Introduction à la programmation orientée objet (POO)

I) Classe et objet : syntaxe de base

1) Le principe d'encapsulation

La POO a été inventée pour simplifier la réutilisation de morceaux de code.

Programmer de manière orientée objet, c'est créer du code source (potentiellement complexe), mais que l'on masque en le plaçant à l'intérieur d'un **objet** à travers lequel on ne voit pas ce code source. C'est le principe d'**encapsulation**.

Ce code source pourra alors être utilisé par un autre développeur à travers des objets. Le développeur pourra utiliser ces objets pour leur faire faire des actions sans avoir besoin de connaître le code qu'ils contiennent.

Ex : Un distributeur de billet est un objet qu'un utilisateur peut utiliser en appuyant sur des boutons sans pour autant comprendre le fonctionnement interne de l'objet.

Ex : PDO.

2) Création d'une classe

Lorsqu'on crée une classe, on la place dans un fichier à son nom avec une majuscule : Vehicule.php.

```
class Vehicule
{
   public $name;
   public $roue;

   public function accélérer()
   {
      echo 'Vroum Vroum !!!'
   }
}
```

Il y a 2 types d'éléments dans la classe :

- Des attributs (= variables de classe)
- Des méthodes (= fonctions de classe)

3) Instanciation et utilisation d'un objet

Pour utiliser une classe, il faut **l'instancier** pour créer un objet, on peut ensuite utiliser ses méthodes et ses attributs :

```
require("Vehicule.php");
$dacia = new Vehicule();        //Instanciation (création) de l'objet $dacia avec la classe (le moule) Vehicule

$dacia->accelerer();            // Affiche : Vroum Vroum !!!

$dacia->name = 'Dacia';                         // On initialise la variable name de l'objet $dacaia à 'Dacia'
echo  'Ce véhicule est une '. $dacia->name;     // Affiche : Ce véhicule est une Dacia
```

II) Quelques méthodes de classe utiles

1) Visibilité, Getter et setters

Dans la classe Vehicule on remarque que devant les attributs et les méthodes, il y a le mot clé public pour définir la visibilité de l'attribut ou de la méthode. Cela signifie qu'on peut y accéder hors de la classe comme on l'a fait quelques lignes plus hautes.

Cela peut poser problème, car on a la possibilité de fixer le nombre de roues pour le véhicule à 13 :

```
$dacia->roue = 13;          //Un véhicule à 13 roues n'a pas de sens
```

Il faut pouvoir interdire la possibilité de fixer autant de roues que l'on souhaite. C'est pour cela que généralement :

- Les variables sont définies comme private (inaccessible en dehors de la classe) pour les protéger de modifications hasardeuses.
- Les méthodes sont définies comme public (accessible en dehors de la classe)

On peut donc réécrire la classe Vehicule :

```
class Vehicule
{
   private $name;
   private $roue;

   public function accélérer()
   {
      echo 'Vroum Vroum !!!'
   }
}
```

Avec ces nouvelles visibilités, le code précédent lève une erreur :

```
require("Vehicule.php");
$dacia = new Vehicule();        //Instanciation de l'objet $dacia

$dacia->accelerer();            // Affiche : Vroum Vroum !!!

$dacia->name = 'Dacia';         // Lève une erreur car on essaye d'accéder hors de la classe à un attribut private
```

Pour résoudre ce problème, on va créer dans la classe Vehicule des nouvelles méthodes, appelées **getters**, qui permettront d'accéder aux l'attribut name et roue :

```
class Vehicule
{
   private $name;
   private $roue;

   public function accélérer()
   {
      echo 'Vroum Vroum !!!'
   }

   public function getName()
   {
      return $this->name;          // Une méthode ou un attribut doit être appelés sur un objet.
   }                               Ici aucun objet existe, car nous sommes dans la classe.
                                   On utilise alors $this pour les appeler sur la classe elle-même.
   public function getRoue()
   {
      return $this->roue;
   }
}
```

Ainsi, il est maintenant possible de récupérer la variable name de l'objet $dacia, en faisant : $dacia->getName()
On peut récupérer l'attribut name, mais on ne peut plus fixer sa valeur. C'est exactement pour cela qu'on a passé la visibilité en private, afin de se protéger des modifications hasardeuses.

Pour résoudre ce problème, on va créer dans la classe Vehicule des nouvelles méthodes, appelées **setters**, qui permettront de fixer les valeurs des attribut name et roue tout en se laissant la possibilité de contrôler leurs valeurs.

```
class Vehicule
{
```

```
    private $name;
    private $roue;

    public function accélérer()
    {
        echo 'Vroum Vroum !!!'
    }

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

    public function setName(string $name)              //name doit être une chaine de caractère
    {
        $this->name = $name;
    }

    public function getRoue()     // roue doit être un entier entre 2 et 4 (2 et 3 roues pour une moto et 4 pour une voiture).
    {
        return $this->Roue;
    }

    public function setRoue(int $number)
    {
        if ($number >=2 && $number <= 4) {
            $this->roue = $number;
    }
}
```

Ainsi pour que le code de départ soit fonctionnel et protégé de modifications hasardeuses, on écrira :

```
require("Vehicule.php");
$dacia = new Vehicule();        //Instanciation de l'objet $dacia

$dacia->accelerer();            // Affiche : Vroum Vroum !!!

$dacia->setName('Dacia');                   // On initialise la variable name de l'objet $dacaia à 'Dacia' grâce au setter
echo  'Ce véhicule est une '. $dacia->getName();     // Affiche : Ce véhicule est une Dacia grâce au getter
```

2) Récapitulatif sur la visibilité

Il existe 3 types de visibilités :

- private : Accessible uniquement à l'intérieur de la classe.
- protected : Accessible à l'intérieur de la classe et de ses classes filles.
- public : Accessible partout, c'est-à-dire dans la classe et en dehors de la classe (dans le code).

3) La méthode magique __construct

__construct est une méthode qui se déclenche automatiquement lorsqu'un objet est instancié (new Objet())

```
class Vehicule
{
    private $name;
    private $roue;

    public function __construct ($name) {
        $this->name = $name;
    }
```

```
    public function accélérer()
    {
        echo 'Vroum Vroum !!!'
    }

    // getters et setters
}
```

Grâce au constructeur on peut déterminer l'attribut name dès l'instanciation de $dacia en passant le nom en paramètre :

```
require("Vehicule.php");
$dacia = new Vehicule('Dacia');              //Instanciation de l'objet $dacia

$dacia->accelerer();                 // Affiche : Vroum Vroum !!!

echo  'Ce véhicule est une '. $dacia->getName();      // Affiche : Ce véhicule est une Dacia
```

III) Le principe de l'héritage

Si on peut dire "Classe A EST une Classe B", alors c'est que cela a du sens de faire un héritage.

- "Une Voiture Est un Vehicule" (donc Voiture hérite de Véhicule)
- "Une Moto Est un Vehicule" (donc Moto hérite de Véhicule)

Pour créer une classe Voiture qui **hérite** de Vehicule, on utilise le mot clef extends.
On dit aussi que Vehicule est la classe **mère** et Voiture la classe **fille**.

```
require ('Vehicule.php);
class Voiture extends Vehicule
{

 }
```

La classe Voiture, héritant de la classe Vehicule, va récupérer tous les attributs et méthodes de véhicule a conditions qu'ils soient public (accessible hors de la classe) ou protected (accessible hors de la classe uniquement dans les classe filles).
Récrivons la classe Vehicule pour qu'elle transmette à Voiture tous ses attributs.

```
class Vehicule
{
    protected $name;
    protected $roue;

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

    public function accélérer()
    {
        echo 'Vroum Vroum !!!'
    }

    // getters et setters
}
```

Si on le souhaite, en plus des attributs et méthodes hérités de Vehicule, on peut en ajouter de nouveaux dans Voiture.

```
require ('Vehicule.php);
class Voiture extends Vehicule
{
    private $porte       // Tous les véhicules n'ayant pas de porte (ex : moto), cet attribut n'a pas à être défini dans Vehicule.
}
```

On peut aussi réécrire des attributs ou méthodes présentes dans la classe mère afin de décrire des comportements différents :

```
require ('Vehicule.php);
class Moto extends Vehicule
{
   Public function accelere()
   {
      echo "Vrimmmmmmmmmmmmmm !!!'
   }
}
```

On pourra alors écrire le code ci-dessous :

```
require("Moto.php");
$yamaha = new Moto('Yamaha');          // Instanciation de l'objet $yamaha

$yamaha->accelerer();                  // Affiche : Vrimmmmmmmmmmmmmm !!!

echo  'Ce véhicule est une '. $yamaha->getName();  // Affiche : Ce véhicule est une Yamaha
```

IV) Les namespaces

1) Le rôle des namespaces

La POO ayant été créée pour permettre la réutilisation de code entre développeur, il n'est pas rare de voir des classes créées par des développeurs différents portant le même nom.
Comme pour les fonctions, deux classes portant le même nom ne peuvent coexister dans le même code. Les **namespaces** ont donc été inventés pour éviter les collisions de noms de classes.

2) Utilisation des namespaces

Avant de définir une classe, on a la possibilité de définir son **namespace** qui représentera une architecture de dossier virtuelle dans laquelle la classe se situe.

Imaginons que Jojo le développeur ai créé un bundle nommé JojoBundle qu'il laisse à la disposition de tous.
Dans son code, une classe Voiture se trouve dans le répertoire Model/Vehicule.
La classe Voiture a été définie avec un namespace (grâce au mot clef namespace) reprenant l'architecture de ses dossiers.

```
namespace JoJoBundle\Model\Vehicule          //attention, on utilise ici un antislash

class Voiture
{
   // code pour définir la classe
}
```

Pour utiliser cette classe Voiture, on écrira :

```
require (JojoBundle/Model/Vehicule/Voiture.php);
$dacia = new JojoBundle\Model\Vehicule\Voiture('Dacia');
```

Si je souhaite créer ma propre classe Voiture, il n'y aura plus de confusion possible entre la sienne et la mienne :

```
require (JojoBundle/Model/Vehicule/Voiture.php);          // Classe créée par Jojo
require ('src/Model/Voiture.php);                         // Classe créée par moi avec le namespace App\Model

$dacia = new JojoBundle\Model\Vehicule\Voiture('Dacia');
$ferrari = new App\Model\Voiture('Ferrari');
```

Remarque : En générale, tous le code que l'on crée est placé dans un dossier à la racine du projet nommé src. On remplace en général src par App dans le namespace.

3) Eviter la répétition du namespace

Le problème de cette solution est que les classes deviennent lourdes à écrire. Pour ne pas à avoir à réécrire tous le namespace, on utilise le mot clef use au début du fichier.

```
use JojoBundle\Model\Vehicule\Voiture;
require (JojoBundle/Model/Vehicule/Voiture.php);

$dacia = new Voiture('Dacia');
```

Attention : si deux classes portant le même nom sont utilisées dans le même script, alors soit on les écrit avec leur namespace, soit on utilise le mot clef as pour les différencier

```
use JojoBundle\Model\Vehicule\Voiture as JojoVoiture;
use App\Model\Vehicule\Voiture as MaVoiture;

require ('Bundle/CamileBundle/Model/Vehicule/Voiture.php);
require ('src/Model/Voiture.php);

$dacia = new JojoVoiture('Dacia');
$ferrari = new MaVoiture('Ferrari');
```

4) Autoloader de composer

Le problème de l'utilisation des namespaces est qu'on a résolu un problème en alourdissant le code avec des use et des require avant de pouvoir utiliser les classes.

Pour régler ce problème, on va utiliser un autoloader qui va charger automatiquement les classes sans avoir besoin de charger le fichier avec require.

Pour installer un autoloader, entrez dans votre conteneur php : docker exec -ti php sh
Initialiser composer : composer init
Répondez aux questions en acceptant le chargement de l'autoloader.
Dans votre index.php, appelez l'autoloader : require 'vendor/autoload.php';

Si composer est déjà initialisé vous pouvez gérer le chargement de l'autoloader manuellement.
En chargeant l'autoloader : composer dump-autoload
Et en configurant l'autoloading PSR-4 dans le composer.json

```
"autoload": {
        "psr-4": {
                "App\\": "src/"
        }
}
```

Maintenant, vous devez respecter la norme PSR-4 et toutes vos classes devront avoir un namespace qui correspond à votre architecture des dossiers en remplaçant src/ par MonProjet\App.

Ainsi grâce au use et à l'autoloader, vous n'avez plus besoin d'utiliser require.

Si l'autoloader est bien chargé dans index.php, le code peut maintenant s'écrire :

```
use JojoBundle\Model\Vehicule\Voiture as JojoVoiture;
use App\Model\Vehicule\Voiture as MaVoiture;

$dacia = new JojoVoiture('Dacia');
$ferrari = new MaVoiture('Ferrari');
```