Votre Formation Sur Mesure

LA PROGRAMMATION ORIENTEE OBJET EN PHP

ELAN
202 avenue de Colmar - 67100 STRASBOURG
☎ 03 88 30 78 30 🖷 03 88 28 30 69
elan@elan-formation.fr
www.elan-formation.fr

SAS ELAN au capital de 37 000 € -
RCS Strasbourg B 390758241 – SIRET 39075824100041 – Code APE : 8559A
N° déclaration DRTEFP 42670182967 - Cet enregistrement ne vaut pas agrément de l'Etat

SOMMAIRE

I. Introduction

L'apprentissage d'un langage informatique nécessite au départ de tester une ribambelle de fonctions, tests, boucles en tous genres et, bien souvent, de reprendre son code dans tous les sens lorsqu'arrivé à un niveau de développement dit "lourd" (l'application emmagasinant de plus en plus de fonctionnalités et de données).

Même avec tous les commentaires et toute l'indentation du monde, relire son code et le corriger deviennent vite une plaie, et dans un contexte professionnel, une perte de temps significative.

De plus, quel que soit le soin apporté à la logique du code et à son efficacité, on ne sort jamais du cadre "procédural" de la programmation : les données et les traitements sont au mieux scindés dans leur gestion par des fonctions nombreuses et dédiées à des actions précises. Mais le code se répète souvent inutilement et sa réutilisation dans d'autres projets ; où souvent les fonctionnalités sont les mêmes (gestion d'un compte client, news, etc.) ; est compromise.

C'est pourquoi la programmation orientée objet, théorisée dès les années 1960, a vu son succès grandir à partir des années 1980 avec l'apparition de langages qui seront totalement ou partiellement conçus pour l'appliquer (Java, C++, Eiffel, Ruby, Python…).

En outre, la POO apporte une conséquente logique au code procédural, une structuration solide. De plus, créer une application orientée objet permettra à tout ou partie de son code d'être réutilisé ailleurs, puisqu'en fin de compte, on peut très vulgairement commencer par comprendre qu'un programme OO (Orienté Objet) n'est qu'un ensemble de blocs de code autonomes rassemblés au sein d'une même application.

Ce support de cours a pour objectif de vous familiariser avec les nombreux concepts liés à la POO (Programmation Orientée Objet) en langage PHP. Bien entendu, ces concepts sont applicables dans les mêmes conditions dans les autres langages de programmation permettant une telle approche.

Néanmoins, la pratique, matérialisée par le code-exemple illustrant l'application d'un de ces concepts, sera adaptée à la syntaxe de PHP.

II. Contexte

Tout au long du cours, nous prendrons comme contexte la gestion du parc informatique d'un centre de formation. Afin de créer une application fonctionnelle et réaliste, ce contexte comporte quelques spécificités et caractéristiques, qui sont pêle-mêle :

- Gestion d'un poste de travail (PC, portable, composants, périphériques)
- Contrôle des logiciels installés sur les postes et leur version
- Composition d'une salle informatique composée de plusieurs postes et d'une capacité d'accueil limitée
- Possibilité d'aménager du matériel additionnel (vidéoprojecteur, imprimante, écran géant…) dans une salle

III. La théorie

Toutes ces prérogatives pourront être évoquées au fur et à mesure de l'avancée du cours. Mais depuis le début de votre lecture, la notion d'« objet » peut vous paraître bien floue…

La POO, c'est tout simplement faire de son site un ensemble d'objets qui interagissent entre eux. En d'autres termes : tout est objet.

Avant toute chose, essayons de comprendre ce qu'est un objet, et par là, ce qu'est une classe !

1. Qu'est-ce qu'un objet ?

Des objets, vous en avez tout autour de vous : votre souris, votre clavier, l'ordinateur, la chaise sur laquelle vous êtes assis…

Bien entendu, un clavier et une chaise sont des objets bien différents l'un de l'autre, mais la notion d'objet, de manière un peu abstraite, peut leur être attribuée.

Prenons l'ordinateur devant lequel vous vous trouvez. Bien qu'il puisse être lui-aussi différent d'un autre ordinateur, nous pouvons décrire ses caractéristiques de manière à le formaliser comme n'importe quel ordinateur :

- Il possède un écran d'une taille X ou Y
- Il a été construit par une marque précise et répond à un nom de modèle
- Il possède des composants plus ou moins puissants, de diverses capacités (disque dur, RAM…)
- Il est relié à des périphériques
- Il possède un système d'exploitation et des logiciels
- Etc.

Ces valeurs nous permettent d'assembler un ordinateur de manière plus ou moins précise.

De plus, cet ordinateur peut être manipulé comme tous les ordinateurs de la planète :

- On peut l'allumer, l'éteindre
- On peut brancher ou débrancher ses périphériques
- On peut aussi installer des logiciels, les mettre à jour
- On peut le connecter à Internet
- On peut le débrancher et le déplacer ailleurs
- Etc.

Ce sont pour ainsi dire les fonctions de notre ordinateur. Vous pourriez ajouter « eh, mais on peut aussi retoucher des photos, allez sur Facebook, configurer Windows, jouer à Call of Duty, etc. ». D'accord, mais vous sortez de l'idée qu'il faut considérer l'ordinateur comme un objet tangible, tout ce que l'on peut réaliser avec ne nous intéresse pas dans le contexte cité plus haut.

Ce qui nous pousse à conserver solidement, tout au long du projet, une certaine réflexion : se demander si tout est bien utile ! Tout comme pour les bases de données, il n'est pas pour objectif de TOUT prévoir, de TOUT gérer, mais uniquement ce qui sera nécessaire au problème de gestion pour lequel vous développez cette application.

Revenons à notre objet, notre ordinateur. Il nous faut créer un fichier qui va décrire cet objet dans le cadre que nous venons d'établir. Ce fichier ne contiendra que ça, c'est ce qu'on appelle une classe.

```
<?php
        Class Ordinateur{
            private $_marque;
            private $_modele;
            private $_ecran;
            private $_statut = 0;

            public function allumer(){
                    $this->_statut = 1;
            }
        }
        $poste = new Ordinateur();
        $poste->allumer();
```

Pour l'instant, la classe Ordinateur reste très simple, voyons un peu ce qui est écrit ici :

- Une classe en php commence par le mot-clé Class suivi du nom de celle-ci
- Cette classe englobe ses attributs et ses méthodes dans des accolades {…}
- Les attributs sont déclarées au début, portent un nom précédé d'un underscore (c'est une bonne pratique pour différencier les paramètres envoyés aux méthodes des attributs de la classe) et peuvent si besoin prendre une valeur dès le départ (ici statut prend la valeur 0, ce qui veut dire que l'ordinateur est éteint par défaut)
- Enfin, on définit les méthodes qui sont finalement des fonctions dédiées chacune à une fonctionnalité qui va « jouer » avec les attributs (ici, allumer l'ordinateur fait passer $_statut à 1)

Un mot sur la variable $this : c'est par elle qu'on fait référence à l'instance de l'objet courant dans une méthode. En d'autres termes, l'objet Ordinateur qu'on créera juste après modifiera lui-même son attribut $_statut lorsqu'on appellera sa méthode allumer !

Les deux dernières lignes de code manipulent la classe précédemment déclarée, dans l'ordre :

- Un nouvel ordinateur est créé dans une variable $poste avec le mot-clé new. En des termes plus corrects, on dit que $poste instancie un objet de classe Ordinateur. Si on en voulait plusieurs, on aurait créé autant de $poste que d'ordinateurs désirés.

- Puis on allume cet ordinateur $poste en appelant sa méthode allumer, de la même manière qu'une fonction classique en php, cependant la flèche -> intime l'ordre à l'objet d'utiliser SA méthode.

Quelque chose à savoir : un objet ne contient rien d'autre que l'identifiant d'une instance de classe, les données de l'objet en lui-même son stockées en mémoire par le système, et nulle part ailleurs (un peu comme l'identifiant d'un enregistrement dans une table par rapport à toutes les données de celui-ci). Voir le chapitre « la fonction clone » pour plus de détails.

2. Private ? Public ?

C'est ce qu'on appelle le principe d'encapsulation, c'est-à-dire la visibilité des éléments en dehors des accolades de la classe. Il en existe 3 et doivent précéder l'élément à encapsuler :

- Private : un élément privé est uniquement accessible depuis la classe où il est déclaré
- Public : un élément public sera accessible depuis n'importe où dans l'application
- Protected : un élément protégé n'est visible que dans la classe où il est déclaré et dans les autres classes héritant de celle-ci (voir Héritage)

Pour des raisons logiques, il ne devrait pas être possible d'effacer la marque d'un ordinateur n'importe où dans votre code, seule la classe Ordinateur doit avoir ce droit. Par contre, à tout moment l'ordinateur pourra s'allumer ou s'éteindre, donc la méthode en question doit être publique. Mais dans certains cas, il est possible d'avoir une méthode privée ou protégée, pour des raisons de sécurité.

Généralement, tous vos attributs doivent être privés. Pour les méthodes, peu importe leur visibilité.

3. Le constructeur

Le constructeur d'une classe a pour rôle principal d'initialiser l'objet à créer, c'est-à-dire de mettre en place la valeur des attributs soit en assignant directement des valeurs spécifiques, soit en appelant diverses méthodes qui ont cette fonction.

Le constructeur est ce qu'on appelle une méthode magique, il en existe d'autres prévues par la gestion de l'orienté objet de PHP, nous les verrons au besoin au fil de ce support. Une méthode magique DOIT être précédée de deux underscores __ et porte un nom prédéfini (on ne peut pas l'appeler comme on veut).

Le constructeur de notre classe Ordinateur pourrait être celui-ci :

```
<?php
        Class Ordinateur{
            private $_marque;

            public function __construct($marque){
                    $this->_marque = $marque;
            }
        }
        $poste = new Ordinateur("Samsung");//créera un objet Ordinateur de marque Samsung
```

A noter qu'il n'est pas obligatoire d'avoir un constructeur dans une classe, la fonction new créera l'instance de la classe de toute manière, mais dans ce cas ses attributs seront tout simplement vides.

4. Les accesseurs et les mutateurs

Etant donné que les attributs d'un objet ne sont manipulables qu'à l'intérieur de la classe (autrement dit, ils sont en private), seules des méthodes publiques de cette classe doivent permettre de les modifier et de les récupérer.

Un accesseur (getter) est une méthode qui permet de récupérer un attribut de l'objet instancié, cette méthode porte par convention le préfixe get.

Ici un exemple d'accesseur qui récupère la marque de l'ordinateur :

```php
<?php
//ordinateur.class.php
		Class Ordinateur{
		    private $_marque;

		    public function __construct($marque){
				$this->_marque = $marque;
		    }
		    public function getMarque(){
				return $this->_marque;
		    }
		}
		$poste = new Ordinateur("Samsung");
		echo $poste->getMarque(); //affichera "Samsung"
```

Un mutateur (setter) est une méthode qui permettra la modification d'un attribut, et uniquement cela (en d'autres termes, elle ne renvoie aucune information).

Ci-dessous un mutateur qui modifie la vitesse d'horloge du processeur d'un ordinateur (dans le cas d'un overclocking, par exemple).

```php
<?php
//ordinateur.class.php
		Class Ordinateur{
		    private $_marque;
		    private $_cpuClock;

		    public function __construct($marque, $cpuClock){
				$this->_marque = $marque;
				$this->_cpuClock = $cpuClock;
		    }
		    public function getMarque(){
				return $this->_marque;
		    }
		    public function getCpuClock(){
				return $this->_cpuClock;
		    }
		    public function setCpuClock($speed){
				$this->_cpuClock = $speed;
		    }
		}
		//l'objet est instancié avec un processeur de 2.4 GHz
		$poste = new Ordinateur("Samsung", 2.4);
		$poste->setCpuClock(3);
		echo $poste->getCpuClock()." GHz"; //affichera "3 GHz"
```

5. Les constantes

Pour rappel, l'opérateur « -> » permet d'accéder à un élément de tel objet, mais il est possible qu'il faille obtenir une information prédéfinie à la création d'un ordinateur. Pour cela, utilisons les CONSTANTES (en majuscule, par convention).

Dans notre exemple, le disque dur est disponible en trois capacités : 250Go (HDD_SMALL), 500Go (HDD_MEDIUM) et 1To (HDD_BIG).

Nous allons les définir d'emblée dans la classe grâce au mot-clé const et ensuite les utiliser à deux moments :

- Dans le mutateur setHdd(), afin d'attribuer à un objet Ordinateur une des capacités disponibles
- Lors de la création de l'objet, pour lui indiquer quel disque dur sera installé dedans

A la différence des attributs classiques, l'opérateur d'accès à une constante est le double-deux points « :: », aussi appelé opérateur de résolution de portée. A l'intérieur de la classe, ce n'est plus "$this" qu'il faut utiliser pour obtenir la valeur constante mais "self" suivi des "::", ex : self::HDD_SMALL;

Pourquoi "self" ? Parce qu'une constante appartient à la classe dans laquelle elle se trouve et jamais aux objets instanciés. A l'extérieur de la classe, il n'est donc pas nécessaire d'instancier la classe Ordinateur pour y avoir accès (il suffit de demander Ordinateur::HDD_SMALL pour obtenir la valeur associée).

Pour finir, l'intérêt des constantes de classes consiste à prévoir des valeurs sensibles (la capacité d'un disque dur est très souvent préétablie) et à éviter ce qu'on appelle le code muet (du code qui ne signifie rien au premier coup d'œil). Il est tout à fait possible de créer un objet Ordinateur en laissant la capacité du disque dur au choix mais c'est au risque de créer un ordinateur avec un disque dur de capacité nulle, négative ou irréaliste (12100To par exemple), ce qui nous obligerait à plus de code pour vérifier l'information renseignée par l'utilisateur…

```php
<?php
//ordinateur.class.php
        Class Ordinateur{
            private $_marque;
            private $_cpuClock;
            private $_hdd;

            const HDD_SMALL = "250Go";
            const HDD_MEDIUM = "500Go";
            const HDD_BIG = "1To";

            public function __construct($marque, $cpuClock){
                    $this->_marque = $marque;
                    $this->_cpuClock = $cpuClock;
            }
            public function getHdd (){
                    return $this->_hdd;
            }
            public function setHdd($capacite){
                    // On vérifie qu'on nous donne bien une des trois capacités prédéfinies plus haut.
                    if (in_array($capacite, [self::HDD_SMALL, self::HDD_MEDIUM, self::HDD_BIG])){
                            $this->_hdd = $capacite;
                    }
            }
        }
        //l'objet est instancié avec un processeur de 2.4 GHz et on y installe un petit disque dur
        $poste = new Ordinateur("Samsung", 2.4);
        $poste->setHdd(Ordinateur::HDD_SMALL);
        echo "Le disque dur est d'une capacité de ".$poste->getHdd() ; //affiche 250Go
```

6. Les attributs et méthodes statiques

Il est encore plus utile de prévoir des actions ou des informations propres à la classe Ordinateur, indépendamment de tout objet instancié. Prenons le cas d'un compteur du nombre de postes informatiques qu'on incrémente au fur et à mesure :

```
<?php
        Class Ordinateur{
            private $_marque;
            private $_cpuClock;

            private static $_nbPostes = 0;//attribut statique

            public function __construct($data){
                    $this->_marque = $data[0];
                    $this->_cpuClock = $ data[1];
                    self::$_nbPostes++;//le compteur est incrémenté
            }
            public function setMarque($marque){
                    $this->_marque = $marque;
            }
            public function setCpuClock($speed){
                    $this->_cpuClock = $speed;
            }
            public static function combien(){//méthode statique
                    echo self::$_nbPostes."<br/>";
            }
        }
        Ordinateur::combien();//affiche 0
        $poste = new Ordinateur(array("Samsung", 2.4));
        $poste2 = new Ordinateur(array("Hitachi", 1.6));
        Ordinateur::combien();//affiche 2
```

Quelques explications :

- $_nbPostes est un attribut statique. Le mot-clé static nous indique que l'attribut appartient à la classe Ordinateur elle-même, et pas aux objets qui l'instancieront. Ce qui signifie que nous n'aurons pas besoin de créer un poste pour obtenir cette information, mais…
- Puisque $_nbPostes est un attribut statique privé (private), donc il est impossible de l'utiliser en dehors de la classe. C'est pourquoi la méthode publique statique combien() a pour fonction de l'afficher. De la même manière, une méthode statique appartient à sa classe, on pourra donc demander le nombre de postes sans avoir à en créer au préalable.

- Suite à l'attribution des valeurs de l'objet créé, l'attribut statique $_nbPostes est incrémenté dans le constructeur grâce au mot-clé self puisque $_nbPostes est indépendant de tout objet instancié, il sera le même pour tous les ordinateurs.

- Les dernières lignes du code ci-dessus montrent bien que la méthode statique combien() est demandée à la classe Ordinateur directement, et non à un poste en particulier. A noter qu'on peut aussi demander $poste->combien(), mais cela n'a aucun intérêt puisque la méthode statique appelée est celle de la classe de $poste.

IV. Objets et BDD

Nous savons désormais tout ce qu'il nous faut pour concevoir dans les grandes lignes une application Orientée Objet. Mais en imaginant qu'après avoir créé plusieurs postes informatiques, une fois la page rafraîchie ou redémarrée, il nous faut recommencer la saisie des ordinateurs. Il nous faut donc persister nos machines en base de données, et pour cela, nous allons mettre en place quelques techniques.

1. Une classe = un rôle

Cette règle à conserver dans la tête tout le long du développement contient le plus gros de la réflexion que vous aurez à établir : une classe possède un rôle dans votre application. Elle n'est pas forcément le reflet exact d'une table dans la base de données : on peut imaginer que la marque de l'ordinateur est une seconde table les contenant toutes et que la référence "marque" dans la table Ordinateur sera l'identifiant d'une marque précise.

Il sera alors totalement inutile de créer une classe Marque si l'application n'a juste besoin que de connaître le nom de la marque d'un poste. On fera alors en sorte de récupérer uniquement le nom de la marque en base de données pour l'attribuer aux objets concernés.

Le rôle de la classe Ordinateur n'est donc pas de représenter un enregistrement de la table Ordinateur, mais de représenter un poste informatique dans l'application, et cette nuance est importante !

2. Hydrater un objet

C'est ainsi que l'on nomme l'action de récupérer des valeurs stockées et de les injecter dans un objet.

L'hydratation d'un objet pourrait prendre pour source toute autre forme de stockage de données : fichiers, simple tableau php… Mais ici, nous allons utiliser une base de données MySQL et un ami qui nous veut du bien : PDO !

Pour rappel, PDO est une API php, autrement dit une extension intégrée au noyau du langage dont le rôle est de faire le lien entre nos bases de données et le serveur de traitement Apache sur lequel php est exécuté. Et PDO, pour ce faire, est une extension... orientée objet !

Nous créions déjà une instance de classe PDO ($db = new PDO(…)) et nous nous servions de ses méthodes query(), prepare() et execute(), entre autres. Nous manipulions déjà un objet php depuis tout ce temps !!

Nous voulons donc conserver nos ordinateurs en base de données, il nous faut donc une entité qui reflètera la classe Ordinateur. Ci-dessous l'exemple d'une table MySQL Ordinateur :

#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
1	id_ordinateur	int(11)			Non	Aucune	AUTO_INCREMENT	Modifier Supprimer Primaire Uniq
2	marque_ordinateur	varchar(50)	utf8_general_ci		Non	Aucune		Modifier Supprimer Primaire Uniq
3	modele_ordinateur	varchar(50)	utf8_general_ci		Non	Aucune		Modifier Supprimer Primaire Uniq
4	cpuclock_ordinateur	float			Non	Aucune		Modifier Supprimer Primaire Uniq
5	ecran_ordinateur	int(2)			Non	Aucune		Modifier Supprimer Primaire Uniq
6	os_ordinateur	varchar(50)	utf8_general_ci		Non	Aucune		Modifier Supprimer Primaire Uniq
7	hdd_ordinateur	varchar(10)	utf8_general_ci		Non	Aucune		Modifier Supprimer Primaire Uniq

Pour simplifier l'exemple, tous les champs de cette table auront un équivalent en tant qu'attributs dans la classe Ordinateur. Attention cependant au nommage des champs, il sera très important par la suite de respecter rigoureusement la même structure de nom !

Pour la suite, il nous faut quelques enregistrements dans la table ordinateur :

	id	marque	modele	cpuclock	ecran	os	hdd
Modifier Copier Effacer	1	Samsung	PC24	2.5	17	Windows 7	256Go
Modifier Copier Effacer	2	Asus	ROG240	3.5	17	Windows 8.1	1To

et qu'un fichier dbconnect.php soit déjà créé et instancie PDO dans une variable $db au préalable :

```php
<?php
        $dsn = 'mysql:dbname=park;host=localhost';
        $user = 'root';
        $password = '';

        try {
            $db = new PDO($dsn, $user, $password);
            $db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
        }
        catch (PDOException $e) {
            echo 'Connexion échouée : ' . $e->getMessage();
        }
```

Enfin, il nous faut également la classe Ordinateur définie comme précédemment, avec tous ses getters et setters.

3. Les étapes de l'hydratation d'un objet

Hydrater un objet reviendra alors à demander une instance de la classe Ordinateur et le constructeur s'occupera de remplir ses attributs comme ceci :

1. Au moment opportun dans l'application, l'instance de la classe Ordinateur est créée avec new Ordinateur($data), où $data est un tableau associatif contenant les informations récupérées de la base de données

2. Le constructeur appelé avec new appelle à son tour la méthode hydrate() de la classe Ordinateur

3. Hydrate() parcourt le tableau $data avec foreach en récupérant à chaque position la clé et la valeur de la position lue

4. On crée une variable $methode qui va contenir le nom du setter exact (à partir du nom du champ de la table Ordinateur) en le formatant (dans l'exemple ci-dessous, on rajoute "set" et une majuscule au nom de la clé du tableau)

5. On teste si le setter ainsi reconstruit dans la variable $methode est présent dans la classe à l'aide de la fonction is_callable()

6. Enfin la méthode en question, si existante, est appelée, ainsi la valeur du tableau devient attribut de l'objet courant.

Voici ce que cela donne concrètement au sein de la classe Ordinateur (le code ci-dessous est épuré pour se concentrer sur l'essentiel de l'hydratation) :

```php
<?php
	Class Ordinateur{
		private $_marque;
		private $_cpuClock;
		…
		public function __construct($valeurs = array()){
			if(!empty($valeurs))
				$this->hydrate($valeurs);
		}
		public function hydrate(array $donnees) {
			foreach($donnees as $attribut => $valeur) {
				$methode = 'set'.str_replace(' ', '', ucwords(str_replace('_', ' ', $attribut)));
				if(is_callable(array($this, $methode))){
					$this->$methode($valeur);
				}
			}
		}
		…
		public function setMarque($marque){
			$this->_marque = $marque;
		}
		public function setCpuClock($speed){
			$this->_cpuClock = $speed;
		}
		…
	}
```

Mais... d'où viennent les données du tableau $data ? Où utilise-t-on PDO ?? C'est là que le pattern de conception MVC entre en jeu.

Pour pouvoir respecter la dissociation Modèles-Vues-Contrôleurs de l'application, et ainsi coder proprement, il va falloir comprendre une notion très importante : la classe Ordinateur ne devra JAMAIS accéder directement à la base de données !! C'est son modèle qui devra le faire, sur ordre d'un contrôleur.

Cela veut dire que pour respecter le pattern MVC, il va falloir penser à chaque classe en deux fichiers : le premier définit très exactement ce qu'un ordinateur est, et l'autre s'occupera de le faire communiquer avec la base de données. Ce dernier est clairement le modèle, que l'on appelle Manager en POO ! Ci-dessous un exemple de manager Ordinateur :

```php
<?php
	Class OrdinateurManager{
		private $_db;//l'objet PDO créé par le fichier dbconnect.php

		public function __construct($db){
			$this->setDb($db);
		}
		public function getList(){
			$postes = [];
			$q = $this->_db->query('SELECT id, marque, modele
						FROM ordinateur
						ORDER BY marque');
			while ($data = $q->fetch()){
				$postes[] = new Ordinateur($data);
			}
			return $postes;
		}
		public function setDb(PDO $db){
			$this->_db = $db;
		}
	}
```

Ci-dessous, le code très simple d'un fichier qui va, grâce à var_dump(), afficher les instances de la classe Ordinateur créées à partir de la base de données :

```php
<?php
	require_once("lib/dbconnect.php");//le fichier de connexion à la BDD
	require_once("ordinateur.class.php");//la classe Ordinateur
	require_once("ordinateurManager.class.php");//le manager

	$manager = new OrdinateurManager($db);
	$postes = $manager->getList();

	var_dump($postes);
```

Et le résultat à l'écran :

```
Ordinateur n°2
object(Ordinateur)[4]
  private '_id' => string '2' (length=1)
  private '_marque' => string 'Asus' (length=4)
  private '_modele' => string 'ROG240' (length=6)
  private '_cpuClock' => null
  private '_ecran' => null
  private '_hdd' => null
  private '_statut' => int 0

Ordinateur n°1
object(Ordinateur)[5]
  private '_id' => string '1' (length=1)
  private '_marque' => string 'Samsung' (length=7)
  private '_modele' => string 'PC24' (length=4)
  private '_cpuClock' => null
  private '_ecran' => null
  private '_hdd' => null
  private '_statut' => int 0
```

Les objets correspondent exactement aux enregistrements de la base et à la manière dont ils ont été récupérés. Deux choses sont alors à constater, en l'état :

- Certains attributs sont vides, et c'est tout à fait normal étant donné qu'ils n'ont pas été stipulés dans la requête de la méthode getList(). C'est une illustration simple de la règle une classe = un rôle citée précédemment : le manager adapte sa récupération des données de la base aux besoins de l'application, et les objets seront créés en conséquence. La POO est au service de l'application, pas de la base de données (qui n'est qu'un support parmi d'autres).

- Le manager ci-dessus est très simple, mais si vous avez tout suivi, il est un modèle selon le pattern MVC ! C'est lui qui aura pour rôle d'ajouter, supprimer, modifier un ordinateur de la base, entre autres manipulations possibles. Conclusion : chaque manipulation nécessaire sera une méthode de la classe OrdinateurManager !!

4. En aparté : l'autoload

Il est très pratique, pour mieux organiser le code et la structure de fichiers d'une application Orientée Objet, de réserver un fichier par classe nécessaire. Mais comme on peut le voir dans le dernier fichier présenté plus haut, il va falloir appeler chaque fichier, chaque classe sollicitée une par une et dans l'ordre. Dans le cas d'une application assez fournie en classes, cela va très vite devenir problématique.

La fonction spl_autoload_register() en php va nous permettre de régler ce problème. Elle va créer une pile d'auto-chargement de fonctions (nos classes, donc) lorsqu'un new Classe quelconque sera demandé, et cela alors même que les fichiers ne sont mentionnés nulle part dans le code.

```
<?php
        function chargerClasse($classe){
            require $classe . '.class.php';
        }
        spl_autoload_register('chargerClasse');
```

Placé en début de code, ces quelques lignes vont faire en sorte, lorsqu'une classe est sollicitée par l'application, d'aller chercher le fichier qui y est associé et par l'intermédiaire de spl_autoload_register(), de la charger en mémoire et ainsi de pouvoir l'utiliser à la volée. Attention toutefois, ce code d'exemple pose le principe mais ne prend pas en compte la possible complexité de l'arborescence des dossiers de l'application et dépend du nom des fichiers de classe (ici : nomdelaclasse.class.php).

V. Héritage

Notre classe Ordinateur, à l'heure actuelle, est la seule à définir un poste informatique. Plus les attributs et les méthodes vont s'ajouter afin de gérer un maximum de paramètres, plus la classe contiendra de code et, au final, être une plaie à exploiter. De plus, tous les postes sont définis de la même manière, ce qui pose un problème si dans une salle certains disposent d'attributs que d'autres n'ont pas (un portable ou une tour, un Mac ou un PC ne proposeront pas les mêmes choses…)

1. Principe

L'héritage en POO est une notion très puissante : en somme, des classes vont dépendre d'autres classes, pour nous faciliter la vie. Imaginons deux classes, Mac et PC, qui héritent de notre classe Ordinateur. Par définition :

- Mac et PC sont des classes filles de Ordinateur, qui spécifient des attributs et méthodes propres à elles
- Ordinateur est la classe mère de Mac et PC, qui leur spécifie leur comportement commun (un Mac et un PC sont, somme toute, des postes informatiques tous les deux…)

On dira alors que Mac et PC héritent de Ordinateur.

Reprenons comme base l'exemple du code présenté dans la partie « attributs et méthodes statiques », en y ajoutant l'hydratation, et implémentons une classe PC héritant de Ordinateur :

```php
<?php
        Class Ordinateur{
            …
            private static $_nbPostes = 0;
            …
        }
        Class PC extends Ordinateur{
            private $_windows;
            public function __construct($data, $windows){
                        parent::__construct($data);
                        $this->_windows = $windows;
            }
        }
        Ordinateur::combien();//affiche 0

        $poste = new Ordinateur(array("marque" => "Samsung", "cpuClock" => 2.4));
        $poste2 = new PC(array("marque" => "Hitachi", "cpuClock" => 1.6), "Vista");

        Ordinateur::combien();//affiche 2
```

De nouvelles notions sont à expliquer ici :

- Le mot-clé extends permet à PC d'hériter de la Classe Ordinateur, en d'autres termes, PC est une classe fille de Ordinateur qui récupère toutes ses méthodes publiques (les getters et setters, la méthode statique combien(), etc.) et devra comporter les mêmes attributs (marque, modèle, etc.).
- L'attribut $_windows est propre à PC, il est donc déclaré dans cette classe
- Le constructeur de la classe PC dispose en première ligne de « parent ::__construct(…) ». Ce code oblige la classe PC à respecter les paramètres du constructeur de sa classe mère, donc à disposer du tableau de données $data. La classe mère Ordinateur ayant toujours la responsabilité de l'attribut statique $_nbPostes, c'est elle qui s'occupera de l'incrémenter, comme avant. On dit alors que la méthode __construct a été réécrite !
- $poste2 est un new PC ! Et les paramètres renseignés dans les parenthèses doivent suivre l'ordre mère-enfant ($data d'abord, puis $windows)

Mais attention, la classe PC n'a pas accès directement aux attributs de la classe Ordinateur, ceux-ci étant privés (private). Un constructeur dans la classe PC comme celui-ci :

```php
<?php
        Class PC extends Ordinateur{
            private $_windows;
            public function __construct($windows){
                    $this->_marque = "no name";//on essaie de modifier l'attribut $_marque de la classe mère
                    $this->_windows = $windows;
            }
        }
        $poste = new PC("Vista");
        $poste->getMarque();//n'affiche rien du tout !
```

Cela ne renverra aucune erreur fatale, bizarrement, mais PC tente d'écrire (en le définissant à « no name ») l'attribut $_marque de sa classe mère Ordinateur qui, puisque privé, est tout simplement inaccessible (cf la dernière ligne) !

2. La visibilité protected à l'oeuvre

Nous avions très vite évoqué la possibilité d'une troisième visibilité pour les attributs, et c'est maintenant qu'elle va beaucoup nous servir : protected ! Il suffit d'encapsuler tous les attributs d'Ordinateur dont la classe fille PC a besoin directement, en les récupérant de sa classe mère, en protected, et le tour est joué : PC pourra afficher et modifier $_marque, par exemple.

Ainsi, le même code que ci-dessus avec « protected $_marque ; » à la place de « private $_marque ; » dans la classe mère Ordinateur nous renverra bien « no name » !

Il est possible également de réécrire n'importe quelle méthode mère dans la classe fille, si tant est qu'elles aient toutes les deux la même visibilité.

En définitive : seuls les attributs et méthodes publics ou protégés sont accessibles par la classe fille ! Mais il est par ailleurs, comme vu plus haut, encore possible d'instancier un objet Ordinateur directement, ce qui est un peu bête…

VI. Notions importantes

1. Classes abstraites et classes finales

Deux notions intéressantes et simples à comprendre pour continuer :

- Une classe abstraite (abstract) est une classe dont on interdit l'instanciation directe (le cas d'Ordinateur juste avant), afin que dans l'application, la création d'un poste informatique ne peut être autre chose qu'un Mac ou un PC.

- Une classe finale (final) est, à contrario, une classe dont on interdit l'héritage. En définissant les classes filles PC et Mac comme cela, on sécurise alors la possibilité de faire hériter une autre classe de PC ou Mac (ces classes ne pourront avoir de classes filles).

Un exemple :

```
<?php
        abstract Class Ordinateur{
            …
        }
        final Class PC extends Ordinateur{
            private $_windows;
            public function __construct($data, $windows){
                        parent::__construct($data);
                        $this->_windows = $windows;
            }
        }
        final Class Mac extends Ordinateur{
            private $_macOS;
            public function __construct($data, $macOS){
                        parent::__construct($data);
                        $this->_macOS = $macOS;
            }
        }
```

2. La résolution statique à la volée

Une nouvelle notion un peu plus complexe pour finir : en prenant cet exemple :

```php
<?php
        Class Ordinateur{
            public static function afficherType(){
                    self::typeDePoste();
            }
            public static function typeDePoste(){
                    echo "Je suis un ordinateur lambda !";
            }
        }
        Class Mac extends Ordinateur{
            public static function typeDePoste(){
                    echo "Je suis un Mac !";
            }
        }

        Mac::afficherType();
```

La dernière ligne affichera « Je suis un ordinateur lambda », alors que la méthode afficherType a été demandée à la classe fille Mac… C'est la fonction statique typeDePoste() de la classe mère qui a été renvoyée !!

Pourtant, on a vu plus haut (avec getMarque()) qu'une classe fille héritait des méthodes publiques de sa classe mère. Mais le mot-clé self est ici en cause : pour rappel, il permet d'accéder à un élément statique de la classe où cet élément est défini. Comme le seul endroit où typeDePoste() est exécuté est dans la classe mère Ordinateur, c'est donc celle de Ordinateur qui s'affiche.

Pour régler ce problème, une simple modification suffit :

```php
<?php
        Class Ordinateur{
            public static function afficherType(){
                    static::typeDePoste();//on remplace self par static
            }
            …
        }
        Class Mac extends Ordinateur{
            …
        }
        Mac::afficherType();
```

Le mot-clé static, en lieu et place de self dans la méthode afficherType() de la classe mère, permet à cette dernière de savoir quelle classe a été initialement appelée pour invoquer la méthode dans laquelle on se trouve.

3. Méthodes magiques

Sous cette dénomination se regroupent toutes les méthodes préexistantes qu'une classe peut implémenter. Une méthode magique sera appelée lorsqu'un évènement particulier se produira au sein d'une classe. Vous en avez déjà vu une, d'ailleurs : la méthode __construct().

Sachez que la méthode __construct() n'est pas obligatoire dans une classe ! Vous avez sans doute remarqué que les derniers exemples de code n'en avaient pas, et pour cause : nous n'avions pas d'objet à instancier ! Et même si nous en avions un (exemple : $poste = new Mac() ;), l'objet sera pourtant bien instancié alors qu'il n'y a toujours pas de constructeur dans sa classe.

Tout est normal : __construct() est une méthode magique qui est appelée dès que le mot-clé new est rencontré. C'est donc à l'évènement de création d'un objet que __construct() est appelée automatiquement, s'occupant d'instancier en mémoire un objet selon la classe voulue.

Un objet est instancié en mémoire jusqu'à ce que php ait fini d'exécuter le code de vos pages. Les objets sont alors détruits (traduction : la mémoire allouée au stockage des instances de classes exploitées par le code est libérée). __destruct() est la méthode qui répond à cet évènement :

```
<?php
        class Ordinateur{
            public function __construct(){
                    echo 'Je suis un nouvel ordinateur !<br/>';
            }
            public function __destruct(){
                    echo 'Plus personne n'a besoin de moi, je disparais !';
            }
        }
        $poste = new Ordinateur();
```

En exécutant ce code, vous verrez les deux textes s'afficher.

Un bref listing d'autres méthodes magiques qui peuvent être utiles (toujours deux underscores obligatoires avant !) :

- __set et __get : sont appelées lorsque, respectivement, on essaie d'assigner ou d'accéder à un attribut qui soit n'existe pas, soit n'est pas accessible (private par exemple). __set prend deux arguments : le nom de l'attribut souhaité et la valeur à laquelle il devait être assigné. __get n'en prend qu'un : le nom de l'attribut.

- __call et __callStatic : la première s'exécutera lorsqu'une méthode est demandée alors qu'elle n'est soit pas existante, soit inaccessible (private). __callStatic fait la même chose mais seulement lorsqu'on a souhaité une méthode statique alors qu'elle n'existe pas ou n'est pas effectivement statique. Ces deux méthodes magiques ont pour arguments le nom de la méthode voulue et les paramètres qu'on lui a passé. Attention : __callStatic, si elle est réécrite, doit être statique elle-même !

- __toString : une méthode magique très pratique : elle s'exécute lorsqu'on essaie d'afficher un objet tout entier ! (exemple : echo $poste ;). Vous pourrez alors lui demander de renvoyer une chaîne de caractères complexe, par exemple un paragraphe répertoriant tous les attributs de l'objet, son état, etc.

- __invoke : permet de se servir d'un objet comme d'une fonction ! Par exemple, le fait d'écrire « $poste('version') ; » pourrait nous afficher la version du système d'exploitation sous forme d'une phrase complète. Le nombre d'arguments à renseigner est selon vos besoins, puisque vous les exploiterez vous-même dans la méthode __invoke() !

Il en existe quelques autres, peut-être un peu moins utiles, n'hésitez pas à vous renseigner sur le web !

4. La fonction clone

Nous en avions brièvement parlé plus haut : un objet ne contient pas réellement toutes les informations de sa classe, mais seulement l'identifiant de l'instance de classe auquel l'objet est rattaché en mémoire. Autrement dit, l'objet n'est qu'une référence à la zone mémoire de php qui, elle, stocke tout son comportement et ses données (la classe, quoi !) jusqu'à ce que php finisse d'interpréter le code de la page.

Ainsi, si vous tentez de faire :

```
< ?php
        Class Ordinateur{
            public $_nomPC ;
            …
        }
        $poste1 = new Ordinateur(…) ;
        $poste1->nomPC = "Samsung";
        $poste2 = $poste1 ;
        echo $poste1->nomPC; //affichera "Samsung"
        $poste2->nomPC = "Hitachi";
        echo $poste1->nomPC; //affichera "Hitachi"
```

Les objets $poste1 et $poste2 font ici référence à la même instance de la classe Ordinateur, deux pointeurs vers la même zone mémoire contenant les informations. Il n'y a donc qu'un seul objet Ordinateur ! Comment alors créer une copie d'un objet, indépendamment de l'objet initial ? En utilisant la fonction clone :

```php
<?php
        $poste2 = clone $poste1;
```

$poste2 est devenu une copie, un clone de l'objet $poste1. Vous pourrez ainsi le manipuler sans risque de modifier l'objet d'origine.

Il existe également une méthode magique __clone(), qui écoute l'évènement produit par cette fonction, et ainsi permettre d'incrémenter un compteur d'instances dans une classe par exemple.

5. La sérialisation

Lorsque php termine d'exécuter le code d'une page, manipulant des objets, la mémoire allouée au stockage des instances de classe est vidée. A moins de les avoir persistés en base de données, vos objets sont donc effacés.

Vous savez que la superglobale $_SESSION permet de stocker une variable pour la passer de page en page. Si vous tentez d'enregistrer en session un objet pour le récupérer sur une autre page, vous faites transiter l'instance de la classe dans une variable de session. Exemple :

```php
<?php //page 1
        $session_start();
        Class Ordinateur{
            …
        }
        $poste = new Ordinateur(…);
        $_SESSION['poste'] = $poste;
        echo "Aller en page 2 : <a href='page2.php'>page2</a>";
```

```php
<?php //page 2
        $session_start();
        var_dump($_SESSION['poste']);
```

La page 2 va vous afficher une chose très étrange : l'objet transité par la session est une instance de "__PHP_Incomplete_Class_Name". Normal, nous ne redéfinissons pas la classe Ordinateur en page 2, donc l'objet $poste ne fait référence à aucune classe déclarée.

Vous pourriez avoir besoin de passer un objet par la session php mais ne pas forcément vouloir redéfinir sa classe dans le fichier cible (pensez d'abord à spl_autoload_register…), pour diverses raisons, par exemple enregistrer un objet tout entier dans le champ d'une table !

La fonction serialize() va convertir votre objet en une simple chaîne de caractères, ce qui permettra de l'inscrire en base de données (dans un champ de type string), l'écrire dans un fichier ou même le passer en paramètre dans une url (via la superglobale $_GET).

```
<?php //page 2
        $session_start();
        $sposte = serialize($_SESSION['poste']);
        echo $sposte;
```

La dernière ligne affiche ce genre de chaîne :

```
O:3:"Mac":3:{s:6:"_macOS";s:13:"Mountain Lion";s:10:"*_marque";s:5:"Apple";s:21:"Ordinateur_cpuClock";i:3;}
```

Et pour rétablir l'objet comme instance de sa classe, on utilise unserialize() :

```
<?php //page 2
        $session_start();
        $unsposte = unserialize($_SESSION['poste']);
```

Attention toutefois à re-déclarer la classe correspondante afin de ne pas provoquer une erreur disant que la classe spécifiée à l'objet n'existe pas !

6. La fonction Instanceof

Pour finir, la fonction instanceof se révèle très pratique utilisée ainsi :

```
<?php
        class Ordinateur{
            …
        }
        $poste = new Ordinateur(..);

        if($poste instanceof Ordinateur){
            echo "l'objet est bien une instance de la classe Ordinateur";
        }
        else echo "L'objet n'est pas une instance de Ordinateur";
```

Instanceof renvoit true si l'objet est une instance de la classe à vérifier, ou false si ce n'est pas le cas. Plus intéressant encore, dans le cas d'un objet dont la classe hérite d'une autre :

```php
<?php
        class Ordinateur{
            ...
        }
        class Mac extends Ordinateur{
            ...
        }

        $poste = new Mac(..);

        if($poste instanceof Ordinateur){
            echo "l'objet est bien une instance d'une classe qui hérite de la classe Ordinateur";
        }
        else echo "L'objet n'est pas une instance d'une classe qui hérite de Ordinateur";
```

Instanceof peut nous renseigner sur le fait qu'un objet est d'une classe héritant d'une autre !

VII. Pour aller plus loin

- Tutoriel Programmation Orienté Objet en PHP (de A à Z, très accessible) :
  http://openclassrooms.com/courses/programmez-en-oriente-objet-en-php
- Sur developpez.com, un peu plus technique :
  http://g-rossolini.developpez.com/tutoriels/php/cours/?page=poo
- Sur le blog Grafikart, avec une vidéo de démonstration :
  http://www.grafikart.fr/formations/programmation-objet-php