PHP – OBJET - 1 - BASES

https://www.w3schools.com/php/php_oop_classes_objects.asp
https://www.php.net/manual/fr/language.oop5.php

SOMMAIRE

PHP OBJET – PARTIE 1 – BASES

Vocabulaire de Programmation orientée objet

Intérêt de la POO

- Le code, sans POO, devient vite compliqué et difficile à faire évoluer.
  - L'intérêt de la POO est d'avoir un **code plus facile à faire évoluer.**
- La POO permet de **créer des « boites noires » : les classes**, qui correspondent à des objets du monde réel
  - Une classe permet de décrire facilement les objets du monde réel.
  - Une classe masque la complexité en donnant des outils simples à utiliser.
  - Les classes permettent des créer des objets auxquels on pourra appliquer des fonctions prédéfinies dans la classe.
  - La classe, c'est le plan qui permet de créer et d'utiliser l'objet.
- Les classes sont pratiques pour gérer des données de la BD
  - Une classe correspond à une table de la BD.
  - Une ligne d'une table correspond à un objet de la BD.
  - Circuler de la BD aux objets correspond à l' « **ORM** » : Object Relationnal Mapping

Classe

- Une classe, c'est un **type**, comme un entier, un réel, un caractère, une string ou un booléen.
- Une classe est un **moule** qui permet de créer des objets, qui sont des variables de type Classe.
- En général, une classe correspond à l'équivalent d'un tableau associatif du PHP : elle contient des **couples de clé-valeur**.
  - Les différentes clés sont appelées « attribut ».
  - Les valeurs seront données aux objets créées à partir de la classe.
- En plus, on associe des fonctions à une classe : on les appelle alors « méthode ».

⇨ **Une classe est un plan pour créer et utiliser des objets.**

Objet

- Un objet c'est une variable de type Classe.
- Quand on crée un objet avec des valeurs pour les couples clé-valeur (pour les attributs), on dit qu'on instancie un objet : c'est une « **instanciation** » :
  - Ca passe par la commande « **new** ».
- Les objets sont ce qu'on appelle les « **instances** » de la classe.

⇨ **Instance : une réalisation concrète du plan : un objet de la classe**

⇨ **Instancier : créer un objet à partir d'une classe.**

Méthode

- Les méthodes sont des **fonctions qui sont attachées à une classe**.
- Elle ne se sont utilisables que par les objets de la classe.
- On écrit : objet->methode() pour appeler la méthode pour l'objet en question :
  - c'est comme si on avait passé l'objet en paramètre de la méthode.
- Il y a différents types de méthodes. Principalement :
  - les « **constructeurs** », pour créer un objet (on peut avoir plusieurs constructeurs avec des paramètres différents). Leur nom est celui de la classe.
  - les « **setter** », pour donner une valeur à un attribut. Leur nom est du type : setAttribut(value)
  - Les « **getter** », pour récupérer une valeur d'un attribut. Leur nom est du type : attribut() ou getAttribut()

Encapsulation

- C'est le fait de **cacher le contenu technique d'une classe** pour ne laisser que l'accès aux fonctionnalités nécessaires pour l'utilisateur (l'utilisateur d'une classe, c'est le programme qui utilise des objets de la classe).
- En général, les attributs sont encapsulés : ils sont dit « **private** ».
- Les méthodes ne sont pas encapsulées : elles sont dites « **public** ».

Polymorphisme

- Le polymorphisme, c'est le fait que :
  - pour une même ligne de code, le comportement est différent.
- Le principe est que, en fonction de l'objet instancié, c'est une méthode ou une autre qui est utilisée.
- C'est un mécanisme un peu compliqué qui passe par la notion d'interface ou de classe abstraite mais qui est centrale en programmation objet pour avoir un code facilement adaptable aux changements de cahier des charges.

Exceptions

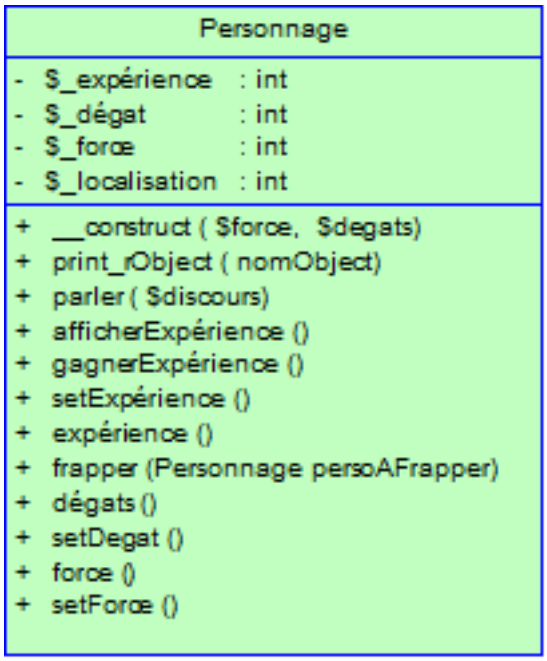
- En cas d'erreur, en programmation objet on passe par des objets de classe Exception.
  - Ca se fait avec un « **try** » « **catch** »
- try : on essaie d'exécuter une suite d'instruction
- catch : si la suite d'instructions exécutée a produit une erreur sous la forme d'une exception, on passe dans le bloc catch
  - Le catch précise le nom de l'objet exception qu'on va traiter.
  - On peut alors accéder à des informations par la méthode getMessage() par exemple.

Guides de style

- Les guides de style PHP sont un peu variables.
- On retrouve le classique CamelCase
  - https://fr.wikipedia.org/wiki/CamelCase
- Ensuite, on a des usages variables selon les framework
  - https://eilgin.github.io/php-the-right-way/#code_style_guide
- Les différentes normes : PSR-12, PEAR, Zend, Symphony, etc.

Classe et objet - exemple

Créer une classe : le plan UML



- Les attributs privés (symbole – en UML) sont écrit $_nomAttribut.
- Les méthodes donc publiques.
- Le constructeur s'appelle : __construct( ) : deux _ devant le nom.
- Le « this » dans les méthodes permet d'utiliser l'objet en cours.
- Les fonctions « set », appelées « setter » permettent de donner une valeur à un attribut : ici les fonctions setDegats(), setForce(), setExperience().
- Les fonctions « nomAttribut», appelées « getter » permettent de récupérer la valeur d'un attribut : ici les fonctions degats(), force() et experience().

Créer une classe : le code

```php
<?php
class Personnage
{
    private int $force;
    private int $experience; // def: 0
    private int $degats; // def: 0
    private array $localisation; // def: x=0, y=0

    public function printInfo(
    ): void {
        echo "Force : " . $this->force . "<br>";
        echo "Expérience : " . $this->experience . "<br>";
        echo "Dégâts : " . $this->degats . "<br>";
        echo "Localisation : x = " . $this->localisation['x'] .
", y = " . $this->localisation['y'] . "<br>";
        echo "<br>";
    }

    // méthode qui déplace le personnage de dX et dY
    public function deplacer(
      int $dX,
      int $dY
    ): void {
      // à écrire
    }

    // méthode qui frappe un personnage
    public function frapper($personnage){
      // à écrire : si les forces sont égales,
      //         degats + 2 pour l'attaqué
      // si la force de l'attaquant est plus faible que celle de
      //         l'attaqué, dégats -2 pour l'attaqué (minimum 0)
      // si la force de l'attaquant est plus forte que celle de
      //         l'attaqué, dégats +5 pour l'attaqué
    }

    // méthode qui augmente l'experience du personnage
    //           de dExperience
    public function gagnerExperience(
      int $dExperience
    ): void {
      $this->experience += $dExperience;
    }
}
```

- A noter :
  - pas de balise fermante du PHP
  - les attribut sont typés, localisation est un « array » : c'est un tableau associatif.
  - les fonctions qui ne retournent rien sont « void »
  - on choisit de lister les paramètres des fonctions l'un en dessous de l'autre

Créer une classe : les bons usages

Les 3 règles fondamentales

- Bien nommer ses variables et ses fonctions.
- Bien indenter le programme.
- Bien commenter le programme.

Usages de ce cours

- nom des **classes** commence par une **majuscule**
- **Constantes** en **majuscules** avec des **_ pour séparer les parties du nom**.
- **double underscore** précédent les noms d'éléments **magiques** (constructeur, etc.)

Constructeur

Le constructeur est une méthode dite « magique » : elle s'appelle « construct » et commence par 2 « _ ».

Exemple : notez les $this->attribut : le $ disparaît !

```php
<?php
class Personnage
{
   private int $force;
   private int $experience; // def: 0
   private int $degats; // def: 0
   private array $localisation; // def: x=0, y=0

   public function __construct(
     int $force,
     array $localisation = ['x' => 0, 'y' => 0],
     int $experience = 0,
     int $degats = 0
   ) {
       $this->force = $force;
       $this->experience = $experience;
       $this->degats = $degats;
       $this->localisation = $localisation;
   }
   Etc.
```

- A noter :
  - Pour donner une valeur par défaut, on passe par le constructeur.
  - Avec ce système, on peut mettre 1, 2, 3 ou 4 paramètres au constructeur.

Utiliser une classe

Principes

- Dans un fichier « usages.php » on écrit un code de création d'un objet et d'utilisation de l'objet.
- Le mot clé « new » permet de créer un nouvel objet. C'est la méthode __construct() qui est appelée à cette occasion.
- A partir d'un objet, on peut utiliser les méthodes définies dans la classe : print_rObject(), parler(), gagnerExperience(), etc.

usages.php

```php
<?php
// création de $perso1
$perso1 = new Personnage(50);
$perso1->printInfo();
$perso1->parler('Bonjour, je m\'appelle perso_1');
$perso1->gagnerExperience(5); // On gagne de l'expérience.
echo 'Nouvelle expérience de perso_1: '. $perso1->experience() .'</br>';
echo "<br>";
$perso1->printInfo('$perso_1');


// création de $perso2
$perso2 = new Personnage(50, ['x' => 10, 'y' => 10]);
$perso2->parler('Bonjour, je m\'appelle perso_2');


// $perso1 frappe $perso2
echo 'perso_1 frappe perso_2 '. '</br>';
$perso1->frapper($perso2);
echo 'perso_2 après le coup de perso_1: '. '</br>';
$perso2->printInfo('$perso_2');
echo "<br>";
```

- A noter :
  - Les '</br>' pour passer à la ligne.
  - Pas de fermeture de la balise PHP

Programme de test : fichier index.php

- On charge la classe avec un « require »
- On charge le code d'usage avec un « include »

fichier index.phhp

- C'est la page HTML :
  - il « require » le fichier contenant la classe
  - il inclut le fichier avec les usages.
- On choisit d'organiser ça comme ça pour séparer les fichiers.

```php
<?php
echo "Entrée dans l'index : " . "<br>";
require 'Personnage.php'; // On utilise la classe.
include 'usages.php';     // On inclut les usages.
```

require, include, require_once, include_once

http://php.net/manual/fr/function.require.php

Require produit une **erreur** si le fichier requis n'existe pas. Le require est orienté « inclusion de morceaux PHP » : si on ne lest a pas, on peut considérer que le programme ne pourra pas fonctionner.
Include produit un **warning** si le fichier requis n'existe pas. L'include est orienté « inclusion de morceaux HTML » : si on ne les a pas, alors on peut considérer que ce n'est pas grave.
Si on recharge une classe déjà chargée, ça va planter !
Le _once permet d'éviter de recharger un fichier déjà inclus. C'est utile pour les classes et les fonctions.
En final, on tend à utiliser **include_once** et **require_once** tout le temps.

TP 1

Tester le code proposé. Rajouter les codes de fonctions nécessaire.
Remarque : vous pouvez utiliser chatGPT pour générer du code automatiquement, mais attention à bien poser vos question à relire le résultat proposer.

Actualisation du code sous MAMP sur MAC :

Dans le fichier de configuration php : …/conf/php.ini (trouver le chemin sur phpInfo)
Changer dans la zone OPcache le 60 du revalidate_freq en 0

```
[OPcache]
; opcache.revalidate_freq=60
  opcache.revalidate_freq=0
```

Autoload, spl_autoload_register()

Auto-chargement des classes : spl_autoload_register et pile d'autoload

https://www.php.net/manual/fr/language.oop5.autoload.php
https://www.php.net/manual/en/function.spl-autoload-register.php

Problème : charger plusieurs fois la même classe

Si on require deux fois la même classe, ça va générer une erreur.
Pour éviter ça, on peut utiliser le require_once.
En général, on met une classe par fichier : ça oblige donc à faire un require_once par classe : c'est lourd !
Pour éviter ça, on peut aussi utiliser la fonction spl_autoload_register qui gère un chargement automatique des classes.

__autoload() : obsolète

La méthode magique permettait de charger les classes automatiquement. Il fallait spécifier le code pour chaque classe => c'est obsolète et remplacé par la fonction spl_autoload()

spl_autoload_register et fonction requireClass

La fonction spl_autoload_register permet de remplacer la méthode __autoload() d'une classe.
spl_autoload_register() est une sorte d'__autoload() généralisé : il permet d'exécuter une méthode à chaque instanciation d'une classe : on parle de pile d'autolaod().
C'est une fonction avec un « callback » en paramètre, c'est-à-dire une fonction passée en paramètre.
On peut définir la fonction qu'on passe en paramètre en même temps qu'on la passe : c'est alors une fonction anonyme.
Ainsi, on n'a plus besoin de charger la classe.

```php
<?php
echo "Entrée dans l'index : " . "<br>";
// On crée une pile d'autoload avec le spl_autoload
spl_autoload_register(function ($class_name) {
   include $class_name . '.php';
});

include 'usages.php';     // On inclut les usages.
```

TP 2

Tester l'autoload. Rajouter les codes de fonctions nécessaire.

Attribut de classe, constante de classe, méthode magique

Attribut et fonction de classe : static, self, ::

Principe

Un **attribut de classe**, ou attribut **static**, est un attribut qui ne concerne pas un objet en particulier mais **concerne toute la classe**.

Une **fonction de classe**, ou fonction **static** est une fonction qui ne concerne pas un objet en particulier mais **concerne toute la classe**.

Exemple

On compte le nombre d'instance de la classe Personnage.
On se dote d'une variable de classe : $nbObjects (variable privée accessible via son getter).
On se dote d'un getter : function nbObjects()
On incrémente $_nbObjects dans le constructeur (à chaque nouvelle instance).
Noter l'usage du mot clé « self ».

```php
<?php
class Personnage
{
   // $_nombre : variable de classe : static, initialisée à 0
   static private $nbObjects=0;
   // function static
   static public function nbObjects(
   ): int {
     // on accede à la variable static par self et ::
     // self represente la classe, on pourrait metre Personnage
     return self::$nbObjects;
   }
```

```php
   public function __construct(
       ... etc .
       // on compte le nombre d'instances de la classe
       self::$nbObjects++;
       ... etc.
```

Utilisation d'une fonction statique -- Class ::methode() ou objet->methode()

```
echo Personnage::nbObjects().' personnages instanciés';
```

Personnage:: pour accéder à une fonction statique.

Remarque : on peut aussi passer par un objet pour accéder à la méthode static :

```
echo $perso2-> nbObjects();
```

```
echo Personnage::nbObjects() ,
  ' personnages instancié' ,
  Personnage::nbObjects()<2 ? "" : "s" , // on gère le pluriel
  '<br><br>' ;
```

TP 3

Tester l'exemple proposé.

Constante de classe – const <=> public static

Principe : const MA_CONSTANTE = 20 ;

On peut déclarer des **constantes** (**attributs avec une valeur qui ne change pas**) dans la classe.
Ces attributs sont toujours **const** : ce qui veut dire : **public** et **static**.

On les écrit **en majuscules** avec des _ pour séparer les parties du nom.

Les constantes participent à l' **encapsulation du code** : l'utilisateur de la classe sait qu'il existe une FORCE_PETITE mais ne connaît pas forcément sa valeur. Ca rend le **code plus clair** et **plus facile à maintenir**.

Exemple

```php
<?php
class Personnage
{
   // Déclarations des constantes en rapport avec la force.
   const FORCE_PETITE = 20;
   const FORCE_MOYENNE = 50;
   const FORCE_GRANDE = 80;
```

```php
   public function __construct(
     int $force, etc.
   ) {
          etc.
      // le setter vérifie l'intégrité des données fournies
      // On vérifie que $force vaut « FORCE_PETITE »,
      // ou « FORCE_MOYENNE », ou « FORCE_GRANDE ».
      if (in_array($force,
           [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]
         )) {
        $this->force = $force;
      }
      else{ // sinon on donne une petite force par défaut !
        $this->force = self::FORCE_PETITE;
      }
          etc.
```

Utilisation de constantes : Classe ::CONSTANTE

```
$perso1 = new Personnage(Personnage::FORCE_MOYENNE);
```

TP 4

Tester l'exemple proposé.

Première méthode magique : __toString()

Principe

http://php.net/manual/fr/language.oop5.magic.php#object.tostring
__toString détermine comment l'objet doit réagir lorsqu'il est traité comme une chaîne de caractères.
Ainsi, __toString retourne une chaine de caractères.
Et par exemple, en faisant un echo($unObjet), on affiche la chaine de caractère retournée.

Dans la classe :

```php
// méthode magique,
// on return une chaine qui sera traitée par un echo
public function __toString(): string {
  return
    'Force = ' . $this->force .
    ' - Localisation X = ' . $this->localisation['x'] .
    ' - Localisation Y = ' . $this->localisation['y'] .
    ' - Experience = ' . $this->experience .
    ' - Degats = ' . $this->degats .
    '<br>';
}
```

Dans le programme qui utilise la classe :

```php
// création de $perso1
$perso1 = new Personnage(Personnage::FORCE_MOYENNE);
echo($perso1); // méthode magique
```

Remarque

On peut aussi se passer de méthodes « magiques » et coder des méthodes « normales », type Java, comme par exemple la méthode toString().
Elle fera la même chose que __toString() et s'utilisera normalement :

```php
// création de $perso1
$perso1 = new Personnage(Personnage::FORCE_MOYENNE);
$perso1->toSring(); // méthode toString() « normale »
```

TP 5

Tester l'exemple proposé.

TP 6

Créez une classe qui gère des armes.
Une arme a un type (comme un nom) et une puissance.
Les personnages portent une arme et une seule.
Ajoutez une arme pour chaque personnage.
On se dote d'un printArme() pour afficher l'arme, d'un modifierPuissance() qui permet d'augmenter ou de diminuer la puissance de l'arme.
La puissance de l'arme augmente pendant les combats, comme les dégats.
La méthode printPersonnage() affichera en plus les caractéristiques de l'arme du personnage.

POO – Tests unitaires

Principes

- La notion de « test unitaire » n'est pas liée à la POO : un test unitaire, c'est un programme qui teste une fonction. On parle de « TU » ou « UT » pour ces tests.
- Toutefois, on la retrouve particulièrement en POO :
- Les tests unitaires vont permettre de valider le bon fonctionnement d'une classe, donc de plusieurs fonctions (méthodes) associées à des attributs.
- L'idée est de garder le code de test pour le faire évoluer si on change quelque chose à la classe ou à son contexte et de pouvoir facilement lancer une procédure de tests qui permette de valider que la classe fonctionne bien.
- Dès qu'on programme objet, il faut faire des tests unitaires sur ses classes.
- https://openclassrooms.com/fr/courses/4087056-testez-unitairement-votre-application-php-symfony/7828665-faites-vos-premiers-pas-avec-phpunit-et-les-tests-unitaires

Technique – 1 – PHP Unit et les framework

- Il y a de nombreuses façons de faire en fonction du langage et des framework utilisés.

- PHP utilise souvent **PHP Unit** qui est utilisé par les framework symfony et laravel.
  - On intérêt à utiliser PHP unit si on utilise ces frameworks.
  - https://laravel.com/docs/10.x/testing

- Sans framework, on peut créer ses programmes de tests : un fichier « _TU » par classe.
- Par exemple, au fichier « Personnage.php », on ajoute le fichier « personnage_TU.php »
  - On écrit « personnage_TU.php » en commençant par une minuscule, car ce fichier n'est pas une classe et qu'on ne met que les fichier qui correspondent à des classes avec une majuscule en première lettre.

Technique – 2 – sans framework

Principes

- Si on part de l'exemple « personnage_TU.php » : que faut-il faire ?
- Ecrire un code qui est un programme qui include la classe Personnage.php (require direct et pas auto_load) et qui teste toutes les méthodes (dont le constructeur) pour tous les cas.
    - Il faut donc réfléchir à toutes les entrées possibles.
    - Afficher les résultats prévus.
    - Afficher des résultats.

Ordre

- On commence par les constructeurs. Les tests vont utiliser les méthodes printInfos() et toString() et d'autres s'il y a d'autres méthodes d'affichage d'un objet.
- Ensuite, on a intérêt à commencer par les méthodes sans dépendance :
    - qui n'utilise pas d'autres méthodes qu'on a écrit,
    - qui n'utilisent pas d'autres objets qu'on a créé.
    - Le principe est qu'on teste d'abord de qui est indépendant d'autres parties du code.

Technique – 3 : notion d'assertion

- On va rapidement avoir beaucoup d'affichage quand on fait les tests et ce n'est pas pratique.
- On veut donc finalement n'afficher que les problèmes et afficher « Tests OK » si tout va bien.
- Pour cela, il faut savoir, pour chaque méthode testée, si ça s'est bien passé ou pas.
- Si ça s'est mal passé, on peut afficher un n° d'erreur et le texte de l'erreur.
- Si ça s'est bien passé, on ne fait rien.
- Pour savoir si ça s'est bien passé, on compare le résultat à celui attendu (pour un constructeur, on fait un toString() que l'on compare à celui attendu ; pour un retour de fonction, on compare le retour au retour attendu, etc.).
    - Si on a le bon résultat, on ne fait rien,
    - Sinon on affiche le code qui n'a pas fonctionné, le résultat attendu et le résultat obtenu, en plus du n° d'erreur.
    - La notion d'assertion est ici : on soutient comme vrai que le résultat sera celui attendu. Si c'est le cas, l'assertion est vraie, sinon elle est fausse.
- A la fin, s'il n'y a eu aucune erreur, on affiche « Tests OK ».

- Les framework comme Symfony ou Laravel permettent de simplifier l'écriture du code de test unitaire.
    - Dans un premier temps, on fait ça « à la main » !
    - Il faut trouver une façon d'écrire son code puis faire des copier-coller qu'on adapte aux différentes classes à tester.

TP 7

A partir du TP 6, créez des tests unitaires pour les armes et pour les personnages.

POO – Synthèse de syntaxe et autres exemples

La classe Membre

```
< ?php
class Membre {
  /* ici on décrira la class */
}
$robert = new Membre() ;
$li = new Membre() ;
```

- $robert ne s'appellera plus « variable » mais « objet ».
- La class Membre contiendra par exemple le nom, le pseudo, le mot de passe, et des fonctions permettant de gérer le membre, par exemple : modifier le mot de passe, changer ses données personnelles, etc.
- On peut créer autant d'objets qu'on veut.

Organisation du code : séparation des classes et des objets

- On crée un fichier séparé avec la déclaration des classes.
- On ne ferme pas la balise ouvrante php du début.
- Pour utiliser ce fichier on fait un « require » ou un « require_once » pour qu'il ne soit inclus 2 fois.
- On peut aussi utiliser le mécanisme d'auto-load.

Définition d'une classe

Une classe contient des attributs et des méthodes.

Création des attribut

```
<?php
class Membre {
  private string $pseudo ;
  private string $email ;
  private bool $actif ;
}
```

- Les attributs sont typés (ce n'est pas obligatoire, mais c'est mieux).
- private : signifie que la variable n'est pas accessible !!! A quoi servent-elle alors ?
- Elles vont servir pour les fonctions qui elles seront accessibles. C'est le principe de la POO : donner des outils simples à utiliser, en l'occurrence les méthodes, en cachant la complexité.

Création des fonctions (= méthodes)

```
< ?php
class Membre {
  private string $pseudo ;
  private string $email ;
  private bool $actif ;

  // getter pour récupérer l'attribut
  public function pseudo(
  ): string {
    return $this->pseudo ;
  }

  // setter pour mettre à jour l'attribut
  public function setPseudo(
    $pseudo
  ) : void {
    $this->pseudo=$pseudo ;
}
```

this-> : this c'est l'objet qui utilise la classe. La -> permet d'accéder à une variable de la classe.
La fonction pseudo() renvoie le pseudo. On met son type après les parenthèses, avant l'accolade de début.
Pourquoi avoir récupérer directement la variable ? Ça permet, par exemple, de rajouter du code, de faire des vérifications par exemple, ou des transformations, etc.
La fonction pseudo() ne renvoie rien : on met « void » après les parenthèses, avant l'accolade de début.
Dans cette fonction, on peut faire des tests :

```
  public function setPseudo(
    $pseudo
  ) : void {
    if (!empty($nouveauPseudo) AND strlen($nouveauPseudo) < 15) {
      $this->pseudo=$pseudo ;
    }
  }
```

- Ici on vérifie que le pseudo n'est pas vide et que la taille est <15.
- A noter que le $pseudo passé en paramètre dans setPseudo n'a rien à voir avec le « pseudo » de l'objet qui correspond au $pseudo de la classe. On pourrait l'appeler $varPseudo pour bien distinguer. Mais ce n'est pas l'usage.

Utilisation des fonctions

Dans la page qui utilise la class, ou pourra écrire :

```
< ?php
include_once('Membre.class.php')

$robert = new Membre() ;
$robert->setPseudo(« Robert ») ;
echo &robert->getPseudo() ;
```

Autre écriture de fonctions, sans paramètre

```
public function bannir(){
        $this->actif=false ;
}
public function debannir(){
        $this->actif=true;
}
publid function getActif(){
        return $this->actif ;
}
```

Terminologie des fonctions

Les fonctions get sont appelées : getter ou accesseur
Les fonctions set sont appelées : setter ou mutateur
Dans la page qui utilise la class, ou pourra écrire :

Les constructeurs

Le constructeur est une fonction qui permet d'instancier un objet.
Par défaut, c'est le nom de la classe : $robert=new Membre() ;
On peut créer ses propres constructeurs, en leur faisant faire des actions particulières.
Par exemple, on veut un constructeur qui donne tout de suite la valeur du pseudo au membre et qui le positionne comme membre actif.

```
public function __construct ($pseudo){ /* deux _ et construct */
        $this->pseudo=$pseudo ;
        $this->actif=true ;
}
```

Quand on code un constructeur, il s'appelle toujours : __construct.

Constructeur sans paramètre

```
$robert = new Membre() ;
$robert->setPseudo(« Robert ») ;
echo &robert->getPseudo() ;
$robert->bannir() ;
```

Avec la POO, le code est beaucoup plus lisible et donc plus facile à mettre à jour.

Constructeur avec paramètre :

```
$robert = new Membre(« Robert ») ; /* on crée l'objet directement avec un pseudo */
echo &robert->getPseudo() ;
$robert->bannir() ;
```

Garbage collector

Pour supprimer un objet, il suffit que sa référence (le nom de la variable objet) soit setté à NULL ; $objet=NULL ;
Le garbage collector (ramasse-miette) gère le fait de libérer la mémoire mais on ne sait pas quand c'est fait.
On peut forcer l'appel avec la fonction : gc_collect_cycles();

Attributs et méthodes de classe :

```
static private $_nbObjects=0;     // attribut de classe
const FORCE_PETITE = 20;          // constante de classe

static public function nbObjects(){ // méthode de classe
self::$_nbObjects;                // accès à un attribut de classe
self::FORCE_PETITE                // accès à un attribut de classe
Personnage::FORCE_MOYENNE         // accès à un attribut de classe
public function __toString()      // méthode magique
echo($objet);                     // <=> echo $objet->__toString()
```