Tutorial sur la POO en php

Ce document est destiné à l'équipe de développement du projet de création d'un réseau social professionnel. Il a pour but d'introduire des notions de programmation orientée objet.

Sommaire

Tutorial sur la POO en php....1
Présentation de la POO (programmation orientée objet) en PHP....1
Les classes....1
Les types de visibilité....3
Notion d'objet....3
Créer un objet (instanciation d'une classe)....3
Constructeurs et destructeurs....5
Visibilité des propriétés et méthodes....6
L'héritage....7
Classes abstraites....8
Surcharger les méthodes....9
L'opérateur de résolution de portée :: 10
Les constantes de classe....10
Les méthodes statiques....11
Les attributs statiques....11
Surcharge et méthodes magiques....12
La surcharge....12
Conclusion....13

Présentation de la POO (programmation orientée objet) en PHP

La POO (programmation orientée objet) est une forme particulière de programmation destinée à faciliter la maintenance et la réutilisation / adaptation de vos scripts PHP. Elle consiste à représenter des objets (du monde réel ou non) sous une forme d'entités informatiques. On représente généralement un objet global par ce que l'on appelle une classe. Une classe va regrouper un ensemble de fonctions et de propriétés pouvant agir sur l'objet. Si on prend par exemple une voiture dans le monde réel, on peut modéliser une voiture par une classe "Voiture" qui aura comme propriétés le nombre de roues, le nombre de portes, etc ...

Les classes

Une classe regroupe des fonctions et des variables (appelées cette fois "attributs", car il s'agit des attributs d'une classe) qui interragissent avec l'objet. C'est à dire que pour un objet "voiture" par exemple, vous aurez une classe nommée "Voiture" et vous pourrez avoir une fonction qui modifie le niveau de carburant (le niveau de carburant étant un attribut de la classe que l'on ne peut modifier que via une fonction (appelée "méthode") qui ira modifier cet attribut). On appelle ce principe l'encapsulation des données, le but de l'encapsulation des données étant de ne pas pouvoir accéder aux données de l'objet directement mais via des fonctions (appelées ici "méthodes"). Chaque

attribut peut donc disposer de droits d'accès à l'extérieur de la classe. Nous verrons tout ceci au fur et à mesure que vous lisez cette page.

Avant toute chose, PHP 5 a amélioré le support objet de PHP par rapport à PHP 4, nous allons donc utiliser PHP 5 pour tous nos exemples. Si vous avez besoin de savoir comment PHP 4 fonctionne avec l'objet, je vous conseille d'aller jeter un oeil dans la rubrique codes sources orientés objet du site, qui vous permettra d'obtenir des sources de tous types.

Voici comment une classe peut être codée en PHP 5 :

```
<?php
     class Voiture
     {
          /**
          * Déclaration des attributs
          */

          private $niveau_carburant;
          private $nombre_portes;
          private $nombre_roues;

          /**
          * Cette méthode un peu spéciale est le constructeur, ell
e est exécutée lorsque vous "créez" votre objet. Elle doit
initialiser les attributs de la classe.
          */

          public function __construct()
          {
               $this->niveau_carburant = 50;
               $this->nombre_portes = 3;
               $this->nombre_roues = 4;
          }

          /**
          * Première méthode accessible par tous et modifiant le n
iveau de carburant
          */

          public function modifier_carburant(int $niveau)
          {
               $this->niveau_carburant = $niveau;
          }

          /**
          * Seconde méthode accessible à tous et modifiant le
nombre de portes
          */

          public function modifier_nb_portes(int $nb_portes)
          {
               $this->nombre_portes = $nb_portes;
          }
```

```
    }
?>
```

Les types de visibilité

Il existe 3 types de visibilité pour les méthodes et les attributs qui seront mis en place:

- **public** : ce type de visibilité nous laisse beaucoup de liberté. On peut accéder à l'attribut ou à la méthode de **n'importe où**. Toute classe fille aura accès aux éléments publics.
- **private** : ce type est celui qui nous laisse le moins de liberté. On ne peut accéder à l'attribut ou à la méthode **que depuis l'intérieur de la classe qui l'a créé**. Toute classe fille n'aura pas accès aux éléments privés.
- **Protected (intervient sur les notions d'héritage, nous y reviendrons plus loin dans le tutorial) :** le type de visibilité protected est en fait une petite modification du type private : il a **exactement** les mêmes effets que private, à l'exception que toute classe fille aura accès aux éléments protégés.

D'une manière générale, tous les attributs d'une classe seront déclaré en **private**. C'est ce qu'on appelle le principe d'encapsulation qui consiste à masquer les attributs. Si un attribut s'avérait être déclaré en **public** alors l'utilisateur aurait la possibilité de modifier les atttributs de la classe ce qui serait une catastrophe.

Cependant, pour pouvoir quand même accéder aux valeurs des attributs, nous avons inventé les accesseurs. Ce sont des méthodes simples qui ne font que renvoyer le contenu de l'attribut. Par convention, il faut précéder chacune de ces méthodes par **get**.

Notion d'objet

Imaginions que vous souhaitiez créer deux voitures dans votre code. Sans programmation orientée objet, on aurait pu stocker les voitures et leurs attributs dans un tableau, ce qui devient vite impossible à gérer. Avec la programmation orientée objet, vous allez pouvoir créer deux objets différents en deux lignes de code. Créer un objet se fait en "instanciant" une classe. Quand on instancie une classe, on crée une version de l'objet ayant des caractéristiques propres. Si vous créez un deuxième objet, il est indépendant du premier, bien qu'ils utilisent tous les deux la même classe (ici nos deux objets seront des voitures, ils utiliseront donc la même classe "Voiture", mais seront bien différents pour autant. Nous pourrons par exemple avoir une voiture ayant trois portes et la seconde 5 portes).

Créer un objet (instanciation d'une classe)

Voici comment on crée un objet Voiture en PHP (on supposera que vous avez inclus le fichier contenant la classe, ou alors que le code de la classe se trouve au dessus du code que vous allez voir) :

```
<?php
     $objet_voiture = new Voiture();
?>
```

La variable $objet_voiture représente l'objet qui est ici une voiture. Lorsque vous exécutez ce code, la méthode __construct() de la classe est exécutée. Comme il s'agit d'une fonction, elle peut prendre

elle aussi des paramètres. Tout dépend comment vous souhaitez coder votre classe, mais vous pourriez très bien avoir une fonction __construct() qui initialise les attributs en fonction des paramètres que vous lui fournissez. Voici ce que ça pourrait donner :

```php
<?php
	class Voiture
	{
		/**
		* Déclaration des attributs
		*/

		private $niveau_carburant;
		private $nombre_portes;
		private $nombre_roues;

		/**
		* Cette méthode un peu spéciale est le constructeur, elle est exécutée lorsque vous "créez" votre objet. Elle doit initialiser les attributs de la classe.
		*/

		public function __construct(int $nb_carburant, int $nb_portes, int $nb_roues = 4)
		{
			$this->niveau_carburant = $nb_carburant;
			$this->nombre_portes = $nb_portes;
			$this->nombre_roues = $nb_roues;
		}

		/**
		* Première méthode accessible par tous et modifiant le niveau de carburant
		*/

		public function modifier_carburant(int $niveau)
		{
			$this->niveau_carburant = $niveau;
		}

		/**
		* Seconde méthode accessible à tous et
modifiant le nombre de portes
		*/

		public function modifier_nb_portes(int $nb_portes)
		{
			$this->nombre_portes = $nb_portes;
		}
	}
?>
```

Lorsque vous créerez l'objet voiture, vous allez pouvoir sans passer par les méthodes appropriées lui fixer un niveau de carburant, un nombre de portes et un nombre de roues (par défaut 4). Voici deux

façons de créer l'objet :

```
<?php
     $objet_voiture = new Voiture(50, 3); //50 : niveau de carbura
nt et 3 portes, on a pas besoin de spécifier le nombre de roues ca
r il est de 4 par défaut

     $autre_voiture = new Voiture(10, 5, 6); //10 : niveau de carb
urant, 5 portes et 6 roues
?>
```

Il est important de signifier que les objets $objet_voiture et $autre_voiture sont deux objets différents qui peuvent avoir leurs propriétés propres. Vous commencez peut-être maintenant à comprendre avec quelle simplicité vous allez pouvoir créer autant d'objets que vous le souhaitez ;)

Constructeurs et destructeurs

Les constructeurs et destructeurs sont des méthodes particulières. D'une part, elle commencent par deux signes "underscores" accolés (touche 8 du pavé alphanumérique). D'autre part, elles sont exécutées à des moments précis.

Le constructeur est appelé automatiquement quand vous créez votre objet. Généralement, cette méthode sert à donner une valeur de départ aux différents attributs de la classe pour vous permettre de "construire" l'objet. Ce n'est cependant pas une obligation et vous pouvez très bien ne rien mettre dans cette méthode.

Le destructeur est appelé à la fin d'exécution de votre script. La méthode s'appelle cette fois-ci __destruct().

```
<?php
     class Voiture
     {
          /**
          * Déclaration des attributs
          */

          private $niveau_carburant;
          private $nombre_portes;
          private $nombre_roues;

          /**
          * Cette méthode un peu spéciale est le constructeur, ell
e est exécutée lorsque vous "créez" votre objet. Elle doit initial
iser les attributs de la classe.
          */

          public function __construct(int $nb_carburant, int $nb_p
ortes, int $nb_roues = 4)
          {
               $this->niveau_carburant = $nb_carburant;
               $this->nombre_portes = $nb_portes;
               $this->nombre_roues = $nb_roues;
          }
```

```
            /**
            * Destructeur, appelé quand l'objet est détruit
            */

            public function __destruct()
            {
                  echo 'L\'objet a été détruit';
            }
      }
?>
```

Visibilité des propriétés et méthodes

PHP 5 introduit la notion de visibilité de méthodes et d'attributs. Chaques attributs et méthodes peuvent se voir attribuer un droit d'accès. Le principe de l'encapsulation voudrait que l'on mette tous les attributs uniquement modifiables et accessibles à l'intérieur de la classe, et les méthodes accessibles de l'extérieur. En pratique, ce n'est pas toujours le cas.

Les trois mots permettant de gérer les accès sont ceux-ci :

- public : n'importe qui a accès à la méthode ou à l'attribut demandé.
- protected : seule la classe ainsi que ses sous classes éventuelles (classes héritées, on verra ce que c'est plus loin).
- private : seule la classe ayant défini l'élément peut y accéder.

Pour la classe Voiture, vous pouvez constater que les attributs ne sont pas modifiables à l'extérieur de la classe, il faut passer par les méthodes appropriées.

Pour accéder à un attribut d'une classe, on utilise ce code : $objet->attribut

```
<?php
      class Voiture
      {
            /**
            * Déclaration des attributs
            */

            private $niveau_carburant;
            public $nombre_portes;
            private $nombre_roues;

            /**
            * Cette méthode un peu spéciale est le constructeur, ell
e est exécutée lorsque vous "créez" votre objet. Elle doit initial
iser les attributs de la classe.
            */

            public function __construct(int $nb_carburant, int $nb_p
ortes, int $nb_roues = 4)
            {
                  $this->niveau_carburant = $nb_carburant;
                  $this->nombre_portes = $nb_portes;
                  $this->nombre_roues = $nb_roues;
            }
```

```
    }
?>
```

Maintenant, on instancie la classe :

```
<?php
    $voiture = new Voiture(50, 3);

    echo $voiture->nombre_portes; //va afficher "3" car
l'attribut est en accès public

    echo $voiture->nombre_roues; //Erreur, on ne peut pas y accéd
er car l'attribut est en accès privé !
?>
```

Notez que l'on ne met pas de signe $ pour accéder ou définir des valeurs aux attributs d'une classe. Le mot-clé $this est un peu spécial et il désigne la classe courante.

L'héritage

L'héritage consiste à utiliser une classe parente et une ou plusieurs classes filles qui héritent des propriétés de la classe parente. Par exemple, si vous avez une classe **Vehicule**, vous pouvez avoir une classe **Voiture** qui hérite de certaines propriétés de la classe **Vehicule**, ainsi qu'une autre classe **Moto** qui va hériter de certaines propriétés de la classe **Voiture** tout en ajoutant des propriétés propres. Le mot utilisé pour dire à PHP qu'une classe hérite d'une autre est le mot-clé **extends**. Prenons l'exemple d'une classe Vehicule :

```
<?php
    class Vehicule
    {
        /**
        * Déclaration des attributs
        */

        protected $prix; //On souhaite que les classes qui en hé
ritent puissent y accéder

        /**
        * Cette méthode un peu spéciale est le constructeur, ell
e est exécutée lorsque vous "créez" votre objet. Elle doit initial
iser les attributs de la classe.
        */

        public function __construct(int $prix_vehicule)
        {
            $this->prix = $prix_vehicule;
        }

        /**
        * Cette méthode permet de modifier le prix du véhicule
        */

        public function modifier_prix;(int $nouveau_prix)
        {
```

```
                $this->prix = $nouveau_prix;
            }
        }
?>
```

Comme vous pouvez le constater, on a mis dans la classe Vehicule tout ce qui sera commun aux différents véhicules que nous allons pouvoir créer. Ici, j'ai mis un attribut "prix" car une voiture ou une moto ont toutes les deux un prix. On met donc tout ce que les véhicules ont en commun dans une même classe, et cette classe sera étendue par d'autres classes plus spécifiques.

Classes abstraites

On a vu que l'on pouvait instancier n'importe quelle classe afin de pouvoir exploiter ses méthodes. Sachez qu'il est possible aussi d'empêcher **quiconque d'instancier telle classe**. C'est le principe d'abstraction.

En déclarant une classe comme **abstraite**, on ne pourra pas se servir **directement** de la classe. La seule façon d'exploiter ses méthodes sera de créer une ou plusieurs classes **héritant de la classe abstraite**.

Prenons l'exemple d'un personnage et de ses classes filles. Imaginons que nous ne créerons **jamais** d'objet **Personnage**, mais uniquement des objets **Magicien**, **Guerrier**, **Brute**, etc. Dans ce cas, à quoi cela nous servirait d'instancier la classe **Personnage** si notre but est de créer un tel type de personnage ?

On va donc considérer la classe **Personnage** comme étant une classe **modèle** dont toute classe fille possèdera les méthodes et attributs.

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

```
<?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.
?>
```

Il est aussi possible de définir une méthode abstraite. Dans ce cas, 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**). Comme ceci:

- **abstract** public function frapper ($perso);

Il est aussi possible de définir une classe comme « **final** ». Ce qui signifie qu'on ne pourra créer de classe fille héritant de la classe définie comme final. Final existe aussi pour les méthodes. Une classe fille de la classe comportant la méthode final pourra hériter de cette méthode mais **ne pourra être surchargée.**

Le mot-clé pour une méthode ou classe finale est « **final** ». Cet aspect de la POO est peu important aussi je n'irai pas plus en détail.

Surcharger les méthodes

Voici maintenant une classe Voiture qui étend les propriétés de la classe Vehicule, nous allons surcharger le constructeur avec de nouvelles propriétés qui cette fois seront propres à la classe Voiture :

Attention: 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.

```
<?php
	class Voiture extends Vehicule
	{
		/**
		* Déclaration des attributs
		*/

		private $climatisation;

		/**
		* Constructeur de la classe Voiture
		*/

		public function __construct(int $prix_vehicule, bool $cl
im)
		{
			parent::__construct($prix_vehicule); //On appelle l
e constructeur de la classe Vehicule en lui fournissant le prix
			$this->climatisation = $clim;
		}
	}
?>
```

Pour utiliser une méthode parente dans une classe fille, on utilise le mot-clé **parent** suivi de l'opérateur de résolution de portée « **::** » que nous expliciterons plus loin.

Voici ce que ça donne au niveau de l'instanciation :

```
<?php
	$voiture = new Voiture(17000, TRUE); //On crée une voiture va
lant 17000 euros et ayant la climatisation

	$voiture->modifier_prix(15000); //On peut modifier le prix de
```

```
la voiture
?>
```

Comme vous pouvez le constater, on peut accéder à la classe parente comme si on accédait à la classe Voiture (avec la méthode **modifier_prix()**). Si vous aviez mis la méthode **modifier_prix()** en accès privé, vous n'auriez pu effectuer la modification directement. Le niveau de protection le plus restrictif permettant cette modification est le niveau **protected**.

L'opérateur de résolution de portée ::

Cet opérateur de résolution de portée, appelé « double deux points » est utilisé pour appeler des attributs et méthodes dits **statiques** et des **constantes.** Ces attributs sont dits constants car leur valeur ne change pas

Les constantes de classe

Pour déclarer une constante, vous devez faire précéder son nom du mot-clé const. Une constante ne prend pas de $ devant son nom ! Ainsi, voilà comment créer une constante :

```
<?php

    class Personnage
    {
        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;

        public function __construct ($forceInitiale)
        {
            $this->force = $forceInitiale;
        }

    }
?>
```

Par convention, le nom de la constante est écrit en majuscule.

Ce qui permet par la suite de faire lors de l'instanciation d'un objet de la classe Personnage:

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

On note l'utilisation de l'opérateur de résolution de portée.

Les méthodes statiques

les méthodes statiques sont des méthodes pouvant être appelées sans avoir besoin de créer d'objet. On pourra toujours les appeler depuis des objets déjà créés, mais aussi « à la volée » via l'opérateur :: .Pour déclarer une méthode statique, vous devez faire précéder le mot-clé function du mot-clé static, après le type de visibilité:

```
<?php
    class Personnage
    {
        private $force;
        private $localisation;
        private $experience;
        private $degats;

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

        public function __construct ($forceInitiale)
        {
            $this->force = $forceInitiale;
        }

        public static function parler()
        {
            echo 'Je vais tous vous tuer !';
        }
    }
?>
```

et on pourra faire:

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

ou encore:

```
<?php
    $perso = new Personnage (Personnage::FORCE_GRANDE);
    $perso->parler();
?>
```

Les attributs statiques

Le principe est le même. La déclaration d'un attribut statique se fait en faisant précéder son nom du mot-clé static, comme ceci :

```
<?php
    class Personnage
    {
        private $force;
        private $localisation;
        private $experience;
        private $degats;

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

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

        public function __construct ($forceInitiale)
        {
            $this->force = $forceInitiale;
        }

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

Utilité: une méthode statique n'a pas accès aux attributs de la classe mais **elle peut demander un attribut statique** grâce au mot-clé **self** suivi du « double deux points »!

Surcharge et méthodes magiques

Une méthode magique est une méthode qui, si elle est présente dans votre classe, sera appelée lors de tel ou tel événement. Si la méthode n'existe pas et que l'événement est exécuté, aucun effet « spécial » ne sera ajouté, l'événement s'exécutera normalement. Le but des méthodes magiques est d'intercepter un événement, dire de faire ça ou ça et retourner une valeur utile pour l'événement si besoin il y a.

Une des méthodes magiques les plus connue et utilisée fréquemment est celle du constructeur:

__construct

cette méthode est appelée lors de l'**évènement** « création de l'objet ».

Voici les contextes dans lesquels les méthodes magiques interviennent:

1. Constructeur et Destructeur [facile]
2. L'auto-chargement de classes [facile]
3. Surcharge de propriétés [intermédiaire]
4. Surcharge de méthodes [intermédiaire]
5. Sérialisation intelligente [difficile]
6. Affichage simplifié d'un objet [facile]
7. Clonage intelligent d'un objet [intermédiaire]

Le lien suivant entre dans les détails et donne quelques exemples d'utilisation de ces méthodes:

http://www.phpfrance.com/tutoriaux/index.php/2006/05/11/43-les-methodes-magiques-avec-php-5

La surcharge

La surcharge en PHP permet de créer dynamiquement des propriétés et des méthodes. Ces entités dynamiques sont traitées via les méthodes magiques (vu précédemment) établies dans une classe pour diverses types d'actions.

Pour simplifier, la surcharge d'attributs ou méthodes consiste à prévoir le cas où l'on appelle un attribut ou méthode qui n'existe pas ou du moins, auquel on n'a pas accès (par exemple, si un

attribut ou une méthode est privé(e)).

Attention: L'interprétation PHP de la "surcharge" est différente de la plupart des langages orientés objet. La surcharge, habituellement, fournit la possibilité d'avoir plusieurs méthodes portant le même nom mais avec une quantité et des types différents d'arguments.

Voici quelques méthodes magiques utilisées:

mixed indique qu'un paramètre peut accepter plusieurs (mais pas nécessairement tous) types.

1. void **__set** (string `$name` , mixed `$value`)
2. mixed **__get** (string `$name`)
3. bool **__isset** (string `$name`)
4. void **__unset** (string `$name`)

- **__set()** est sollicitée lors de l'écriture de données vers des propriétés inaccessibles.
- **__get()** est sollicitée pour lire des données depuis des propriétés inaccessibles.
- **__isset()** est sollicitée lorsque isset() ou la fonction empty() sont appelés avec des propriétés inaccessibles.
- **__unset()()** est sollicitée lorsque unset() est appelée avec des propriétés inaccessibles.

L'argument *$name* est le nom de la propriété qui interagit. L'argument *$value* de la méthode **__set()** spécifie la valeur de la propriété *$name* qui doit être définie.

La surcharge des propriétés ne fonctionne que dans les contexte objet. Ces méthodes magiques ne seront pas lancées dans un contexte statique. Par conséquent, ces méthodes ne devraient pas être déclarées comme statiques.

Le lien ci-après montre des exemples d'utilisation de ces méthodes:

http://www.siteduzero.com/tutoriel-3-147171-les-methodes-magiques.html

Conclusion

Ce tutorial fait le tour des principaux aspects de la programmation orientée objet. Il reprend de nombreux exemples et explications des sites suivants qui vous seront utiles pour des explications plus en détails:

- **Le site du zéro** (de bonnes explications adaptées à une compréhension rapide surtout pour les novices avec des exemples clairs):

http://www.siteduzero.com/tutoriel-3-147182-introduction-a-la-poo.html

- **Le manuel PHP** (disponible en français, comprend tous ce qu'il existe sur la POO en php, mais des exemples en anglais et des explications réservées à des personnes relativement à l'aise avec le concept de POO et ayant de bonnes notions en php):

http://fr.php.net/manual/fr/language.oop5.php

- **Php France** (un tutorial bien réalisé sur les méthodes magiques):

http://www.phpfrance.com/tutoriaux/index.php/2006/05/11/43-les-methodes-magiques-avec-php-5