La programmation orientée objet en PHP

par Wes Shell (Auteur) Joris Crozier (Traducteur)

Date de publication : 03/11/2009

Dernière mise à jour :

La programmation orientée objet est une technique adaptée en premier lieu par les développeurs de jeux vidéos et ensuite reprise par les développeurs traditionnels applicatifs et web qui utilisent des langages tels que PHP, JSP et ASP.NET. Les programmeurs procéduraux ont trouvé le passage vers la programmation orientée objet tel un challenge dans leur carrière car cela change totalement la manière de penser ses données, et les opérations que le programme réalise.

I - Introduction

J'étais l'un de ces programmeurs "old school" qui ont lutté pour réaliser que ce que je venais de découvrir était en fait un concept simple. J'espère que mon expérience va aider à expliquer comment passer du style procédural au style objet. Mais pour commencer en POO, vous allez devoir connaitre ces concepts :

- Les classes
- L'instanciation d'objets
- L'encapsulation
- L'héritage

Deux autres concepts en POO que nous n'aborderons pas dans cet article mais que nous aurons l'occasion de voir sont les suivants :

- Les classes abstraites et statiques
- Le polymorphisme

II - Les classes

Voir aussi ***Créer une classe en PHP***

Les classes sont les squelettes des objets. Quand vous faites de la POO vous devez commencer à penser votre programme ou votre application comme si c'était un objet du monde réel faisant des actions réelles. Les classes sont l'endroit où vous définissez les caractéristiques et les capacités de ces choses.

Bien, vous dites-vous, mais que cela veut-il dire ? Pour mieux comprendre, vous devez prendre du recul sur votre projet et jeter un il sur ce qui est autour. La meilleure façon de commencer est d'écrire une explication de votre programme, ce qu'il fait, comment il marche, et comment on interagit avec. Pour un gros programme, cela peut prendre beaucoup de temps, mais une fois cela fait, vous vous en féliciterez.

Nous allons utiliser un exemple très simple :

Ce programme est un zoo. Quand le programme démarre, 3 animaux sont montrés à l'utilisateur. Un chat blanc, un chien marron et un koala gris. L'utilisateur sera en mesure de sélectionner un animal et de cliquer sur "parler " qui aura pour effet de faire parler l'animal. Une fois sélectionné, le chat dira "miaou ", le chient "wouf ", et le koala dira "Bonjour monsieur ".

La première chose que nous devons faire est d'identifier tout ce dont nous avons besoin pour notre programme. Le meilleur moyen de le faire simplement est de lister tous les noms du paragraphe précédent :

Programme, zoo, animaux, utilisateur, chat, chien, koala.

Ensuite nous devons éliminer les éléments évidents. Programme et utilisateur ne sont pas inclus car l'utilisateur est en dehors du programme, et programme est ce qui sera composé de toutes nos classes.

Zoo, animaux, chat, chien, koala.

Vous vous demandez sûrement pourquoi nous avons "animaux" dans notre liste sachant que chat, chien et koala représentent déjà des animaux. La réponse est le concept que nous appelons héritage, que nous verrons plus loin. Pour le moment nous allons ignorer " animaux ", car notre chien, notre chat et notre koala sont nos animaux.

La chose suivante à faire est de repérer les caractéristiques de nos objets. Ceci peut être fait en reprenant le paragraphe précédent et en mettant en avant tous les adjectifs qui décrivent nos objets.

*Ce programme est un **zoo**. Quand le programme démarre, 3 **animaux** sont montrés à l'utilisateur. Un **chat blanc**, un **chien marron** et un **koala gris**. L'utilisateur sera en mesure de sélectionner un animal et de cliquer sur "parler" qui aura pour effet de faire parler l'animal. Une fois sélectionné, le **chat** dira "miaou ", le **chien** "wouf", et le **koala** dira "Bonjour monsieur ".*

- Le chat - {blanc}
- Le chien - {marron}
- Le koala - {gris}

Une fois que les caractéristiques sont repérées, nous devons repérer les actions que les objets peuvent faire. On le fait en mettant en avant les verbes attachés à nos objets :

*Ce programme est un zoo. Quand le programme démarre, 3 animaux sont **montrés** à l'utilisateur. Un chat blanc, un chien marron et un koala gris. L'utilisateur sera en mesure de sélectionner un animal et de cliquer sur "parler" qui aura pour effet de faire **parler** l'animal. Une fois sélectionné, le chat **dira** "miaou", le chien "wouf", et le koala **dira** "Bonjour monsieur".*

- Zoo - {}(montré)
- Le chat - {blanc}(dira)
- Le chien - {marron}(dira)
- Le koala - {gris}(dira)

Maintenant regardez nos actions. Pour que les choses aient un peu plus de sens nous devons modifier légèrement notre descriptif. En commençant par le zoo ; le zoo doit "montrer" trois animaux. Donc nous changeons "montré "par "montrer". Ensuite les animaux "diront" quelque chose alors que nous voulons qu'ils "parlent". Donc nous allons changer "dira" en "parle".

- Zoo - {}(montrer)
- Le chat - {blanc}(parle)
- Le chien - {marron}(parle)
- Le koala - {gris}(parle)

Ce procédé peut requérir pas mal de réflexion et pas mal de jugement de votre part pour avoir les bonnes formulations, mais la chose importante est que toutes vos actions sont en fait représentées dans vos classes. Pour voir le parallèle entre les mots collectés dans le paragraphe et le code, vous pouvez jeter un il à l'exemple suivant. Si vous êtes déjà familiarisé avec la syntaxe vous pouvez tenter de créer le code suivant par vous-même. Si vous êtes débutant, je vous encourage à lire l'article "**Créer une classe en PHP**".

```
class Zoo
{
    var $_animaux = Array();

    function Montrer()
    {
 echo "<h2>Les animaux dans le zoo:</h2>";

 foreach ($this->_animaux as $animal)
 {
      echo "<a href='?action=parler&animal=" . get_class($animal)
                . "'>" . get_class($animal) . "</a><br />" ;
        }
    }

    function __construct()
    {
  $this->_animaux["Chat"] = new Chat("Blanc");
 $this->_animaux ["Chien"] = new Chien("Marron");
 $this->_animaux ["Koala"] = new Koala("Gris");
    }
}

class Chat
```

```
{
    var $_couleur;

    function Parler()
    {
  return "Miaou";
    }

    function __construct($couleur)
    {
        $this->_couleur = $couleur;
    }
}

class Chien
{
    var $_ couleur;

    function Parler()
    {
 return "Wouf";
    }

    function __construct($couleur)
    {
 $this->_couleur = $couleur;
    }
}

class Koala
{
    var $_couleur;

    function Parler()
    {
  return "Bonne journée monsieur";
    }

    function __construct($couleur)
    {
 $this->_couleur = $couleur;
    }
}

$zoo = new Zoo();

if (isSet($_REQUEST['action']))
{
    $animal = $zoo->_animaux[$_REQUEST['animal']];

    echo "Le " . get_class($animal)
        . " ". $animal->_couleur. " dit '" . $animal->Parler() . "'";
}

$zoo->Montrer();
```

III - Instanciation des objets

L'instanciation est le fait de créer une nouvelle instance d'un objet depuis une classe. Avec tous ces grands mots cela peut sembler effrayant mais c'est en réalité le concept le plus simple que vous devez apprendre. Quand vous créez une classe, en somme vous définissez le squelette de vos objets. Ce code en lui-même ne peut rien faire sans vos objets (les classes statiques sont une exception dont nous discuterons plus tard).

Une fois vos objets instanciés, toutes les caractéristiques et les actions de l'objet sont stockées dans celui-ci, et peuvent être utilisées par votre programme. Le mot clef **new** est en PHP le mot qui vous permet d'instancier. Regardons la fin du code de l'exemple précédent. Regardons comment nous créons une nouvelle instance de la classe Zoo.

```
$zoo = new Zoo();
```

Maintenant que nous avons une instance de notre zoo, nous pouvons regarder les $_animaux de celui-ci et appeler la méthode Montrer(). Ceci se fait en utilisant un opérateur spécial -> après la variable qui stocke notre objet.

```
$animaux = $zoo->_animaux;

$zoo->Montrer();
```

La seule autre chose que vous devez vraiment savoir à propos de l'instanciation est que vous pouvez avoir plus d'une seule instance d'un même objet au même moment. Chaque instance pouvant comprendre différentes valeurs pour leurs caractéristiques. Ce qui veut dire que l'un de nos zoos peut avoir un chat blanc tandis qu'un autre peut avoir un chat noir. Dans notre exemple, pour réaliser ceci, nous aurions du changer la méthode du constructeur de la classe zoo et utiliser une autre méthode pour ajouter des animaux dedans. Essayez donc de le faire.

IV - L'encapsulation

L'encapsulation est la manière d'abstraire les caractéristiques ou des champs de vos objets pour n'importe quoi utilisant ceux-ci. Le but de cette man□uvre est de contrôler strictement la manière dont vos données sont manipulées, peuplées et accessibles. Par défaut tous nos champs et méthodes sont ce que nous appelons "publique ", ce qui veut dire qu'elles peuvent toutes être accédées depuis les instances de classes. $_animaux est un champ publique qui peut être accessible depuis la classe Zoo.

Cela marche bien car nous pouvons facilement récupérer nos animaux de notre zoo et avoir des informations sur ceux-ci. Le problème avec ça est que nous pouvons aussi écraser les données des animaux de notre zoo tant que l'on peut y accéder :

```
$zoo->_animaux = "Hello World";
```

Et maintenant, nos animaux ne sont plus des animaux du tout, ce sont simplement une chaîne de caractères. Ceci pouvant être à la tête de toute sorte de problèmes dans notre programme si ça venait à se produire. D'une part, $_animaux doit être un tableau donc autant de fois que nous voudrons accéder à cette valeur en tant que tableau nous obtiendrons une erreur. De plus, notre programme s'attend à pouvoir extraire des objets de ce tableau pour accéder à leur méthode Parler(). Pour se prémunir de ce genre de chose nous utilisons l'encapsulation pour cacher notre champs $_animaux de tous dans un premier temps et ensuite pour fournir un moyen d'y accéder en utilisant certaines règles à respecter.

Les mots clefs que vous allez utiliser sont 'public', 'private' et 'protected'. Comme mentionné avant, toutes les méthodes et les champs de classe sont 'public' par défaut. C'est toujours une bonne idée de libeller vos champs publics avec le mot clef public, comme ça votre code est plus simple à lire et facile à comprendre pour les autres.

Private veut dire que ce champ peut seulement être utilisé à l'intérieur de la classe et par les méthodes de celle-ci. Ce qui veut dire que si $_animaux est en private je ne peux plus y accéder via une instance de $zoo, mais je peux toujours l'utiliser depuis l'intérieur de la classe. Cela vous permet de protéger $_animaux contre les gens qui voudraient changer sa valeur pour une chaîne de caractères.

```
class Zoo
{
    private $_animaux = Array();
    ...
}
```

Maintenant si nous essayons d'accéder à $_animaux via une instance de $zoo, cela devrait renvoyer une erreur. C'est bien, cela permet de protéger $_animaux contre les gens qui voudraient y renseigner autre chose qu'un tableau d'animaux. Maintenant nous devons fournir aux autres un moyen d'accéder à la valeur de $_animaux. Pour ce faire, nous allons créer une méthode dans notre classe qui va retourner la valeur de $_animaux.

```
public function Animaux()
{
    return $_animaux;
}
```

Maintenant nous avons une méthode pour extraire nos animaux de notre zoo. Si l'on appel notre méthode publique Animaux() depuis une instance de $zoo, cela va retourner de manière sécurisée notre tableau d'animaux stocké dans notre classe, et personne ne peut modifier la valeur contenue dans notre champs $_animaux.

```
$zoo = new Zoo();
$animaux = $zoo->Animaux();
```

Maintenant les choses ont l'air d'être un peu plus stable. Mais il reste un problème, nous n'avons aucun moyen d'ajouter des animaux dans notre zoo. Nous avons créé une méthode capable de retourner la valeur stockée. Ce qui veut dire que nous sommes capables de créer une méthode qui va pouvoir ajouter des animaux dans notre zoo.

```
public function AjouterAnimal($animal)
{
    if(get_class($animal) != "Chat"
        && get_class($animal) != "Chien"
        && get_class($animal) != "Koala")
        {
            throw new Exception("$animal n'est pas un type d'animal autorisé");
 }

    $this->_animaux[] = $animal;
}
```

Maintenant, nous pouvons donc ajouter d'autres animaux dans notre zoo, et nous savons que les animaux qui peuvent être ajoutés doivent être des chats, des chiens ou des koalas, qui sont tous trois des objets. Maintenant il n'y a aucune chance que notre champs $_animaux contiennent autre chose qu'un tableau d'animaux.

Ce processus montre un exemple d'encapsulation. Nous avons encapsulé notre champ $_animaux. C'est en général une bonne idée d'encapsuler ces champs, surtout si vous voulez donner un accès complets a ceux-ci. Ce n'est pas seulement une bonne pratique de développement mais c'est aussi moyen de rendre votre code plus facile à lire.

V - L'héritage

L'héritage dans la POO est le fait pour une classe fille de prendre toutes les caractéristiques de sa classe mère ou de base. Ce qui veut dire que tous les champs et les méthodes d'une classe sont implémentées dans une classe qui étend celle-ci. L'héritage est utilisé quand plusieurs objets partagent les mêmes caractéristiques mais possèdent aussi leurs propres éléments.

Nous avons cette situation dans notre application Zoo. Tous les animaux partagent la caractéristique couleur. Ils peuvent tous aussi être ajoutés dans le champ $_animaux. Leur plus gros point commun est d'être des animaux. Souvenez vous de notre phrase originale. Nous enlevons tous les animaux de la phrase sachant que Chat, Chien et Koala en sont. C'est ici que nous allons donc profiter des bienfaits de l'héritage.

Pour ajouter nos animaux comme décrit dans notre phrase nous créons simplement une classe Animal. Cette classe contient toutes les caractéristiques et les méthodes communes qu'une classe aura besoin quand elle héritera de la classe Animal. Dans ce cas notre champs commun est $_couleur et notre méthode commune est Parler().

```
class Animal
{
    private $_couleur;
    function Parler()
    {
  return "Je dis : bien!";
    }
    function __construct($couleur)
```

```
    {
 $this->_couleur = $couleur;
    }
}
```

Avec ce code nous pouvons maintenant créer un animal générique qui possède une couleur et qui peut parler. Ce n'est pas un grand accomplissement cependant nous pouvons maintenant refaire tous nos objets "animaux" et les hériter des caractéristiques de la classe Animale. Pour ce faire nous avons simplement besoin du mot clef "extends" après le nom de notre classe suivi du nom de la classe qu'elle étend.

```
class Chat extends Animal
{

}
```

Maintenant quand nous créons un objet Chat, il a automatiquement toutes les caractéristiques de la classe générique Animal. Comme les champs et les méthodes sont hérités nous n'avons pas besoin de les redéfinir dans notre classe. Cela laisse notre classe Chat assez nue, mais avec les mêmes fonctionnalités qu'elle avait auparavant.

Il y'a juste un petit problème. Maintenant quand on demande au chat de parler, cela appelle la méthode Parler() de la classe parente et il va dire "Je dis : bien !" au lieu de "Miaou". C'est là que le principe de surcharge entre en jeux. La surcharge est le fait de remplacer les caractéristiques héritées par celle de son propre cru. Nous pouvons utiliser cette technique pour surcharger la méthode Parler() de la classe Animal pour notre classe Chat.

```
class Chat extends Animal
{
    function Parler()
    {
  return "Miaou";
    }
}
```

Maintenant notre chat va utiliser sa propre méthode Parler() quand elle sera appelée depuis son instance. Bien, mais pourquoi tout ces ennuis juste pour écraser les méthodes ? La meilleure raison est qu'il reste des caractéristiques comme la couleur qui ne seront pas surchargées et qui profiteront à tous les animaux. Ce la nous permet de na pas avoir à écrire à chaque fois le même code.

Cela fournit aussi d'autres avantages, jeton un □il sur notre méthode AjouterAnimal() de notre classe Zoo. Actuellement, elle nous permet seulement d'ajouter des Chat, des Chiens et des Koalas. En fait nous pourrions très bien ajouter notre nouvel objet Animal sans pour autant ajouter de conditions dans cette méthode. Imaginez le nombre de conditions si nous venions à accepter d'autres animaux dans notre Zoo ? Sans décrire tout ce qui en découle, nous devrions revenir sur notre code à chaque ajout d'un nouvel animal.

En ayant des objets hérités de notre classe Animal, nous pouvons maintenant procéder à une vérification plus générique avant de les ajouter au Zoo. Nous le faisons en utilisant la fonction PHP is_subclass_of(). Cette fonction renverra true si l'objet passé en paramètre est directement ou indirectement hérité de la classe Animal. Ce qui veut dire que nous pourrions créer une classe Panthère qui hérite de Chat qui hérite de Animal. Pas besoin de se soucier du nombre de fois où une classe est héritée, sa base est toujours la classe Animal et peut donc entrer dans notre Zoo.

```
public function AjouterAnimal($animal)
{
    if(is_subclass_of($animal, "Animal"))
    {
  $this->_animaux[get_class($animal)] = $animal;
    }
}
```

Maintenant nous pouvons créer toutes les sortes d'animaux que nous voulons pour notre zoo et faire hériter leurs caractéristiques de la classe Animal. Nous savons que quel que soit l'animal que nous voulons, il aura toutes les

caractéristiques requises pour faire fonctionner notre programme en supprimant totalement les erreurs de run-time. Ce concept s'appelle l'héritage.

VI - Conclusion

Avec ce que vous avez appris dans cet article, vous êtes en mesure de créer des applications basiques en utilisant les principes de la POO. Quand vous serez à l'aise avec ces concepts, je vous recommande de lire les articles sur les classes statiques, abstraites et sur le polymorphisme.

VI - Le code complet

```php
class Zoo
{
    private $_animaux = Array();

    public function Animaux()
    {
        return $this->_animaux;
    }

    public function AjouterAnimal($animal)
    {
        if(!is_subclass_of($animal, "Animal"))
        {
            throw new Exception("$animal n'est pas un objet Animal.");
        }

        $this->_animaux[get_class($animal)] = $animal;
    }

    public function Montrer()
    {
        echo "<h2>Animaux dans le zoo:</h2>";

        foreach ($this->_animaux as $animal)
        {
            echo "<a href='?action=parler&animal=" . get_class($animal) .
                "'>" . get_class($animal) . "</a><br />" ;
        }
    }

    function __construct()
    {
        $this->_animaux["Chat"] = new Chat("Blanc");
        $this->_animaux["Chien"] = new Chien("Marron");
        $this->_animaux["Koala"] = new Koala("Gris");
    }
}

class Animal
{
    public $_couleur;

    public function parler()
    {
        return "Je dis : bien!";
    }

    function __construct($couleur)
    {
        $this->_couleur = $couleur;
    }
}

class Chat extends Animal
{
    function Parler()
```

```
    {
        return "Miaou";
    }
}

class Chien extends Animal
{
    function Parler()
    {
        return "Woof";
    }
}

class Koala extends Animal
{
    function Parler()
    {
        return "Bonne journée monsieur";
    }
}

$zoo = new Zoo();

$monAnimal = new Animal("Jaune");
$zoo->AjouterAnimal($monAnimal);

if (isSet($_REQUEST['action']))
{
    $animaux = $zoo->Animaux();
    $animal = $animaux[$_REQUEST['animal']];

    echo "Le " . get_class($animal)
        . " ". $animal->_couleur. " dit '" . $animal->Parler() . "'";

}

$zoo->Montrer();
```

VII - Liens

Vous pouvez aussi aller voir mes autres **traductions**.