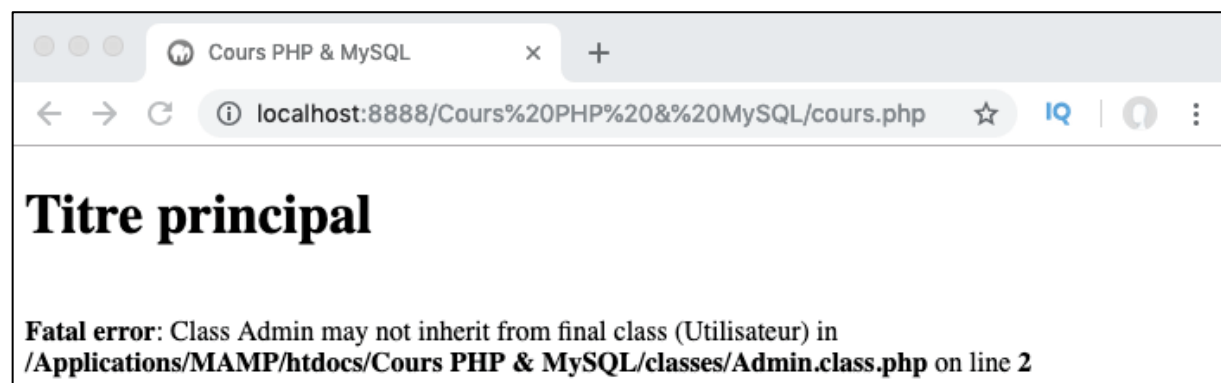
    public function setPrixAbo(){

    }

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

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

    public function plusUn(){
        $this->x++;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
    public function moinsUn(){
        $this->x--;
        echo '$x vaut ' . $this->x. '<br>';
        return $this;
    }
}
?>
```



Notez ici que déclarer une classe comme abstraite et finale n'a aucun sens puisqu'une classe abstraite est par définition une classe qui va laisser à ses classes étendues le soin d'implémenter certains de ses éléments alors qu'une classe finale ne peut justement pas être étendue.

Par définition, une classe finale est une classe dont l'implémentation est complète puisqu'en la déclarant comme finale on indique qu'on ne souhaite pas qu'elle puisse être étendue. Ainsi, aucune méthode abstraite n'est autorisée dans une classe finale.

Résolution statique à la volée - late static bindings

Dans cette nouvelle leçon, nous allons découvrir une fonctionnalité très intéressante du PHP appelée la résolution statique à la volée ou « late static binding » en anglais et comprendre les problèmes qu'elle résout.

Définition et intérêt de la résolution statique à la volée

La résolution statique à la volée va nous permettre de faire référence à la classe réellement appelée dans un contexte d'héritage statique.

En effet, lorsqu'on utilise le `self::` pour faire référence à la classe courante dans un contexte statique, la classe utilisée sera toujours celle dans laquelle sont définies les méthodes utilisant `self::`.

Cela peut parfois produire des comportements inattendus. Regardez plutôt l'exemple ci-dessous pour vous en convaincre.

```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    protected $x = 0;
    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 static function getStatut(){
        self::statut();
    }
    public static function statut(){
        echo 'Utilisateur';
    }

    public function getNom(){
        echo $this->user_name;
    }
    public function getPrixAbo(){
        echo $this->prix_abo;
    }
    abstract public function 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 static function statut(){
        echo 'Admin';
    }
    public function getNom(){
        echo $this->user_name. '(Admin)';
    }
    public function setBan(...$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;
        }
    }
}
?>

```

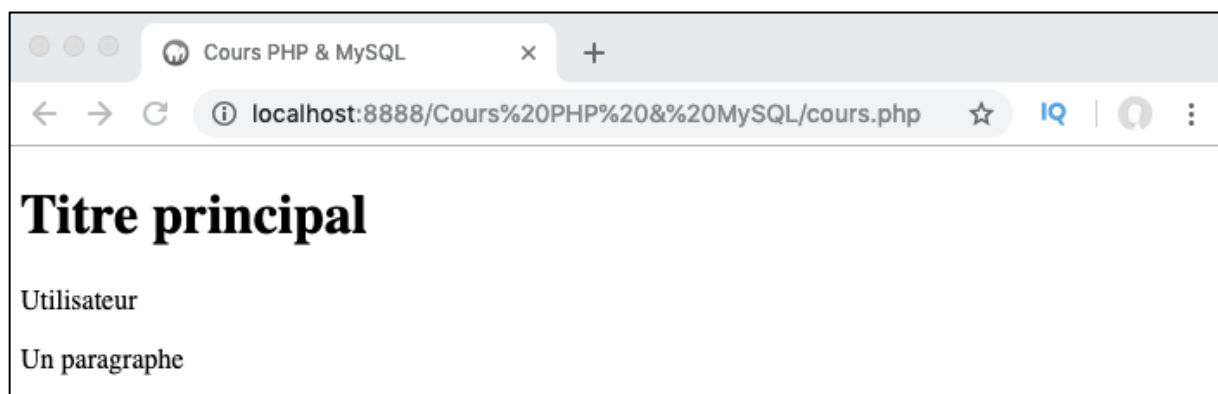
```

<!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
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      /*On n'a pas besoin d'instancier nos classes ici puisqu'on se contente
      *d'utiliser une méthode statique (= qui appartient à la classe)*/
      Admin::getStatut();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on réutilise nos classes `Utilisateur` (classe mère) et `Admin` (classe étendue). Dans notre classe `Utilisateur`, on définit deux méthodes statiques `getStatut()` et `statut()`.

La méthode `statut()` de la classe `Utilisateur` renvoie le mot « Utilisateur ». La méthode `getStatut()` sert elle à exécuter la méthode `statut()` de la classe courante.

On surcharge ensuite notre méthode `statut()` dans notre classe étendue `Admin` afin qu'elle renvoie le texte « Admin ».

Finalement, dans notre script principal, on appelle notre méthode `getStatut()` depuis notre classe `Admin`.

Comme vous pouvez le voir, le résultat renvoyé est « Utilisateur » et non pas « Admin » comme on aurait pu le penser instinctivement. Cela est dû au fait que le code `self::` dans notre méthode `getStatut()` va toujours faire référence à la classe dans laquelle la méthode a été définie, c'est-à-dire la classe `Utilisateur`.

Ainsi, `self::statut()` sera toujours l'équivalent de `Utilisateur::statut()` et renverra toujours la valeur de la méthode `statut()` définie dans la classe `Utilisateur`.

La résolution statique à la volée a été introduite justement pour dépasser ce problème précis et pour pouvoir faire référence à la classe réellement utilisée.

Utilisation de la résolution statique à la volée et de `static::`

La résolution statique à la volée va donc nous permettre de faire référence à la classe réellement utilisée dans un contexte statique.

Pour utiliser la résolution statique à la volée, nous allons simplement devoir utiliser le mot clef `static` à la place de `self`. Ce mot clef va nous permettre de faire référence à la classe utilisée durant l'exécution de notre méthode.

Reprenons l'exemple précédent et changeons `self::` par `static::` dans le code de notre méthode `getStatut()`.


```

<?php
abstract class Utilisateur{
    protected $user_name;
    protected $user_region;
    protected $prix_abo;
    protected $user_pass;
    protected $x = 0;
    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 static function getStatut(){
        static::$statut();
    }
    public static function statut(){
        echo 'Utilisateur';
    }

    public function getNom(){
        echo $this->user_name;
    }
    public function getPrixAbo(){
        echo $this->prix_abo;
    }
    abstract public function setPrixAbo();
}
?>

```

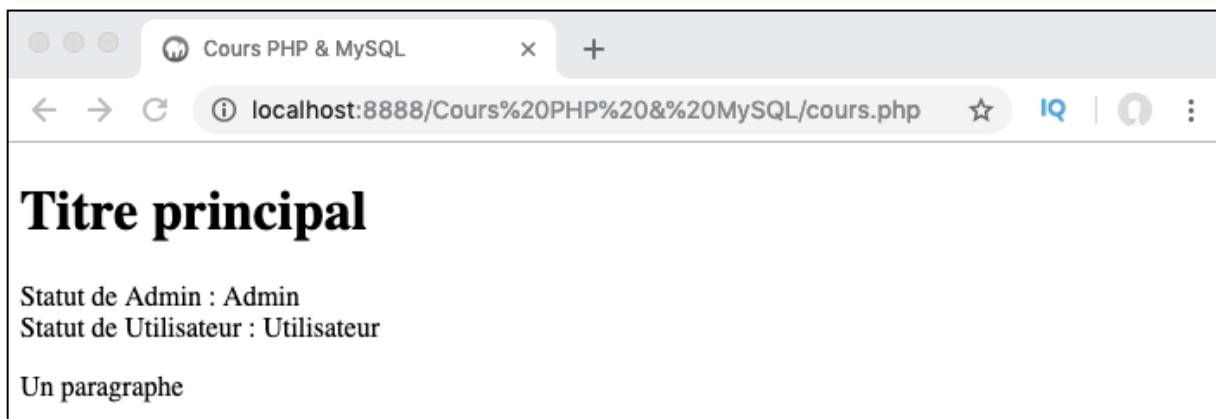
```

<!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
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      /*On n'a pas besoin d'instancier nos classes ici puisqu'on se contente
      *d'utiliser une méthode statique (= qui appartient à la classe)*/
      echo 'Statut de Admin : ' ;
      Admin::getStatut();
      echo '<br>Statut de Utilisateur : ' ;
      Utilisateur::getStatut();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Cette fois-ci, la méthode `getStatut()` va exécuter la méthode `statut()` de la classe utilisée, c'est-à-dire de la classe `Admin` et c'est donc la valeur « Admin » qui va être renvoyée.

Les traits

Dans cette nouvelle leçon, nous allons découvrir une fonctionnalité PHP qui va nous permettre de réutiliser du code dans des classes indépendantes et qu'on appelle « traits ».

Définition et intérêt des traits

Les traits sont apparus avec la version 5.4 du PHP. Très simplement, les traits correspondent à un mécanisme nous permettant de réutiliser des méthodes dans des classes indépendantes, repoussant ainsi les limites de l'héritage traditionnel.

En effet, rappelez-vous qu'en PHP une classe ne peut hériter que d'une seule classe mère.

Or, imaginons que nous devons définir la même opération au sein de plusieurs classes indépendantes, c'est-à-dire des classes qui ne partagent pas de fonctionnalité commune et pour lesquelles il n'est donc pas pertinent de créer une classe mère et de les faire étendre cette classe.

Dans ce cas-là, nous allons être obligé de réécrire le code correspondant à la méthode que ces classes ont en commun dans chacune des classes à moins justement d'utiliser les traits qui permettent à plusieurs classes d'utiliser une même méthode.

Par exemple, on peut imaginer qu'un site marchand possède deux classes **Utilisateur** et **Produit** qui vont être indépendantes mais qui vont posséder certaines méthodes en commun comme une méthode de comptage **plusUn()** par exemple.

Comme ces classes sont indépendantes et qu'on ne veut donc pas les faire hériter d'une même classe mère, on va être obligé de réécrire le code de notre méthode dans les deux classes si on n'utilise pas les traits :

```
< cours.php X cours.css X utilisateur.class.php X produit.class.php X abonne.class.php X a
... htdocs > Cours PHP & MySQL > classes > utilisateur.class.php > Ln: 34 Col: 1 UTF-8
1 <?php
2 class Utilisateur{
3     protected $user_name;
4     protected $user_region;
5     protected $prix_abo;
6     protected $user_pass;
7     protected $nombre;
8     public const ABONNEMENT = 15;
9
10    public function __construct($n, $p, $r, $nb){
11        $this->user_name = $n;
12        $this->user_pass = $p;
13        $this->user_region = $r;
14        $this->nombre = $nb;
15    }
16    public function __destruct(){
17        //Du code à exécuter
18    }
19
20    public function getNom(){
21        echo $this->user_name;
22    }
23
24    public function plusUn(){
25        $this->nombre++;
26        echo $this->nombre. '<br>';
27        return $this;
28    }
29
30    //D'autres méthodes...
31 }
32 ?>
```

```
< cours.php X cours.css X utilisateur.class.php X produit.class.php X abonne.class.php X a
... htdocs > Cours PHP & MySQL > classes > produit.class.php > Ln: 25 Col: 1 UTF-8
1 <?php
2 class Produit{
3     protected $nom;
4     protected $nombre;
5
6     public function __construct($n, $nb){
7         $this->nom = $n;
8         $this->nombre = $nb;
9     }
10
11    public function getNom(){
12        echo $this->nom;
13    }
14
15    public function plusUn(){
16        $this->nombre++;
17        echo $this->nombre. '<br>';
18        return $this;
19    }
20
21    //D'autres méthodes...
22 }
23 ?>
```

Si on tente ensuite d'utiliser notre méthode, cela va bien évidemment fonctionner mais il ne sera pas optimisé puisqu'on a dû réécrire notre méthode deux fois.

```

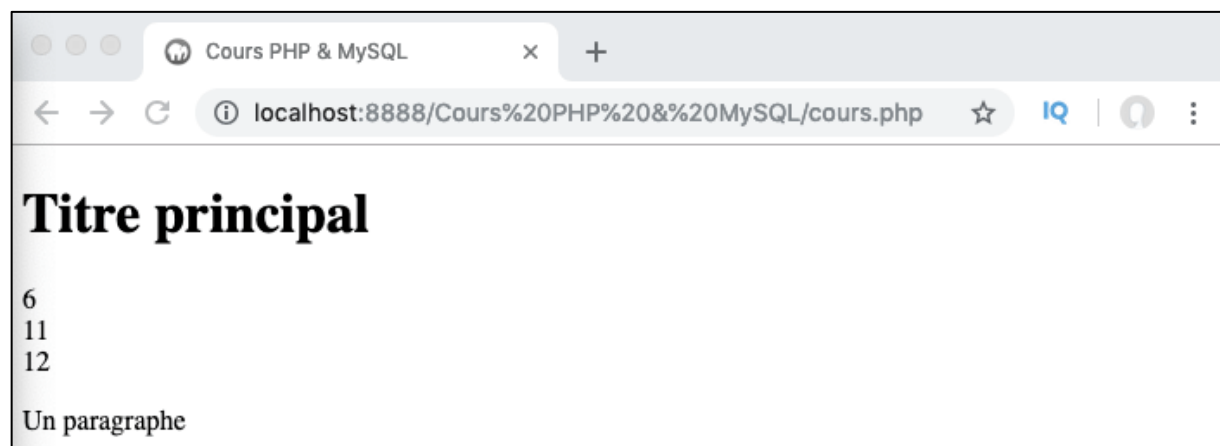
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      $pierre = new Utilisateur('Pierre', 'abcdef', 'Sud', 5);
      $yeti = new Produit('Yeti', 10);

      $pierre->plusUn();
      $yeti->plusUn()->plusUn();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Dans le cas présent, ce n'est pas trop grave mais imaginez maintenant que nous ayons des dizaines de classes utilisant certaines mêmes méthodes... Cela va faire beaucoup de code écrit pour rien et en cas de modification d'une méthode il faudra modifier chaque classe, ce qui est loin d'être optimal !

Pour optimiser notre code, il va être intéressant dans ce cas d'utiliser les traits. Un trait est semblable dans l'idée à une classe mère en ce sens qu'il sert à grouper des fonctionnalités qui vont être partagées par plusieurs classes mais, à la différence des classes, on ne va pas pouvoir instancier un trait.

De plus, vous devez bien comprendre que le mécanisme des traits est un ajout à l'héritage « traditionnel » en PHP et que les méthodes contenues dans les traits ne vont pas être «

héritées » dans le même sens que ce qu'on a pu voir jusqu'à présent par les différentes classes.

Utiliser les traits en pratique

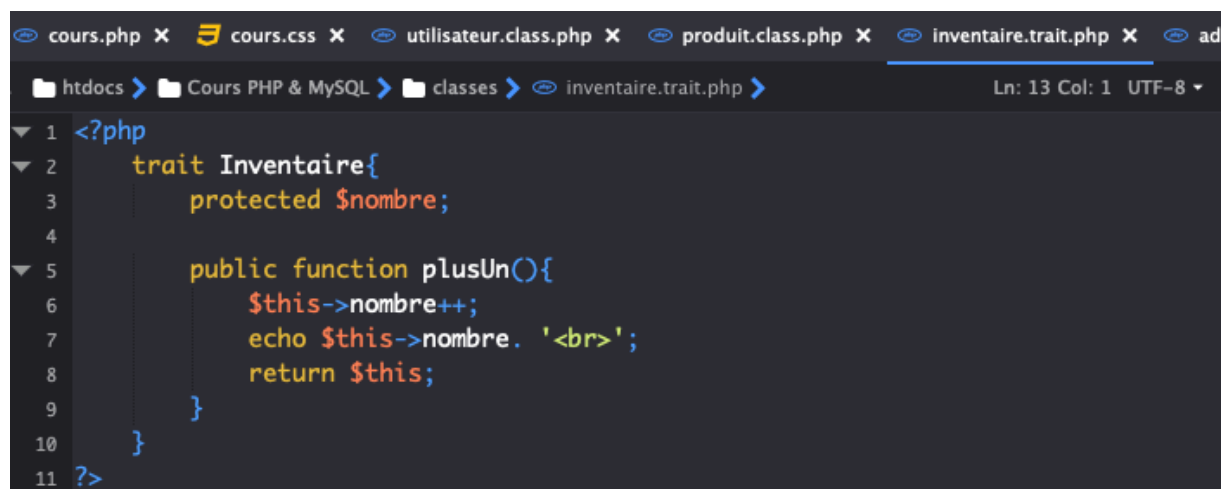
On va définir un trait de façon similaire à une classe, à la différence que nous allons utiliser le mot clef `trait` suivi du nom de notre trait.

Une bonne pratique consiste à utiliser un nouveau fichier pour chaque nouveau trait (on inclura ensuite les traits dans les classes qui en ont besoin). Ici, on peut déjà créer un trait qu'on va appeler `Inventaire`.

Dans ce trait, nous allons définir une propriété `$nombre` et une méthode `plusUn()`.

Notez qu'on peut tout à fait inclure des propriétés dans nos traits. Il faut cependant faire bien attention à la visibilité de celles-ci et ne pas abuser de cela au risque d'avoir un code au final moins clair et plus faillible.

Notez également que si on définit une propriété dans un trait, alors on ne peut pas à nouveau définir une propriété de même nom dans une classe utilisant notre trait à moins que la propriété possède la même visibilité et la même valeur initiale.



```
1 <?php
2 trait Inventaire{
3     protected $nombre;
4
5     public function plusUn(){
6         $this->nombre++;
7         echo $this->nombre. '<br>';
8         return $this;
9     }
10 }
11 ?>
```

Une fois notre trait défini, nous devons préciser une instruction `use` pour pouvoir l'utiliser dans les différentes classes qui vont en avoir besoin. On peut également en profiter pour supprimer la propriété `$nombre` et la méthode `plusUn()` de ces classes.

```

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

    public function __construct($n, $p, $r, $nb){
        $this->user_name = $n;
        $this->user_pass = $p;
        $this->user_region = $r;
        $this->nombre = $nb;
    }
    public function __destruct(){
        //Du code à exécuter
    }

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

    //D'autres méthodes...
}
?>

```

```

<?php
class Produit{
    use Inventaire;
    protected $nom;

    public function __construct($n, $nb){
        $this->nom = $n;
        $this->nombre = $nb;
    }

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

    //D'autres méthodes...
}
?>

```

Dans notre script principal, nous allons également devoir inclure notre trait pour l'utiliser. On va faire cela de manière « classique », c'est-à-dire en dehors de la fonction `spl_autoload_register()` ici car il faudrait la modifier pour qu'elle accepte un fichier en `.trait.php`.

Les classes vont maintenant pouvoir utiliser les propriétés et les méthodes définies dans le trait et notre code va à nouveau fonctionner.


```

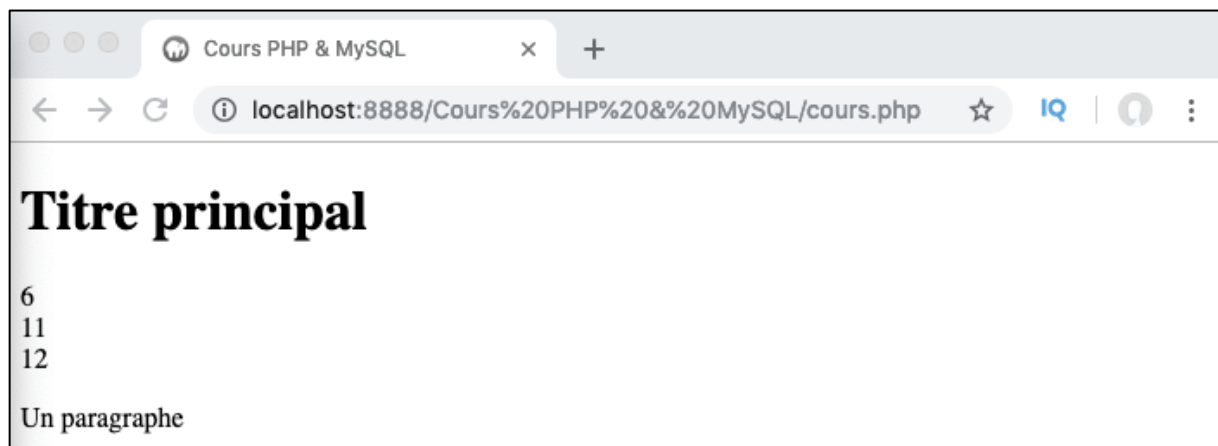
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/inventaire.trait.php';
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      $pierre = new Utilisateur('Pierre', 'abcdef', 'Sud', 5);
      $yeti = new Produit('Yeti', 10);

      $pierre->plusUn();
      $yeti->plusUn()->plusUn();
    <?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Utiliser un trait ici nous a permis de pouvoir réutiliser notre méthode `plusUn()` et notre propriété `$nombre` dans des classes indépendantes.

Ordre de précedence (ordre de priorité)

Dans le cas où une classe hérite d'une méthode d'une classe mère, celle-ci va être écrasée par une méthode (du même nom, bien évidemment) provenant d'un trait.

En revanche, dans le cas où une classe définit elle-même une méthode, celle-ci sera prédominante par rapport à celle du trait.

Ainsi, une méthode issue de la classe elle-même sera prioritaire sur celle venant d'un trait qui sera elle-même prédominante par rapport à une méthode héritée d'une classe mère. Pour illustrer cela, nous allons commencer par définir une méthode `precedence()` dans notre trait qui va renvoyer un texte.

```
<?php
trait Inventaire{
    protected $nombre;

    public function plusUn(){
        $this->nombre++;
        echo $this->nombre. '<br>';
        return $this;
    }

    public function precedence(){
        echo 'Méthode issue du trait<br>';
    }
}
```

Ensuite, nous allons récupérer notre classe `Utilisateur` et notre classes étendue `Admin` créée précédemment et allons utiliser notre trait dans chacune d'entre elles.

Nous allons également redéfinir notre méthode `precedence()` dans la classe `Utilisateur` mais pas dans `Admin`.

```

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

    public function __construct($n, $p, $r, $nb){
        $this->user_name = $n;
        $this->user_pass = $p;
        $this->user_region = $r;
        $this->nombre = $nb;
    }
    public function __destruct(){
        //Du code à exécuter
    }

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

    public function precedance(){
        echo 'Méthode issue de Utilisateur<br>';
    }
    //D'autres méthodes...
}
?>

```

```

<?php
class Admin extends Utilisateur{
    use Inventaire;
    public function getNom(){
        echo $this->user_name. '(Admin)';
    }

    //Plus de méthodes...
}
?>

```

Finalement, on instancie nos deux classes et on appelle notre méthode via nos deux objets créés pour observer le résultat.

```

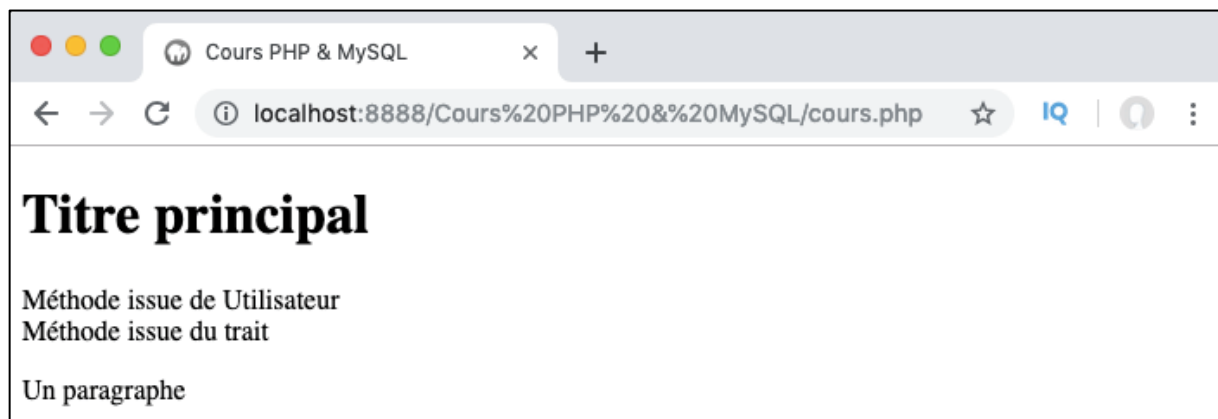
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      require 'classes/inventaire.trait.php';
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      $pierre = new Utilisateur('Pierre', 'abcdef', 'Sud', 5);
      $mathilde = new Admin('Math', 123456, 'Nord', 2);

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

```



Comme on peut le constater, la méthode définie dans **Utilisateur** est celle utilisée par-dessus celle définie dans le trait pour l'objet issu de cette classe.

En revanche, pour l'objet issu de **Admin**, c'est la méthode du trait qui va être utilisée par-dessus celle de la classe mère.

Inclusion de plusieurs traits et gestion des conflits

L'un des intérêts principaux des traits est qu'on va pouvoir utiliser plusieurs traits différents dans une même classe.

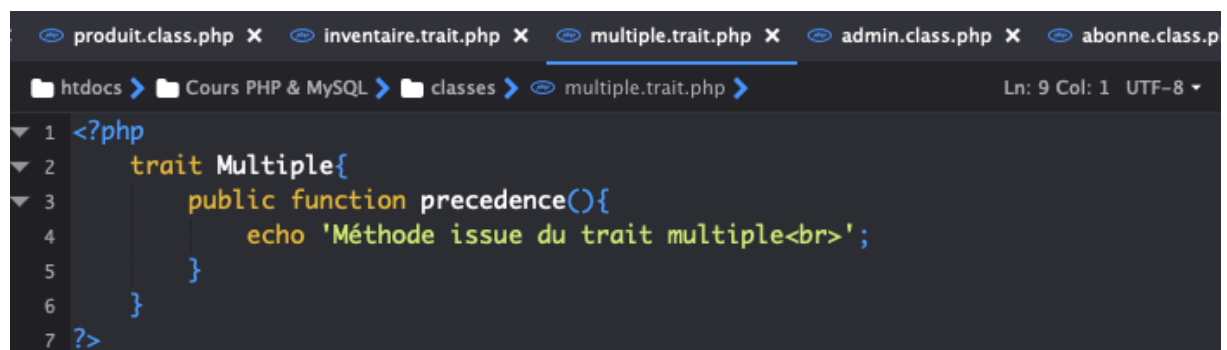
Cependant, cela nous laisse à priori avec le même problème d'héritage multiple vu précédemment et qui interdisait à une classe d'hériter de plusieurs classes parents.

En effet, imaginez le cas où une classe utilise plusieurs traits qui définissent une méthode de même nom mais de façon différente. Quelle définition doit alors être choisie par la classe ?

Dans ce genre de cas, il va falloir utiliser l'opérateur `insteadof` (« plutôt que » ou « à la place de » en français) pour choisir explicitement quelle définition de la méthode doit être choisie.

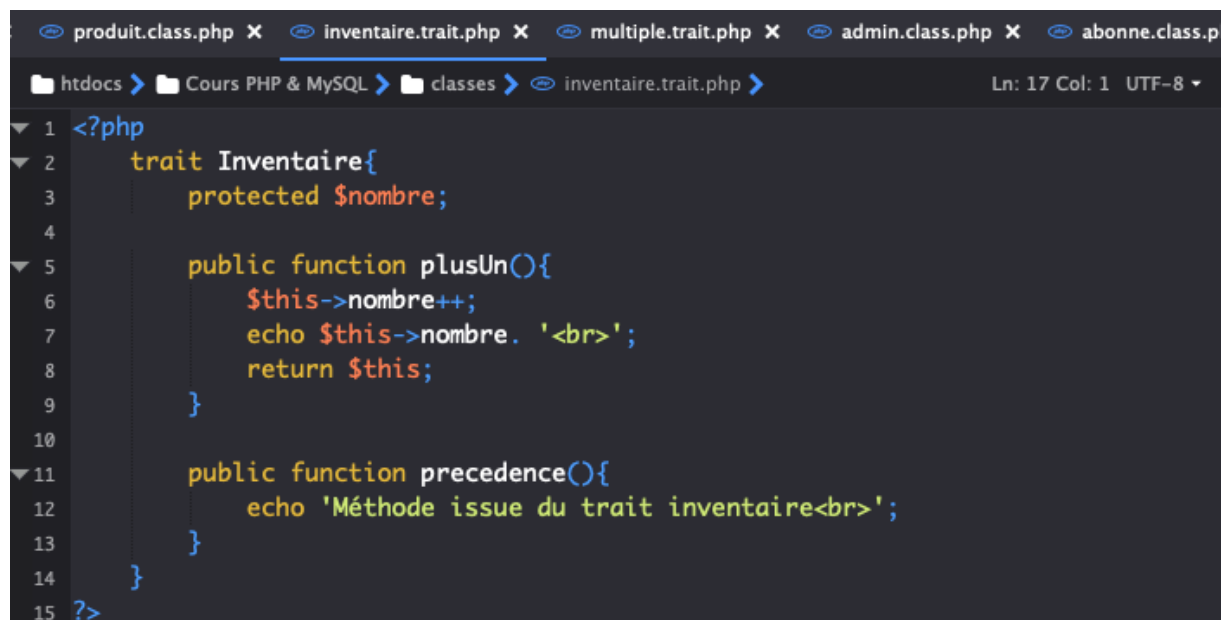
Utiliser l'opérateur `insteadof` nous permet en fait d'exclure les définitions d'une méthode qui ne devront pas être utilisées.

Pour illustrer cela, on peut par exemple créer un nouveau trait qu'on appellera `multiple` et qui va redéfinir la méthode `precedence()`.



```
1 <?php
2 trait Multiple{
3     public function precedence(){
4         echo 'Méthode issue du trait multiple<br>';
5     }
6 }
7 ?>
```

The screenshot shows a code editor with several tabs open: produit.class.php, inventaire.trait.php, multiple.trait.php (active), admin.class.php, and abonne.class.p. The active file is multiple.trait.php, showing a PHP trait named Multiple with a public function precedence() that echoes a message.



```
1 <?php
2 trait Inventaire{
3     protected $nombre;
4
5     public function plusUn(){
6         $this->nombre++;
7         echo $this->nombre. '<br>';
8         return $this;
9     }
10
11     public function precedence(){
12         echo 'Méthode issue du trait inventaire<br>';
13     }
14 }
15 ?>
```

The screenshot shows the same code editor with the tab inventaire.trait.php active. It displays a PHP trait named Inventaire with a protected property \$nombre, a public function plusUn() that increments the property and echoes its value, and a public function precedence() that echoes a message.

On va ensuite inclure nos deux traits dans notre classe `Produit` et utiliser l'opérateur `insteadof` pour définir laquelle des deux définitions de notre méthode doit être utilisée avec la syntaxe suivante :

```

<?php
class Produit{
    use Inventaire, Multiple{
        Multiple::precedence insteadof Inventaire;
    }
    protected $nom;

    public function __construct($n, $nb){
        $this->nom = $n;
        $this->nombre = $nb;
    }

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

    //D'autres méthodes...
}
?>

```

Ici, on déclare qu'on souhaite utiliser la définition de notre méthode `precedence()` du trait `Multiple` plutôt que celle du trait `Inventaire`.

```

<!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
            //On pense bien à inclure nos traits
            require 'classes/inventaire.trait.php';
            require 'classes/multiple.trait.php';
            spl_autoload_register(function($classe){
                require 'classes/' . $classe . '.class.php';
            });

            $yeti = new Produit('Yeti', 10);
            $yeti->precedence();

        ?>
        <p>Un paragraphe</p>
    </body>
</html>

```



Notez que dans le cas (rare) où on souhaite utiliser les différentes définitions de méthodes portant le même nom définies dans différents traits, on peut également utiliser l'opérateur `as` qui permet d'utiliser une autre version d'une méthode de même nom en lui choisissant un nouveau nom temporaire.

Retenez ici bien que l'opérateur `as` ne renomme pas une méthode en soi mais permet simplement d'utiliser un autre nom pour une méthode juste pour le temps d'une inclusion et d'une exécution : le nom d'origine de la méthode n'est pas modifié.

```
<?php
class Produit{
    use Inventaire, Multiple{
        Inventaire::precedence insteadof Multiple;
        Multiple::precedence as prece;
    }
    protected $nom;

    public function __construct($n, $nb){
        $this->nom = $n;
        $this->nombre = $nb;
    }

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

    //D'autres méthodes...
}
?>
```

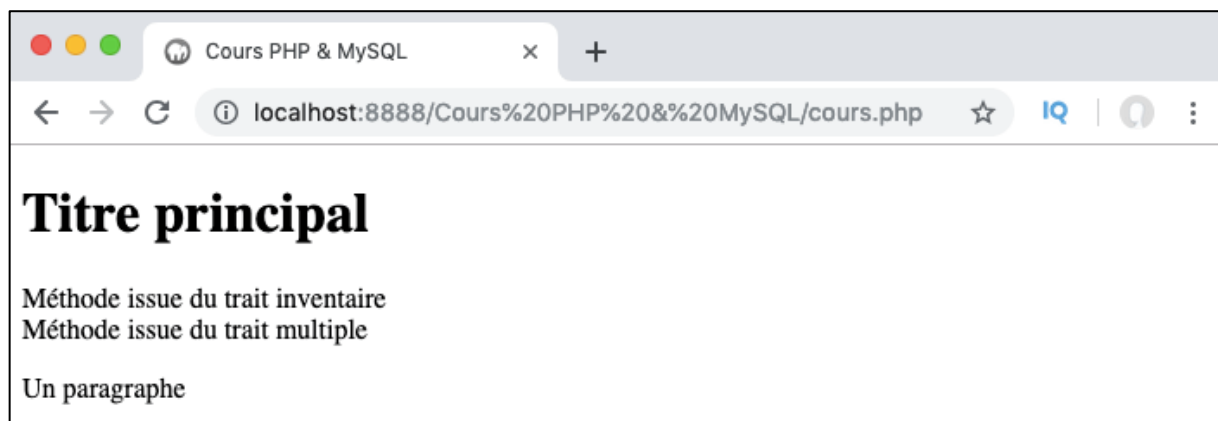
```

<!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
      //On pense bien à inclure nos traits
      require 'classes/inventaire.trait.php';
      require 'classes/multiple.trait.php';
      spl_autoload_register(function($classe){
        require 'classes/' . $classe . '.class.php';
      });

      $yeti = new Produit('Yeti', 10);
      $yeti->precedence();
      $yeti->prece();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on commence par définir quelle définition de `precedence()` doit être utilisée grâce à l'opérateur `insteadof`. Ensuite, on demande à également utiliser la méthode `precedence()` du trait `Multiple` en l'utilisant sous le nom `prece()` pour éviter les conflits.

Une nouvelle fois, il est très rare d'avoir à effectuer ce genre d'opérations mais il reste tout de même bon de les connaître, ne serait-ce que pour les reconnaître dans les codes d'autres développeurs.

Par ailleurs, vous devez savoir qu'on peut également définir une nouvelle visibilité pour une méthode lorsqu'on utilise l'opérateur `as`. Pour cela, il suffit de préciser la nouvelle visibilité juste avant le nom d'emprunt de la méthode.

Les traits composés (héritage de traits)

Vous devez finalement savoir que des traits peuvent eux-mêmes utiliser d'autres traits et hériter de tout ou d'une partie de ceux-ci.

Pour cela, on va à nouveau utiliser le mot clef `use` pour utiliser un trait dans un autre.

On pourrait ainsi par exemple utiliser notre trait `Multiple` dans notre trait `Inventaire` (même si cela ne fait pas beaucoup de sens dans le cas présent) en utilisant la syntaxe suivante :

```
<?php
trait Inventaire{
    use Multiple;
    protected $nombre;

    public function plusUn(){
        $this->nombre++;
        echo $this->nombre. '<br>';
        return $this;
    }

    public function precedence(){
        echo 'Méthode issue du trait inventaire<br>';
    }
}
?>
```

Notez que selon le système utilisé et l'emplacement de vos différents traits, vous devrez peut être spécifier le chemin complet du trait ou utiliser une instruction `require` pour inclure le trait directement dans l'autre avant d'utiliser `use`.

L'interface Iterator et le parcours d'objets

Dans cette nouvelle leçon, nous allons apprendre à parcourir rapidement les propriétés visibles d'un objet en utilisant une boucle `foreach`. Nous allons également découvrir l'interface `Iterator` et implémenter certaines de ses méthodes.

Parcourir un objet en utilisant une boucle foreach

Le PHP nous permet simplement de parcourir un objet afin d'afficher la liste de ses propriétés et leurs valeurs en utilisant une boucle `foreach`.

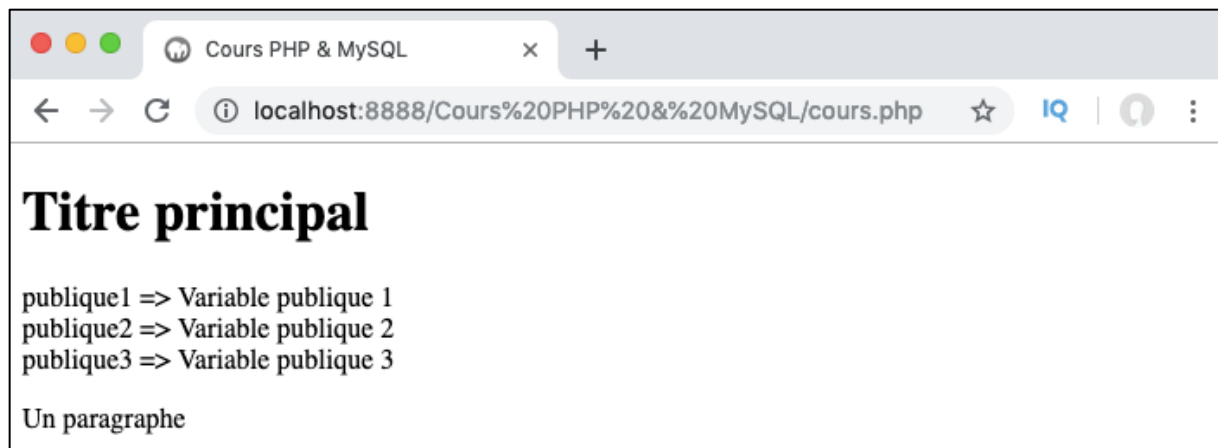
Attention cependant : par défaut, seules les propriétés visibles (publiques) seront lues. Prenons immédiatement un exemple pour illustrer cela.

```
<!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
      //Définition d'une classe
      class Test{
        public $publique1 = 'Variable publique 1';
        public $publique2 = 'Variable publique 2';
        public $publique3 = 'Variable publique 3';

        protected $protegee = 'Variable protégée';
        private $privee = 'Variable privée';
      }
      //Instanciation de la classe
      $test = new Test();

      //Parcours et affichage des propriétés visibles
      foreach ($test as $clef => $valeur){
        echo $clef. ' => ' . $valeur. '<br>';
      }
    ?>
    <p>Un paragraphe</p>
  </body>
</html>
```



Ici, on crée une classe **Test** qui contient cinq propriétés dont une protégée et une privée. Notez qu'ici je place tout le code sur une seule page pour plus de simplicité. Ce n'est cependant pas recommandé en pratique (rappelez-vous : une classe = un fichier).

Ensuite, on instancie la classe et on utilise une boucle **foreach** sur l'objet créé afin d'afficher la liste des propriétés visibles qu'il contient et leurs valeurs associées.

Comme vous pouvez le constater, seules les propriétés publiques sont renvoyées par défaut.

On va pouvoir gérer la façon dont un objet doit être traversé ou parcouru en implémentant une interface **Iterator** qui est une interface prédéfinie en PHP.

L'interface **Iterator** définit des méthodes qu'on va pouvoir implémenter pour itérer des objets en interne. On va ainsi pouvoir passer en revue certaines valeurs de nos objets à des moments choisis.

L'interface **Iterator** possède notamment cinq méthodes qu'il va être intéressant d'implémenter :

- La méthode **current()** ;
- La méthode **next()** ;
- La méthode **rewind()** ;
- La méthode **key()** ;
- La méthode **valid()**.

Une nouvelle fois, rappelez-vous bien ici qu'une interface n'est qu'un plan de base qui spécifie une liste de méthodes que les classes qui implémentent l'interface devront implémenter.

Les interfaces prédéfinies comme **Iterator** ne servent donc qu'à produire un code plus compréhensible et plus structuré (notamment car les autres développeurs vont « reconnaître » l'utilisation d'une interface prédéfinie et donc immédiatement comprendre ce qu'on cherche à faire).

Nous allons donc devoir ici implémenter les méthodes définies dans **Iterator** dans la classe qui implémente cette interface. Bien évidemment, une nouvelle fois, on peut définir n'importe quelle implémentation pour nos méthodes mais dans ce cas utiliser une interface perd tout son sens.

Généralement, on va se baser sur le nom des méthodes pour définir une implémentation cohérente et utile. En effet, vous devez savoir que les fonctions `current()`, `next()`, `rewind()` et `key()` existent toutes déjà en tant que fonctions prédéfinies en PHP. Nous allons donc les utiliser pour définir l'implémentation de nos méthodes.

```

<body>
  <h1>Titre principal</h1>
  <?php
    //Définition d'une classe
    class Test implements Iterator{
      private $tableau = [];

      public function __construct(array $tb){
        $this->tableau = $tb;
      }

      public function rewind(){
        echo 'Retour au début du tableau<br>';
        reset($this->tableau);
      }

      public function current(){
        $tableau = current($this->tableau);
        echo 'Élément actuel : ' . $tableau. '<br>';
        return $tableau;
      }

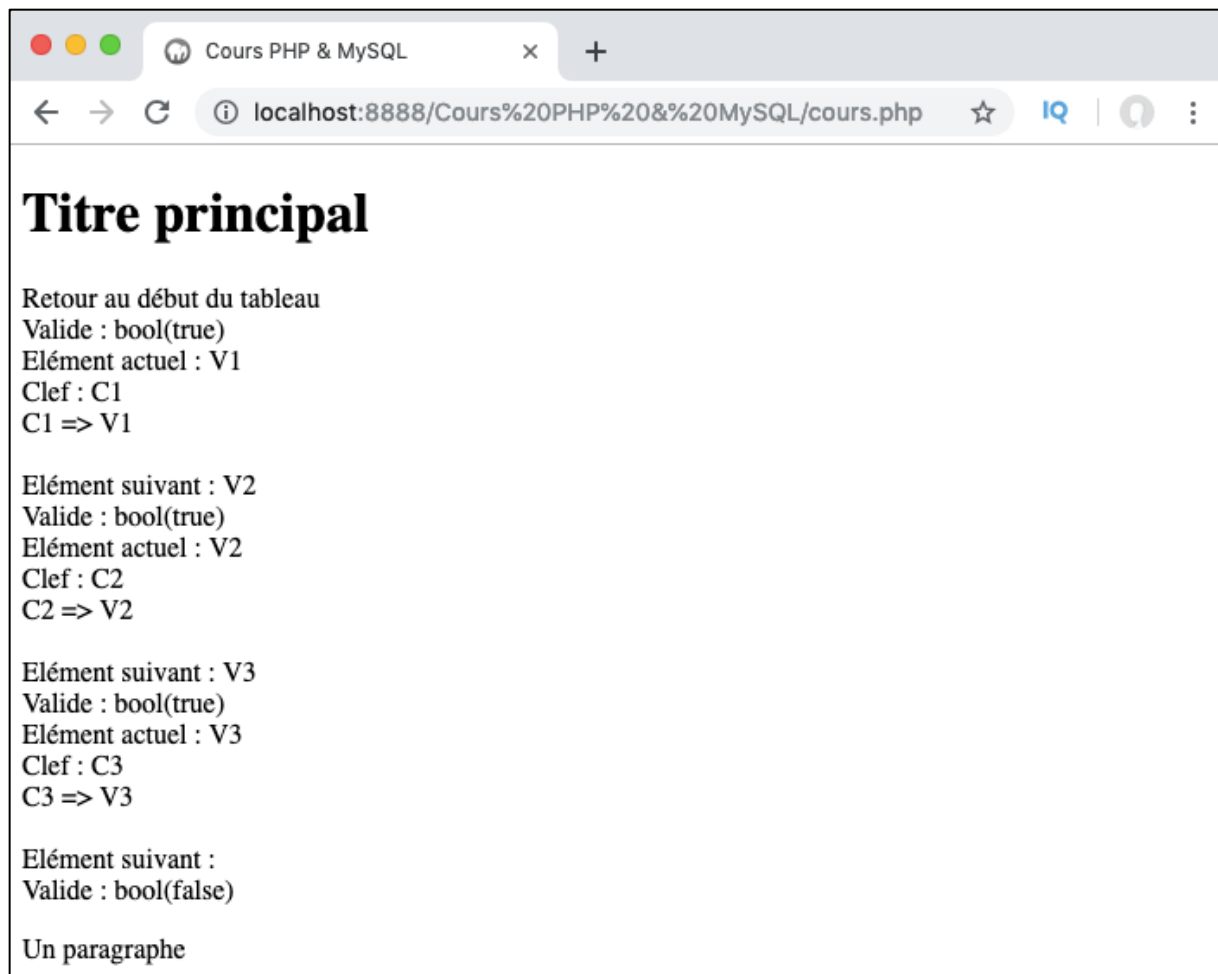
      public function key(){
        $tableau = key($this->tableau);
        echo 'Clef : ' . $tableau. '<br>';
        return $tableau;
      }

      public function next(){
        $tableau = next($this->tableau);
        echo 'Élément suivant : ' . $tableau. '<br>';
        return $tableau;
      }

      public function valid(){
        $clef = key($this->tableau);
        $tableau = ($clef !== NULL && $clef !== FALSE);
        echo 'Valide : ';
        var_dump($tableau);
        echo '<br>';
        return $tableau;
      }
    }

    $tbtest = ['C1' => 'V1', 'C2' => 'V2', 'C3' => 'V3'];
    $objet = new Test($tbtest);
    foreach ($objet as $c => $v){
      echo $c. ' => ' . $v. '<br><br>';
    }
  ?>
  <p>Un paragraphe</p>
</body>

```



Ce code contient de nombreuses choses intéressantes à expliquer et qui devraient vous permettre de mieux comprendre l'orienté objet en PHP en soi.

Tout d'abord, vous devez bien comprendre qu'une méthode de classe est un élément différent d'une fonction en PHP. En effet, la fonction `current()` par exemple est une fonction prédéfinie (ou prête à l'emploi) en PHP et on ne peut donc pas la redéfinir.

En revanche, la méthode `current()` n'est pas prédéfinie et on va donc pouvoir définir son implémentation. Ici, en l'occurrence, on utilise la fonction prédéfinie `current()` pour implémenter la méthode `current()`.

Notre classe `Test` implémente l'interface `Iterator`. Elle possède ici une propriété privée `$tableau` qui est un tableau vide au départ et définit un constructeur qui accepte un tableau en argument et le place dans la propriété privée `$tableau`. Ensuite, la classe se contente d'implémenter les méthodes de l'interface `Iterator`.

Ensuite, je pense qu'il convient d'expliquer ce que font les fonctions prédéfinies `current()`, `next()`, `rewind()` et `key()` pour comprendre le code ci-dessus.

La fonction `reset()` replace le pointeur interne au début du tableau et retourne la valeur du premier élément du tableau (avec une instruction de type `return`).

La fonction `current()` retourne la valeur de l'élément courant du tableau, c'est-à-dire de l'élément actuellement parcouru (l'élément au niveau duquel est situé le pointeur interne du tableau).

La fonction `key()` retourne la clef liée à la valeur de l'élément courant du tableau.

La fonction `next()` avance le pointeur interne d'un tableau d'un élément et retourne la valeur de l'élément au niveau duquel se situe le pointeur.

Une fois notre classe définie, on l'instancie en lui passant un tableau associatif qui va être utilisé comme argument pour notre constructeur.

Finalement, on utilise une boucle `foreach` pour parcourir l'objet créé à partir de notre classe.

Ce qu'il faut alors bien comprendre ici est que le PHP a un comportement bien défini lorsqu'on utilise un objet qui implémente l'interface `Iterator` et notamment lorsqu'on essaie de le parcourir avec une boucle `foreach`.

Notez par ailleurs que la plupart des langages web ont des comportements prédéfinis lorsqu'on utilise des éléments prédéfinis du langage et c'est généralement tout l'intérêt d'utiliser ces éléments.

Expliquons précisément ce qu'il se passe dans le cas présent. Tout d'abord, on sait qu'une interface impose aux classes qui l'implémentent d'implémenter toutes les méthodes de l'interface. Lorsqu'on crée un objet qui implémente l'interface `Iterator`, le PHP sait donc que l'objet va posséder des méthodes `rewind()`, `current()`, `key()`, `next()` et `valid()` et il va donc pouvoir les exécuter selon un ordre prédéfini.

Lorsqu'on utilise une boucle `foreach` avec un objet qui implémente l'interface `Iterator`, le PHP va automatiquement commencer par appeler `Iterator::rewind()` avant le premier passage dans la boucle ce qui va dans notre cas `echo` le texte « Retour au début du tableau » et va placer le pointeur interne du tableau au début de celui-ci.

Ensuite, avant chaque nouveau passage dans la boucle, `Iterator::valid()` est appelée et si `false` est retourné, on sort de la boucle. Dans le cas contraire, `Iterator::current()` et `Iterator::key()` sont appelées.

Finalement, après chaque passage dans la boucle, `Iterator::next()` est appelée et on recommence l'appel aux mêmes méthodes dans le même ordre (excepté pour `rewind()` qui n'est appelée qu'une fois en tout début de boucle).

Passage d'objets : identifiants et références

Dans cette nouvelle leçon, nous allons tenter d'illustrer des notions relativement complexes et abstraites concernant la façon dont les objets sont passés. Comprendre ces choses vous permettra de bien comprendre ce qu'il se passe en PHP orienté objet dans la plupart des cas d'assignation.

Le passage de variables par valeur ou par « référence » (alias)

On a vu plus tôt dans ce cours qu'il existait deux façons de passer une variable (à une fonction par exemple) en PHP : on pouvait soit la passer par valeur (ce qui est le comportement par défaut), soit par référence en utilisant le symbole `&` devant le nom de la variable.

Lorsqu'on parle de « passage par référence » en PHP, on devrait en fait plutôt parler d'alias au sens strict du terme et pour être cohérent par rapport à la plupart des autres langages de programmation.

Une « référence » en PHP ou plus précisément un alias est un moyen d'accéder au contenu d'une même variable en utilisant un autre nom. Pour le dire simplement, créer un alias signifie déclarer un autre nom de variable qui va partager la même valeur que la variable de départ.

Notez qu'en PHP le nom d'une variable et son contenu ou sa valeur sont identifiés comme deux choses distinctes par le langage. Cela permet donc de donner plusieurs noms à un même contenu (c'est-à-dire d'utiliser plusieurs noms pour accéder à un même contenu).

Ainsi, lorsqu'on modifie la valeur de l'alias, on modifie également la valeur de la variable de base puisque ces deux éléments partagent la même valeur.

Au contraire, lorsqu'on passe une variable par valeur (ce qui est le comportement par défaut en PHP), on travaille avec une « copie » de la variable de départ. Les deux copies sont alors indépendantes et lorsqu'on modifie le contenu de la copie, le contenu de la variable d'origine n'est pas modifié.

Regardez plutôt l'exemple ci-dessous pour bien vous en assurer :


```

<!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
      $x = 1;
      $y = $x;
      $z = &$y;
      $y = 2;

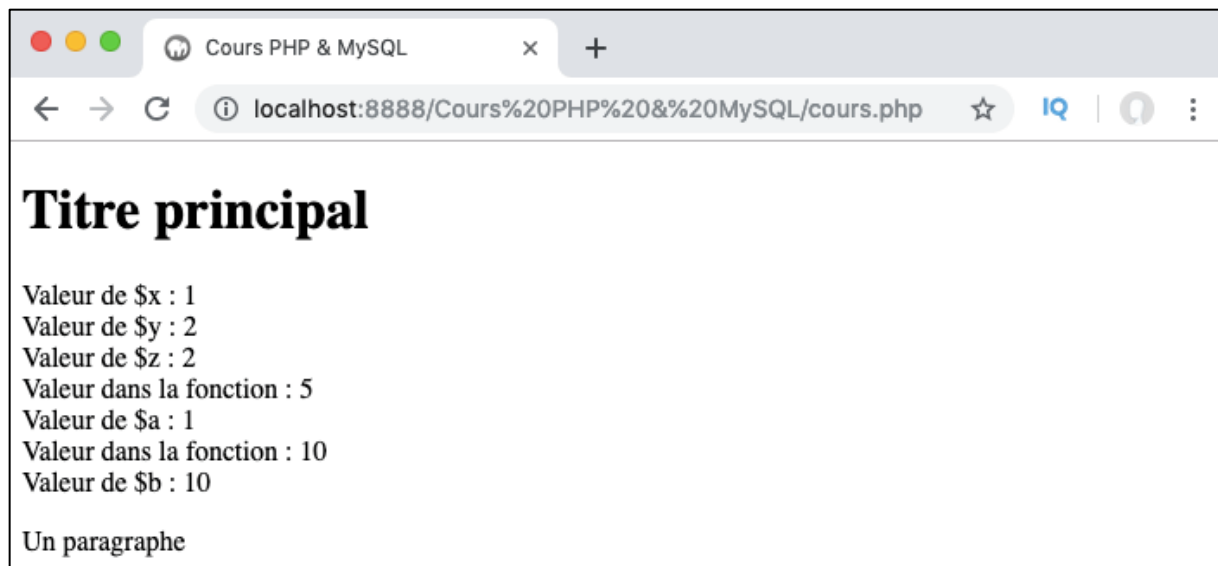
      echo 'Valeur de $x : ' . $x. '<br>'
        . 'Valeur de $y : ' . $y. '<br>'
        . 'Valeur de $z : ' . $z. '<br>';

      $a = 1;
      $b = 2;

      function parValeur($valeur){
        $valeur = 5;
        echo 'Valeur dans la fonction : ' . $valeur. '<br>';
      }
      parValeur($a);
      echo 'Valeur de $a : ' . $a. '<br>';

      function parReference(&$reference){
        $reference = 10;
        echo 'Valeur dans la fonction : ' . $reference. '<br>';
      }
      parReference($b);
      echo 'Valeur de $b : ' . $b. '<br>';
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on commence par déclarer une variable `$x` à laquelle on assigne la valeur `1`.

Ensuite on définit une variable `$y` en lui assignant le contenu de `$x`. Par défaut, le passage se fait par valeur ce qui signifie qu'une copie de `$x` est créée. Si on manipule ensuite la copie (c'est-à-dire `$y`), le contenu de la variable de base `$x` n'est pas modifié puisqu'on a bien deux éléments indépendants ici stockant chacun une valeur bien différenciée.

Finalement, on définit une troisième variable `$z` en lui assignant le contenu de `$y` mais cette fois-ci on passe le contenu par référence avec le signe `&`. Notre variable `$z` est donc ici un alias de `$y`, ce qui signifie que `$z` et `$y` sont deux noms utilisés pour faire référence (= pour accéder ou pour manipuler) à la même valeur.

Nos deux variables `$y` et `$z` partagent donc ici la même valeur et lorsqu'on change la valeur assignée à l'une cela modifie forcément le contenu assigné à la seconde puisqu'encore une fois ces deux variables partagent la même valeur.

Ici, il faut bien comprendre une nouvelle fois qu'en PHP le nom d'une variable et son contenu sont deux éléments clairement identifiés. En fait, lorsqu'on assigne une valeur à un nom, on indique simplement au PHP qu'à partir de maintenant on va utiliser ce nom pour accéder à la valeur assignée.

Il se passe exactement la même chose lors du passage des arguments d'une fonction. Comme vous pouvez le voir, on passe l'argument dans notre fonction `parValeur()` par valeur. Lorsqu'on modifie la valeur de l'argument à l'intérieur de la fonction, la valeur de la variable externe n'est pas impactée puisque ces deux éléments sont bien différents.

En revanche, on passe l'argument de la fonction `parReference()` par référence. Ainsi, on crée un alias qui va servir de référence vers la même valeur que la variable qu'on va passer en argument à la fonction. Lorsqu'on modifie la valeur à l'intérieur de la fonction, on modifie donc également ce que stocke la variable à l'extérieur de la fonction.

Le passage des objets en PHP

Lorsqu'on crée une nouvelle instance de classe en PHP et qu'on assigne le résultat dans une variable, vous devez savoir qu'on n'assigne pas véritablement l'objet en soi à notre variable objet mais simplement un identifiant d'objet qu'on appelle également parfois un « pointeur ».

Cet identifiant va être utilisé pour accéder à l'objet en soi. Pour l'expliquer en d'autres termes, vous pouvez considérer que cet identifiant d'objet est à l'objet ce que le nom d'une variable est à la valeur qui lui est assignée.

Notre variable objet créée stocke donc un identifiant d'objet qui permet lui-même d'accéder aux propriétés de l'objet. Pour accéder à l'objet via son identifiant, on va utiliser l'opérateur `->` qu'on connaît bien.

Ainsi, lorsqu'on passe une variable objet en argument d'une fonction, ou lorsqu'on demande à une fonction de retourner une variable objet, ou encore lorsqu'on assigne une variable objet à une autre variable objet, ce sont des copies de l'identifiant pointant vers le même objet qui sont passées.

Comme les copies de l'identifiant pointent toujours vers le même objet, on a tendance à dire que « les objets sont passés par référence ». Ce n'est cependant pas strictement vrai : encore une fois, ce sont des identifiants d'objets pointant vers le même objet qui vont être passés par valeur.

Regardez plutôt l'exemple suivant :

```

<!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
      class Utilisateur{
        protected $user_name;

        public function __construct($n){
          $this->user_name = $n;
        }
        public function getNom(){
          echo $this->user_name;
        }
        public function setNom($nom){
          return $this->user_name = $nom;
        }
      };

      $pierre = new Utilisateur('Pierre');
      $victor = $pierre;
      $victor->setNom('Victor');
      $pierre->getNom();
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on crée une classe `Utilisateur` qu'on instancie une première fois. On assigne un identifiant d'objet à la variable objet `$pierre`.

On définit ensuite une deuxième variable `$victor` en lui assignant le contenu de `$pierre`. Notre variable va donc devenir de fait une variable objet et va stocker une copie de l'identifiant pointant vers le même objet que `$pierre`.

C'est la raison pour laquelle lorsqu'on accède à l'objet via `$victor->` pour modifier la valeur de la propriété `$user_name` de l'objet, la valeur de `$user_name` de `$pierre` est également modifiée.

En effet, `$pierre` et `$victor` contiennent deux copies d'identifiant permettant d'accéder au même objet. C'est la raison pour laquelle le résultat ici peut faire penser que nos objets ont été passés par référence.

Ce n'est toutefois pas le cas, ce sont des copies d'identifiant pointant vers le même objet qui sont passées par valeur. Pour passer un identifiant d'objet par référence, nous allons une nouvelle fois devoir utiliser le signe `&`.

Regardez le nouvel exemple ci-dessous pour bien comprendre la différence entre un passage par référence et un passage par valeur via un identifiant.

```

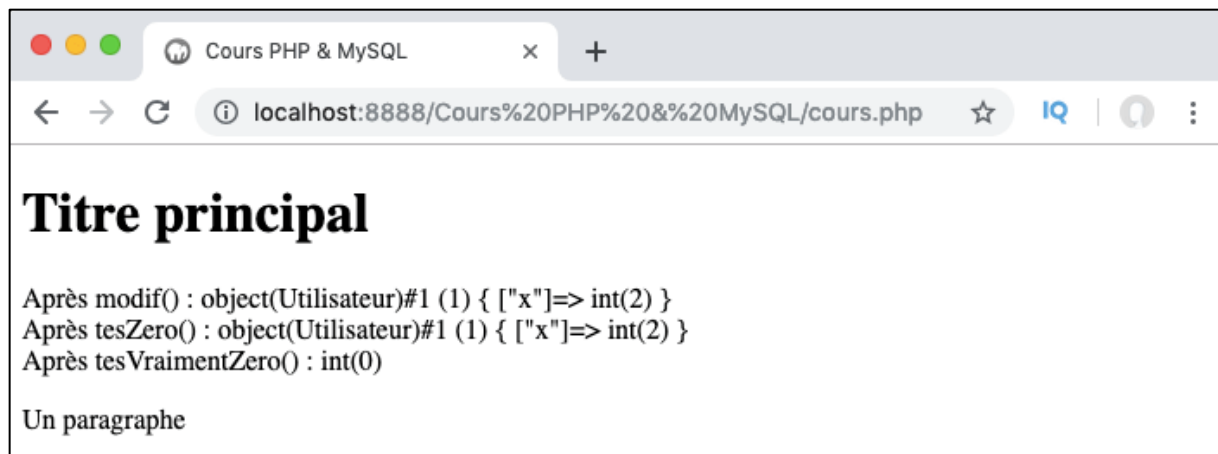
<!DOCTYPE html>
<html>
  <head>
    <title>Cours PHP & MySQL</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
  </head>

  <body>
    <h1>Titre principal</h1>
    <?php
      class Utilisateur{
        public $x = 1;

        public function modif(){
          $this->x = 2;
        }
      }
      function tesZero($obj){
        $obj = 0;
      }
      function tesVraimentZero(&$obj){
        $obj = 0;
      }

      $pierre = new Utilisateur();
      $pierre->modif();
      echo 'Après modif() : ' ;
      var_dump($pierre);
      tesZero($pierre);
      echo '<br>Après tesZero() : ' ;
      var_dump($pierre);
      tesVraimentZero($pierre);
      echo '<br>Après tesVraimentZero() : ' ;
      var_dump($pierre);
    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on définit une classe qui contient une propriété et une méthode publiques et on instancie notre classe puis on assigne l'identifiant d'objet à notre variable objet `$pierre`.

On définit également deux fonctions en dehors de notre classe.

On appelle ensuite notre méthode `modif()` donc le rôle est de modifier la valeur de la propriété `$x` de l'objet courant puis on affiche les informations relatives à notre objet grâce à `var_dump()`. On constate que notre propriété `$x` stocke bien la valeur 2.

Ensuite, on utilise notre fonction `tesZero()` en lui passant `$pierre` en argument. Le rôle de cette fonction est d'assigner la valeur 0 à la variable passée en argument. Pourtant, lorsqu'on `var_dump()` à nouveau `$pierre`, on s'aperçoit que le même objet que précédemment est renvoyé.

Cela est dû au fait qu'ici notre fonction `tesZero()` n'a modifié que l'identifiant d'objet et non pas l'objet en soi.

Notre fonction `tesVraimentZero()` utilise elle le passage par référence. Dans ce cas-là, c'est bien une référence à l'objet qui va être passée et on va donc bien pouvoir écraser l'objet cette fois-ci.

Je tiens ici à préciser que ces notions sont des notions abstraites et complexes et qu'il faut généralement beaucoup de pratique et une très bonne connaissance au préalable du langage pour bien les comprendre et surtout comprendre leurs implications. J'essaie ici de vous les présenter de la manière la plus simple possible, mais ne vous inquiétez pas si certaines choses vous échappent pour le moment : c'est tout à fait normal, car il faut du temps et du recul pour maîtriser parfaitement un langage.

Le clonage d'objets

Dans la leçon précédente, on a vu que nos variables objets stockaient en fait des identifiants d'objets servant à accéder aux objets.

Lorsqu'on instancie une fois une classe, on crée un objet et on stocke généralement un identifiant d'objet dans une variable qu'on appelle une variable objet ou un objet par simplification.

Si on assigne le contenu de notre variable objet dans une nouvelle variable, on ne va créer qu'une copie de l'identifiant qui va continuer à pointer vers le même objet.

Cependant, dans certains cas, on voudra plutôt créer une copie d'un objet en soi. C'est exactement ce que va nous permettre de réaliser le clonage d'objet que nous allons étudier dans cette nouvelle leçon.

Les principes du clonage d'objets

Parfois, on voudra « copier » un objet afin de manipuler une copie indépendante plutôt que l'objet original.

Dans ces cas-là, on va « cloner » l'objet. Pour cela, on va utiliser le mot clef `clone` qui va faire appel à la méthode magique `__clone()` de l'objet si celle-ci a été définie. Notez qu'on ne peut pas directement appeler la méthode `__clone()`.

Lorsque l'on clone un objet, le PHP va en fait réaliser une copie « superficielle » de toutes les propriétés relatives à l'objet, ce qui signifie que les propriétés qui sont des références à d'autres variables (objets) demeureront des références.

Dès que le clonage d'objet a été effectué, la méthode `__clone()` du nouvel objet (le clone) va être automatiquement appelée. Cela va généralement nous permettre de mettre à jour les propriétés souhaitées.

Exemple de clonage d'objets

Pour cloner un objet en pratique nous allons simplement devoir utiliser le mot clef `clone` et éventuellement pouvoir définir une méthode `__clone()` qui va nous permettre de mettre à jour les éléments du clone par rapport à l'original.


```

<!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
      class Utilisateur{
        protected $nom;

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

        public function __clone(){
          $this->nom = $this->nom. ' (clone)';
        }
        public function getNom(){
          echo $this->nom;
        }
      }

      $pierre = new Utilisateur('Pierre');
      $victor = clone $pierre;

      var_dump($pierre);
      echo '<br>';
      var_dump($victor);
      echo '<br>';
      $pierre->getNom();
      echo '<br>';
      $victor->getNom();

    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



On crée ici une classe qu'on appelle à `Utilisateur`. Cette classe possède une propriété `$nom`, un constructeur dont le rôle est d'assigner une valeur à `$nom` pour l'objet courant, une méthode `getNom()` qui sert à renvoyer la valeur de `$nom` de l'objet courant et finalement une méthode `__clone()`.

Le rôle de la méthode `__clone()` est ici de modifier la valeur stockée dans `$nom` pour le clone en lui ajoutant « (clone) ».

On instancie ensuite notre classe et on assigne le résultat à une variable objet `$pierre` qu'on va cloner grâce au mot clef `clone`.

Lors du clonage, notre clone `$victor` va recevoir une copie superficielle des propriétés et des méthodes de l'objet de départ. Dès que le clonage est terminé, la méthode `__clone()` est appelée et va ici mettre à jour la valeur de la propriété `$nom` pour notre clone.

On voit bien ici qu'on a créé une copie indépendante de l'objet de départ (et donc une nouvelle instance de la classe) et qu'on ne s'est pas contenté de créer une copie d'un identifiant pointant vers le même objet dès qu'on affiche le contenu de nos deux objets et qu'on tente d'accéder à la valeur de leur propriété `$nom`.

Comparer des objets

Plus tôt dans ce cours nous avons appris à comparer différentes variables qui stockaient des valeurs simples (une chaîne de caractères, un chiffre, un booléen, etc.).

On va également pouvoir comparer des variables objets de manière similaire, ce qui va pouvoir être très utile pour s'assurer par exemple qu'un objet est unique.

Principe de la comparaison d'objets

Pour comparer deux variables objets entre elles, nous allons à nouveau devoir utiliser des opérateurs de comparaison. Cependant, étant donné que les valeurs comparées vont cette fois-ci être des valeurs complexes (car un objet est composé de diverses propriétés et méthodes), nous n'allons pas pouvoir utiliser ces opérateurs de comparaison aussi librement que lors de la comparaison de valeurs simples.

La première chose à savoir est qu'on ne va pouvoir tester que l'égalité (en valeur ou en identité) entre les objets. En effet, cela n'aurait aucun sens de demander au PHP si un objet est « inférieur » ou « supérieur » à un autre puisqu'un objet regroupe un ensemble de propriétés et de méthodes.

En utilisant l'opérateur de comparaison simple `==`, les objets vont être considérés comme égaux s'ils possèdent les mêmes attributs et valeurs (valeurs qui vont être comparées à nouveau avec `==` et si ce sont des instances de la même classe).

En utilisant l'opérateur d'identité `===`, en revanche, les objets ne vont être considérés comme égaux que s'ils font référence à la même instance de la même classe.

Comparer des objets en pratique

Pour illustrer la façon dont le PHP va comparer différents objets et pour commenter les résultats renvoyés, nous allons nous baser sur l'exemple suivant :

```

<!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>

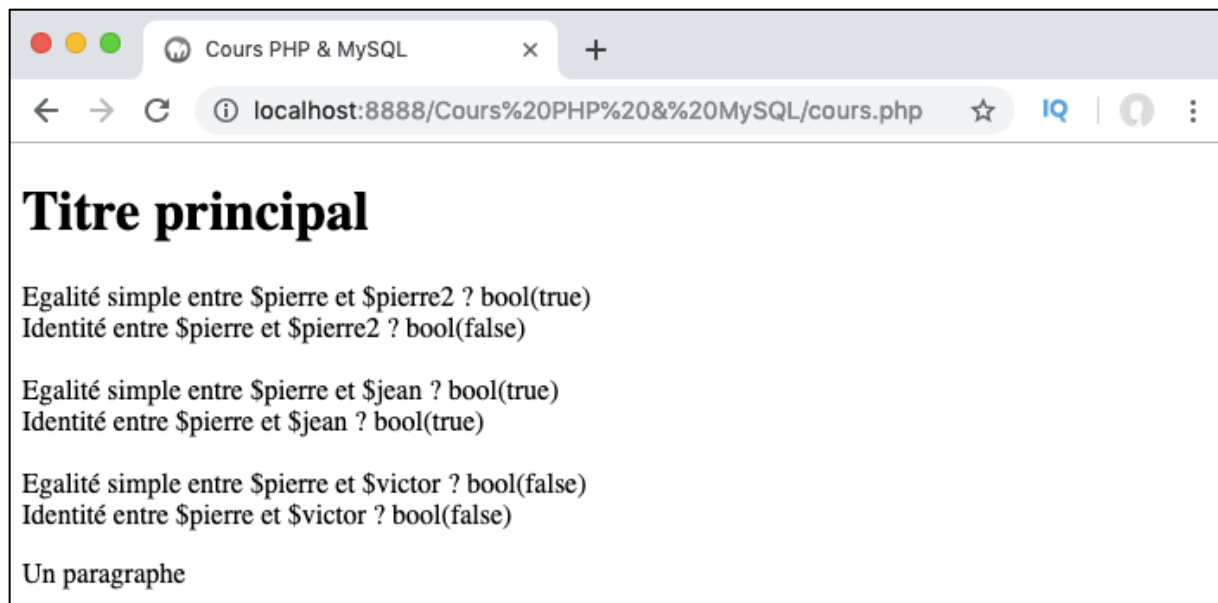
  <body>
    <h1>Titre principal</h1>
    <?php
      class Utilisateur{
        protected $nom;
        public function __construct($n){
          $this->nom = $n;
        }
        public function __clone(){
          $this->nom = $this->nom. ' (clone)';
        }
        public function getNom(){
          echo $this->nom;
        }
      }

      $pierre = new Utilisateur('Pierre');
      $pierre2 = new Utilisateur('Pierre');
      $jean = $pierre;
      $victor = clone $pierre;

      echo 'Egalité simple entre $pierre et $pierre2 ? ';
      var_dump($pierre == $pierre2);
      echo '<br>Identité entre $pierre et $pierre2 ? ';
      var_dump($pierre === $pierre2);
      echo '<br><br>Egalité simple entre $pierre et $jean ? ';
      var_dump($pierre == $jean);
      echo '<br>Identité entre $pierre et $jean ? ';
      var_dump($pierre === $jean);
      echo '<br><br>Egalité simple entre $pierre et $victor ? ';
      var_dump($pierre == $victor);
      echo '<br>Identité entre $pierre et $victor ? ';
      var_dump($pierre === $victor);

    ?>
    <p>Un paragraphe</p>
  </body>
</html>

```



Ici, on réutilise notre classe `Utilisateur` créée précédemment. Cette classe définit une propriété `$nom` ainsi qu'un constructeur qui va initialiser notre propriété, une méthode `getNom()` dont le rôle est de renvoyer la valeur de `$nom` de l'objet courant et une méthode `__clone()` qui va mettre à jour la valeur de la propriété `$nom` d'un clone.

On instancie ensuite deux fois notre classe `Utilisateur` et on stocke le résultat dans deux variables objet `$pierre` et `$pierre2`.

En dessous, on assigne le contenu de `$pierre` dans une nouvelle variable objet `$jean`. Je vous rappelle ici que nos deux objets contiennent deux copies d'un identifiant pointant vers le même objet (la même instance de la classe).

Finalement, on crée un clone de `$pierre` grâce au mot clef `clone` qu'on appelle `$victor`. Ici, une véritable copie de l'objet est créée et donc nos deux variable objets vont bien représenter deux instances différentes et indépendantes de la classe. Dès que le clonage est terminé, notre méthode `__clone()` est appelée et la valeur de `$nom` de `$victor` est mise à jour.

L'idée va alors ici être de comparer nos différents objets. Pour cela, on utilise les opérateurs de comparaison `==` et `===` qui renvoient 1 si la comparaison réussit ou 0 si la comparaison échoue. On passe la valeur renvoyée à la fonction `var_dump()` afin qu'elle nous renvoie le type de résultat (la valeur 1 correspond au booléen `true` tandis que 0 correspond à `false`).

La première comparaison simple entre `$pierre` et `$pierre2` réussit. En effet, nos deux objets sont issus de la même classe et leur propriété `$nom` contient la même valeur puisqu'on a passé « Pierre » au constructeur dans les deux cas.

En revanche, la comparaison en termes d'identité échoue. La raison est que nos deux objets ont été créés en instanciant la classe à chaque fois et représentent donc deux instances différentes de la classe.

Cela n'est pas le cas pour notre objet `$jean` qui contient la copie d'un identifiant pointant vers le même objet (la même instance de la classe) que `$pierre` et qui va donc pouvoir être comparé à `$pierre` en termes d'identité avec succès.

Notre dernier objet, `$victor`, est un clone (ou une copie) de `$pierre` et stocke donc une instance différente de `$pierre`. De plus, juste après que le clonage ait été terminé, la valeur de `$nom` du clone a été mise à jour. La comparaison entre `$victor` et `$pierre` échoue donc à la fois en termes d'identité et en termes de comparaison simple.