POO

Programmation
Orienté
Objet

PHP : Programmation orientée objet (P.O.O.)

Préambule : penser en terme d'objet

Il existe plusieurs moyens de conceptualiser un programme informatique. L'approche classique consiste à considérer un programme comme une série d'instructions exécutées séquentiellement, ce qu'on a l'habitude d'appeler "programmation procédurale".

PHP, comme d'autres langages de programmation, est un lagage "orienté objet". C'est là plus qu'un concept à la mode. C'est une véritable philosophie de la programmation. La programmation orientée objet est une façon d'élaborer des programmes informatiques qui tend à refléter la manière dont les objets à gérer sont assemblés et en communication dans le monde réel. Ainsi, une voiture est un ensemble complexe de composants en interactions les uns avec les autres mais elle forme un tout cohérent, capable d'agir (démarrer, freiner…), et doté de caractéristiques propres (sa couleur, son numéro d'immatriculation, sa vitesse…).

Pour parvenir à ce type de représentation du monde réel, la logique de programmation orientée objet s'appuie sur quelques principes qu'il s'agit ici de découvrir, avant de pouvoir les mettre en œuvre. Bien entendu, comme toute technologie, la programmation orientée objet introduit un nouveau vocabulaire, correspondant aux concepts sousjacents, auquel il faudra bien se familiariser… Nous aborderons les sujets suivants :

- l'organisation des programmes en éléments appelés **classes** et l'utilisation de ces classes pour définir des **objets** manipulables par programmation
- la définition d'une classe sous les deux aspects de sa *structure* : les **méthodes** (= comportements) dont elle dispose et les **attributs** (= données) qui la caractérisent
- les **relations** entre les classes, comme **l'héritage**, qui permet à une classe bénéficier des fonctionnalités d'une autre classe
- les différents types **d'association entre objets**

Une classe pour créer des objets à manipuler

Un objet est un ensemble cohérent, rassemblant à la fois des données (qui le caractérisent) et des morceaux de programmation (qui le font évoluer).
Le concept de base lié à la P.O.O est que, dans un programme, **"tout est objet"** (ou considéré comme tel) : un produit, un client, mais aussi un fichier, une connexion, etc.

> "Une **classe** est un ensemble cohérent de code, qui contient généralement à la fois des variables (appelées **attributs**) et des fonctions (appelées **méthodes**), et qui va nous servir de plan pour créer des objets. Le but d'une classe va donc être de créer des objets, que nous allons ensuite pouvoir manipuler dans un ou plusieurs projets. "

Une classe peut ainsi être considérée comme un **moule à partir duquel on peut créer des objets**, chaque objet représentant un élément du monde réel à gérer par le programme.
Une classe décrit la **structure interne d'un objet** : les **données** qu'il regroupe et les **actions** qu'il est capable d'assurer sur ses données.
Ainsi dans un projet de site internet, une classe `Utilisateur` décrira les données et les actions que peuvent exécuter, entre autres, tous les clients ; d'un autre côté, chaque client à gérer sera une **instance** (= un exemplaire) de cette classe `Utilisateur`.

Attention : un développeur écrit le code des classes, mais à l'exécution, c'est le code qui instanciera et manipulera les objets qui réalisera les opérations concrètes.
Par exemple, un fichier de classe `Utilisateur` que nous aurions à créer pour Jarditou pourra être intégré au projet de notre site internet. Mais lorsqu'un client voudra créer un compte, c'est bien le script de validation du formulaire d'inscription qui va instancier la classe `Utilisateur` en créant un objet $client, dans lequel on ira stocker le nom, le n° SIRET et l'adresse du nouveau client, et réaliser les opérations permettant de hasher son mot de passe avant de l'enregistrer en base de données...

Déclaration d'une classe

Il y a quelques règles à connaître et à respecter lorsque l'on définit une nouvelle classe.

Conventions de nommage :

- Une classe ne peut avoir un nom qui est déjà un mot réservé en PHP (ex: `Date`, `Clone`, etc.)
- Le nom de la classe doit commencer soit par une lettre majuscule, soit par un underscore (_)
- Le nom de la classe ne peut contenir que des caractères alphanumérique et des underscores

Chaque classe doit par ailleurs être définie dans un fichier qui lui est propre, et qui porte par convention le nom de la classe suivi de l'extension (ex: `NomClasse.class.php`).

En PHP, on déclare une nouvelle classe avec le mot-clé `class`.
Nous allons donc nous en servir pour créer une première classe, que nous appellerons `Animal` (le fichier se nommera donc *Animal.class.php*) :

```php
<?php
class Animal
{
    // code du contenu de la classe
}
?>
```

Voilà ! Nous avons défini la classe `Animal` (une définition certes basique mais une définition tout de même !).

Notre classe `Animal` est donc le plan de création d'un animal.
Nous allons pouvoir y définir les **caractéristiques** qu'aura chaque animal (une espèce, une taille, un poids, un régime alimentaire, un nombre de pattes, etc.) ainsi que les **actions** qu'il pourra effectuer (manger, dormir, se reproduire, se déplacer, etc.) :

```php
<?php
class Animal
{
    public $espece;
    private $taille;
    private $poids;
    private $nbpattes;

    public function manger() {
        // ...
    }

    public function avancer(int $nbpas) {
        // ...
    }
}
?>
```

Instanciation d'une classe

L'instanciation d'une classe est l'opération consistant à créer un **objet** à partir de la **classe** en question. On parle aussi de *créer une instance*.

Cette opération s'effectue dans les scripts qui souhaitent avoir recours à la classe, et par l'intermédiaire du mot clé `new` :

```php
$UnChien = new Animal();
```

Une **instance** correspond à un « exemplaire » d'une classe. Lorsqu'on instancie une classe, on *crée un objet* ; on peut considérer que l'instanciation est l'opération qui « donne vie » à l'objet.

Pour manipuler un objet et l'utiliser, il nous faut stocker l'objet créé dans une variable (c'est cette variable que nous appellerons par la suite notre **objet**).
Ainsi, nous pourrons avoir autant d'objets que nous le souhaitons depuis une même classe :

```php
$UnChien = new Animal();
$UnAutreChien = new Animal();
$UnChat = new Animal();
$UnLapin = new Animal();
```

Le grand intérêt ici, est qu'on va pouvoir effectuer des opérations différentes sur chaque instance d'une classe (= un exemplaire) sans pour autant affecter les autres instances (= les autres objets).

Lorsque nous créons ici des animaux à partir de la classe `Animal` (ou que *nous instancions cette classe*), chaque animal instancié est défini par toutes les caractéristiques de sa classe : tous les animaux appartiendront à une espèce, auront un poids et un régime alimentaire, par exemple.
Mais les *valeurs* de ces caractéristiques et de ces actions seront, elles, spécifiques à chaque animal. L'espèce sera différente d'un animal à l'autre, tous les animaux ne pèseront pas le même poids, et tous n'auront pas le même régime alimentaire.

De même, les actions définies pour la classe seront les mêmes pour tous les animaux, mais ces actions auront un impact différents selon l'animal (l'objet) qui les exécutent. Par exemple, un animal de grande taille parcourera une plus grande distance lorsqu'il se déplacera, qu'un animal plus petit...

Attributs et méthodes de classe

Prenons maintenant l'exemple d'une classe `Vehicule` :

```php
class Vehicule
{
    public $_marque;
    public $_vitesseMax;
    protected $_vitesseCourante;
    private $_nbPassagers;

    public function demarrer() {
        // code permettant de démarrer l'objet instancié
    }
    public function accelerer(int $nbKm) {
        // code permettant d'accélérer
    }
    public function avancer(int $nbKm) {
        // code permettant d'avancer
    }
    public function ajouterPassager() {
        // code permettant d'ajouter un passager
    }
}
```

Attributs d'instance

Notre classe est caractérisée par 4 informations :

- sa marque : `$_marque`
- sa vitesse maximale : `$_vitesseMax`
- sa vitesse courante : `$_vitesseCourante`
- le nombre de passagers à bord : `$_nbPassagers`

Toutes les données, appelée **attributs** seront **représentatives d'un véhicule spécifique** ; autrement dit, chaque objet véhicule aura sa propre copie des données : on parle alors **d'attribut d'instance**.

--> Comprendre le fonctionnement des attributs

Méthodes d'instance

Le même raisonnement s'applique directement aux méthodes.
Notre classe `Vehicule` possède 4 actions :

- possibilité de `demarrer()`
- possibilité d'`accelerer()`
- possibilité d'`avancer()`
- possibilité d'`ajouterPassager()`

Il est clair que la méthode `demarrer()` s'applique individuellement à chaque véhicule. Tous les objets Vehicule instanciés dans le projet ne vont pas démarrer en même temps, il faudra spécifier quelle instance de la classe nous souhaitons démarrer.
En outre, la méthode `avancer()` va clairement utiliser ou modifier les attributs d'instance de l'objet auquel elle va s'appliquer, par exemple la `vitesseCourante` du véhicule en question. Ce sont des **méthodes d'instance**.

--> Comprendre le fonctionnement des méthodes

Visibilité et sécurité

--> Choisir la bonne visibilité

--> Déclarer les accesseurs et les mutateurs

Constructeur de classe ?

Côté serveur, la création/instanciation d'un nouvel objet est constituée de deux phases :

- une phase du ressort *de la classe* (et du système d'exploitation) : allouer de la mémoire pour le nouvel objet et lui fournir un contexte d'exécution
- une phase du ressort *de l'objet* : initialiser ses attributs d'instance

En PHP, ces opérations sont réalisées dans une *méthode spéciale* : le **constructeur** de la classe ; cette méthode est appelée automatiquement lorsqu'on instancie la classe en utilisant l'opérateur `new`, et le code que contient la fonction de constructeur est de suite exécuté sur l'objet instancié.

La déclaration du constructeur est facultative en PHP (attention : elle est obligatoire dans certains langages !).

Le constructeur peut être déclaré dans la classe avec la méthode suivante : `__construct()`. Cette méthode peut recevoir des arguments.

--> Utiliser le constructeur

Ici, on pourrait ajouter un constructeur à notre classe, permettant d'instancier un véhicule avec une marque à définir obligatoirement, et certaines caractéristiques données par défaut :

```php
class Vehicule
{
    public $_marque;
    public $_vitesseMax;
    protected $_vitesseCourante;
    private $_nbPassagers;

    public function __construct ($marque, $vMax = 200, $nbPassagers = 0) {
        $this->_marque = $marque;
        $this->_vitesseMax = $vMax;
        $this->_vitesseCourante = 0;
        $this->_nbPassagers = $nbPassagers;
    }

    public function demarrer() {
        // code permettant de démarrer l'objet instancié
    }

    // ...
}
```

Exemple d'instanciation :

```php
// Instanciation d'un vehicule
$maVoitureDeFonction = new Vehicule("Lotus", 220, 1);
```

Conventions de nommage

- Préfixer les noms de méthodes et d'attributs privés avec un underscore afin de les distinguer plus rapidement à la relectue du code.
- Placer les attributs et méthodes en accès privé ou bien en accès protégé si l'on souhaite dériver la classe dans le futur.
- Utiliser autant que possible les conventions de nommage pour les accesseurs et les mutateurs (`getNomAttribut()` et `setNomAttribut()`).

NB: ces conseils et réflexes sont à mettre en oeuvre en développement orienté objet afin d'uniformiser et de standardiser le code. Ces astuces syntaxiques ne sont pas des techniques personnelles puisque ce sont des conventions éprouvées par des spécialistes des langages orientés objet. Malgré tout, ne le prenez pas comme des paroles d'évangile ! Vous êtes libres d'adopter les conventions et règles syntaxiques de votre choix.

Encapsulation

Le premier principe de l'orienté objet, **l'encapsulation**, fait référence à l'état d'un objet comme étant un **ensemble indissociable de données et de traitements sur les données**.

Un objet rassemble en lui-même (au sein de sa classe) ses données (les attributs, représentant l'état de l'objet), et le code capable d'agir dessus (les méthodes).
On dit que les attributs et les méthodes sont **encapsulés** dans la classe : les variables ont une visibilité au sein de la classe, et il faut dans tous les cas utiliser la classe pour accéder au contenu de ces variables (elles ne peuvent pas l'être directement).

En savoir plus sur l'encapsulation

Héritage et hiérarchie de classes

Le deuxième principe de l'orienté objet, **l'héritage**, est l'idée selon laquelle on peut **définir des classes à partir d'autres classes** - ce qui évite de tout réécrire quand des classes ont des caractéristiques ou des actions en commun.

L'héritage est un concept majeur de la programmation orientée objet ; c'est un mécanisme permettant à une classe d'hériter de (de pouvoir utiliser) toutes les méthodes et attributs d'une autre classe.

Prenons un exemple, à partir de notre classe `Animal` définie en amont.

Si nous devons considérer des chiens, des chats et des canaris, nous pouvons alors créer trois classes dérivées de la classe `Animal` : les classes `Chien`, `Chat` et `Canari`.
Peu importe que cette division ne soit pas pertinente dans l'univers réel ; il suffit qu'elle le soit dans celui du problème à traiter.
Nous représenterons cette division de l'univers de la façon suivante :

heritage

Un chien est bien un animal spécifique ; on dit que la classe `Chien` **hérite**, **étend** ou **dérive** de la classe `Animal`.
D'un autre côté, la classe `Animal` est une généralisation des classes `Chien`, `Chat` et `Canari`.

L'héritage dénotant une relation de généralisation / spécialisation, on peut traduire toute relation d'héritage par la phrase suivante : « La classe dérivée **est une version spécialisée** de sa classe de base ».
On parle également de **relation *est un*** pour traduire le principe de généralisation / spécialisation. Ici, un chien est bien un animal, une *sorte* d'animal.

Nous pouvons ensuite, si nécessaire, créer encore de nouvelles classes dérivées, par exemple :

heritage dans l'heritage

Il est d'usage de représenter ainsi la hiérarchie des classes, sous forme d'arbre inversé. La classe la plus générale se trouve à la racine (en haut, puisque l'arbre est inversé).

Bien entendu, chacune des classes symbolisées ci-dessus est dotée d'attributs et méthodes.
Mais en y regardant de plus près, un animal, qu'il soit chien ou chat peut avoir un `$_nom`, et il est capable de `seNourrir()` ou `seDeplacer()`.
Ces attributs et méthodes communs seront décrits uniquement dans la classe `Animal`. Le mécanisme de l'héritage permettra ensuite directement de faire en sorte que chacune des classes dérivées en héritera automatiquement.

Il ne restera ainsi à écrire dans les classes spécialisées que ce qui est réellement spécifique : `ronronner()` et `miauler()` pour un `chat`, `aboyer()` pour un `chien`, ou encore `siffler()` pour un `canari`.

--> Utiliser l'héritage

Par ailleurs, il est également possible que plusieurs classes dérivées contiennent des méthodes similaires mais nécessitant un fonctionnement spécifique pour la réaliser, et donc un code différent. On parle alors de **polymorphisme**.

Par exemple, un chien et un canari pourront `avancer()`, mais un chien utilisera ses pattes alors que le canari s'envolera, et le chien marchera moins rapidement qu'un canari qui vole.
Si l'on veut gérer ces deux déplacements dans un jeu en 3D, on appellera dans tous les cas la fonction `avancer()` ; mais le code programmé dans cette méthode sera différent dans la classe de l'un et de l'autre.

De la même manière, on pourrait avoir une méthode `avancer()` déjà présente dans la classe mère `Animal` ; auquel cas, toute méthode `avancer()` déclarée dans une classe fille viendra écraser le traitement hérité.

--> Maîtriser le polymorphisme

Conclusion

Il faut bien noter que nous avons abordé dans cette séquence uniquement les bases.
La programmation orientée objet présente en effet des notions beaucoup plus avancées (clonage, itérateurs, classes abstraites, interfaces, traits, design patterns, introspection, héritage multiple dans certains langages...).
Pour aller plus loin, vous pouvez lire ici quelques notions de P.O.O. avancée

Exercices

Exercice 1 : création d'un personnage

1 - Dans un dossier nommé `POO`, créez un dossier `classes/` dans lequel vous créez un fichier *Personnage.class.php*.

2 - Créez une classe `Personnage` qui comprend les attributs suivants:

- nom
- prenom
- age
- sexe

Votre classe doit encpsuler les attributs à l'aide de propriétés privées et utiliser des accesseurs pour permettre l'accès aux attributs.

3 - Créez un deuxième fichier `MonPerso.php` permettant de vérifier le bon fonctionnement de votre classe, en vous inspirant du code ci-dessous :

```php
    $p = new Personnage();
    $p->setNom("Lebowski");
    $p->setPrenom("Jeff");

    echo ($p);
```

Exercice 2 : gestion des employés

La société Jarditou, d'envergure nationale, souhaite se doter d'un S.I.R.H. pour gérer ses employés dans les différents magasins à travers la France.

1 - Ecrire une classe *Employe* avec les informations (propriétés) suivantes :

- Nom
- Prénom
- Date d'embauche dans l'entreprise
- Fonction (Poste) dans l'entreprise
- Salaire en K euros brut annuel
- Service dans lequel se situe l'employé (Comptabilité, Commercial...)

2 - Dans la classe *Employe*, écrire une méthode permettant de savoir depuis combien d'années l'employé est dans l'entreprise.

3 - Chaque année, l'employé reçoit une prime calculée sur le salaire annuel (5% du brut) et sur l'ancienneté (2% du brut pour chaque année d'ancienneté). Cette prime est versée au 30/11 de chaque année. Dans la classe *Employe*, écrire le(s) méthode(s) permettant de déduire le montant de cette prime et de donner l'ordre de transfert à la banque le jour du versement.

L'ordre de transfert à la banque sera juste un message écrit spécifiant que l'ordre de transfert a été envoyé à la banque avec mention du montant.

Afin de tester les éléments ci-dessus, créer au minimum 5 objets *Employe* avec des informations sensiblement différentes. Ecrire dans le programme principal l'affichage du montant des primes de chaque employé.

La date du versement de la prime doit également être vérifiée. Pour tester, fixer cette date à la date du jour, faire les conditions nécessaires dans le code et tester en changeant de date afin de voir si le message voulu s'affiche.

4 - L'entreprise est constituée de magasins implantés sur tout le territoire français.
Un employé fait partie d'un (et un seul) magasin. Un magasin dispose d'un nom, d'une adresse, d'un code postal, d'une ville. Ecrire une nouvelle classe *Magasins* qui contient tous ces éléments et modifier la classe *Employe* afin que celui-ci soit rattaché à un magasin.

5 - En ce qui concerne les repas, les magasins ne disposent pas toutes d'un restaurant d'entreprise. Les employés se trouvant dans les magasins qui n'ont pas de restaurant d'entreprise bénéficient en contrepartie de tickets restaurants. Chaque magasin dispose donc de son propre mode de restauration.

Modifier la classe *Magasin* pour gérer ce mode de restauration.

Afficher chaque mode de restauration de chaque employé selon le magasin dans lequel il est affecté.

6 - L'entreprise souhaite intégrer dans ce système informatique les activités du comité d'entreprise. Des chèques-vacances sont distribués aux employés à condition que ceux-ci aient une ancienneté d'au moins un an.

Modifier la classe *Employe* afin de savoir si celui-ci peut disposer de chèques-vacances ou non.

7 - Chaque année, des chèques Noël sont distribués aux enfants des employés. Le montant du chèque Noël dépend de l'âge des enfants :

- 20 € pour les enfants de 0 à 10 ans,
- 30 € pour les enfants de 11 à 15 ans,
- 50 € pour les enfants de 16 à 18 ans.

Modifier le programme afin de gérer l'attribution des chèques Noël aux enfants des salariés.

Afficher si l'employé a le droit d'avoir des chèques Noël (Oui/Non). Pour ce faire, établir les conditions nécessaires dans le programme.

Si la réponse est *Oui*, afficher combien de chèques de chaque montant sera distribué à l'employé.

Si aucun chèque n'est distribué pour une tranche d'âge, ne pas afficher.