

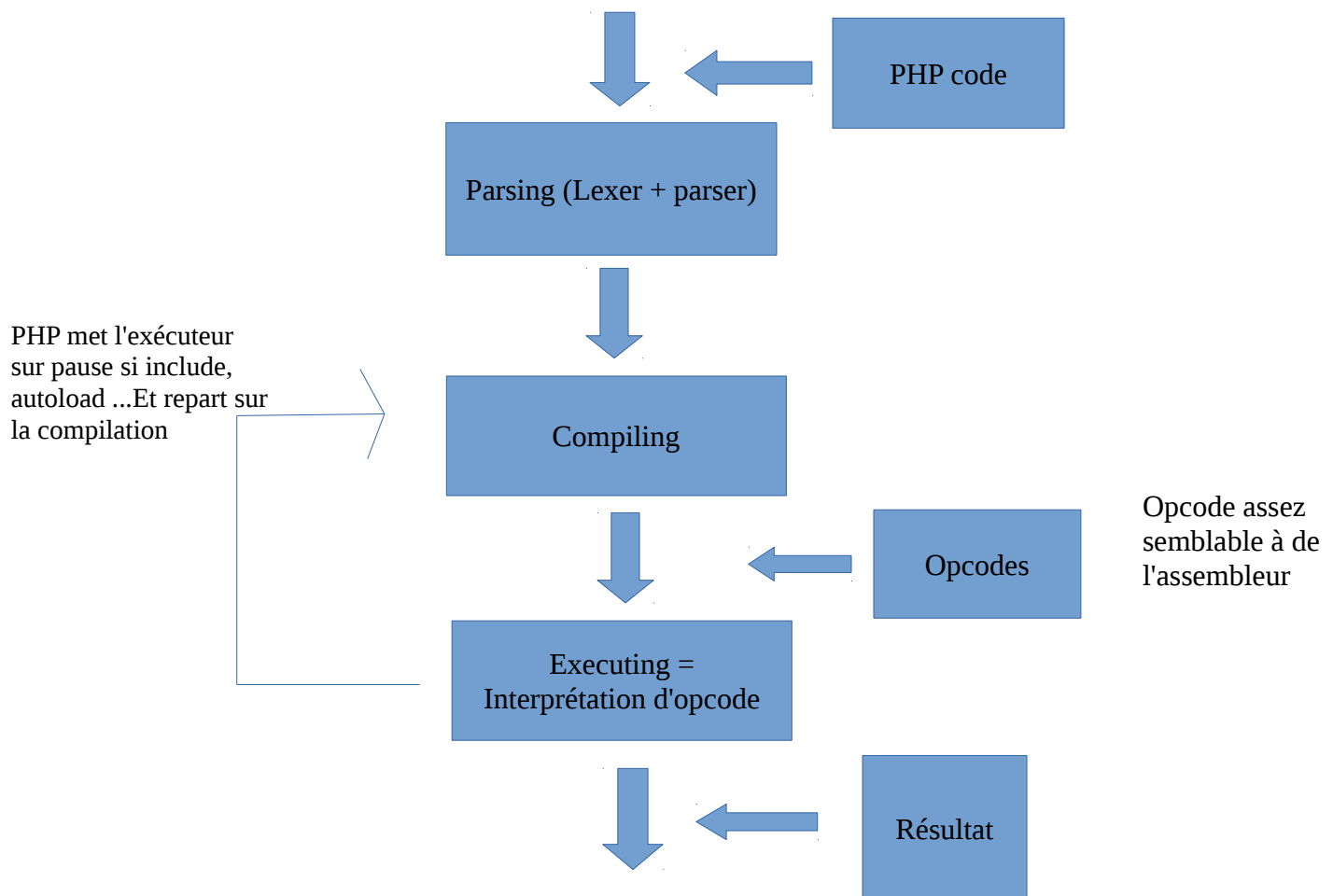
PHP

Antoine Lucsko

1 Introduction

PHP n'est pas un langage orienté objet, mais il permet d'orienter son code en objets.

1.1 Fonctionnement de PHP



La première chose que PHP fait c'est de parser le code (analyse de la syntaxe), il essaye de comprendre ce que l'on a écrit, il vérifie qu'il n'y a pas de **parser error**, si c'est le cas il arrête tout et **retourne un message d'erreur**.

PHP compile le code si aucune erreur n'est levée, dans un code intermédiaire que l'on appelle l'opcodes.

L'opcode sera alors exécuté par la machine virtuelle de PHP pour produire un résultat.

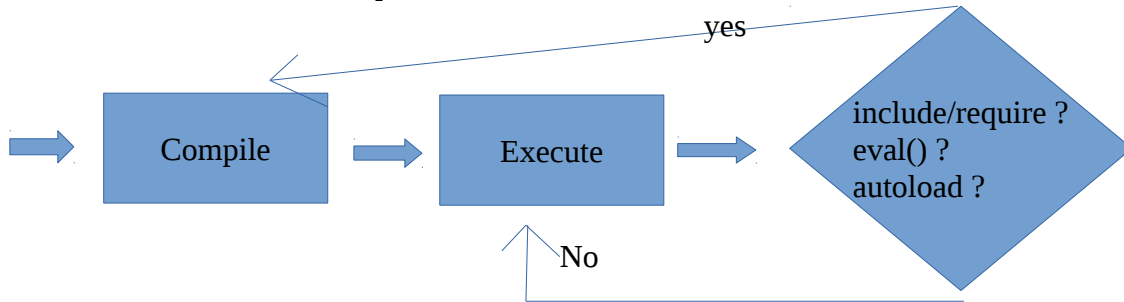
Def du machine virtuelle, c'est un logiciel qui exécute des instructions produisant des résultats identiques quelque soit la plat forme matériel sous-jacente. L'opcode exécuté par cette machine virtuel produira sous Windows ou Linux, par exemple, des résultats identiques.

Remarque importante. PHP compile le code puis exécute l'opcodes et vide entièrement sa mémoire une fois qu'il a terminé.

Compile, execute, forget, Compile, execute, forget, Compile, execute, forget, ...

Comment marche la compilation, une exécution peut déclencher une compilation, si on a dans le

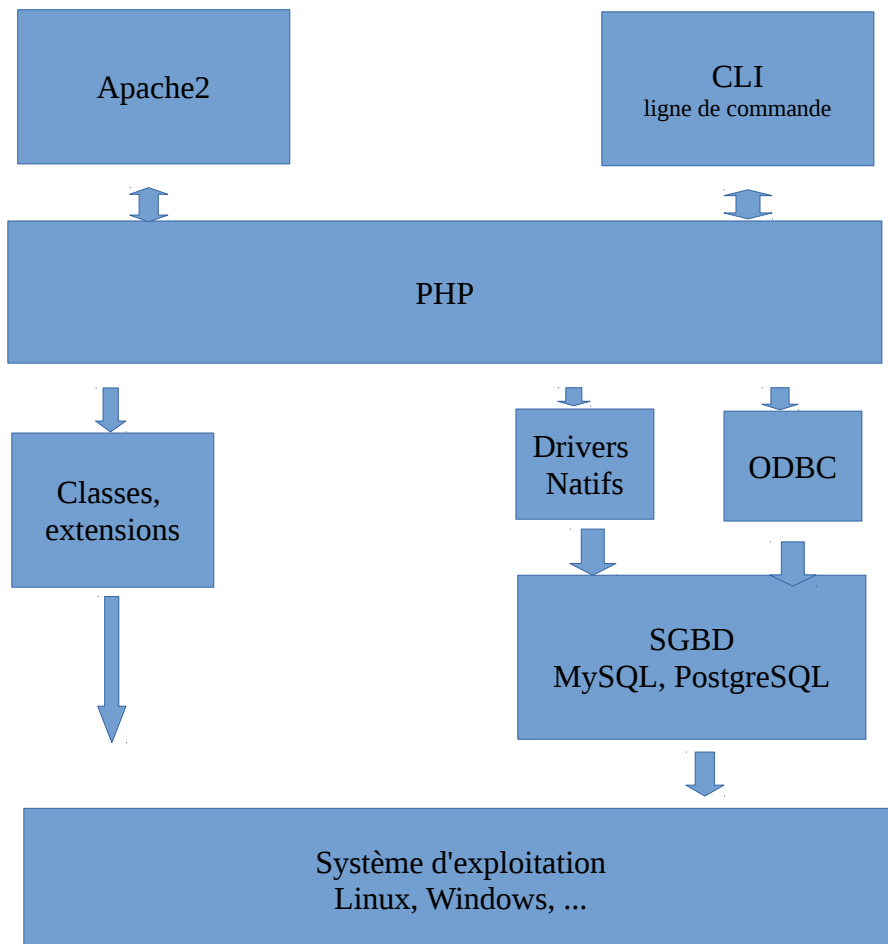
script des includes, require, autoload, ... Un code qui s'exécute peut donc s'arrêter pour compiler du code include, ... Puis reprendre son exécution.



Quelques remarques sur l'installation de PHP

L'intégration de PHP dans le serveur Apache peut se faire de deux manières différentes :


- En tant que **module Apache**, **PHP est directement intégré dans le serveur Web** => **le code PHP sera directement exécuté par le même processus que celui du serveur Web.**
- Via l'interface CGI (common Gateway Interface), c'est un standard qui indique comment transmettre la requête du serveur HTTP au programme et comment récupérer la réponse. CGI est indépendant du langage de programme, on peut avoir Python, PHP, ... Mais cette technique nécessite pour chaque requête un lancement d'un nouveau processus. Cependant, il existe une évolution de CGI : **FastCGI**, elle permet de ne lancer le programme CGI qu'une seule fois, et pas à chaque requête. Cette technique nécessite pour chaque langage concerné une bibliothèque supplémentaire installée, pour PHP c'est **PHP-FPM**. Dans un premier temps sur les machines on a PHP en module Apache, pour Wamp et MAMP.




Pour savoir si son installation Apache2 + PHP est en CGI ou en module Apache2, on utilise la fonction `phpinfo()` :

```
$ echo "<?php phpinfo();" > info.php
```

On exécute ce script sur son serveur :

PHP Version 5.5.16-1+deb.sury.org~trusty+1		
System	Linux antoine-VPCS12J1E 3.13.0-32-generic #57-Ubuntu SMP Tue Jul 15 03:51:08 UTC 2014 x86_64	
Build Date	Aug 25 2014 10:24:16	
Server API	Apache 2.0 Handler	
Virtual Directory Support	disabled	
Configuration File (php.ini) Path	/etc/php5/apache2	
Loaded Configuration File	/etc/php5/apache2/php.ini	
Scan this dir for additional .ini files	/etc/php5/apache2/conf.d	
Additional .ini files parsed	/etc/php5/apache2/conf.d/05-opcache.ini, /etc/php5/apache2/conf.d/10-pdo.ini, /etc/php5/apache2/conf.d/20-curl.ini, /etc/php5/apache2/conf.d/20-gd.ini, /etc/php5/apache2/conf.d/20-json.ini, /etc/php5/apache2/conf.d/20-mcrypt.ini,	

Si PHP était installé en tant que CGI, ici FastCGI, on aurait pour la section **Server API** :

PHP Version 5.6.0-1+deb.sury.org~trusty+1		
System	Linux vps99142 2.6.32-042stab092.3 #1 SMP Sun Jul 20 13:27:24 MSK 2014 x86_64	
Build Date	Aug 28 2014 15:00:02	
Server API	FPM/FastCGI	
Virtual Directory Support	disabled	
Configuration File (php.ini) Path	/etc/php5/fpm	
Loaded Configuration File	/etc/php5/fpm/php.ini	
Scan this dir for additional .ini files	/etc/php5/fpm/conf.d	
Additional .ini files parsed	/etc/php5/fpm/conf.d/05-opcache.ini, /etc/php5/fpm/conf.d/10-pdo.ini, /etc/php5/fpm/conf.d/20-curl.ini, /etc/php5/fpm/conf.d/20-json.ini, /etc/php5/fpm/conf.d/20-mcrypt.ini, /etc/php5/fpm/conf.d/20-readline.ini	
PHP API	20131106	

1.2 Un mot sur le fichier `php.ini`

PHP cherche un fichier nommé `php.ini`, si PHP est installé en tant que module apache ce fichier se

trouve dans le dossier php5/apache2/, en général sous Linux par exemple on le trouve dans /etc/php5/apache2/php.ini. On retiendra que PHP cherche le fichier php.ini en fonction de la SAPI ,Server Application Programming Interface, (CGI, FastCGI, CLI, ...). Il faut donc savoir où se trouve ce fichier.

Pour MAMP il se trouve a priori dans Applications/MAMP/config/php5/php.ini.

Pour WAMP on le trouvera dans wamp\bin\apache\apache2.2.21\bin\php.ini et pour le mode CLI (ligne de commande) dans wamp\bin\php\php5.6\php.ini

On va ouvrir le fichier php.ini concernant le PHP installé en tant que module Apache2, à l'aide d'un éditeur de code comme sublime par exemple. Et on vérifie les points suivants :

max_execution_time=30, pour limiter le temps d'exécution d'un script, c'est une protection contre les boucles infinies.

max_input_time=60, pour limiter la durée maximale pour recevoir les données d'entrée, via POST, GET, et FILES.

memory_limit=64M, limite la mémoire vive allouée par PHP et par ses extensions (gestion des images par exemple)

remarque

Pour connaître la mémoire utiliser par PHP on peut utiliser dans son script la fonction

```
<?php
```

```
memory_get_usage(true) ;
```

include_path=". :/var/www/lib:/usr/lib", cette ligne indique les dossiers dans lesquels recherchés pour inclure des fichiers dans les scripts PHP

Remarque lorsqu'on précise de manière intelligente le chemin des fichiers à inclure, c'est-à-dire en chemin absolu, PHP ne fait aucun calcul pour inclure le dit fichier :

```
<?php
```

```
require __DIR__.'./monScript.php' ;
```

session.auto_start=off, off ici indique que les sessions ne doivent pas être automatiquement lancées au début de chaque script.

display_error=On, indique à PHP d'afficher les erreurs PHP, il faut lorsqu'on est en phase de développement mettre cette directive à On, et Off en prod

error_reporting=E_ALL , en dev on mettra ce niveau de report d'erreurs

Pour la journalisation des erreurs en PHP, on y reviendra, mais les logs sont importants à analyser en phase de développement si on a des problèmes.

```
log_errors=on
```

```
error_log=/var/log/php-error.log
```

2 PHP Objet

2.1 Historique

PHP est un langage web interprété créé en 1998 par Rasmus Lerdorf, sa syntaxe est inspirée du C, son modèle objet est inspiré de Java. Il est écrit en C.

PHP est extensible (on peut créer des fonctions les compilés et les ajouté au langage), simple efficace et massivement déployé (Facebook, Yahoo, ...)

L'objet existe depuis la version 3 de PHP, à ce moment on n'a pas vraiment un langage de programmation objet.

Ce n'est qu'à partir de la version 5 que PHP introduit les véritables concepts de l'objet.

Bien que PHP soit un langage de script on va programmer tout en objet dans la suite de ce cours.

2.2 Introduction à l'objet

2.2.1 Organisation des dossiers et fichier pour le cours

```
PHPObj  
  chap1/  
    example.php  
    exercices/  
    ...
```

On rappellera l'organisation des dossiers et fichiers dans la suite.

2.2.2 Définitions

Def Classe

Une classe est la somme des propriétés et attributs d'un objet. C'est une représentation abstraite d'un objet.

Def Objet

Un objet est une instance d'une classe.

Def Attributs et méthodes d'une classe

Les attributs d'une classe sont les variables d'une classe et les méthodes sont les fonctions de la classe. Plus généralement on appelle membre d'une classe les attributs ou méthodes d'une classe.

Exemples

Ci-dessous une représentation abstraite d'un personnage, la classe « Persona » n'est pas, par définition, concrète. La classe représente l'implémentation des attributs (variables de la classe) et méthodes (fonctions de la classe), le code que l'on écrit dans la classe. Pour rendre « concrète » son utilisation, on fera une **instance** de la classe Persona.

```
<?php
// représentation abstraite
class Persona{

    private $force ;
    private $secret="my secret" ;

    public function fight()
    {
        echo "missile";
    }
}
```

// Un objet est une instance de classe, c'est une variable dans le script courant.

```
$mario = new Persona() ;
```

//On peut créer à partir de la classe autant d'objets que l'on souhaite, chaque instance est une nouvelle variable dans le script courant :

```
$luigi = new Persona() ;
```

```
$peach = new Persona() ;
```

Pour accéder aux membres d'une classe on utilise le symbole « -> » :

```
<?php
$mario->fight() ;
```

2.2.3 Visibilité d'un attribut ou d'une méthode

Si un membre de la classe est privé il est impossible d'y accéder à l'extérieur de la classe, c'est-à-dire à partir de l'objet dans le script courant.

```
<?php
echo $peach->secret ;
// Fatal error: Cannot access private property Persona::$secret
```

Si on souhaite afficher la valeur de l'attribut **\$secret** de la classe Persona, dans le script courant, il faut modifier sa visibilité et la passer en public :

```
<?php
...
public $secret="my secret" ;
...
```

// dans le script :

```
echo $peach->secret ;
```

Pour la méthode fight pas de problème puisqu'elle est publique on peut donc l'appeler directement dans le script courant :

```
<?php
...
```

```
$peach->fight();
```

2.2.4 Méta-variable \$this

Pour qu'une méthode puisse manipuler une variable de classe à l'intérieur de la classe elle-même, elle utilise la méta-variable `$this`. C'est une référence à une instance unique de la classe dans le script.

```
<?php
class Persona{

    private $force ;
    private $secret="my secret" ;

    ...
    public function getSecret()
    {
        return $this->secret ;
    }
}

$toad = new Persona() ;
echo $toad->getSecret() ;
```

2.2.5 Principe d'encapsulation

Si un attribut ou une méthode est privé il est donc impossible d'y accéder dans le script courant. Plus généralement les attributs seront privés et les méthodes publiques.

Définition

Les données (attributs) ne peuvent être modifiées dans le script courant directement (ils sont privés), seuls les méthodes qui contrôlent les données peuvent le faire.

Une boîte noire dans un avion par exemple aura un programme possédant des attributs privés et des méthodes publiques ; pour modifier les attributs privés, si un événement exceptionnel dans l'avion se produit, seuls les méthodes publiques peuvent le faire.

Exemple

```
<?php

class BlackBox
{
    private $strength = 1.8;
    private $type = "A390";
    private $data = [];

    public function setData($speed)
    {
        if(is_numeric($speed)){
```



```

        $this->data[]=$speed*$this->strength;
    }else{
        $this->data[]="error1234";
    }
}
}

// si événement exceptionnel...Création d'un objet
$crash = new BlackBox;

// enregistre la vitesse de l'avion
$crash->setData(900);

// var_dump affiche le contenu de la variable objet $crash
var_dump($crash);

```

```

object(BlackBox)[1]
  private 'strength' => float 1.8
  private 'type' => string 'A390' (length=4)
  private 'data' =>
    array (size=1)
      0 => float 1620

```

2.2.6 Accéder aux attributs : **accesseur ou getter**

On va implémenter des méthodes de la classe qui auront comme seul objectif de retourner les valeurs des attributs privées dans le script courant :

Par convention cette méthode commencera par « get » puis le nom de l'attribut à retourner dans le script courant. Bien sûr cette méthode est publique.

Exemple pour l'attribut \$secret on aura :

```

<?php
...
public function getSecret()
{
    return $this->secret;
}
...

```

2.2.7 Modifier un attribut : **mutateur ou setter**

```

<?php
...
public function setSecret($secret)
{
    // si ce n'est pas un string on sort de la méthode en retournant un message, principe
    d'encapsulation on vérifie la cohérence des données :
    if(!is_string($secret))
    {
        echo 'setter function only accepts string. Input was: '.$secret;
        return ;
    }
}

```

```

    }
    $this->secret=$secret;
}
...

```

2.2.8 Le constructeur __construct()

Si la méthode magique __construct est implémentée dans une classe alors lorsqu'on instancie celle-ci, cette méthode est appelée en premier. Cela permet d'initialiser l'objet avec des valeurs particulières.

Exemple dans la classe Persona

```

<?php

class Persona{

    ...
    // constructeur avec paramètres obligatoires
    public function __construct($force) {
        if(is_numeric($force)){
            $this->force=$force;
        }
        $this->force=1000;
    }
    ...
}

```

Attention, ici le fait d'avoir mis un argument obligatoire au constructeur impose le fait de mettre une valeur de force à l'instanciation de la classe Persona, si on ne le fait pas il y aura une erreur :

```

//Warning: Missing argument 1 for Persona::__construct()
$mario = new Persona ;

```

Il faut passer une valeur à l'instanciation de la classe Persona, pour cela on utilise les parenthèses, comme une fonction mais ici pour la classe, comme suit :

```

$mario = new Persona(1000) ;

```

Bien sûr pour plus de souplesse on peut mettre une valeur par défaut au constructeur, cela rend l'argument facultatif pour le constructeur :

```

<?php

class Persona{

    ...
    public function __construct($force=1000) {
        if(is_numeric($force)){
            $this->force=$force;
        }
    }
}

```

```

    $this->force=1000;
}
....
}

```

Dans ce cas on peut instancier la classe de deux manières différentes :

```

$mario = new Persona ;
$koopas = new Persona(0);

```

Exercice

Dans le dossier exercices on crée le fichier **construcPrivate.php**.

Que se passerait-il si on implémente le constructeur de la classe Persona avec la visibilité private ?

2.2.9 Single Responsibility

Lorsqu'on programmera en objet on devra toujours garder à l'esprit le principe suivant :

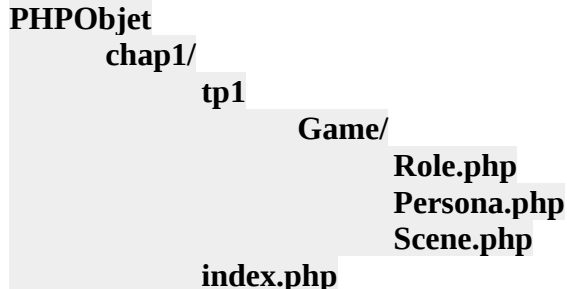
Une classe un rôle, ou attribuer à chaque classe une responsabilité unique définie et bornée.

Par exemple la classe Persona, Scene, Role.

Il faudra également respecter le principe suivant, une classe un fichier, c'est comme cela que l'on programme en objet pas autrement.

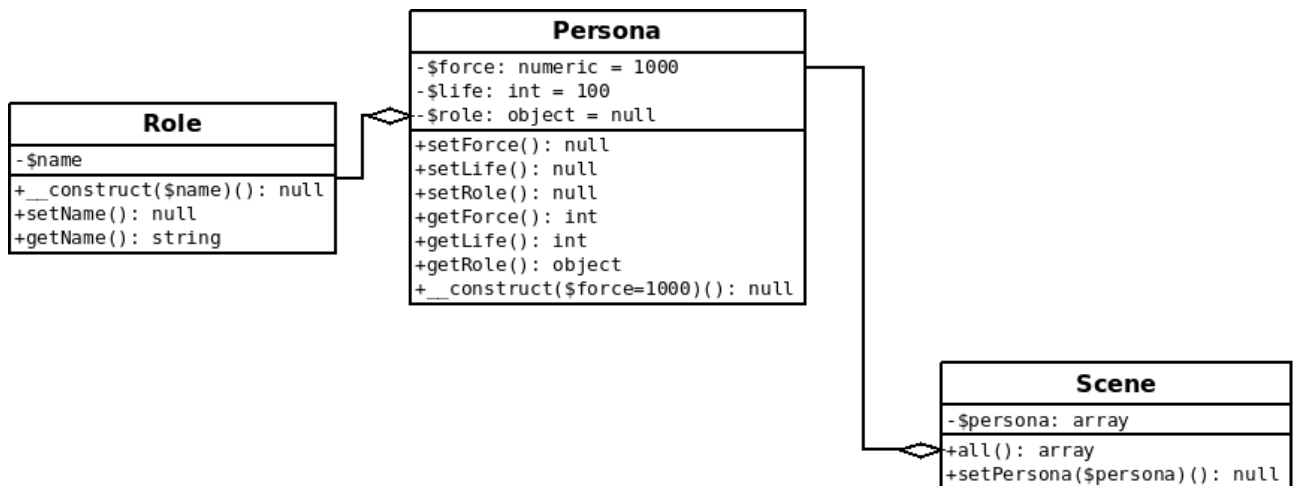
Tp1 **

Organisation fichiers et dossiers



On va créer trois classes. La première « Persona » permet de créer des personnages de jeu avec de la force, un niveau de vie et un rôle, la classe Role définira ce rôle. La classe Scene possède une unique méthode setPersona pour ajouter des objets de type Persona dans un tableau.

Pour créer ces classes on va suivre le schéma UML suivant, un schéma UML permet de voir entièrement les classes à implémenter et de préciser également les relations entre les classes. Ici nous avons deux relations l'attribut \$role dans la classe Persona enregistre un objet de type Role et la classe Scene enregistre des objets de type Persona dans un tableau. Pour tester tout cela on va inclure toutes les classes dans le fichier index.php et instancier les classes (voir après la figure).



Fichier index.php

```
<?php
```

// l'utilisation de la constante `__DIR__` rend les inclusions plus portable, si on change de dossier ..., cette constante est compilée elle est donc performante.

```
require_once __DIR__.'/Game/Persona.php';
require_once __DIR__.'/Game/Role.php';
require_once __DIR__.'/Game/Scene.php';
```

```
...
```

// définition d'un rôle

```
$savior = new Role();
$savior->setName('savior');
```

// création d'un personnage et ajout d'un rôle, on ajoute bien un objet dans un autre !

```
$mario = new Persona;
$mario->setRole($savior);
```

```
...
```

2.2.10 Auto-chargement des classes

Souvent dans une application objet, les scripts n'utilisent pas toutes les classes de l'application, il est donc assez problématique de charger toutes les classes dans un script donné. De plus rien ne dit que toutes les classes ainsi chargées soient toutes utiliser dans le script.

On perd d'ailleurs en performance si on charge trop de classes dans un script. Il faut donc un mécanisme de chargement automatique de classe. Si on a besoin de tel ou tel classe dans un script à un moment donné, on déclenche un mécanisme qui inclut la classe dont on a besoin et c'est tout.

PHP possède un mécanisme d'auto-chargement de classes, il se base sur le fait que la classe n'existe pas au moment ou on essaye d'instancier la classe :

```
<?php
```

```
$mario = new Persona(1000) ;
```

```
Fatal error: Class 'Persona' not found
```

On peut lancer une fonction d'autoload lors de cette erreur pour inclure la classe au bon moment :

```
<?php
```

```
spl_autoload_register('autoload') ;
```

```
function autoload($className){
```

```
    die ($className) ;
```

```
}
```

Exercice

```
PHPObjet
```

```
    /chap1/
```

```
        Autoload/
```

```
        ...[reprendre le dossier et fichiers de l'exercice précédent]
```

Copier coller l'exercice précédent dans un dossier Autoload et implémenter l'autoloader dans le fichier index.php. On utilisera les fonctions `file_exists` pour vérifier que le fichier existe bien avant de l'inclure.

Historiquement l'autoload en PHP a été mis en place pour gagner en performance. Ce gain de performance est fondamental dans les applications PHP objet, il est donc indispensable de l'avoir compris et de savoir le mettre en place.

Par ailleurs, on le verra plus tard, Composer qui est un programme d'installation en ligne de commande de paquets PHP, possède un mécanisme d'autoloader standardisé pour les applications PHP en objet.

`spl_autoload_register` gère une pile de fonction d'autoload :

```
<?php
```

```
// autoloading
```

```
spl_autoload_register('autoload1');
```

```
spl_autoload_register('autoload2');
```

```
spl_autoload_register('autoload3');
```

2.2.11 Méthodes magiques

Les méthodes magiques sont appelées automatiquement lors d'un événement particulier. Dans la suite on en présente quelques unes. Mais celles que l'on utilisera le plus sont :

- `__construct`
- `__set`
- `__get`
- `__call`
- `__toString`.

- les autres sont utilisées dans des cas très spécifiques, donc peu utilisées

Notons également que si on met trop de « magie » dans les classes cela peut impacter les performances d'une application, cela n'empêche pas que l'on utilise beaucoup __construct en objet.

__destruct

C'est une méthode appelée automatiquement lors de la destruction de l'objet :

- soit en détruisant l'objet, c'est-à-dire en l'effaçant de la mémoire unset(\$objet)
- soit à la fin du script, dans ce cas c'est PHP qui efface tout de la mémoire

Exercice

```
PHPObjet
    /chap1/
        exercices/
            destruct.php
```

Implémenter un exemple avec un destructeur dans la classe Foo. On fera un echo dans le destructeur pour visualiser le déclenchement de celui-ci dans le script :

```
<?php
$foo=new Foo ;
unset($foo) ; // destruction en mémoire de l'objet $foo
echo 'Hello world' ;
```

2.2.12 Surcharge magique des attributs et méthodes d'une classe : __set, __get, __call

def surcharge

Définition surcharge PHP

Attention, la notion de surcharge en PHP est différente de celle de la plupart des langages orientés objets. Dans les autres langages elle définit la possibilité d'avoir plusieurs méthodes portant le même nom mais avec un nombre d'arguments différents pour chacune de ces méthodes. En PHP si on fait cela, on aura une erreur fatal, car on ne peut pas redéfinir la même méthode dans la même classe, même principe pour les fonctions, les constantes ou la classe elle-même dans le script courant.

En PHP la surcharge magique ce fait lorsqu'on essaye d'appeler dans le script courant un attribut ou une méthode de la classe qui n'existe pas ou qui est privé (ou protégé que l'on verra plus loin).

__get(\$name), accesseur magique

On écrira dans le fichier exemple.php, la classe Post ci-dessous, pour tester cet exemple :

```
<?php
class Post {
```

```

private $title="attribut privé";

public function __get($name) {
    echo "je n'existe pas ou je suis inaccessible dans le script ($name)";
}

}

(new Post)->title;
(new Post)->content;

```

De même il existe un mutateur magique `__set($name, $value),`

```

<?php
...
public function __set($name, $value) {
    if (property_exists($this, $name)) {
        $this->$name = $value;
    }
}
...

$post=new Post;
$post->title="PHP 5.6";

var_dump($post);
object(Post)[1]
  private 'title' => string 'PHP 5.6' (length=7)

```

2.2.13 `__call($name, $arguments)`

Cette méthode magique est appelée automatiquement lorsqu'on essaye d'appeler dans le script courant une méthode qui n'existe pas dans la classe ou qui est protégée :

```

<?php
...
private function renderPrivate(){
    echo "Je suis une méthode privée non accessible dans le script courant";
}

public function __call($method, $arguments) {
    if(method_exists($this, $method)){
        $this->$method();
    }else{
        echo "<p>je suis la méthode $method, et voici mes paramètres:</p>";
        echo "<ul>";
        foreach($arguments as $v){
            echo "<li>$v</li>";
        }
        echo "</ul>";
    }
}

```

```
}
```

```
...
```

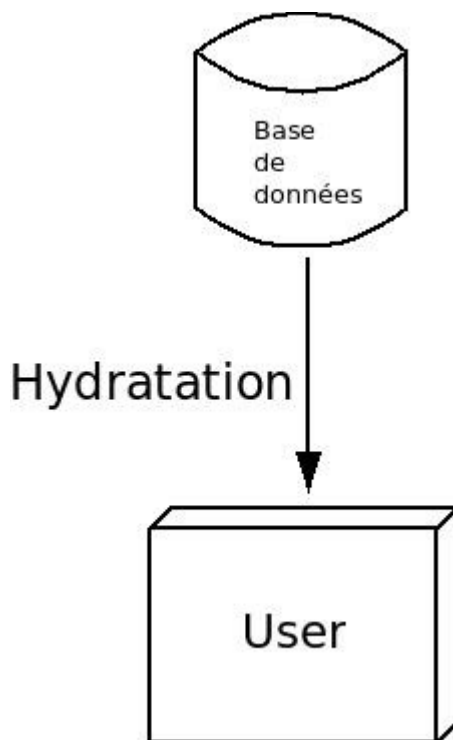
2.2.14 __toString

Méthode magique qui est appelée automatiquement lorsqu'on fait un echo sur l'objet lui-même :

```
// toString
class Bar{
    public function __toString() {
        return "<p>un message</p>";
    }
}
echo (new Bar);
```

2.3 Hydratation TP

L'hydratation est le fait de nourrir un objet de données spécifiques en respectant le principe d'encapsulation.



2.3.1 Introduction préparation de l'environnement de travail

Dans la suite du cours, on va créer une base de données « blog », un utilisateur « tony » (à personnaliser).

Dans un premier temps on va configurer les machines pour avoir PHP et MySQL en ligne de commande :

- Windows (WAMP)
 - On installe cmdr et Wamp selon l'architecture de la machine a priori 64bits, on met dans les variables d'environnement les chemins pour PHP et MySQL (voir les variables d'environnement Windows)
- Mac (MAMP),
 - on crée un fichier `.bash_profile` à la racine de son compte(on tape dans un terminal `$ cd ~` pour accéder à la racine de son compte).
On écrira dans ce fichier la ligne suivante (à adapter si besoin) vers le dossier dans lequel se trouve les exécutable, on utilise vim un éditeur en ligne de commande :
`$ vim .bash_profile`
Dans le fichier :
export PATH=/Applications/MAMP/Library/bin:\$PATH
export PATH=/Applications/MAMP/bin/php/php5.5/bin:\$PATH
Pour quitter le fichier on tapera shift + wq (enregistre et quitte).
 - Relancer le terminal pour voir les changements : on quitte le terminal et on le relance.
- Pour Linux si on a LAMP tout est installé

On vérifie dans un terminal la version de PHP et l'accès au serveur de base de données :

```
$ php -v
```

```
$ mysql -u root -p
```

Voici un site pour avoir l'essentiel des commandes de Vim (éditeur en ligne de commande)
[<http://www.tuteurs.ens.fr/unix/editeurs/vim.html>]

Organisation des dossiers et fichiers

PHPObjet

```

/chap1/
  hydratation/
    script/
      install.php
    Db/
      Post.php
      Connect.php
    bootstrap.php
    index.php

```

Script de déploiement pour la base de données, création des tables et d'un utilisateur de la base de données :

Mettre le script qui suit dans un fichier `install.php`, pour exécuter le script en ligne de commande on tapera. On utilise `mysqli_*` qui est une API PHP permettant de communiquer avec un serveur de base de données :

```
$ php install.php
```

```
<?php
```

```

$link=mysqli_connect('localhost', 'root', 'antoine') or die("pb connexion");

// réinitialisation
$link->query("
    DROP DATABASE IF EXISTS blog;
    ") or die("drop database blog impossible");
$link->query("
    DELETE FROM mysql.user WHERE user='tony' and host='tony';
    ") or die("drop user tony impossible");

$link->query(
    "CREATE DATABASE IF NOT EXISTS `blog` DEFAULT CHARACTER SET utf8
    COLLATE utf8_general_ci;"
    ) or die("pb create database blog");

echo "ok database \n";

$link->query(
    "GRANT ALL PRIVILEGES ON blog.* to 'tony'@'localhost' IDENTIFIED BY
    'tony' WITH GRANT OPTION;"
    ) or die("impossible de créer l'utilisateur tony");

echo "Ok add user tony \n";

mysqli_close($link);

$link=mysqli_connect('localhost', 'tony', 'tony', 'blog') or die("pb connexion");

$link->query(
    "CREATE TABLE IF NOT EXISTS `posts` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `title` VARCHAR(30) NOT NULL,
    `content` TEXT NOT NULL,
    `date_created` DATETIME,
    `status` ENUM('publish', 'unpublish', 'draft', 'trash') NOT NULL DEFAULT 'publish',
    PRIMARY KEY (`id`)
    ) ENGINE=InnoDB AUTO_INCREMENT=1 ;"
    ) or die("pb create table");

$link->query("
    INSERT INTO posts (title, content, date_created) VALUES
    ('PHP 5.6', 'blabla', NOW()), ('script php', 'blabla', NOW());
    ");

mysqli_close($link);
echo "ok tout est bon";

```

On vérifiera que tout c'est bien passé en tapant dans un terminal :

\$ mysql -u tony -p on se connecte avec cet utilisateur et son mot de passe à donner ensuite.

`>show databases ;` montre tous les bases de données qui appartiennent à « tony »

`$ USE blog ;` on se connecte à la base de données blog

`$ DESCRIBE posts ;` on affiche une description de la table par exemple

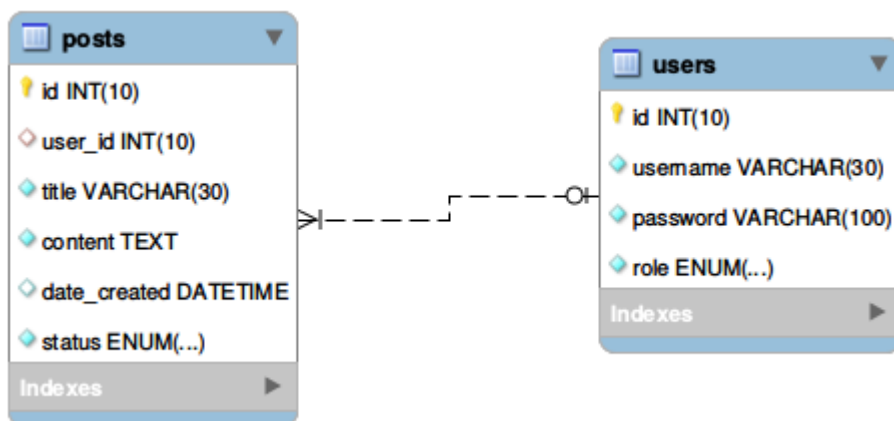
Moralité, même si on a pas écrit un « super script » on a le code en PHP qui nous permet de déployer rapidement notre travail.

Exercice

Reprendre le script précédent et ajouter une table users, la table posts possède maintenant une clef étrangère user_id qui se référence à la clef primaire de la table posts. On ajoutera un utilisateur avec un mot de passe.

Pour créer ce mot de passe on utilisera la fonction PHP suivante : `password_hash('Admin', PASSWORD_DEFAULT)`.

Relancer le script qui réinitialisera l'installation de la base de données et des tables. On donne ci-dessous le schéma des tables (fait avec mysqlworkbench) :



2.3.2 Introduction à PDO

On utilisera PDO à la place de l'API `mysqli_*`, qui est spécifique à MySQL. PDO est plus souple car il ne dépend pas de la SGDB, en clair d'un type de base de données particulier. PDO s'utilise de la même manière pour MySQL, PostgreSQL, ...

C'est une couche d'abstraction (une couche objet) des fonctions d'accès aux bases de données.

Attention, lorsqu'on établira une connexion avec une base de données il faudra indiquer quel type de SGDB à utiliser.

```
<?php
```

```
try {
```

```
    $options = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8",
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION);
    $pdo = new PDO("mysql:host=localhost;dbname=blog", 'tony', 'tony',
        $options);
```

```
// $pdo = null; // ferme la connexion ...
```

```
} catch (PDOException $e) {
```

```
    echo "Erreur: à la ligne:" . $e->getLine() . "on a l'erreur:" . $e->getMessage();
```

```
}
```

PDO est une classe, pour créer une connexion à la base de données, on crée une instance de celle-ci.

DNS, data name source, ou chaîne de connexion, pilote, serveur et bd, correspond à

mysql:host=localhost;dbname=gestion_user dans les arguments de la classe. Il existe plusieurs DNS selon le type de SGDB (Oracle, SQLite, ...).

SET NAME utf8 indique le jeu de caractères utilisé depuis et vers le serveur de base de données, échange entre les modules PHP et MySQL.

PDO::ERRMODE_EXCEPTION indique que l'on va utiliser la classe **PDOException** pour la gestion des exceptions, par exemple **PDO::ERRMODE_SILENT** ne rapporte aucune erreur !
mysql:host=localhost;dbname=gestion_user le DNS .

\$pdo est un objet, si la connexion n'est pas faite avec la base de données cet objet renvoi une exception qui sera catcher (attrapé).

Pour écraser/stopper la connexion on écrira dans le script **\$pdo=null**

Dans la suite on va utiliser la méthode query de PDO pour faire les requêtes sur la base de données.

On utilisera cette méthode comme suit :

- On prépare la requête, qui n'est qu'une chaîne de caractères, en faisant attention à l'insertion des valeurs utilisateurs ou variables de script dans cette chaîne, puis on fait la requête. On utilisera **sprintf** pour gérer la chaîne de caractères et l'insertion de valeur dans cette chaîne. En supposant que \$pdo est l'objet de connexion à la base de données, on a :

```
<?php
```

```
$query = sprintf('
    SELECT *
    FROM `posts`
    WHERE status=%s', 'publish'
);
```

```
// exécution de la méthode query
```

```
$stmt=$pdo->query($query) ;
```

- La méthode query retourne un objet de type PDOStatement, dans notre exemple avec un **SELECT** on peut :
 - soit retourner un résultat : **\$stmt->fetch()**
 - soit retourner tous les résultats de la requête : **\$stmt->fetchAll()**

Deux stratégies pour extraire les données en base :

- Si on fait un **\$stmt->fetch()**, cette méthode retourne un résultat puis déplace son pointeur sur le résultat suivant. Donc si on refait un appel à cette méthode, on obtiendra le résultat suivant, ainsi de suite, jusqu'à arriver au bout du tableau (plus de valeur à retourner), dans ce cas la méthode retournera false.
- Si on fait un **\$stmt->fetchAll()**, on retourne tous les résultats d'un seul coup.

On utilisera la méthode **fetch** qui techniquement **consomme moins de mémoire**. En effet, elle ne retourne qu'une ligne d'un tableau de résultats à chaque appel. On lui passera un argument pour préciser que l'on souhaite avoir les données sous forme d'un tableau associatif :

```
$stmt->fetch(PDO::FETCH_ASSOC) ;
```

Concrètement dans le script on écrira :

```
<?php
```

```
while ($data = $q->fetch(PDO::FETCH_ASSOC)) {  
    echo $data['title'];  
}
```

Exercice

On va créer un fichier bootstrap.php, dans lequel on écrira l'autoloader et on établira une connexion à la base de données. Il faudra également inclure le fichier bootstrap.php dans le fichier index.php (point d'entrée de notre application).

On créera également la classe Connect. Ci-dessous le diagramme UML de cette classe ; on passera les paramètres de connexion dans un tableau PHP \$database, pour se connecter à la base de données, dans le constructeur de la classe :

Connect
+\$pdo: PDO
+__construct(\$database:array): null
+getPdo(): Objet PDO
+disconnect(): null

On testera ensuite dans le fichier index.php que tout est en place :

```
<?php
```

```
require_once __DIR__.DIRECTORY_SEPARATOR."bootstrap.php";
```

```
var_dump($pdo) ; // doit retourner l'objet PDO
```

Exercice

1/ On crée la classe Post et User. Ces deux classes doivent avoir des attributs correspondants aux noms des champs des tables posts et users respectivement. Il faut également implémenter les setters et getters sur ces attributs.

2/ Par ailleurs, on implémentera des setter et getter magiques (__set et __get) qui appelleront les méthodes setTitle, getTitle, setContent, getContent, ... Voir l'exemple qui suit pour implémenter :

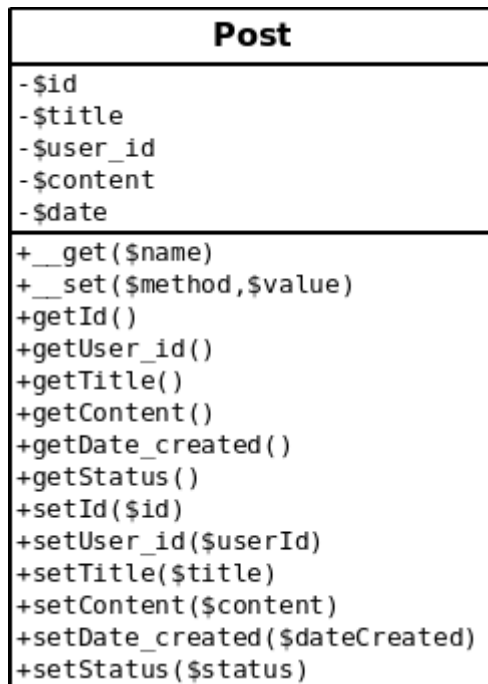
```
<?php
```

```
$post=new Post ;  
echo $post->title ;
```

```
<?php
```

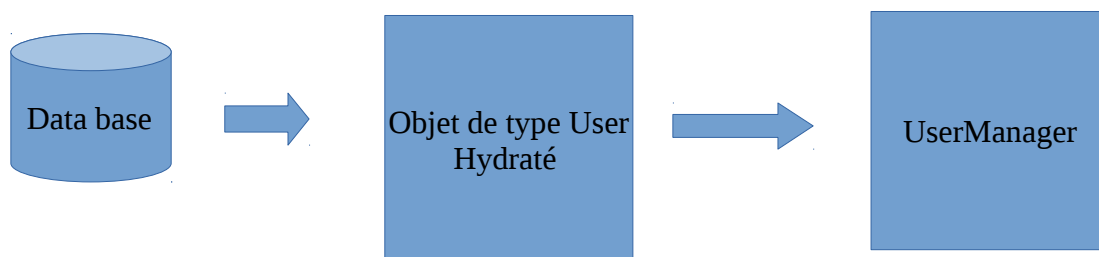
```
...  
public function __get($name) {  
    $method = 'get' . ucfirst($name);  
    if (method_exists($this, $method)) {  
        return $this->$method();  
    }  
}  
...  
...
```

Voici le diagramme UML de la classe Post, la classe User suit le même principe



On va s'occuper maintenant de la classe UserManager et PostManager

Le rôle de ces classes c'est d'hydrater la classe User et Post des données de la base, cela permet de rester cohérent entre la base de données et le monde objet, ainsi on aura pas de problème de type de données puisque les setter et getter sont là pour vérifier la cohérence.



On va implémenter la classe UserManager et PostManger, elles permettrons d'hydrater les objets User et Post que l'on peut voir comme des « structures de données avec des méthodes pour agir sur les données ».

Voici le diagramme UML de la classe UserManager

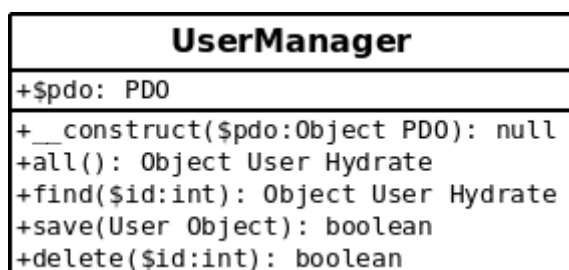


Table des matières

1	Introduction.....	1
1.1	Fonctionnement de PHP.....	1
2	Historique.....	4
2.1	Introduction à l'objet.....	4
2.1.1	Organisation des dossiers et fichier pour le cours.....	4
2.1.2	Définitions.....	4
2.1.3	Visibilité d'un attribut ou d'une méthode.....	5
2.1.4	Méta-variable \$this.....	6
2.1.5	Principe d'encapsulation.....	6
2.1.6	Accéder aux attributs : accesseur ou getter.....	7
2.1.7	Modifier un attribut : mutateur ou setter.....	7
2.1.8	Le constructeur __construct().....	8
2.1.9	Single Responsibility.....	9
2.1.10	Auto-chargement des classes.....	10
2.1.11	Surcharge magique des attributs et méthodes d'une classe : __set, __get, __call.....	12
2.1.12	__call(\$name, \$arguments).....	13
2.1.13	__toString.....	13
2.2	Hydratation TP.....	14
2.2.1	Introduction préparation de l'environnement de travail.....	14
2.2.2	Introduction à PDO.....	17