Php Data Object

Ce cours à pour but de vous former à la programmation orienté objet (POO), nous allons aborder des notions de base du PHP orienté objets en mettant en place une application web avec gestion de données CRUD avec le design pattern MVC.

Pour ce cours, vous devez avoir certains pré-requis en PHP procédurale ainsi que de bonnes bases en HTML/CSS/JS.

Concernant l'IDE de développement, vous êtes libre d'utiliser celui que vous voulez, je vais utiliser VSCODE (Visual Studio Code) qui me permet d'utiliser des extensions très pratique pour la mise en place d'une structure PHP orienté objet.

Pour ceux qui veulent utiliser VsCode ainsi que les extension :

- **Mithril Emmet** : plugin d'autocomplétion pour langage HTML/JS/CSS
- **PHP Getter & Setter** : plugin de génération automatique des accesseurs
- **Php extension Pack** : pack de plugin pour le développement en PHP
- **Php namespace resolveur** : plugin d'autocomplétion de namespace
- **Php docBlocker** : plugin qui permet de commenter automatiquement et de pré remplir les commentaires
- **Phpcs** : plugin de validation de syntaxe PHP

Introduction

Tout d'adord, le PHP orienté objet, qu'est-ce que c'est ?

En PHP, nous avons plusieurs type de variables :

- Entier
- Décimal
- Chaîne de caractère
- Tableaux

Ces variables sont dites "statiques" étant donnée qu'elles permettent de stocker une valeur.

Les objets quant à eux, sont beaucoup plus dynamique étant donnée que nous allons pouvoir faire des actions particulières sur ces objets.

Pour mieux comprendre, nous allons vulgariser un petit peu les objets en PHP.

Si on prend plusieurs objets du quotidien :

- Une voiture
- Un ordinateur
- Une télévision

Chacun de ces objets ont des caractéristiques propre à ce type d'objet

- La voiture

    - A une couleur,
    - A un nombre de place
    - A un nombre de roues
    - Etc....
- Un ordinateur
    - A une marque,
    - A un nombre de mémoire RAM
    - Etc...
- Une télévision
    - A une marque
    - A une taille d'écran
    - Etc...

Mais chacun de ces objets ont également des actions :

- La voiture
    - On peut la démarrer
    - On peut l'éteindre
    - On peut avancer

... Vous avez compris le principe !

En programmation orienté objet (POO), les caractéristiques d'un objet sont appelé des **propriétés**, alors que les actions que les objets peuvent faire s'appelles des **méthodes**.

Par exemple, pour cette introduction, nous allons créer un **objet de type compte bancaire**.

Cet objet compte Bancaire aura des caractéristiques :

- Titulaire
- Solde
- ...

Et sur cet objet, nous allons pouvoir effectuer des actions :

- Déposer de l'argent
- Retirer de l'argent
- Voir le solde
- ...

Avant de débuter la pratique, nous devons aborder les conventions à adopter en POO.

Les conventions

Chaque objet "classe" sera dans son propre fichier, une classe est simplement la définition d'un type d'objet, c'est dans ce fichier/classe que nous allons définir les **propriétés** ainsi que les **méthodes**.

Le nom du fichier doit être identique à celui de la classe (si nous avons une classe User, le fichier doit s'appeler User.php)

Le nom de la classe est écrit en UpperCamelCase -> camelCase avec la première lettre en majuscule.

Création de l'objet

Nous allons maintenant créer notre premier objet !

Un objet en PHP se définit par le mot clé "class". Nous allons donc créer l'objet Compte dans un fichier intitulé **Compte.php** (Notez que nous mettons un **C** majuscule pour respecter le upperCamelCase).

Pour organiser notre projet, nous allons créer ce fichier dans un dossier **classes** qui contiendra tous nos fichier de classes.

Dans ce fichier nous allons déclarer la classe de l'objet compte :

```
class Compte
{

}
```

Une fois que nous avons créer la classe, nous allons pouvoir définir à l'intérieur de cette classe les propriétés de l'objet. Notre compte aura un titulaire et un solde, nous devons donc créer 2 variables public (nous reviendrons plus tard sur le mot clé public) :

```
class Compte
{
  public $titulaire;
  public $solde;
}
```

Pour garder un code propre ainsi que du code maintenable, pensez à commenter vos propriétés, méthodes ainsi que vos classes :

```
/**
* Classe correspondant à un compte bancaire
*/
class Compte
{
  /**
  * Titulaire du compte
  * @var string
  */
  public $titualire;

  /**
  * Solde du compte
  * @var float
  */
```

```
  public $solde;
}
```

N'hésitez pas à prendre cette habitude, un code commenté, c'est la garantie d'un code simple à maintenir et à utiliser.

Instancier l'objet

Maintenant que nous avons déclaré la classe, nous pouvons l'appeler à plusieurs reprises dans des variables différentes.

Cette manipulation s'appelle '**L'instanciation d'objet**' qui permet de créer plusieurs exemplaires de notre objet, chaque exemplaire étant appelé une **Instance**.

Pour créer une instance de notre classe compte bancaire, nous allons créer un fichier index.php, dans lequel nous allons créer notre objet 2 fois.

Dans un premier temps, nous allons devoir appeler le fichier Compte.php afin de pouvoir utiliser la classe qu'il contient :

```
require_once('classes/Compte.php');
```

Une fois que nous avons le fichier, nous pouvons l'instance à 2 reprises :

```
// Première instance de la classe Compte
$compte1 = new Compte();

// Deuxième instance de la classe Compte
$compte2 = new Compte();
```

Si vous faite un `var_dump($compte1)` vous allez avoir le résultat suivant :

```
/app/index.php:7:
object(Compte)[1]
  public 'titulaire' => null
  public 'solde' => null
```

Nous voyons ici que notre objet appartient à la classe compte et que cet objet hérite des propriétés de la classe (titulaire et compte), mais pour le moment, c'est propriétés sont vides, c'est normal, nous ne les avons pas encore définit, c'est ce que nous allons faire maintenant :

```
// On attribut un titulaire
$compte1->titulaire = "Pierre";

// On attribut un solde
$compte1->solde = 500;
```

Maintenant, le var_dump nous renvoie :

```
/app/index.php:13:
object(Compte)[1]
  public 'titulaire' => string 'Pierre' (length=6)
  public 'solde' => int 500
```

Le constructeur

Comme nous venons de le voir, quand on instancie un objet, **ses propriété ne sont pas remplies**.

Nous allons donc créer une méthode (fonction) spécifique appelé constructeur qui va permettre de **construire notre objet lorsque nous l'instancions**.

Cette méthode va être **exécuter automatiquement lors d'une instanciation** de l'objet.

Nous allons modifier la classe, donc nous retournons sur notre fichier Compte.php :

```php
/**
* Constructeur de la classe Compte
* @param string $titulaire Nom du titulaire du compte
* @param float $solde Solde du compte
*/
public function __construct(string $titulaire, float $solde)
{
  // On affecte le titulaire à la propriété titulaire
  $this->titulaire = $titulaire;

  // On affecte le montant à la propriété solde
  $this->solde = $solde;
}
```

Vous remarquez que nous avons utiliser le `$this` qui représente l'instance de classe que nous utilisons (l'objet que nous avons créé dans le fichier index.php).

Étant donné que cette fonction `__construct()` est exécuté automatiquement lors de l'instanciation de cette classe, ça veut dire que lorsqu'on appelle cette classe, nous allons devoir lui passer 2 paramètres : **le titulaire** et **le solde**, sinon nous allons avoir une erreur.

Donc nous allons devoir modifier notre fichier index.php :

```php
// Première instance de la classe Compte
$compte1 = new Compte("Pierre", 500);
```

En faisant ça, nous avons créer une instance de la classe Compte, Mais en même temps, nous lui avons définit le nom du titulaire ainsi que le solde du compte.

Chaque variable que vous passez au construct de la classe doit être rentrée lors de l'instanciation.

Mais pouvez également définir dans votre construct une valeur par défaut, ce que veut dire que si la valeur n'est pas envoyé lors de l'instanciation, on attribut une valeur par défaut à une propriété :

```php
/**
 * Constructeur de notre objet Compte
 * @param string $titulaire Nom du titulaire du compte
 * @param float $solde Solde du compte
 */
public function __construct(string $titulaire, float $solde = 500)
{
    // On affecte le titulaire à la propriété titulaire
    $this->titulaire = $titulaire;

    // On affecte le solde à la propriété solde
    $this->solde = $solde;
}
```

Maintenant, si lors de l'instanciation de la classe Compte, si on ne définit pas le solde du compte, il sera par défaut à 500. Cette méthode permet de ne pas être obliger de passer la valeur lors de la création de l'instance.

Ajouter une méthode

Une méthode est une fonction stockée dans notre classe, qui va permettre de définir les actions de notre objet.

Dans le cas de notre compte, nous devrions pouvoir **afficher le solde**, **déposer** et **retirer** de l'argent.

Pour **afficher le solde du compte**, nous allons créer une méthode que nous placerons directement sous le constructeur.

Notre méthode affichera le solde du compte sous la forme d'un "**echo**" et sera créée comme ceci :

```php
/**
 * Voir le solde du compte
 * @return void
 */
public function showSolde()
{
    echo "Le solde du compte est de $this->solde euros";
}
```

Maintenant, pour afficher le solde du compte1 que nous avons créé, nous avons simplement à écrire dans notre fichier index.php :

```php
$compte1->voirSolde();
```

Maintenant que nous avons fait notre première méthode, nous allons en créer une autre pour retirer de l'argent, cette fois, ce ne sera pas un simple `echo`, nous voulons vérifier d'abord s'il y a assez d'argent sur le compte pour pouvoir le retirer :

```php
/**
* Permet de retirer de l'argent sur le compte
*
* @param float $montant à retirer
* @return void
*/
public function retrait(float $montant)
{
  // On verifie s'il y a assez d'argent et que le montant n'est pas négatif
  if ($montant > 0 && $this->solde >= $montant) {
    $this->solde -= $montant;
  } else {
    echo "Montant invalide ou solde insufisant";
  }
}
```

Les visibilités

Nous avons vu comment mettre en place une **classe**, des **propriétés** et des **méthodes**. Si vous avez bien suivit jusqu'ici, nous avons créé des variables et des fonctions "**public**", pour le moment nous n'avions pas fait le point sur ce mot clé.

En PHP POO, il existe 3 visibilités pour vos méthodes et vos propriétés :

- **public** : La propriété ou la méthode pourront être accessible depuis **l'intérieur ET l'extérieur de la classe**.
- **private** : L'accès à la propriété ou méthode n'est possible uniquement depuis **l'intérieur** de la classe (seulement dans le fichier).
- **protected** : Équivalent à private, mais accessible également dans les classe héritées (nous y reviendrons plus tard).

Donc concrètement, ces visibilités nous permettent de protéger nos propriétés ainsi que les méthodes que nous créons dans une classe.

C'est ce qu'on appelle de l'encapsulation, et cela permet de protéger notre classe (vous ne voulez pas que d'autre fichier/personne puisse modifier votre classe).

Dans 99% des cas, quand nous déclarons des propriétés dans une classe le mot clé n'est pas public, mais **private**.

Pour mieux comprendre, nous allons modifier notre classe en passant nos propriétés en private plutôt qu'en public :

```php
/**
* Titulaire du compte
* @var string
*/
private $titulaire;

/**
* Solde du compte
* @var float
*/
private $solde;
```

Maintenant, si dans notre fichier index.php, nous essayons de redéfinir le nom du titulaire :

```php
$compte1->titulaire = "Paul";
```

Nous allons avoir cette erreur :

Fatal error: Uncaught Error: Cannot access private property Compte::$titulaire in /app/index.php on line 7

Error: Cannot access private property Compte::$titulaire in /app/index.php on line 7

Call Stack

#	Time	Memory	Function	Location
1	0.0158	359320	{main}( )	.../index.php:0

Ce qui est tout a fait normal, nous essayons de modifier une propriété "private" à l'extérieur de la classe, ce qui n'est pas possible.

Les accesseurs

Maintenant que nous avons une belle erreur quand nous essayons de modifier les propriétés, comment faire pour modifier notre objet ?

Avant toute chose, faisons un petit parallèle avec le quotidien, prenons l'exemple d'une voiture.

Vous pouvez utiliser votre voiture sans pour autant devoir faire de la mécanique en modifiant votre moteur non ?

Vous n'avez pas besoin de toucher au moteur pour pouvoir accélérer, parce qu'on vous donne accès à des "commande" sans que vous n'ayez besoin de toucher à la mécanique (afin d'éviter de causer des problèmes moteur).

Le principe de la visibilité et des accesseurs est le même : On va définir les actions qui peuvent être faites sur des propriétés ou méthodes afin d'être sûr que l'utilisateur ne va pas causer de problème dans notre classe.

Afin de pouvoir récupérer le nom du titulaire et également de le définir, nous allons mettre en place des méthode (nos fameux **accesseurs**), nos allons donc créer 2 méthodes :

- 1 getter : qui va permettre de récupérer le nom du titulaire
- 1 setter : qui va permettre de définir le nom du titulaire

Ces méthodes sont à placer en dessous du constructeur et doivent respecter la convention de nommage : getTitulaire() et setTitulaire() :

```php
// Accesseurs
/**
* Récupère le nom du titulaire
*
* @return string
*/
public function getTitulaire(): string
{
  return $this->titulaire;
}

/**
* Définit le nom du titulaire
*
* @param string $titulaire
* @return Compte
*/
public function setTitulaire(string $titulaire): self
{
  $this->titulaire = $titulaire;
  return $this;
}
```

Notez que cette fois, la visibilité de nos accesseurs est public, c'est parce que nous voulons pour récupérer ou modifier le nom du titulaire depuis un autre fichier (à l'extérieur de notre classe).

C'est ce que nous allons voir sur le fichier index.php :

```php
echo $compte1->getTitulaire();

$compte1->setTitulaire("Paul");
```

Maintenant sur le fichier index.php, vous appelez une première fois le nom du titulaire, et ensuite vous le redéfinissez. Voilà comment vous pouvez, de manière sécurisée, laissé possible la modification des propriétés de vos objet.

Faites la même chose pour le solde maintenant.

La methode __toString

Pour information, si vous essayer de faire un echo de votre objet (l'objet entier), vous allez avoir une erreur de PHP (c'est le var_dump qui affiche les infos en mode debug), mais le echo ne pourra pas afficher l'objet.

Pour faire ça, vous allez devoir définir une méthode magique __toString sur votre classe pour qu'automatiquement, si vous souhaitez faire un echo de l'objet, vous renvoyez une chaine de caractère avec les informations de l'objet.

```
/**
* Transforme l'objet en chaine de caractère
*
* @return string
*/
public function __toString()
{
  return "Vous visualez le compte de $this->titulaire, le solde est de {$this->solde}
€";
}
```

Maintenant, si vous faite dans votre fichier index.php :

```
echo $compte1;
```

Vous allez avoir sur votre page le message que vous avez définit dans le __toString de la classe.

Cette méthode permet de ne pas renvoyer d'erreur si on demande à afficher tout un objet et permet de formatter la chaine de caractère qui sera renvoyé.

Les constantes

Une constantes est une variable qui ne peut pas être redéfinit.

Par exemple, pour notre compte, nous voulons créer un taux d'intérêts, ce taux d'intérêt ne pourra pas être redéfinit par la suite.

Pour définir une constante en PHP POO il faut l'écrire :

```
// Constante Taux intérêts
public const INTERETS = 5;
```

Chose importante à savoir sur les constante :

- La convention de nommage veut que nous écrivions le nom de la constante en MAJUSCULE
- Si plusieurs mots, faire du underscore_case => `TAUX_INTERETS`
- On doit définir la valeur de la constante au moment de la déclaration.

Comment accéder à une constante de classe ? C'est un peu particulier, cette fois, pour récupérer une constante de classe, nous allons devoir appeler la classe et ajouter `::` et le nom de la constante.

Par exemple, pour afficher le taux d'intéret de notre compte, nous allons écrire dans le fichier index.php :

```
echo "La taux d'intérêt du compte est : " . Compte::TAUX_INTERETS . "%";
```

Pour afficher le taux d'intérêt, nous appelons d'abord la classe, puis `::`, et ensuite on appelle la constante.

Vous voyez également que sa visibilité est public.

Vous avez maintenant les bases pour mettre en place des objets avec des classes, maintenant, nous allons aborder un sujet très important en POO : l'héritage.

L'héritage

Jusqu'ici, nous avons une classe Compte qui représente un objet compte.

Cependant, il existe plusieurs type de compte bancaire : les comptes courant avec possibilité de découvert, les comptes d'épargne etc..

Pour mettre en place ces 2 types de comptes, nous n'allons pas recréer une classe entière pour chaque objet, nous pouvons créer des "**sous-objets**" qui vont hériter de la classe Compte (qui contient les propriétés principale des comptes).

En PHP, on nous allons créer des classes qui vont hériter de la classe Compte.

Pour ça nous devons d'abord modifier légèrement notre fichier Compte.php :

```
abstract class Compte
{
}
```

Avec le mot clé `Abstract` devant le nom de la classe, cette classe ne pourra pas être instanciée directement, nous devrons forcément créer une classe qui hérite de Compte.

De plus, sur le Compte.php, nous avons définit les propriétés en `private` donc accessible seulement dans la classe, il faut donc changer cette visibilité en la transformant en protected (les propriétés seront disponibles dans les classes héritées).

```php
/**
* Titulaire du compte
* @var string
*/
protected $titulaire;

/**
* Solde du compte
* @var float
*/
protected $solde;
```

Maintenant, nous ne pouvons plus instancier la classe Compte, c'est notre classe principale, tous les type de comptes que nous allons créer vont hériter de cette classe.

Les classes héritées

Nous allons maintenant créer 2 types de compte, le compte courant et le compte et le compte d'épargne.

Pour ça nous devons créer 2 fichiers étant donné que nous voulons créer nouvelles classes.

Commençons avec le compte courant qui aura droit à un découvert. Nous créons donc un fichier nommé CompteEpargne.php

```php
class CompteCourant extends Compte
{
}
```

On dit de cette classe qu'elle étant de la classe Compte, donc on ajoute après le nom de classe le mot clé extends suivi de la classe abstraite Compte.

Ainsi, notre classe CompteEpargne prends automatiquement les propriétés et méthodes de la classe Compte.

Nous faisons pareil pour le compte d'épargne :

```php
// CompteEpargne.php
class CompteEpargne extends Compte
{
}
```

Pour le moment, pour utiliser et instancier nos classes dans le fichier index.php, nous devons appeler les fichiers avec un `require_once`, donc on ajoute les 2 nouvelles classes au fichiers index.php :

```php
require_once 'classes/Compte.php';
require_once 'classes/CompteCourant.php';
require_once 'classes/CompteEpargne.php';
```

Maintenant, si vous essayez d'instancier les classes CompteCourant et CompteEpargne, vous allez voir que c'est possible et qu'il y a déjà les 2 propriétés de notre classe Compte, même si les classe CompteCourant et CompteEpargne sont vide pour le moment.

C'est le principe de l'héritage.

Le compte Courant

Maintenant que nous avons nos classes héritées, nous allons voir comment ajouter des propriétés ainsi que des méthodes à ces classes enfants.

Nous allons commencer avec le compte courant, nous voulons lui rajouter une propriété découvert. Pour ce faire, on le fait de manière classique en ajoutant la propriété dans la classe CompteCourant.

Si nous ajoutons une propriété, nous devons également créer un constructeur pour cette classe afin d'ajouter automatiquement la valeur du découvert au moment de l'instanciation de la classe.

```
/**
* Constructeur de la classe CompteCourant
*
* @param string $titulaire
* @param float $solde
* @param integer $decouvert
*/
public function __construct(string $titulaire, float $solde, int $decouvert = 500)
{
  // On doit appeler le constructeur de la classe parente pour
  // lui passer les propriétés de base
  parent::__construct($titulaire, $solde);

  // On définit la propriétés découvert
  $this->decouvert = $decouvert;
}
```

Ça se complique légèrement, le constructeur de notre classe Compte attend 2 valeur (le nom du titulaire ainsi que le montant du solde).

Donc pour passer les informations au constructeur de la classe parente, on l'appelle simplement avec le mot clé `parent::__construct()` ce mot clé parent fait référence à la classe parente, et ensuite on lui dit quelle méthode on veut utiliser.

Une fois le constructeur de la classe parente appelé, nous avons simplement à lui passer les valeurs.

Maintenant, quand vous allez instancier la classe CompteCourant, vous devrez intégrer 3 valeurs (titulaire, solde et découvert).

Et voilà vous avez votre première classe héritée fonctionnelle !

Mais maintenant, nous avons un problème, sur la classe Compte *(parente)* nous n'avions pas autorisé le retrait si le solde était inférieur à 0, mais le principe d'un découvert autorisé, c'est de pouvoir descendre un peu plus si besoin.

Donc nous allons devoir modifier la méthode retrait, **MAIS PAS directement** depuis la classe Compte, nous voulons simplement réécrire cette méthode seulement pour les objet de type CompteCourant.

Pour ça, vous avez simplement à définir la même méthode dans la classe enfant, avec d'autres instruction :

```
// CompteCourant.php
/**
* Retrait sur le compte avec prise en compte du découvert
*
* @param float $montant
* @return void
*/
public function retrait(float $montant)
{
  // On vérifie si le découvert permet le retrait
  if ($this->solde - $montant > -$this->decouvert) {
    $this->solde -= $montant;
  } else {
    echo 'Solde insufisant';
  }
}
```

Et voilà, maintenant la méthode retrait prend en compte le découvert seulement pour les CompteCourants.

Mais il manque quelque chose dans notre classe CompteCourant vous allez me dire ?

Et bien bravo si c'est le cas, effectivement, il manque les accesseurs de notre propriétés découvert !

Nous l'avons définit en private, il faut donc créer les méthodes pour récupérer la valeur du découvert ainsi que la méthode pour la définir, les **accesseurs**.

Nous allons les créer maintenant :

```
/**
* Get découvert autorisé pour le compte courant
*
* @return int
*/
public function getDecouvert(): int
{
  return $this->decouvert;
}

/**
* Set découvert autorisé pour le compte courant
*
* @param int $decouvert  Découvert autorisé pour le compte courant
```

```
*
* @return CompteCourant
*/
public function setDecouvert(int $decouvert): self
{
  $this->decouvert = $decouvert;

  return $this;
}
```

Maintenant notre classe CompteCourant est un peu plus dynamique !

Le compte d'épargne

Maintenant, sur le compte d'épargne, nous allons devoir procéder comme pour le CompteCourant, mais cette fois, nous voulons créer un taux d'intérêt et pouvoir faire le versement des intérêts sur le compte.

Donc nous devons créer la propriété, créer le constructeur et enfin les accesseurs.

Essayer de le faire seul pour le moment avec ce que nous avons déjà vu ensemble.

Pour ce qui sont largués, voici la correction :

```
/**
* Taux d'intérêt pour le compte d'épargne
*
* @var int
*/
private $interets;

/**
* Constrtucteur de la classe compte épargne
*
* @param string $titulaire
* @param float $solde
* @param integer $taux
*/
public function __construct(string $titulaire, float $solde, int $taux)
{
  parent::__construct($titulaire, $solde);
  $this->interets = $taux;
}

/**
* Get taux d'intérêt pour le compte d'épargne
*
* @return int
*/
```

```php
public function getInterets(): int
{
  return $this->interets;
}

/**
* Set taux d'intérêt pour le compte d'épargne
*
* @param int $interet  Taux d'intérêt pour le compte d'épargne
*
* @return CompteEpargne
*/
public function setInterets(int $interets): self
{
  $this->interets = $interets;

  return $this;
}
```

Et voilà, votre classe Compte Épargne est quasiment finit, il manque simplement de créer la méthode de versement d'intérêts :

```php
/**
* Permet le versement des intérêts
*
* @return void
*/
public function verserInterets()
{
  if ($this->solde > 0) {
    $this->solde = $this->solde + ($this->solde * $this->interets / 100);
  } else {
    echo "Solde insufisant";
  }
}
```

Maintenant, vous avez de bonnes bases pour passer à la suite, les namespaces et l'autoloading !

Les namespaces

Bon pour le moment, nous avons 3 classes qui ont toutes un nom différents, donc jusqu'ici, pas de problème.

Mais nous allons aborder durant ce module le design MVC, qui nous oblige à suivre une structure et organisation de dossiers rigoureuse, nous allons donc devoir faire un peu de structure du projet.

Mais avant de rentrer dans le vif du sujet, prenons un exemple simple, si dans notre projet, les développeurs appellent chacun leur classe du même nom, nous allons avoir un conflit ! PHP ne va pas savoir de quelle classe on lui parle.

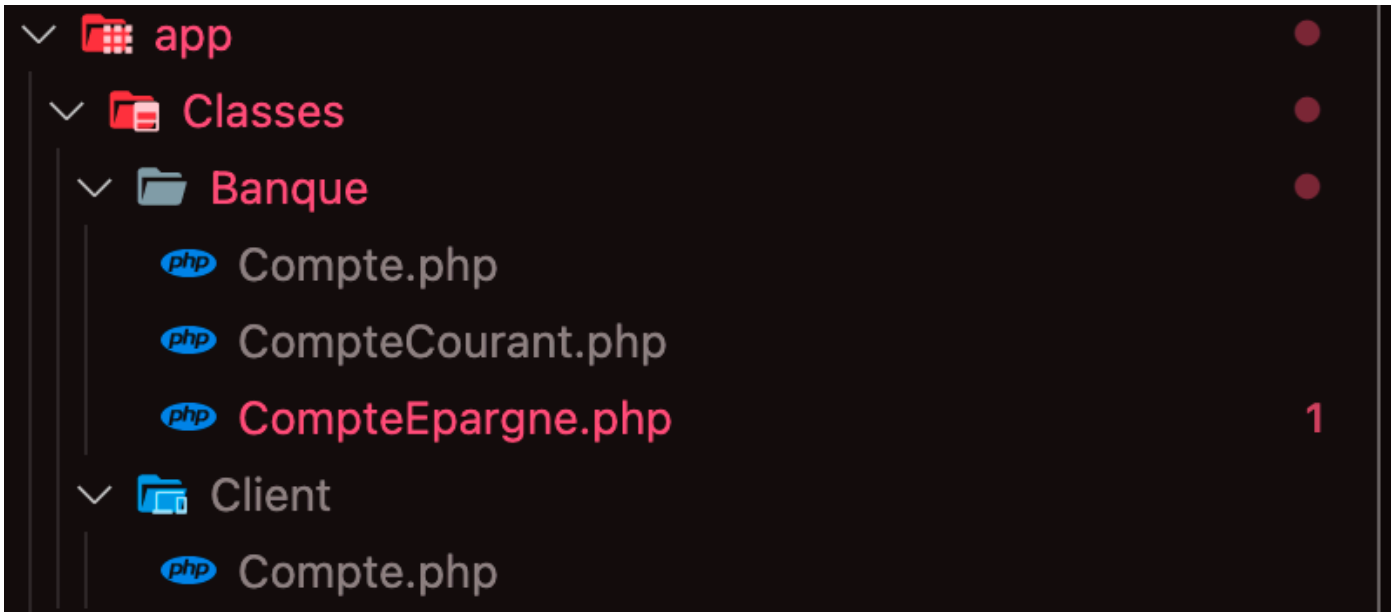
C'est là que les namespaces entre en jeu, les **namespaces** peuvent être considéré comme des dossiers virtuelle pour PHP, afin qu'il puisse être informé de notre organisation de classes, ainsi, quand nous appelleront telle ou telle classe, même si elle porte le même nom, PHP saura de laquelle on lui parle.

Pas encore claire ?

Faisons un exemple pour illustrer :

Nous allons créer une classe AUTRE classe Compte, mais celle-ci sera le compte client !

Avant tout, nous allons modifier la structure du projet en créant dans le fichier classe un dossier Banque et un dossier Client pour ranger nos classes :



Comme vous pouvez le voir, nous avons 2 fichiers qui se nomme Compte.

Maintenant, sur notre index.php, nous allons importer toutes les classes (même la dernière) avec un require_once.

```php
require_once 'classes/Banque/Compte.php';
require_once 'classes/Banque/CompteCourant.php';
require_once 'classes/Banque/CompteEpargne.php';
require_once 'classes/Client/Compte.php';
```

Et là, vous avez cette belle erreur :

Fatal error: Cannot declare class Compte, because the name is already in use in /app/classes/Client/Compte.php on line 3

Call Stack

#	Time	Memory	Function	Location
1	0.0130	358720	{main}( )	.../index.php:0
2	0.0263	362744	require_once('/app/classes/Client/Compte.php)	.../index.php:6

Php ne peut pas inclure la deuxième classe Compte !

C'est là que les namespaces entre en jeu, nous allons déclarer un namespace pour chacune de nos classes, ce namespace ressemblera à un chemin vers un fichier et reprendra la structure de nos dossiers en partant du dossier classes.

Par convention, le dossier "racine" du namespace est App, donc pour définir le namespace de notre fichier Compte.php se trouvant dans le dossier client, nous avons le chemin : classes/Client/Compte.php

Donc pour définir le namespace, nous allons simplement prendre les dossiers (pas le nom du fichier), attention, nous allons utiliser des `\` entre chaque dossier dans le chemin du namespace.

```php
// Client/Compte.php
<?php

  namespace App\Client;
```

Le namespace est la première chose qu'on définit dans le fichier (sous la balise d'ouverture php).

Maintenant, pour toutes les classes dans le dossier Banque nous allons rajouter :

```php
<?php
  namespace App\Banque;
```

Maintenant, si vous rechargez votre navigateur, vous allez trouver une autre erreur :

> **Fatal error: Uncaught Error: Class "CompteCourant" not found in /app/index.php on line *9***
>
> **Error: Class "CompteCourant" not found in /app/index.php on line *9***
>
> **Call Stack**

#	Time	Memory	Function	Location
1	0.0180	358736	{main}( )	.../index.php:0

Php ne trouve plus la classe CompteCourant, c'est normal, nous n'avons pas définit son namespace dans le fichier index.php.

Vous avez donc 2 solutions :

La moins propre est qu'au moment de l'instanciation, vous indiquiez le namespace complet avec la classe

```php
$compte1 = new App\Banque\CompteCourant("Pierre", 1000, 500);
```

La plus utilisée est de rajouter un **use** en haut du fichier (en dessous du namespace) pour dire à PHP où il doit chercher les classes

```php
use App\Banque\CompteCourant
```

Pour chaque classe que vous allez utiliser dans le fichier, vous ajouterez un use

```
use App\Banque\Compte;
use App\Banque\CompteCourant;
use App\Banque\CompteEpargne;
```

Autre chose à savoir, si vous avez des classes qui sont dans le même namespace (comme les classe dans le dossier Banque par exemple), vous pouvez les importer sur 1 seule ligne :

```
use App\Banque\{Compte, CompteCourant, CompteEpargne};
```

Ça vous fera économiser quelques ligne de code.

Dernière chose à savoir sur les uses, si vous avez 2 classes qui porte le même nom, vous pouvez créer un alias sur une des 2 avec le mot clé as :

```
use App\Client\Compte as CompteClient
```

Ce qui veut dire que votre classe Client, qui est dans le namespace App\Client, pourra être appelé dans le fichier sous l'alias CompteClient

Maintenant, si vous avez bien écrit vos namespaces et que vous les avez bien appelés dans votre fichier index.php, tout devrait marché comme il faut.

L'autoload

Maintenant si on regarde le haut de notre fichier index.php, il y a beaucoup de ligne seulement pour importer des fichiers et utiliser nos classes, dans des projets plus conséquents, la liste des fichiers à appeler va se rallonger, et nous allons vite nous retrouver avec énormément de require à faire PLUS les require.

Cette méthode n'est pas viable, donc l'autoload fait son entré en scène !

L'autoload, c'est simplement un **système de chargement de fichier dynamique**.

En résumé, si le serveur PHP trouve une classe qu'il ne connaît pas, il va chercher le fichier correspondant et le charger pour nous.

Donc, nous n'auront plus besoin de faire les require_once en début de fichier, seulement les uses.

Ça va nous faire gagner de l'espace et un peu de temps également.

Mais avant que tout marche de manière dynamique et automatiser, nous allons devoir créer notre autoloader.

Nous allons donc créer une nouvelle classe Autoloader dans laquelle nous allons définir les règles de chargement dynamique.

```php
<?php
namespace App;

class Autoloader
{
}
```

La fonction spl_autoload_register

Maintenant, tout repose sur la fonction PHP appelée **spl_autoload_register**. Cette fonction a pour rôle d'enregistrer une fonction qui sera appelée à chaque fois qu'une classe sera rencontrée dans un fichier.

Ainsi, à chaque fois qu'on va vouloir instancier une classe, nous allons pouvoir mettre en oeuvre un système qui va permettre de trouver le fichier PHP correspondant à la classe et de le charger s'il existe.

Maintenant, dans notre classe Autoloader, nous allons devoir créer une fonction statique "register" qui nous permettra de l'activer sans besoin d'instancier la classe Autoload.

Dans cette méthode, nous allons **récupérer la classe par la constante magique "CLASS"** et déclencher la méthode "**autoload**". Cette constante magique `__CLASS__` contient tout le chemin de la classe incluant le Namespace.

```php
<?php

namespace App;

class Autoloader
{
    static function register()
    {
        spl_autoload_register([
            __CLASS__,
            'autoload'
        ]);
    }
}
```

Grâce à cette première fonction, à chaque fois qu'on instenciera une classe dans un fichier, la classe sera envoyé à la fonction statique que nous allons créer autoload(), cette fonction va permettre d'instancier automatiquement la class (que nous n'ayons plus besoin de faire les require_once à chaque classe).

La méthode "**autoload**" devra donc **retirer le début du namespace "App"** puis **remplacer les "\" par des "/"** pour créer le chemin du fichier.

Ainsi, si nous cherchons à charger la classe "**CompteCourant**", son chemin complet incluant le namespace est "**App\Banque\CompteCourant**", nous retirons "**App**", nous obtenons "**Compte\CompteCourant**", puis nous remplaçons les "****" par des "**/**" pour obtenir "**Banque/CompteCourant**". Il nous reste à inclure le dossier "**classes**" au début et le "**.php**" à la fin.

```php
static function autoload($class)
{
  // On retire App\
  $class = str_replace(__NAMESPACE__ . '\\', '', $class);

  // On remplace les \ par les /
  $class = str_replace('\\', '/', $class);

  $fichier = __DIR__ . '/' . $class . '.php';

  // On vérifie si le fichier existe
  if (file_exists($fichier)) {
    // On include le fichier
    include_once $fichier;
  }
}
```

Maintenant, dans votre fichier Index.php, vous pouvez enlever tous les require_once que nous avions mis et remplacer par :

```php
require_once 'classes/Autoloader.php';

Autoloader::register();
```

Notez qu'il faut quand même faire 1 require_once pour aller chercher le fichier autoloader qui va permettre le chargement dynamique de toutes nos classes.

Rechargez votre page, vous ne devriez pas avoir d'erreur, si c'est le cas, votre autoload est bon, vous n'aurez plus besoin de faire un require_once de vos fichiers quand vous allez utiliser des classes.

Maintenant que nous avons gérer l'autoload ainsi que les namespaces, ajoutez dans votre classe Compte (CLIENT) les propriétés nom, prénom et ville, veillez à créer un constructeur ainsi que les accesseurs.

L'injection de dépendance

Nous commençons à avoir une architecture orienté objet plutôt intéressante.

Mais maintenant, nous avons nouvelle classe Compte (client), mais le problème c'est que dans nos classe Compte, nous définissons déjà le nom du titulaire, pourquoi ne pas directement relié un Compte client à un compte bancaire ?

Et bien c'est ce que nous allons faire grâce à l'injection de dépendance.

Pour ça nous allons modifier notre code pour que les classes CompteCourant et CompteBancaire pour que dans le constructeur de ces classes, on ne demande plus une chaine de caractère pour définir le titulaire, mais un objet de type Compte Client.

On appelle ça une dépendance car la classe Compte (Bancaire) sera dépendante de la classe compte Client !

Pour mettre ça en oeuvre, nous allons devoir modifier le fichier Compte.php pour que dans le constructeur de la classe, on appelle un objet compte (client) et nous allons devoir également modifier les accesseurs :

```php
namespace App\Banque;

use App\Client\Compte as CompteClient;
//......
/**
* Titulaire du compte
* @var CompteClient
*/
protected $titulaire;

//.........

public function __construct(CompteClient $titulaire, float $solde)
{
  // On affecte le titulaire à la propriété titulaire
  $this->titulaire = $titulaire;

  // On affecte le montant à la propriété solde
  $this->solde = $solde;
}

//..........

/**
* Récupère le nom du titulaire
*
* @return CompteClient
*/
public function getTitulaire(): CompteClient
{
  return $this->titulaire;
}

/**
* Définit le nom du titulaire
*
* @param CompteClient $titulaire
* @return Compte
*/
public function setTitulaire(CompteClient $titulaire): self
{
  // On peut vérifier qu'il y a bien au moins 1 caractère envoyé
  if ($titulaire != "") {
    $this->titulaire = $titulaire;
  }
```

```
    return $this;
}
```

Il faudra faire la même modification sur les classes CompteCourant et CompteEpargne.

Une fois que nous avons fais ça, nous n'avons plus qu'à instancier un compte d'épargne en passant en paramètre un compte client :

```
//index.php
$compteClient = new CompteClient("Bertrand", "Pierre", "Chambéry");

$compteCourant = new CompteCourant($compteClient, 1000, 500);

var_dump($compteCourant);
```

Avec nos modifications vous devriez avoir ce résultat sur le navigateur :

```
/app/index.php:15:
object(App\Banque\CompteCourant)[2]
  private 'decouvert' => int 500
  protected 'titulaire' =>
    object(App\Client\Compte)[1]
      private 'nom' => string 'Bertrand' (length=8)
      private 'prenom' => string 'Pierre' (length=6)
      private 'ville' => string 'Chambéry' (length=9)
  protected 'solde' => float 1000
```

Comme vous le voyez, le titulaire du compte est un objet de type Compte Client, nous avons bien réussi notre injection de dépendance.

Félicitation, vous venez de terminer l'initiation à la programmation orienté objet en PHP !

Nous allons maintenant aller plus loins en abordant un nouveau projet avec une connexion en base de donnée SQL avec MySQL ainsi que le développement avec le design pattern MVC.

Rendez-vous sur le prochain PDF !