Programmation orientée objet php 5

1 Souces imprimées

1.1 Ouvrage

N. Borde A; MarhinM. Thévenet, *PHP 5,* Micro Application Référence, 2004, isbn : 2-7429-3217-8

1.2 Sites

1.2.1 Sites généraux

http://www.commentcamarche.net/poo/objet.php3
http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_objet
http://hdd34.developpez.com/cours/artpoo/
http://perso.orange.fr/isabelle.thieblemont/poo/pooch2.html

1.2.2 Sites php 5

http://hachesse.developpez.com/objetphp/#intro
http://www.ac-creteil.fr/util/programmation/scripts/php-pgr-objet.php
http://www.phpteam.net/progresser/php5-nouveaut-s-et-enjeux-4.html

2 Historique

2.1 En terme de programmation, il y a eu plusieurs évolutions successives.

Une des principales fut la programmation structurée, dont le principe premier était de diviser un programme en sous-programmes, afin de pouvoir en gérer la complexité. Ce type de programmation tient avant tout compte des traitements et peut être résumé par la question "Que doit faire le programme ?".

2.2 Les langages orientés objets

Les langages orientés objets sont une nouvelle méthode de programmation qui tend à se rapprocher de notre manière naturelle d'appréhender le monde. Les L.O.O. se sont surtout posé la question "Sur quoi porte le programme ?". En effet, un programme informatique comporte toujours des traitements, mais aussi et surtout des données. Si la programmation structurée s'intéresse aux traitements puis aux données, la conception objet s'intéresse d'abord aux données, auxquelles elle associe ensuite les traitements. L'expérience a montré que les données sont ce qu'il y a de plus stable dans la vie d'un programme, il est donc intéressant d'architecturer le programme autour de ces données.

2.3 Réutilisation et facilité de maintenance en essayant de s'abstraire du code

Depuis, les maîtres mots en matière de programmation sont réutilisation et facilité de maintenance. Le code doit être organisé de façon claire et logique de manière à atteindre ces deux objectifs. La programmation objet fournit justement un cadre idéal à ce genre de réalisations, car elle permet une meilleure conceptualisation, elle rend le principe de fonctionnement d'un programme plus "humain" en permettant de s'abstraire totalement du code pour s'atteler à la tâche principale : trouver la solution à un

2.4 Succession de langages

Le langage Simula-67 jette les prémices de la programmation objet, résultat des travaux sur la mise au point de langages de simulation dans les années 1960 et à partir desquels s'inspira aussi la recherche sur l'intelligence artificielle dans les années 1970-80. Mais c'est réellement par et avec Smalltalk 72 puis Smalltalk 80, inspiré en partie par Simula, que la programmation par objets débute et que sont posés les concepts de base de celle-ci : objet, messages, encapsulation, polymorphisme, héritage (sous-typage ou sous-classification), redéfinition, etc. Smalltalk est plus qu'un langage à objets, il est aussi un environnement graphique interactif complet.

À partir des années 1980, commence l'effervescence des langages à objets : Objective C (début des années 1980), C++ (C with classes) en 1983, Eiffel en 1984, Common lisp object system dans les années 1980, etc. Les années 1990 voient l'âge d'or de l'extension de la programmation par objet dans les différents secteurs du développement logiciel.

S'il avait souvent été reproché à la version 4 de Php de n'avoir implémenté les concepts de la programmation objet qu'à moitié, Php 5 ne souffrira certainement pas des mêmes reproches. Le nouveau moteur Zend offre toutes les possibilités d'un langage de programmation orienté objet moderne: classes, héritage, constructeurs, destructeurs, tout est là.

3 Les principes de bases

3.1 La notion d'objet

La difficulté de cette modélisation consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle (chien, voiture, ampoule, ...) ou bien virtuelle (sécurité sociale, temps, ...).

3.1.1 Définition aussi complète que possible d'un objet.

Un *objet* est avant tout une **structure de données**. Autrement, il s'agit d'une entité chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. En cela, rien ne distingue un *objet* d'une quelconque autre structure de données. La principale différence vient du fait que l'*objet* **regroupe les données et les moyens de traitement de ces données**.

Un *objet* rassemble de fait deux éléments de la programmation *procédurale* :

- Les **champs** :

  Les *champs* sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer (les informations). Tout comme n'importe quelle autre variable, un *champ* peut posséder un type quelconque défini au préalable : nombre, caractère, ..., ou même un type objet. On parle d'attributs.
  Synonymes : "attributs". "champs", "propriétés" ou encore "membres",

- Les **méthodes** :

  Les *méthodes* (les actions) sont les éléments d'un objet qui servent d'interface entre les données et le programme. Sous ce nom obscur se cachent simplement des procédures ou fonctions destinées à traiter les données.

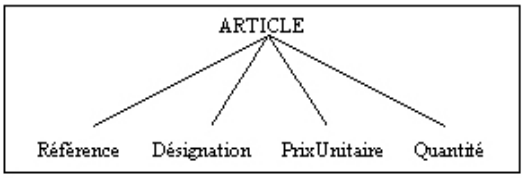
Si nous résumons, un *objet* est donc un type servant à stocker des données dans des *champs* et à les gérer au travers des *méthodes*.
Si on se rapproche du Pascal, un objet n'est donc qu'une extension évoluée des *enregistrements* (type **record**) disposant de procédures et fonctions pour gérer les champs qu'il contient.

- ***Identifiant unique***
  Un objet est un ***groupe de données structurées caractérisé par un identifiant unique*** représentant le monde réel.
  On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, etc.)

3.1.2 Exemples

Exemple 1

Exemple : Un article en stock est un objet, il est caractérisé par une référence, une désignation, un prix unitaire et une quantité.



Un objet est constitué ***attributs*** qui caractérisent la structure de celui-ci.

Exemple : Référence, Désignation, PrixUnitaire et Quantité sont les attributs de l'objet de type ARTICLE.

L'objet est manipulé par des procédures appelées ***méthodes*** qui sont caractérisées par une entête définissant leur nom, les paramètres d'appel et de retour pour chacune d'elle.

Exemple : Nous pouvons définir comme méthodes de l'objet ARTICLE

- *PrixTtc : Méthode permettant de calculer le prix TTC d'un article*
- *SortieArticle : Méthode permettant de diminuer la quantité en stock*
- *EntreeArticle : Méthode permettant d'augmenter la quantité en stock*

On ne peut exécuter une méthode sans préciser l'objet sur lequel elle s'applique.

L'ensemble des propriétés d'un objet (attributs et méthodes) constitue un ensemble appelé ***classe.***

Exemple 2

Imaginons que nous souhaitions simuler un zoo et, plus précisément, le moment où le gardien vient nourrir chaque animal. Nous voulons que chaque animal lance son cri lorsque le gardien le nourrit (le lion va rugir, le mouton bêler, etc.), et nous voulons également que le gardien donne le bon type de nourriture à chaque animal. Ici, nous sommes en présence de deux objets: un gardien et un animal. Chacun des deux sera défini par ses caractéristiques propres et par les actions qu'il peut effectuer. Les caractéristiques du gardien seront par exemple son nom et son prénom, et celles de l'animal duquel il s'approche seront son "type" (lion, koala), son nom et la nourriture qu'il consomme. Au niveau des actions, le gardien peut nourrir un animal et l'animal peut émettre son cri.

3.2 La notion de classe

On appelle **classe** la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule. En réalité on dit qu'un objet est une **instanciation** d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'**objet** ou d'**instance** (éventuellement d'*occurrence*).

Une classe est composée de deux parties :

- **Les attributs** (parfois appelés *données membres*) : il s'agit des données représentant l'état de l'objet
- **Les méthodes** (parfois appelées *fonctions membres*): il s'agit des opérations applicables aux objets

Si on définit la classe *voiture*, les objets *Peugeot 406*, *Renault 18* seront des instanciations de cette classe. Il pourra éventuellement exister plusieurs

objets *Peugeot 406*, différenciés par leur numéro de série. Mieux: deux instanciations de classes pourront avoir tous leurs attributs égaux sans pour autant être un seul et même objet. C'est le cas dans le monde rééel, deux T-shirts peuvent être strictement identiques et pourtant ils sont distincts. D'ailleurs, en les mélangeant, il serait impossible de les distinguer...

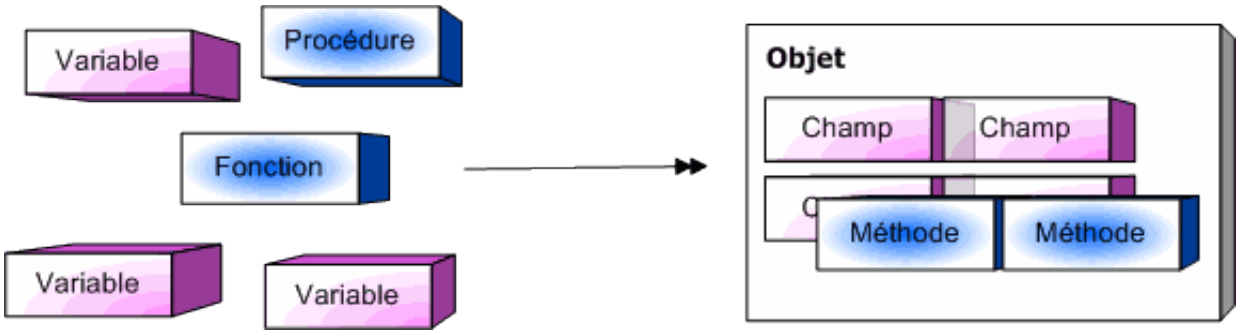
3.3 Les 3 fondamentaux de la POO

La Programmation Orientée Objet est dirigée par 3 fondamentaux qu'il convient de toujours garder à l'esprit : **encapsulation**, **héritage** et **polymorphisme**. Houlà ! Inutile de fuir en voyant cela, car en fait, il ne cachent que des choses relativement simples. Nous allons tenter de les expliquer tout de suite.

3.3.1 Encapsulation

Derrière ce terme se cache le concept même de l'objet : réunir sous la même entité les données et les moyens de les gérer, à savoir les champs et les méthodes.
L'**encapsulation** introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.
Si l'**encapsulation** est déjà une réalité dans les langages procéduraux (comme le Pascal non objet par exemple) au travers des unités et autres librairies, il prend une toute nouvelle dimension avec l'**objet**.
En effet, sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet et donc l'ensemble des procédures et fonctions destinées à la gestion *interne* de l'objet, auxquelles le programmeur final n'aura pas à avoir accès. L'**encapsulation** permet donc de masquer un certain nombre de champs et méthodes tout en laissant visibles d'autres champs et méthodes. Nous verrons ceci un peu plus loin.

Pour conclure, l'**encapsulation** permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.



3.3.2 Héritage

Si l'encapsulation pouvait se faire manuellement (grâce à la définition d'une unité par exemple), il en va tout autrement de l'**héritage**. Cette notion est celle qui s'explique le mieux au travers d'un exemple. Considérons un objet *Bâtiment*. Cet objet est pour le moins générique, et sa définition reste assez vague. On peut toutefois lui associer divers champs, dont par exemple :

- Les murs
- Le toit
- Une porte
- L'adresse
- La superficie

On peut supposer que cet objet *Bâtiment* dispose d'un ensemble de méthodes destinées à sa gestion. On pourrait ainsi définir entre autres des méthodes pour :

- Ouvrir le Bâtiment
- Fermer le Bâtiment
- Agrandir le Bâtiment

Grâce au concept d'**héritage**, cet objet *Bâtiment* va pouvoir donner naissance à un ou des *descendants*. Ces descendants vont tous bénéficier des caractéristiques propres de leur *ancêtre*, à savoir ses champs et méthodes. Cependant, les descendants conservent la possibilité de posséder leur propres champs et méthodes. Tout comme un enfant hérite des caractéristiques de ses parents et développe les siennes, un objet peut hériter des caractéristiques de son ancêtre, mais aussi en **développer de nouvelles**, ou bien encore se **spécialiser**.

Ainsi, si l'on poursuit notre exemple, nous allons pouvoir créer un objet *Maison*. Ce nouvel objet est toujours considéré comme un *Bâtiment*, il possède donc toujours des murs, un toit, une porte, les champs *Adresse* ou *Superficie* et les méthodes destinées par exemple à *Ouvrir le Bâtiment*.

Toutefois, si notre nouvel objet est toujours un *Bâtiment*, il n'en reste pas moins qu'il s'agit d'une *Maison*. On peut donc lui adjoindre d'autres champs et méthodes, et par exemple :

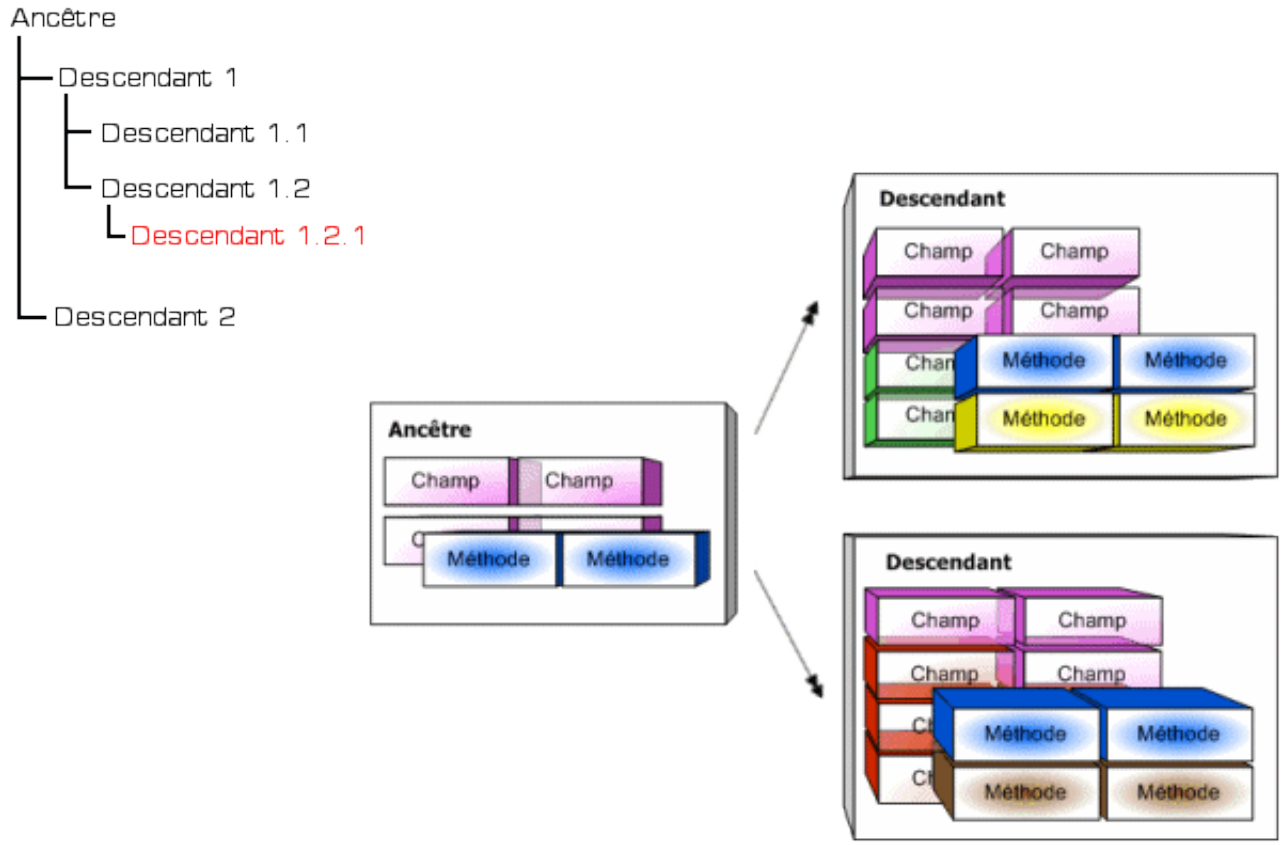
- Nombre de fenêtres
- Nombre d'étages
- Nombre de pièces
- Possède ou non un jardin
- Possède une cave

Notre *Bâtiment* a ici bien évolué. Il s'est **spécialisé**. Avec notre *Maison*, nous sommes allés plus avant dans les détails, et elle est à même de nous offir des services plus évoluées. Nous avons complété ce qui n'était qu'un squelette.

Ce processus d'héritage peut bien sûr être répété. Autrement dit, il est tout à fait possible de déclarer à présent un descendant de *Maison*, développant sa spécialisation : un *Chalet* ou encore une *Villa*. Mais de la même manière, il n'y a pas de restrictions théoriques concernant le nombre de descendants pour un objet. Ainsi, pourquoi ne pas déclarer des objets *Immeuble* ou encore *Usine* dont l'ancêtre commun serait toujours *Bâtiment*.

Ce concept d'**héritage** ouvre donc la porte à un nouveau genre de programmation.

On notera qu'une fois qu'un champ ou une méthode est définie, il ou elle le reste pour tous les descendants, quel que soit leur degré d'éloignement.

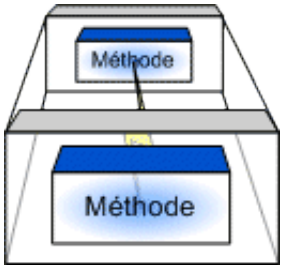


3.3.3 Polymorphisme

Le terme **polymorphisme** est certainement celui que l'on appréhende le plus. Mais il ne faut pas s'arrêter à cela. Afin de mieux le cerner, il suffit d'analyser la structure du mot : *poly* comme *plusieurs* et *morphisme* comme *forme*. Le **polymorphisme** traite de la capacité de l'objet à posséder *plusieurs formes*.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, comme on le sait déjà, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir **redéfinir une méthode** afin de la réécrire, ou de la **compléter**.
On voit donc apparaître ici ce concept de **polymorphisme** : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.
Le **polymorphisme**, en d'autres termes, est donc la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours. Ainsi, si l'on considère un objet *Véhicule* et ses descendants *Bateau*, *Avion*, *Voiture* possédant tous une méthode *Avancer*, le système appellera la fonction *Avancer* spécifique suivant que le véhicule est un *Bateau*, un *Avion* ou bien une *Voiture*.

Attention !
Le concept de **polymorphisme** ne doit pas être confondu avec celui d'*héritage multiple*. En effet, l'héritage multiple - non supporté par le Pascal standard - permet à un objet d'hériter des membres (champs et méthodes) de plusieurs objets à la fois, alors que le **polymorphisme** réside dans la capacité d'un objet à modifier son comportement propre et celui de ses descendants au cours de l'exécution.



4 La programmation orientée objet et PHP

Php dans ses versions antérieures à la version 4, ne proposait pas la possibilité d'utiliser des classes. Le programmeur devait donc se débrouiller pour organiser des fonctions de la manière la plus propre possible, en essayant de les faire communiquer entre elles et de leur faire partager des données sans pour autant rendre le code incompréhensible. La majeure partie des scripts restait lisible, mais **force** est d'avouer que par rapport à des langages objet comme C++ ou Java, Php faisait figure de langage "brouillon" avec lequel il était très difficile de développer du code réutilisable et maintenable facilement. De plus, la création d'applications complexes devenait vite un véritable casse-tête.

Voyons à présent d'un point de vue plus technique de quoi se compose un objet.

4.1 Les objets naissent

La création d'un objet à partir d'une classe se fait grâce au mot **new** suivi du nom de la classe (pas de l'objet).

Il s'agit "d'instanciation" d'une classe. Ici nous utilisons des classes existantes exemple domDocument

Voir site http://fr3.php.net/manual/fr/ref.dom.php

```
< ?php
        $dom = new domDocument() ;
?>
```

4.2 Utilisation d'une méthode

Nous utilisons une méthode de la classe Domdocument pour charger du XML depuis un fichier.

DOMDocument->load() - Charge du XML depuis un fichier

```
$dom->load($fichier_xml)
```

Explication de ->

Nous pouvez vous représenter les objets comme des dossiers sur votre disque dur. Vous pouvez avoir deux fichiers *lisez-moi.txt* sur votre disque dur, tant qu'ils ne sont pas dans le même répertoire. De même que vous devez alors taper le chemin complet jusqu'au fichier, vous devez spécifier le nom complet de la méthode avant de l'employer : en termes PHP, le dossier racine est l'espace de nom global, et le séparateur de dossier est ->. Par exemple, les noms *$cart->items* et *$another_cart->items* représentent deux variables distinctes. Notez que le nom de la variable est alors *$cart->items,* et non pas *$cart->$items* : il n'y a qu'un seul signe *$* dans un nom de variable.

4.3 Autres classes et méthodes utilisées ici

4.3.1 Classe DomDocument

DOMDocument->validate() - Valide un document en se basant sur sa DTD

```
$dom->validate()
```

4.3.2 Classe XSLTProcessor

Voir site http://fr.php.net/manual/fr/ref.xsl.php

et 2 de ses méthodes

XSLTProcessor::importStylesheet - Importe une feuille de style

XSLTProcessor::transformToXML - Transforme en du XML

D'où les lignes

```
$proc = new XSLTProcessor();
$proc -> importStylesheet($xsl);
echo $proc->transformToXml( $dom );
```

Le html est une forme de xml.

Passage de paramètres à une feuille de style.

XSLTProcessor::setParameter — Définit la valeur d'un paramètre

Utilisation

setParameter (string namespace, string name, string value)

Définit la valeur d'un ou plusieurs paramètres pour être utilisés dans une sous-séquence de transformation avec `XSLTProcessor`. Si le paramètre n'existe pas dans la feuille de style, il sera ignoré.

Liste de paramètres

`namespace`

L'URI de l'espace de noms du paramètre XSLT.

`name`

Le nom local du paramètre XSLT.

`value`

La nouvelle valeur du paramètre XSLT.

`options`

Un tableau de paire *nom => valeur*. Cette syntaxe est disponible depuis PHP 5.1.0.

```
$proc -> setParameter(null, 'limit', $limit );
```