PHP OBJET

La programmation procédurale est inefficace.

La programmation orientée objet (ou POO en abrégé) correspond à une autre manière d'imaginer, de construire et d'organiser son code.

Jusqu'à présent, nous avons codé de manière procédurale, c'est-à-dire en écrivant une suite de procédures et de fonctions dont le rôle était d'effectuer différentes opérations sur des données généralement contenues dans des variables et ceci dans leur ordre d'écriture dans le script.

La programmation orientée objet est une façon différente d'écrire et d'arranger son code autour de ce qu'on appelle des **objets**. Un objet est une entité qui va pouvoir contenir un ensemble de fonctions et de variables.

L'idée de la programmation orientée objet va donc être de grouper des parties de code qui servent à effectuer une tâche précise ensemble au sein d'objets afin d'obtenir une nouvelle organisation du code.

Les intérêts principaux de la programmation orientée objet vont être une structure générale du code plus claire, plus modulable et plus facile à maintenir et à déboguer.

La programmation orientée objet va introduire des syntaxes différentes de ce qu'on a pu voir jusqu'à présent et c'est l'une des raisons principales pour lesquelles le POO en PHP est vu comme une chose obscure et compliquée par les débutants.

Au final, si vous arrivez à comprendre cette nouvelle syntaxe et si vous faites l'effort de comprendre les nouvelles notions qui vont être amenées, vous allez vous rendre compte que la POO n'est pas si complexe : ce n'est qu'une façon différente de coder qui va amener de nombreux avantages.

Pour vous donner un aperçu des avantages concrets de la POO, rappelez-vous du moment où on a découvert les fonctions prêtes à l'emploi en PHP. Aujourd'hui, on les utilise constamment car celles-ci sont très pratiques : elles vont effectuer une tâche précise sans qu'on ait à imaginer ni à écrire tout le code qui les fait fonctionner.

Maintenant, imaginez qu'on dispose de la même chose avec les objets : ce ne sont plus des fonctions mais des ensembles de fonctions et de variables enfermées dans des objets et qui vont effectuer une tâche complexe qu'on va pouvoir utiliser directement pour commencer à créer des scripts complexes et complets !

Les frameworks PHP, comme Laravel, Symfony et CodeIgniter, reposent exclusivement sur un PHP orienté objet.
Les systèmes CMS, WORDPRESS,Drupal 8 et October, sont orientés objet à leur base.

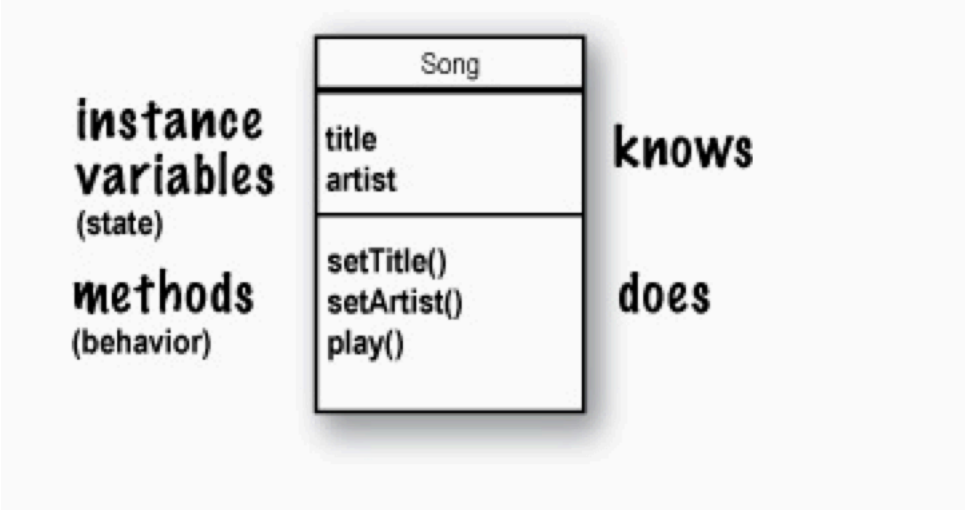
La programmation orientée objet est un style de programmation dans lequel il est d'usage de regrouper toutes les variables et fonctions d'un sujet particulier dans une seule classe.

La programmation orientée objet est considérée comme plus avancée et plus efficace que le style de programmation procédural. Cette efficacité découle du fait qu'elle permet une meilleure organisation du code, offre une modularité et réduit le besoin de se répéter. Ceci étant dit, nous pouvons toujours préférer le style procédural dans des projets simples et de petite taille. Cependant, lorsque nos projets deviennent plus complexes, nous avons intérêt à utiliser le style orienté objet.

- **class**
- **object**
- **method**
- **property**

Nous déclarons la classe avec le mot-clé **class**. Nous écrivons le nom de la classe et mettons la première lettre en majuscule. Si le nom de la classe contient plus d'un mot, nous mettons une majuscule à chaque mot. C'est ce qu'on appelle Camel case. Par exemple, JapaneseCars, AmericanIdol, EuropeTour, etc. Nous plaçons notre code entre les accolades.

Remember: a class describes what an object knows and what an object does



Encapsulation ou dissimulation d’informations

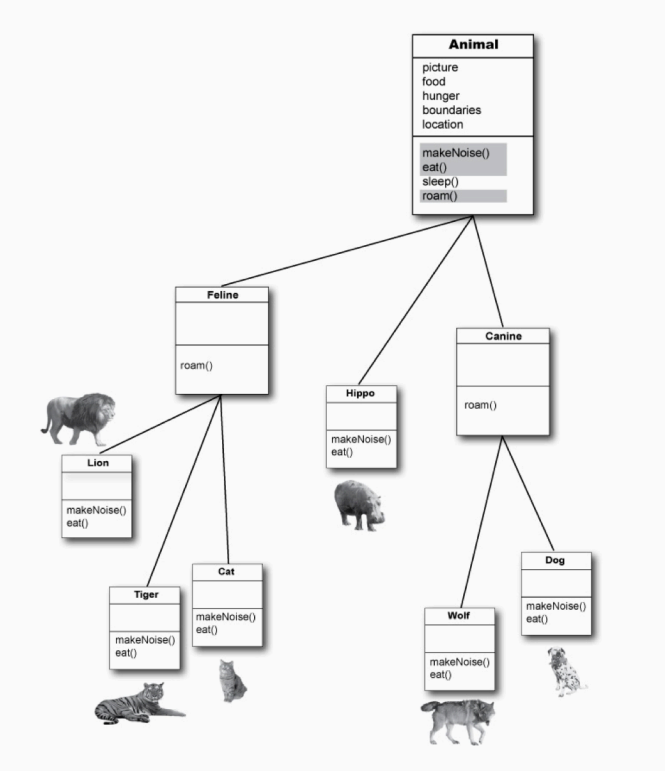
L'encapsulation consiste simplement à envelopper certaines données dans un objet. Le terme "encapsulation" est souvent utilisé de manière interchangeable avec "masquage d'informations".

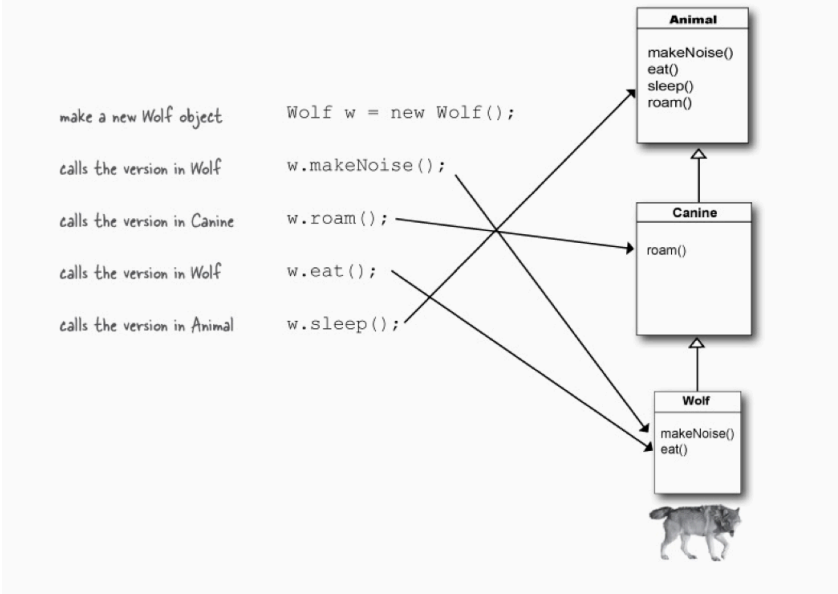
Le fait de cacher les données internes de l'objet protège son intégrité en empêchant les utilisateurs de placer les données internes du composant dans un état invalide ou incohérent.

Vous pouvez utiliser l'encapsulation si les propriétés d'un objet sont privées et que le seul moyen de les mettre à jour est d'utiliser des méthodes publiques.

```php
class User {
 private $name;
 private $gender;

 public function getName() {
  return $this->name;
 }

 public function setName($name) {
  $this->name = $name;
  return $this;
 }

 public function getGender() {
  return $this->gender;
 }

 public function setGender($gender) {
  if ('male' !== $gender and 'female' !== $gender) {
   throw new Exception('Set male or female for gender');
  }
  $this->gender = $gender;
  return $this;
 }
}

$user = new User();
$user->setName('Michal');
$user->setGender('male');
```

Héritage


Animal
picture
food
hunger
boundaries
location
makeNoise()
eat()
sleep()
roam()
Feline
roam()
Canine
roam()
Hippo
makeNoise()
eat()
Lion
makeNoise()
eat()
Cat
makeNoise()
eat()
Tiger
makeNoise()
eat()
Dog
makeNoise()
eat()
Wolf
makeNoise()
eat()


make a new Wolf object
Wolf w = new Wolf();
calls the version in Wolf
w.makeNoise();
calls the version in Canine
w.roam();
calls the version in Wolf
w.eat();
calls the version in Animal
w.sleep();
Animal
makeNoise()
eat()
sleep()
roam()
Canine
roam()
Wolf
makeNoise()
eat()

```php
<?php
class Personnage
{
  private $_experience = 50;

  public function afficherExperience()
  {
    echo $this->_experience;
  }
}

$perso = new Personnage;
$perso->afficherExperience();


//-

<?php
class Personnage
{
  private $_experience = 50;

  public function afficherExperience()
  {
    echo $this->_experience;
  }

  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}

$perso = new Personnage;
$perso->gagnerExperience();   // On gagne de l'expérience.
$perso->afficherExperience(); // On affiche la nouvelle valeur de
l’attribut.
```

Les méthodes peuvent avoir des paramètres

CONSTRUCTEUR:

Les objets peuvent être créés avec des arguments , PHP utilise pour cela un **constructeur**

exemple:

```
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  public function __construct($force, $degats) // Constructeur
demandant 2 paramètres
  {
    echo 'Voici le constructeur !'; // Message s'affichant une
fois que tout objet est créé.
    $this->setForce($force); // Initialisation de la force.
    $this->setDegats($degats); // Initialisation des dégâts.
    $this->_experience = 1; // Initialisation de l'expérience à 1.
  }
```

auto-chargement des classes

Il est préférable de créer chaque classe dans un fichier ainsi l'utilisation d'une classe se fera en important la classe dans un fichier plus global qui utilise cette classe avec la fonction **require**.

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

$objet = new MaClasse;
```

Pour éviter d'effectuer cette fonction pour plusieurs classes
//--

```
<?php
function chargerClasse($classe)
{
  require $classe . '.php'; // On inclut la classe correspondante
au paramètre passé.
}

$perso = new Personnage(); // Instanciation de la classe
Personnage qui n'est pas déclarée dans ce fichier.

//-  ça ne marche pas encore car la classe n'est pas déclarée
(utilisation de spl_autoload_register)
```

```
<?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
instanciera une classe non déclarée.

$perso = new Personnage;
```

CONSTANTES

```
<?php
class Personnage
{
  // Je rappelle : tous les attributs en privé !

  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Déclarations des constantes en rapport avec la force.

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  public function __construct()
  {

  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }
}
```

On accède à ces valeurs avec l’opérateur

```
          self::CONSTANTE
```

Attributs et méthodes statiques

Un attribut statique appartient à la classe et non à un objet. Ainsi, tous les objets auront accès à cet attribut et cet attribut aura la même valeur pour tous les objets.

Cet attribut statique s'accède aussi

```
 public static function parler()
{
     echo self::$_texteADire; // On donne le texte à dire.
}
```

Les attributs statiques servent en particulier à avoir des attributs *indépendants de tout objet*. Ainsi, vous aurez beau créer des tas d'objets, votre attribut aura toujours la même valeur

Classe abstraite:

on ne peut pas se servir directement de la classe.
on ne peut pas l'instancier
on pourra se servir des méthodes héritant de cette classe.

```
<?php
abstract class Personnage // Notre classe Personnage est
abstraite.
{

}

class Magicien extends Personnage // Création d'une classe
Magicien héritant de la classe Personnage.
{

}

$magicien = new Magicien; // Tout va bien, la classe Magicien
n'est pas abstraite.
$perso = new Personnage; // Erreur fatale car on instancie une
classe abstraite.

On s'efforcera d'écrire des classes filles pour toutes les classes
héritant de cette classe

//______

<?php
abstract class Personnage
{
  // On va forcer toute classe fille à écrire cette méthode car
chaque personnage frappe différemment.
  abstract public function frapper(Personnage $perso);

  // Cette méthode n'aura pas besoin d'être réécrite.
  public function recevoirDegats()
  {
    // Instructions.
  }
}

class Magicien extends Personnage
{
  // On écrit la méthode « frapper » du même type de visibilité
que la méthode abstraite « frapper » de la classe mère.
  public function frapper(Personnage $perso)
  {
    // Instructions.
  }
}
```