Programmation Orienté Objet en PHP

I. Définitions

1. Définition d'une classe :

Une classe est une entité cohérente regroupant des variables et des fonctions. Chacune de ces fonctions aura accès aux variables de cette entité.

2. Définition d'un objet :

Un objet est une variable (d'instance) de type correspondant à la classe invoquée dans l'instanciation.

3. Définition d'une instance :

Une instance, c'est tout simplement le résultat d'une instanciation. Une instanciation, c'est le fait d'instancier une classe, c'est à dire créer un objet de type cette classe. « De créer un moulé »

Instancier une classe, c'est se servir d'une classe « un moule » afin qu'elle nous crée un objet « moulé ». En gros, une instance est un objet.

4. Définition de l'encapsulation :

L'encapsulation permet de rassembler les attributs composant un objet et les méthodes pour les manipuler dans une seule entité appelée classe de l'objet.

II. Les classes

Une classe se déclare par le mot clé class suivi d'un identificateur de classe choisi par le programmeur

Créer un nouveau type de données, c'est **modéliser** de la manière la plus juste un objet, à partir des possibilités offertes par un langage de programmation.

Il faudra donc énumérer toutes les propriétés de cet objet et toutes les fonctions qui vont permettre de définir son comportement. Ces dernières peuvent être classées de la façon suivante :

- **Les fonctions d'entrée/sortie**. Comme les données de base du langage, ces fonctions devront permettre de lire et d'écrire les nouvelles données sur les périphériques (clavier, écran, fichier, etc.).
- **Les opérateurs de calcul**. S'il est possible de calculer sur ces données, il faudra créer les fonctions de calcul.
- **Les opérateurs relationnels**. Il faut au moins pouvoir tester si deux données sont égales. S'il existe une relation d'ordre pour le nouveau type de donnée, il faut pouvoir aussi tester si une donnée est inférieure à une autre.
- **Les fonctions de conversion**. Si des conversions vers les autres types de données sont nécessaires, il faudra implémenter également les fonctions correspondantes.
- **Intégrité de l'objet**. La manière dont est modélisé le nouveau type de données n'est probablement pas suffisant pour représenter l'objet de façon exacte. Par exemple, si l'on représente une fraction par un couple de deux entiers, il faut également vérifier que le dénominateur d'une fraction n'est pas nul et que la représentation d'une fraction est unique (simplification).

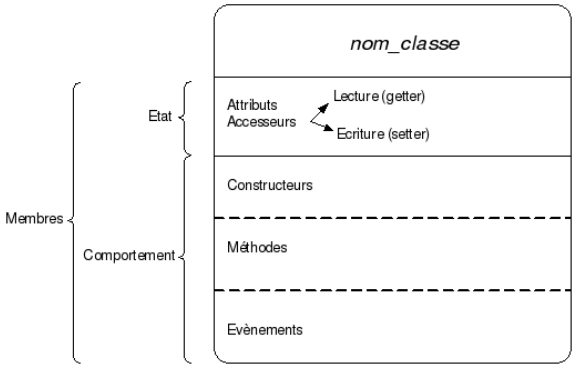
La déclaration d'un nouveau type de données et les fonctions qui permettent de gérer les objets associés constituent une classe de l'objet.

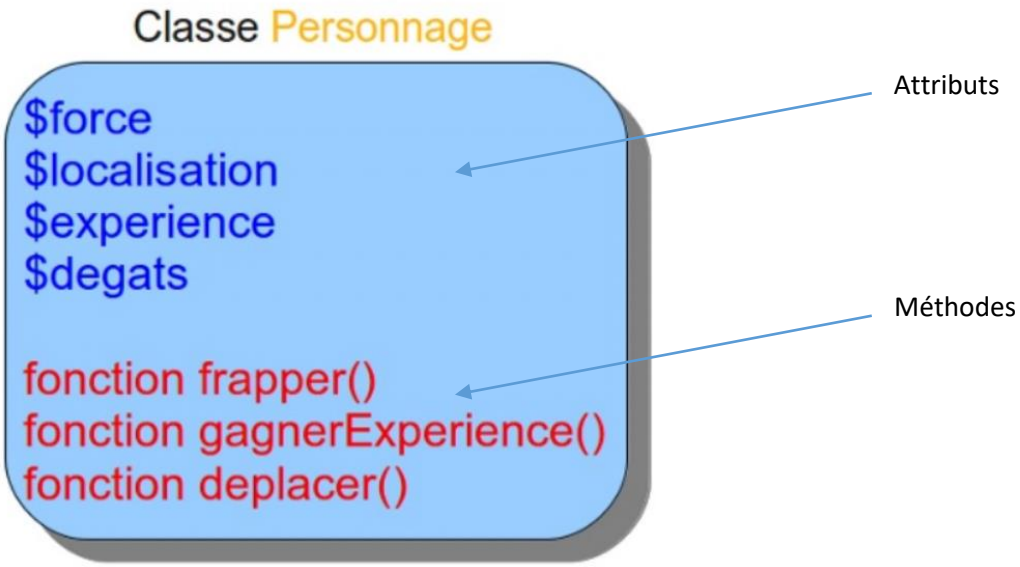
Les propriétés de l'objet seront implantées sous la forme de données membres de la classe.

Les différentes fonctions ou méthodes seront implémentées sous la forme de fonctions membres de la classe.

De façon courante, dans le patois de la programmation orientée objet, données membres et fonctions membres de la classe sont considérées respectivement comme synonymes de attributs et méthodes de l'objet.

Exemple de Classe :





Les propriétés et les méthodes de l'objet seront déclarées et implémentées dans le bloc { } contrôlé par le mot clé class.

Cela constitue l'implémentation du concept d'encapsulation

```
class Personnage
{
        private $force;
        private $localisation;

        public function deplacer()
        {// Une méthode qui déplacera le personnage .

        }

        public function frapper()
        {// Une méthode qui frappera un personnage.
        }
}
```

1. Visibilité d'un attribut ou d'une méthode

La visibilité d'un attribut ou d'une méthode indique à partir d'où on peut y avoir accès. Nous allons voir ici deux types de visibilité : `public` et `private`. (Il en existe un 3eme qui sera abordé au moment de l'héritage)

Le premier, `public`, est le plus simple. Si un attribut ou une méthode est `public`, alors on pourra y avoir accès depuis n'importe où, depuis l'intérieur de l'objet (dans les méthodes qu'on a créées), comme depuis l'extérieur. Je m'explique. Quand on crée un objet, c'est principalement pour pouvoir exploiter ses attributs et méthodes. L'extérieur de l'objet, c'est tout le code qui n'est pas *dans* votre classe. En effet, quand vous créerez un objet, cet objet sera représenté par une variable, et c'est à partir d'elle qu'on pourra modifier l'objet, appeler des méthodes, etc. Vous allez donc dire à PHP « dans cet objet, donne-moi cet attribut » ou « dans cet objet, appelle cette méthode » : c'est ça, appeler des attributs ou méthodes depuis l'extérieur de l'objet.

Le second, `private`, impose quelques restrictions. On n'aura accès aux attributs et méthodes *seulement* depuis l'intérieur de la classe, c'est-à-dire que seul le code voulant accéder à un attribut privé ou une méthode privée écrit(e) *à l'intérieur* de la classe fonctionnera. Sinon, une jolie erreur fatale s'affichera disant que vous ne pouvez pas accéder à telle méthode ou tel attribut parce qu'il ou elle est privé(e).

Là, ça devrait faire *tilt* dans votre tête : le principe d'encapsulation ! C'est de cette manière qu'on peut interdire l'accès à nos attributs.

2. Création d'attributs

Pour déclarer des attributs, on va donc les écrire entre les accolades, les uns à la suite des autres, en faisant précéder leurs noms du mot-clé `private`, comme ça :

Code : PHP

```php
<?php

class Personnage

{
  private $_force;          // La force du personnage
  private $_localisation;   // Sa localisation
  private $_experience;     // Son expérience
  private $_degats;         // Ses dégâts
}
```

Vous pouvez constater que chaque attribut est précédé d'un underscore (« _ »). Ceci est une notation qu'il est préférable de respecter (il s'agit de la notation PEAR) qui dit que chaque nom d'élément privé (ici il s'agit d'attributs, mais nous verrons plus tard qu'il peut aussi s'agir de méthodes) doit être précédé d'un underscore.

Vous pouvez initialiser les attributs lorsque vous les déclarez (par exemple, leur mettre une valeur de 0 ou autre).

3. Implantation des méthodes

Pour la déclaration de méthodes, il suffit de faire précéder le mot-clé `function` à la visibilité de la méthode. Les types de visibilité des méthodes sont les mêmes que les attributs. Les méthodes n'ont en général pas besoin d'être masquées à l'utilisateur, vous les mettrez souvent en `public` (à moins que vous teniez absolument à ce que l'utilisateur ne puisse pas appeler cette méthode, par exemple s'il s'agit d'une fonction qui simplifie certaines tâches sur l'objet mais qui ne doit pas être appelée n'importe comment).

Code : PHP

```php
<?php
class Personnage
{
  private $_force;         // La force du personnage
  private $_localisation; // Sa localisation
  private $_experience;    // Son expérience
  private $_degats;        // Ses dégâts

  public function deplacer() // Une méthode qui déplacera le
personnage (modifiera sa localisation).
  {

  }

  public function frapper() // Une méthode qui frappera un
personnage (suivant la force qu'il a).
  {

  }

  public function gagnerExperience() // Une méthode augmentant
l'attribut $experience du personnage.
  {

  }
}
?>
```

III. Instanciation

1. Création d'objets

Pour qu'un objet ait une existence, il faut qu'il soit instancié. Une même classe peut être instanciée plusieurs fois, chaque instance ayant des attributs ayant des valeurs spécifiques.

Nous allons voir comment créer un objet, c'est-à-dire que nous allons utiliser notre classe afin qu'elle nous fournisse un objet. Pour créer un nouvel objet, vous devez faire précéder le nom de la classe à instancier du mot-clé ***new***, comme ceci :

Code : PHP

```php
<?php
$perso = new Personnage();
?>
```

Ainsi, `$perso` sera un objet de type `Personnage`. On dit que l'on *instancie* la classe `Personnage`, que l'on crée une instance de la classe `Personnage`.

2. Accès aux propriétés et aux méthodes

Pour appeler une méthode d'un objet, il va falloir utiliser un opérateur : il s'agit de l'opérateur `->` (une flèche composée d'un tiret suivi d'un chevron fermant). Celui-ci s'utilise de la manière suivante. À gauche de cet opérateur, on place **l'objet** que l'on veut utiliser. Dans l'exemple pris juste au-dessus, cet objet aurait été `$perso`. À droite de l'opérateur, on spécifie le nom de la méthode que l'on veut invoquer.

Code : PHP

```php
<?php
// Nous créons une classe « Personnage ».
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Nous déclarons une méthode dont le seul but est d'afficher un texte.
  public function parler()
  {
    echo 'Je suis un personnage !';
  }
}

$perso = new Personnage();
$perso->parler();
```

Remarque : Il est impossible d'atteindre un attribut ou une méthode privée depuis l'extérieur de l'objet.

Depuis l'intérieur de la classe

La pseudo-variable **$this**, permet d'accéder aux méthodes et attributs de la classe à l'intérieur d'elle-même.

(Nota : sauf pour les membres statiques que l'on verra plus loin)

Code : PHP

```php
<?php
class Personnage
{
  private $_experience = 50;

  public function afficherExperience()
  {
    echo $this-> experience;
  }
}

$perso = new Personnage();
$perso->afficherExperience();
```

3. Exiger des objets en paramètre

En php, lorsqu'une méthode prend un argument en entrée, il peut être utile de préciser le type attendu. Ce type ne peut être précisé que s'il s'agit d'un objet (système ou personnel) ou d'un tableau.

Si un objet est requis et qu'une variable d'un autre type est passée, cela génère une erreur d'accès. En effet dans la méthode, on va invoquer une méthode de l'objet et cela n'est pas possible sur un autre type.

Pour cela, il faut ajouter le nom de la Classe avant la variable d'entrée, ou Array dans le cas d'un tableau.

Code : PHP

```php
<?php
class Personnage
{
  // …

  public function frapper(Personnage $persoAFrapper)
  {
    // …
  }
}
```

4. Conventions d'écriture

Les classes - *PascalCase (UpperCamelCase)*

Considérations spéciales : Les noms de classe doivent correspondre au nom du fichier qui contient la classe. Les noms doivent être alphanumériques uniquement.

Les interfaces - *PascalCase (UpperCamelCase)*

Considérations spéciales : Voir Les classes.

Les fonctions et méthodes - *camelCase*

Considérations spéciales : alphanumériques uniquement, à l'exception de '_' comme décrit ici.Le nom doit décrire le comportement de la fonction ou de la méthode. Les méthodes qui sont déclarées avec modificateur privé ou protégé de visibilité doit commencer par un _.

Les variables - *camelCase*

Considérations spéciales : Les noms doivent décrire les données dans la variable. Les variables déclarées avec des modificateurs de visibilité privé ou protégé doivent commencer par un _.

Les constantes - *ALL_CAPS*

Considérations spéciales : alphanumériques et '_' sont autorisés cependant '_' doit être utilisé pour séparer les mots dans des constantes.

5. Méthodes d'accès aux données / Accesseur

Si les propriétés sont verrouillées, on ne peut plus y avoir accès de l'extérieur de la classe. Il faut donc créer des méthodes dédiées à l'accès aux propriétés pour chacune d'elles. Ces méthodes doivent permettre un accès dans les deux sens :

- Pour connaître la valeur de l'attribut. Ces méthodes sont appelées méthodes de type Get ou Getter. La réponse de l'objet, donc la valeur retournée par la méthode Get, doit être cette valeur.
- Pour modifier la valeur d'un attribut. Ces méthodes sont appelées méthodes Set ou setter. Cette méthode ne retourne aucune réponse. Par contre, un paramètre de même nature que l'attribut doit lui être indiquée.

L'intérêt de passer par des fonctions Set est de pouvoir y localiser des contrôles de validité des paramètres passés pour assurer la cohérence de l'objet, en y déclenchant des exceptions par exemple.

```php
class Personnage
{
    private $_nom;
    private $_prenom;

    public function get_nom(){
        return $this->_nom;
    }

    public function set_nom($_nom){
        $this->_nom = $_nom;
    }

    public function get_prenom(){
        return $this->_prenom;
    }

    public function set_prenom($_prenom){
        $this->_prenom = $_prenom;
    }

}
```

Vous pouvez utiliser un générateur de Getter et Setter, soit dans l'IDE , soit en ligne (exemple : http://mikeangstadt.name/projects/getter-setter-gen/)

6. Constructeur

Quand une instance d'une classe d'objet est créée au moment de l'instanciation d'une variable avec ***new***, une fonction particulière est exécutée. Cette fonction s'appelle le constructeur. Elle permet, entre autre, d'initialiser chaque instance pour que ses attributs aient un contenu cohérent.

Un constructeur est déclaré comme les autres fonctions membres à trois différences près :

- Le nom de l'identificateur du constructeur est : `__construct`, avec deux underscores au début.
- Un constructeur ne renvoie pas de résultat.
- Le constructeur doit forcément être public

Comme son nom l'indique, le constructeur sert à *construire* l'objet. Ce que je veux dire par là, c'est que si des attributs doivent être initialisés, c'est par ici que ça se passe. Comme dit plus haut, le constructeur est exécuté *dès la création de l'objet* et par conséquent, aucune valeur ne doit être retournée, même si ça ne génèrera aucune erreur.

Bien sûr, et comme nous l'avons vu, une classe fonctionne très bien sans constructeur, il n'est en rien obligatoire ! Si vous n'en spécifiez pas, cela revient au même que si vous en aviez écrit un vide (sans instruction à l'intérieur). Dans ce cas, les parenthèses sont inutiles lors de l'appel (exemple $objet=new MaClasse)

Code : PHP

```php
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  public function __construct($force, $degats) // Constructeur demandant 2 paramètres
  {
    echo 'Voici le constructeur !'; // Message s'affichant une fois que tout objet est créé.
    $this->setForce($force); // Initialisation de la force.
    $this->setDegats($degats); // Initialisation des dégâts.
    $this->_experience = 1; // Initialisation de l'expérience à 1.
  }
```

Notez quelque chose d'important dans la classe : dans le constructeur, les valeurs sont initialisées en appelant les setters correspondants. En effet, si on assignait directement ces valeurs avec les arguments, le principe d'encapsulation ne serait plus respecté et n'importe quel type de valeur pourrait être assigné !

7. L'autoload des classes

Les classes peuvent être chargées dynamiquement (c'est-à-dire sans avoir explicitement inclus le fichier la déclarant) grâce à l'auto-chargement de classe (utilisation de s`pl_autoload_register`).

Pour une question d'organisation, il vaut mieux créer un fichier par classe. Vous appelez votre fichier comme bon vous semble et placez votre classe dedans. Pour ma part, mes fichiers sont toujours appelés « MaClasse.class.php ». Ainsi, si je veux pouvoir utiliser la classe `MaClasse`, je n'aurais qu'à inclure ce fichier :

Code : PHP

```php
<?php
require 'MaClasse.class.php'; // J'inclus la classe.

$objet = new MaClasse(); // Puis, seulement après, je me sers de ma classe.
```

Maintenant, imaginons que vous ayez plusieurs dizaines de classes… Pas très pratique de les inclure une par une ! Vous vous retrouverez avec des dizaines d'inclusions, certaines pouvant même être inutile si vous ne vous servez pas de toutes vos classes. Et c'est là qu'intervient l'auto-chargement des classes. Vous pouvez créer dans votre fichier principal (c'est-à-dire celui où vous créerez une instance de votre classe) une ou plusieurs fonction(s) qui tenteront de charger le fichier déclarant la classe. Dans la plupart des cas, une seule fonction suffit. Ces fonctions doivent accepter un paramètre, c'est le nom de la classe qu'on doit tenter de charger.

On va donc utiliser la fonction `spl_autoload_register` en spécifiant en premier paramètre le nom de la fonction à charger :

Code : PHP

```php
<?php
function chargerClasse($classe)
{
  require $classe . '.class.php'; // On inclut la classe correspondante au paramètre passé.
}

spl_autoload_register('chargerClasse'); // On enregistre la fonction en autoload pour qu'elle soit appelée dès qu'on instanciera une classe non déclarée.

$perso = new Personnage();
?>
```

Si je nomme bien les fichiers contenants mes classes et que j'inclus ce code dans mon programme, je n'aurais pas de problème de classe non trouvée.

Je peux créer plusieurs autoloads si nécessaire.

IV. L'opérateur de résolution de portée ::

L'opérateur de résolution de portée (« `::` »), appelé « double deux points » (« *Scope Resolution Operator* » en anglais), est utilisé pour appeler des éléments appartenant à telle classe et non à tel objet. En effet, nous pouvons définir des attributs et méthodes appartenant à la classe : ce sont des éléments **statiques**. Nous y reviendrons en temps voulu dans une partie dédiée à ce sujet.

Parmi les éléments appartenant à la classe (et donc appelés via cet opérateur), il y a aussi les **constantes de classe**, sortes d'attributs dont la valeur est constante, c'est-à-dire qu'elle ne change pas. Nous allons d'ailleurs commencer par ces constantes de classe.

1. Les constantes de classe

Le principe est à peu près le même que lorsque vous créez une constante à l'aide de la fonction `define`. Les constantes de classe permettent d'éviter tout code *muet*. Un code qui n'est pas explicite.

Une constante est une sorte d'attribut appartenant à la classe dont la valeur ne change jamais. Pour déclarer une constante, vous devez faire précéder son nom du mot-clé `const`. Faites bien attention, une constante ne prend pas de « $ » devant son nom !

Contrairement aux attributs, vous ne pouvez accéder à ces valeurs *via* l'opérateur « `->` » depuis un objet (ni `$this` ni `$perso` ne fonctionneront) mais avec l'opérateur « `::` » car une constante appartient à la classe et non à un quelconque objet.

Pour accéder à une constante, vous devez spécifier le nom de la classe, suivi du symbole double deux points, suivi du nom de la constante.

Code : PHP

```php
<?php
class Personnage
{
  // Je rappelle : tous les attributs en privé !

  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Déclarations des constantes en rapport avec la force.

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;
```

```php
<?php
// On envoie une « FORCE_MOYENNE » en guise de force initiale.
$perso = new Personnage(Personnage::FORCE_MOYENNE);
?>
```

2. Les attributs et méthodes statiques

Les méthodes statiques

Comme je l'ai brièvement dit dans l'introduction, les méthodes statiques sont des méthodes qui sont faites pour agir sur une classe et non sur un objet. Par conséquent, je ne veux voir aucun `$this` dans la méthode !

```php
  // Notez que le mot-clé static peut être placé avant la
 visibilité de la méthode (ici c'est public).
  public static function parler()
  {
    echo 'Je vais tous vous tuer !';
  }
}
?>
```

Et dans le code, vous pourrez faire :

Code : PHP

```php
<?php
Personnage::parler();
?>
```

Les attributs statiques

Le principe est le même, c'est-à-dire qu'un attribut statique appartient à la classe et non à un objet. Ainsi, tous les objets auront accès à cet attribut et cet attribut aura la même valeur pour tous les objets.

La déclaration d'un attribut statique se fait en faisant précéder son nom du mot-clé `static`, comme ceci :

```php
  // Variable statique PRIVÉE.
  private static $ texteADire = 'Je vais tous vous tuer !';
```

Il doit être invoqué dans le code en utilisant self ::

```php
    public static function parler()
    {
      echo self::$ texteADire; // On donne le texte à dire.
    }
```

3. Compteur d'instance

Afin de compter le nombre d'instance d'une classe (le nombre d'objet créé de cette classe), on va ajouter un attribut static compteur, on va incrémenter le compteur dans le construct, ainsi chaque fois que l'on créera un objet le compteur sera mis à jour pour tous les objets. Lorsque l'on voudra savoir combien d'objet auront été créé, il suffira de demander à la classe la valeur de ce compteur.

Code : PHP

```php
<?php
class Compteur
{
  // Déclaration de la variable $compteur
  private static $ compteur = 0;

  public function   construct()
  {
    // On instancie la variable $compteur qui appartient à la
classe (donc utilisation du mot-clé self).
    self::$ compteur++;
  }

  public static function getCompteur() // Méthode statique qui
renverra la valeur du compteur.
  {
    return self::$ compteur;
  }
}

$test1 = new Compteur;
$test2 = new Compteur;
$test3 = new Compteur;

echo Compteur::getCompteur();
?>
```

V. L'héritage

1. Définition

Le concept d'héritage est l'un des trois principaux fondements de la Programmation Orientée Objet, le premier étant l'encapsulation vu précédemment et le dernier étant le polymorphisme qui sera abordé plus loin dans ce document.

L'héritage consiste en la création d'une nouvelle classe dite classe dérivée ou classe fille à partir d'une classe existante dite classe de base ou classe mère. L'héritage permet de :

- récupérer le comportement standard d'une classe d'objet (classe parente) à partir de propriétés et des méthodes définies dans celles-ci,
- ajouter des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée,
- modifier le comportement standard d'une classe d'objet (classe parente) en surchargeant certaines méthodes de la classe parente dans la classe dérivée.

Quand on parle **d'héritage**, c'est qu'on dit qu'une classe B hérite d'une classe A. La classe A est donc considérée comme la classe **mère** et la classe B est considérée comme la classe **fille**.

Soit deux classes `A` et `B`. Pour qu'un héritage soit possible, il faut que vous puissiez dire que `A` *est un* `B`. Par exemple, un magicien est un personnage, donc héritage. Un chien est un animal, donc héritage aussi.

Nous n'aurons pas à redéfinir les attributs et les méthodes de la classe mère dans la classe fille, nous pourrons nous en servir directement.

2. Procéder à un héritage

Pour procéder à un héritage (c'est-à-dire faire en sorte qu'une classe hérite des attributs et méthodes d'une autre classe), il suffit d'utiliser le mot-clé `extends`. Vous déclarez votre classe comme d'habitude (`class MaClasse`) en ajoutant `extends NomDeLaClasseAHeriter` comme ceci :

Code : PHP

```php
<?php
class Personnage // Création d'une classe simple.
{

}

class Magicien extends Personnage // Notre classe Magicien hérite
des attributs et méthodes de Personnage.
{

}
?>
```

3. Protection des données et des méthodes

En plus des mots-clés public et private décrits dans les chapitres précédents, il est possible d'utiliser un niveau de protection intermédiaire de propriétés et des méthodes par le mot-clé protected.

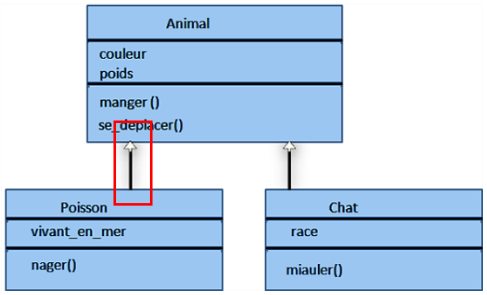
Un attribut (ou une méthode) **public** est accessible depuis n'importe quelle partie du code.

Un attribut (ou une méthode) **private** n'est accessible que depuis la même classe. Les classes qui hériteront n'ont pas accès aux éléments private.

Un attribut (ou une méthode) **protected** est accessible depuis la même classe ET les classes qui en héritent.

Rappel : Un attribut (ou une méthode) static est accessible uniquement depuis la classe, pas depuis les instances

4. Mode de représentation



5. Constructeur et héritage

La classe fille hérite de la classe mère pour l'appel à constructeur aussi

Soit :

- 1 classe-mère Personne qui prend 3 arguments (nom, prenom, age)
- 1 classe-fille Etudiant qui prend 5 arguments (nom, prenom, age, liste_matieres, notes)

```php
Class Personne{
    protected $_nom;
    protected $_prenom;
    protected $age;

    public function __construct($nom,$prenom,$age)
    {
        $this->_nom=$nom;
        $this->_prenom=$prenom;
        $this->_age=$age;
    }
}

Class Etudiant extends Personne{
    private $_matiere;
    private $_note;
    public function __construct($nom,$prenom,$age,$matiere,$note)
    {
        Parent::__construct($nom,$prenom,$age);
        $this->_matiere=$matiere;
        $this->_note=$note;
    }
```

6. Appel aux méthodes de la classe de base

Depuis l'extérieur de la classe fille

L'appel se fait comme si la méthode était défini dans la classe fille, c'est-à-dire $objetFille→methodeMere() ;

Depuis l'intérieur de la classe fille

Depuis une méthode par exemple, il suffit pour cela d'utiliser le mot-clé `parent` suivi du symbole double deux points suivi lui-même du nom de la méthode à appeler.

Parent :: methodeMere() ;

La méthode fille commencera par exécuter la méthode mère puis se poursuivra.

Notez que si la méthode parente retourne une valeur, vous pouvez la récupérer comme si vous appeliez une méthode normalement.

Code : PHP

```php
<?php
class A
{
  public function test()
  {
    return 'test';
  }
}

class B extends A
{
  public function test()
  {
    $retour = parent::test();

    echo $retour; // Affiche 'test'
  }
}
```

7. Surcharger les méthodes

Il s'agit ici de modifier la méthode de la classe mère pour l'adapter à la classe fille. Pour cela, il vous suffit de déclarer à nouveau la méthode et d'écrire ce que bon vous semble à l'intérieur.

Ainsi la méthode fille nommée comme la méthode mère pourrait faire tout autre chose.

Si vous surchargez une méthode, sa visibilité doit être la même que dans la classe parente ! Si tel n'est pas le cas, une erreur fatale sera levée. Par exemple, vous ne pouvez surcharger une méthode publique en disant qu'elle est privée.

VI. Polymorphisme

Le polymorphisme n'existe pas en PHP, cette notion n'est pas implémentée. Elle est moins utile que dans les langages très typés.

Le polymorphisme est le fait de pouvoir définir 2 méthodes avec le même nom mais des signatures différentes à l'intérieur d'une même classe.

VII. Classes abstraites et finales / Interfaces

1. Classe Abstraite

Définition : Classe qui ne peut pas être instancié. Cette classe sert de modèle aux classes qui en hérite.

Pour déclarer une classe abstraite, il suffit de faire précéder le mot-clé `class` du mot-clé `abstract`

Code : PHP

```php
<?php
abstract class Personnage // Notre classe Personnage est abstraite.
{

}

class Magicien extends Personnage // Création d'une classe Magicien
héritant de la classe Personnage.
{

}

$magicien = new Magicien; // Tout va bien, la classe Magicien n'est
pas abstraite.
$perso = new Personnage; // Erreur fatale car on instancie une
classe abstraite.
?>
```

2. Méthode abstraite

Si vous décidez de rendre une méthode abstraite en plaçant le mot-clé `abstract` juste avant la visibilité de la méthode, vous forcerez toutes les classes filles à écrire cette méthode. Si tel n'est pas le cas, une erreur fatale sera levée. Puisque l'on force la classe fille à écrire la méthode, on ne doit spécifier aucune instruction dans la méthode, on déclarera juste son prototype (visibilité + `function` + nomDeLaMethode + parenthèses avec ou sans paramètres **+ point-virgule**).

Code : PHP

```php
<?php
abstract class Personnage
{
  // On va forcer toute classe fille à écrire cette méthode car
chaque personnage frappe différemment.
  abstract public function frapper(Personnage $perso);

  // Cette méthode n'aura pas besoin d'être réécrite.
  public function recevoirDegats()
  {
    // Instructions.
  }
}

class Magicien extends Personnage
{
  // On écrit la méthode « frapper » du même type de visibilité que
la méthode abstraite « frapper » de la classe mère.
  public function frapper(Personnage $perso)
  {
    // Instructions.
  }
}
?>
```

Pour définir une méthode comme étant abstraite, il faut que la classe elle-même soit abstraite !

3. Classe finale

Le concept des classes et méthodes finales est exactement l'inverse du concept d'abstraction. Si une classe est finale, vous ne pourrez pas créer de classe fille héritant de cette classe.

Pour déclarer une classe finale, vous devez placer le mot-clé `final` juste avant le mot-clé `class`

Code : PHP

```php
<?php
// Classe abstraite servant de modèle.

abstract class Personnage
{

}

// Classe finale, on ne pourra créer de classe héritant de
Guerrier.

final class Guerrier extends Personnage
{

}

// Erreur fatale, car notre classe hérite d'une classe finale.

class GentilGuerrier extends Guerrier
{

}
?>
```

4. Méthode finale

Si vous déclarez une méthode finale, toute classe fille de la classe comportant cette méthode finale héritera de cette méthode **mais ne pourra la surcharger**

5. Interface

Une interface est une classe **entièrement** abstraite. Son rôle est de décrire un comportement à notre objet.

Une interface se déclare avec le mot-clé `interface`, suivi du nom de l'interface, suivi d'une paire d'accolades. C'est entre ces accolades que vous listerez des méthodes.

```
< ?php
Interface Mobile
{
        public function seDeplace($destination) ;
}
```

1. Toutes les méthodes présentes dans une interface doivent être publiques.
2. Une interface ne peut pas lister de méthodes abstraites ou finales.
3. Une interface ne peut pas avoir le même nom qu'une classe et vice-versa.

6. Implémenter une interface

```
< ?php
Class Personnage implements Mobile
{
        public function seDeplace($destination)
        {
                // code
        }
}
```

Une classe peut implémenter plusieurs interfaces, en plus de l'héritage

```
< ?php
Class Personnage extends Humain implements Mobile,Personne
{
```

Les interfaces peuvent héritées de plusieurs interfaces

```
<?php
interface iA
{
  public function test1();
}

interface iB
{
  public function test2();
}

interface iC extends iA, iB
{
  public function test3();
}
```