Programmation Orientée Objet en PHP

PWS Programmation Orientée Objet en PHP

Pierre Blarre - Florian Rodriguez

Définition d’un objet

- Exemples : Un ordinateur, une lampe, une chaise, un bureau, etc.
- En programmation : Objets similaires avec des caractéristiques (**attributs**) et des capacités (**méthodes**).
- Exemple :
  - Objet = Personnage (dans un jeu)
  - Attributs : Force, localisation, expérience, dégâts
  - Méthodes : Frapper, gagner de l'expérience, se déplacer

Définition d’une classe d’objet

- Classes = Définition des objets.
- Analogie : Classe = Moule à gâteaux ; Objets = Gâteaux.
- Classe = Plan de fabrication des objets.

Définition d’une instance d’objet

- Une **instance** est le résultat d'une instanciation. Par exemple :
  - "Gâteau" est l’objet.
  - Le moule est la classe qui définit comment seront les gâteaux.
  - Ce gâteau au chocolat est une instance de la classe "Gâteau".

Un premier exemple en php

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

    public function __construct($force) {
        $this->setForce($force);
        $this->experience = 1;
    }

    public function parler() {
        echo 'Je suis un personnage !';
    }

    public function setForce($force) {
        if (!is_int($force)) {
            trigger_error('La force doit être un nombre entier', E_USER_WARNING);
            return;
        }
        if ($force > 100) {
            trigger_error('La force ne peut dépasser 100', E_USER_WARNING);
            return;
        }
        $this->force = $force;
```

Création d'une classe

```php
<?php
class Personnage // Déclaration de la classe
{
    // Déclaration des attributs et méthodes ici.
}
?>
```

Visibilité d'un attribut ou d'une méthode

- La **visibilité** détermine si un attribut ou une méthode est accessible :
  - **public** : accessible de n'importe où.
  - **private** : accessible uniquement depuis la classe.
- On peut interdire l'accès à un attribut ou une méthode depuis l'extérieur de la classe:
  - c'est ce qu'on appelle l'**encapsulation**.

Création d’attributs

```php
<?php
class Personnage
{
    private $force;          // La force du personnage
    private $localisation;   // Sa localisation
    private $experience;     // Son expérience
    private $degats;         // Ses dégâts
}
?>
```

Historiquement, certains projets préfixaient les éléments privés par un underscore (« _ »). Les conventions modernes (PSR) recommandent d'utiliser la visibilité (private/protected/public) sans underscore.

Valeurs initiales des attributs

Les attributs peuvent être initialisés lors de leur déclaration.

```php
<?php
class Personnage
{
    private $force = 50;
    private $localisation = 'Grenoble';
    private $experience = 1;
    private $degats = 0;
}
?>
```

Exemple : Création de méthodes

```php
<?php
class Personnage
{
    public function deplacer() { }
    public function frapper() { }
    public function gagnerExperience() { }
}
?>
```

Récapitulatif

- Une **classe** est un ensemble de **variables** (attributs) et **fonctions** (méthodes).
- Un **objet** est une instance de classe.
- Les attributs sont généralement **privés** (*encapsulation*).
- Les classes se définissent avec le mot-clé `class`.

Utilisation de la classe

Pour créer et manipuler des objets :

```php
<?php
$perso = new Personnage; // Instanciation d'un objet
$perso->parler();        // Appel d'une méthode
?>
```

Pour créer et manipuler des objets, on va utiliser le mot-clé **new**.
La variable `$perso` sera donc un objet de type Personnage.
On dit que l'on *instancie* la classe Personnage, ou encore que l'on crée *une instance* de la classe Personnage.

Getter/Setter (Accesseurs/Mutateurs)

Pour accéder ou modifier les attributs privés :

```php
<?php
class Personnage
{
    public function degats() {
            return $this->degats;
    }
    public function setForce($force) {
        if (!is_int($force)) return;
        if ($force > 100) return;
            $this->force = $force;
    }
}
?>
```

Constructeur

Une méthode spéciale pour initialiser un objet lors de sa création.

```php
<?php
class Personnage
{
    public function __construct($force, $degats)
    {
        $this->setForce($force);
        $this->setDegats($degats);
        $this->experience = 1;
    }
}
$perso1 = new Personnage(60, 0); // 60 de force, 0 dégât
$perso2 = new Personnage(100, 10); // 100 de force, 10 dégâts
?>
```

- Le constructeur peut avoir des paramètres comme n'importe quelle autre méthode.
- Il est appelé lorsque l'on instancie un objet d'une classe

Chargement des classes

La POO sert, entre autres, à pouvoir structurer et organiser le code des applications.

Une bonne pratique consiste à créer un fichier par classe, et de nommer ce fichier avec le nom de la classe. Pour notre classe Personnage par exemple, on va créer un fichier `Personnage.php` qui contiendra *uniquement* le code de cette classe.

Ensuite, lorsque l'on voudra utiliser cette classe dans un programme, on pourra utiliser la fonction `require`:

```php
<?php
require 'MaClasse.php'; // J'inclus la classe.

$objet = new MaClasse; // Puis, seulement après, je me sers de ma classe.
?>
```

Chargement automatique des classes

Cette manière de procéder est acceptable sur un tout petit projet avec une ou deux classes.
Cependant, une application comporte souvent des centaines de classes!
On ne peut pas les inclure une par une, c'est pourquoi PHP permet de faire de «l'auto-loading», comprenez du chargement automatique.
Ce chargement automatique s'effectue grâce à la fonction: `spl_autoload_register('nom_de_la_fonction')` qui va s'exécuter dès que l'on essaiera d'instancier (de créer un objet) une classe qui n'a pas encore été déclarée avec require.

Pour inclure automatiquement des classes non déclarées :

```php
<?php
function chargerClasse($classe) {
    require $classe . '.php'; // On inclut la classe correspondante au paramètre passé.
}
spl_autoload_register('chargerClasse'); // On enregistre la fonction en autoload pour qu'elle soit appelée dès qu'on in
$perso = new Personnage;
?>
```

Récapitulatif

- Les classes sont déclarées avec `class`.
- Les objets se créent avec `new`.
- L'accès à un attribut ou à une méthode d'un objet se fait grâce à l'opérateur flèche : ->
- Le constructeur d'une classe a pour rôle principal d'initialiser l'objet en cours de création.
- Les attributs privés sont accessibles via des getter/setter.
- L'auto-chargement simplifie la gestion des grandes applications.

Éléments statiques

- Un élément statique est un élément qui appartient à la classe, et non à l'objet. (attribut ou méthode)
- On peut y accéder *sans* avoir besoin d'*instancier* la classe.
- On utilise le mot-clé `static`.
- On accède à un élément statique avec le double deux-points `::`
- Les attributs statiques servent en particulier à avoir des attributs indépendants de tout objet. Ainsi, tous les objets vont partager **la même valeur** pour cet attribut. (exemple: compteur d'instances)
- `$this` fait référence à l'objet en cours, tandis que `self` fait référence à la classe en cours.

- Exemple d'éléments statiques:

```php
<?php
class Personnage
{
    private static $texteADire = 'Bonjour !';

    public static function parler()
    {
        echo self::$texteADire;
    }
}

echo Personnage::$texteADire; // Affiche 'Bonjour !'
Personnage::parler(); // Affiche 'Bonjour !'
?>
```

Exemple compteur d'instances

```php
<?php
class Personnage
{
    private static $compteur = 0;

    public function __construct() {
        self::$compteur++;
    }

    public static function getCompteur() {
        return self::$compteur;
    }
}

$perso1 = new Personnage;
$perso2 = new Personnage;
$perso3 = new Personnage;

echo Personnage::getCompteur(); // Affiche 3
?>
```

Les constantes de classe

- Une constante est une valeur qui ne peut pas être modifiée.
- On les déclare avec le mot-clé `const`.
- On y accède avec le double deux-points `::` (comme pour les éléments statiques).
- Voici un code muet:

```php
$perso = new Personnage(50);
```

  - On ne sait pas à quoi fait référence 50.

- Dans notre projet, on va considérer que le paramètre force ne peut prendre que 3 valeurs distinctes lorsque l'on crée un Personnage:
    - 20, qui veut dire que le personnage aura une faible force
    - 50, qui veut dire que le personnage aura une force moyenne
    - 80, qui veut dire que le personnage sera très fort

```php
<?php
class Personnage
{
    const FORCE_PETITE = 20;
    const FORCE_MOYENNE = 50;
    const FORCE_GRANDE = 80;

private $force;

    public function __construct($force)
    {
        if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]))
        {
            $this->force = $force;
        }
    }
}

$perso = new Personnage(Personnage::FORCE_MOYENNE);
?>
```

Récapitulatif

- L'opérateur `->` permet d'accéder à un élément d'un **objet**, tandis que `::` permet d'accéder à un élément de la **classe**.
- Dans une classe on accède à l'**objet** en cours avec `$this`, et à la **classe** en cours avec `self`.
- Les éléments **statiques** ainsi que les constantes de classe sont des éléments **propres à la classe** et sont accessibles **sans** avoir besoin d'**instancier** la classe.

Héritage

- L'héritage est un principe de la POO qui permet de créer une nouvelle classe à partir d'une classe existante.
- La nouvelle classe hérite des attributs et des méthodes de la classe existante.
- La classe existante est appelée **classe mère** ou **classe parente**.
- La nouvelle classe est appelée **classe fille** ou **classe enfant**.
- L'héritage permet de réutiliser du code, et de créer des classes plus spécialisées.
- La classe fille peut redéfinir les méthodes de la classe mère.
- La classe fille peut ajouter de nouvelles méthodes.
- La classe fille peut ajouter de nouveaux attributs.
- Pour qu'un héritage soit possible, il faut qu'on puisse dire que B est un A. Par exemple, un chien est un animal. Un magicien est un personnage.

L'héritage en PHP

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

    public function __construct($force, $degats)
    {
        $this->setForce($force);
        $this->setDegats($degats);
        $this->experience = 1;
    }

    public function frapper(Personnage $persoAFrapper)
    {
        $persoAFrapper->recevoirDegats($this->force);
    }

    public function gagnerExperience()
    {
        $this->experience++;
    }
}
?>
```

```php
class Magicien extends Personnage
{
    private $magie;

    public function lancerUnSort(Personnage $perso)
    {
        $perso->recevoirDegats($this->magie);
    }

    public function gagnerExperience()
    {
        // On appelle la méthode gagnerExperience de la
        parent::gagnerExperience();
        if ($this->magie < 100)
        {
            $this->magie += 10;
        }
    }
}
?>
```

- Pour hériter d'une classe, on utilise le mot-clé `extends`.

Protected

- Rappel:
  - `public` : accessible de n'importe où.
  - `private` : accessible uniquement depuis la classe.
- `protected` : accessible depuis la classe et les classes filles.

Imposer des contraintes

- Il est possible d'imposer des contraintes.
- On parlera d'abstraction ou de finalisation.

Abstraction

- Une classe abstraite est une classe qui ne peut pas être instanciée.
- Elle sert de modèle pour d'autres classes.
- On ne peut pas instancier une classe abstraite.
- On utilise le mot-clé `abstract`.
- Pour exploiter une classe abstraite (et de ses méthodes) , il faut créer une classe fille qui hérite de la classe abstraite.
- Exemple:
  - On pourrait créer une classe abstraite `Personnage`, et des classes filles `Guerrier`, `Magicien`, etc.
  - Si on essaie d'instancier une classe abstraite, on obtient une erreur fatale.
  - Cela garantie que l'on ne créera jamais d'objet `Personnage`

```php
<?php
abstract class Personnage
{
}
class Guerrier extends Personnage
{
}
$guerrier = new Guerrier; // OK
$perso = new Personnage; // Erreur fatale
?>
```

Méthodes abstraites

- Une méthode abstraite est une méthode qui n'a pas de corps.
- Elle doit être redéfinie dans les classes filles.
- On utilise le mot-clé `abstract`.
- Exemple:
  - On pourrait créer une méthode abstraite `attaquer` dans la classe abstraite `Personnage`.
  - Les classes filles `Guerrier`, `Magicien` devront redéfinir cette méthode.

```php
<?php
abstract class Personnage
{
    abstract public function attaquer();
}
class Guerrier extends Personnage
{
    public function attaquer()
    {
        // Code de l'attaque
    }
}
?>
```

Finalisation

- On peut empêcher la redéfinition d'une méthode ou d'une classe.
    - On utilise le mot-clé `final`.
        - Exemple:
        - On pourrait créer une méthode `gagnerExperience` dans la classe `Personnage`.
        - On pourrait empêcher les classes filles de redéfinir cette méthode.

```php
<?php
class Personnage
{
    final public function gagnerExperience()
    {
        // Code de l'expérience
    }
}
class Guerrier extends Personnage
{
    public function gagnerExperience() // Erreur fatale
    {
        // Code de l'expérience
    }
}

?>
```

Récapitulatif

- L'héritage permet de créer des classes plus spécialisées.
- Une classe fille hérite des attributs et des méthodes de la classe mère (`public` et `protected`).
- On utilise le mot-clé `extends` pour hériter d'une classe.
- Il est possible d'interdire l'instanciation d'une classe avec le mot-clé `abstract`.
- Il est possible d'obliger les classes filles à redéfinir une méthode avec le mot-clé `abstract`.
- Il est possible d'interdire la redéfinition d'une méthode ou d'une classe avec le mot-clé `final`.