



POO - PHP - Partie 3 : Composition, traits et namespace

▼ Cours	PHP
▼ Type	Guide
🔗 Supports	https://openclassrooms.com/fr/courses/1665806-programmez-en-orientee-objet-en-php
▼ Difficulté	Moyen

Lorsque le projet devient de plus en plus conséquent, il devient important de respecter plusieurs principes pour avoir un code maintenable dans le temps et plus facile à comprendre. C'est à ce moment précis qu'on peut penser à d'autres architectures comme la composition.

i Le principe des classes est qu'une classe ne devrait posséder qu'une seule responsabilité. (Single Responsibility : Le S de S.O.L.I.D).

Espaces de noms

Imaginons un système de commentaires dans une page blog et un système de commentaires sur le même site web mais dans une autre page quelconque (la boutique par exemple).

On aura besoin d'écrire deux fois la classe "Commentaire" pour différencier les deux systèmes. Cependant, une classe ne peut pas être déclarée deux fois.

Un moyen de contourner cette interdiction est d'utiliser les `namespaces` !

```
<?php

declare(strict_types=1);

namespace Blog;
class Commentaire
{}

namespace Shop;
class Commentaire
{}

```

Pour instancier un objet dans un espace de noms, on utilisera une nouvelle syntaxe :

```
$objet = new \NomNameSpace\MaClasse;
```

le “\” correspond au chemin du `namespace`. De manière globale, le code entier est un `namespace` à lui tout seul.

⚠ Quand on utilise des Namespaces, **le script pensera alors que tout ce qui suit est dans le même espace**. Si l'on souhaite instancier un objet d'une classe plus générale, il faudra donc lui préciser `\` devant.

ℹ Il est possible d'utiliser les namespaces de manière plus précise en entourant tout le script entre `{}` dans le namespace. Cela ajoute la contrainte que tout ce qui est dans ce script, doit être dans ce namespace.

Les namespaces sont beaucoup utilisés dans de gros projets et notamment dans des Framework comme Symfony. Il n'est pas rare de voir des namespaces à plusieurs niveaux du genre : `App\Domain\Messenger`.

Au bout d'un moment, il devient utile de récupérer des namespaces issus d'autres fichiers. On emploiera alors le mot clé `use`.

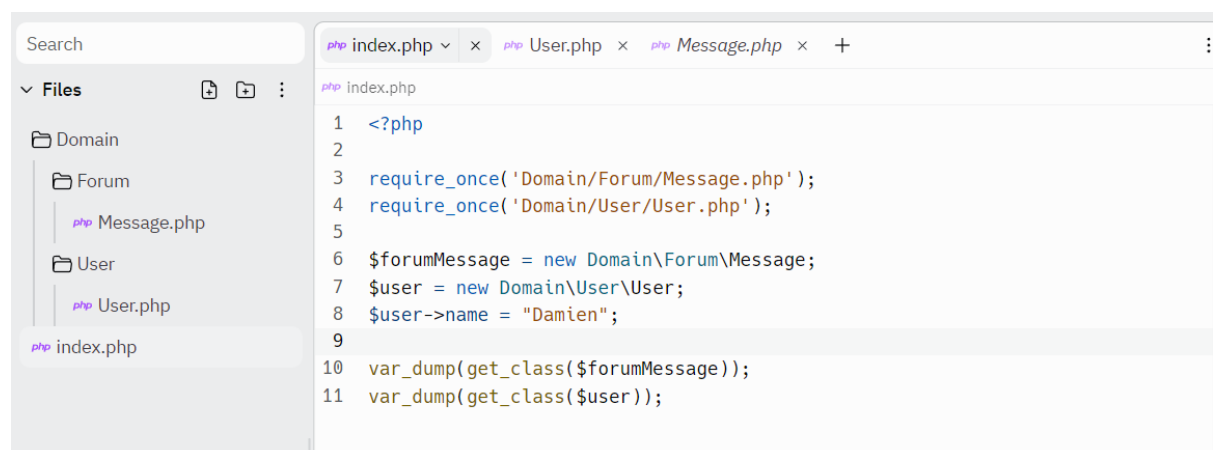
```
namespace {  
    use App\Domain\Messenger\Message;  
  
    $messengerMessage = new Message;  
    var_dump($messengerMessage::class);  
}
```

Structure et organisation

Une règle de programmation consiste à structurer un programme de grande envergure en utilisant **un fichier par classe**.

Cette méthode permet de réduire la taille des fichiers et de n'appeler que ce qui est nécessaire au moment opportun par PHP (optimisation).

Le code ressemble donc à ceci pour une classe `User` et une classe `Message` :



```
1 <?php  
2  
3 require_once('Domain/Forum/Message.php');  
4 require_once('Domain/User/User.php');  
5  
6 $forumMessage = new Domain\Forum\Message;  
7 $user = new Domain\User\User;  
8 $user->name = "Damien";  
9  
10 var_dump(get_class($forumMessage));  
11 var_dump(get_class($user));
```

Ici on voit que l'on a bien séparé `User.php` et `Message.php` qui sont des fichiers de classes, dans deux namespaces différents.

Cependant, avec plusieurs classes et un projet en croissance, on peut se retrouver avec une multitude de `require` ou de `use`.

Pour éviter cela, on peut utiliser la bibliothèque PHP `SPL` pour le chargement automatique des classes.

Chargement automatisé

Avec le SPL, on peut appeler une fonction nommée `spl_autoload_register()`. Cette fonction permet de charger des namespaces qui sont contenus dans des chemins différents. Voici un exemple de code basé sur le précédent :

```
<?php

spl_autoload_register(static function ($fqcn){
    $path = str_replace(['App', '\\'], ['src', '/'], $fqcn . '.php');
    require_once $path;
});

use App\Domain\Forum\Message;
use App\Domain\User\User;

$forumMessage = new App\Domain\Forum\Message;
$user = new App\Domain\User\User;
$user->name = "Damien";

var_dump(get_class($forumMessage));
var_dump(get_class($user));
```

La fonction `spl_autoload_register()` permet de remplacer chaque `use` par un `require`. On prends le chemin et on lui change l'attribut src (nom du dossier original) par App (convention utilisée par Symfony pour stocker les fichiers de classe) et on échange les `\` par `/`. Enfin, on ajoute l'extension `.php` au chemin. cela évite d'avoir à utiliser des `use` et `require` en même temps.

Cette fonction possède une deuxième particularité qui la rend indispensable : Elle stocke les chemins d'accès. Ce qui fait que lorsque PHP ne trouve pas quelque chose, il appellera la fonction `spl_autoload_register()` pour récupérer ce qu'il cherche.

Il est important de savoir que **les arguments ne sont pas obligatoire** dans cette fonction, en réalité elle contient par défaut un ensemble de paramètres :

```
spl_autoload_register(static function(string $fqcn) {
    // $fqcn contient le chemin\chemin\fichier
    // remplaçons les \ par des / et ajoutons .php à la fin.
    $path = str_replace('\\', '/', $fqcn).'.php';

    // puis chargeons le fichier converti en format dossier/dossier/fichier.php
    require_once($path);
});
```

La convention de nommage des namespace se nomme : PSR-4

Les Traits

Un trait fonctionne un peu comme une classe, il permet de regrouper un ensemble de propriétés et de méthodes pour être réutilisé dans une classe indépendante.

Il peut se présenter selon la forme suivante :

```

trait flyCapacity {
    function fly(){}
    $altitude float;
}

class Oiseau{
    use flyCapacity;
}

```

Ici on a défini un trait, la capacité de voler, et nous l'avons intégré dans la classe Oiseau. Dans un programme plus complexe, nous pourrions avoir plusieurs traits tels que voler, nager, manger, marcher, boire etc. et différents oiseaux qui ont ou n'ont pas ces capacités.

Cette façon de faire, sélectionner ce dont la classe hérite, se nomme de l'héritage horizontal. (c'est un premier pas vers la composition)

i Un trait ne peut pas être instancié ! C'est un élément voué à être appelé dans une classe, c'est à dire un composant.

Les Interfaces

Une interface est un modèle de création de classe (ou de trait). C'est à dire qu'elle permet de définir les règles qui régissent la création de classe, ce qui est utile dans un gros projet lorsque l'on veut s'assurer que l'on a toujours respecté le même schéma, ou lorsque l'on ne connaît pas encore les classes qui en découlent, ou encore lorsque l'on travaille à plusieurs et que l'on souhaite éviter que quelqu'un fasse du code qui ne respecte pas ces normes.

En reprenant l'exemple des oiseaux, on peut définir l'interface comme ceci :

```

<?php

namespace App/Birds/Bird;

interface BirdsInterface {
    public function defineComponents(): string;
    public function getComponents(): string;
}

```

Toute classe utilisant cette interface aura l'obligation de définir des composants (fly, run etc.) et d'avoir une fonction permettant d'obtenir une liste de ces composants.

On peut ensuite créer une nouvelle classe d'oiseau comme ceci :

```

<?php

class FlyBird {
    public static printComponents(BirdsInterface $parameters) //Ici on intègre l'interface en tant qu'argument
    {
        echo $parameters;
    }
}

```

Ou comme ceci :

```

<?php

namespace Bird {
    use App/Birds/Bird;
}

```

```
class FlyBird implements BirdsInterface { // Ici il est possible d'implémenter plusieurs interfaces à la suite avec ,
    //ici méthodes de l'interface comme defineComponents() et getComponents()
}
```

Il est également possible qu'une interface hérite d'une autre interface avec le mot clé `extends`.

⚠ Il est important de connaître la **ségrégation des interfaces** pour éviter d'avoir trop de méthodes non utilisées. Si certaines méthodes ne sont pas nécessaires, il vaut mieux diviser les interfaces en plus petites interfaces.

La composition

La composition est un principe qui consiste à créer des objets complexes en les assemblant à partir d'objets plus simples. Au lieu de recourir à l'héritage pour étendre les fonctionnalités d'une classe, la composition favorise la création de relations entre différentes classes, permettant ainsi de construire des structures plus flexibles et modulaires.

Vous l'avez vu précédemment, pour faire de la composition, il suffit d'instaurer des modèles de classes et de leurs ajouter des comportements tout en évitant d'avoir trop de dépendances. Cela rend le code réutilisable, plus cohérent et flexible. Cependant, il est important de noter que le code peut se complexifier s'il y a beaucoup de relations entre plusieurs classes et que la mise en place est plus longue sur les gros projets.

i "Pas d'exemple ? 😞"

Non. Je vous garde ça pour plus tard 😊

La classe Exception

Pour envoyer une erreur, le code attends un `return`

On parle de jeter des exceptions. Pour indiquer des erreurs ou des problèmes dans le code, la classe `Exception` permet de gérer les exceptions retournées en utilisant le mot clé `throw`.

```
if (/* something good */ true) {
    // do something
} else {
    throw new Exception('Une erreur est survenue', 'code erreur au choix');
}
```

Ici on à instancié des objets issus de la classe `Exception`.

La classe `Exception` est riche et possède des sous-classes natives gérées par PHP permettant de spécifier le type d'erreur. Par exemple, si vous voulez indiquer à un développeur qu'il n'a pas correctement utilisé son code, on utilise la classe `LogicException`. Si vous voulez indiquer à un utilisateur qu'il a mal saisi quelque chose, utilisez `RuntimeException`.

Il est également possible de déclarer vos propres exceptions personnalisées.

```
class ExceptionMessage extends RuntimeException {
    public $message = 'Impossible d\'envoyer le message';
}

function sendMessage(string $text):bool {
    if (/* fail */) {
        throw new ExceptionMessage();
    }
}
```

Try ... Catch

La structure `try ... catch` permet de gérer les exceptions de manière plus simple. Elle permet d'exécuter du code sans nécessairement arrêter le script en cas d'erreur. On peut également envoyer des notifications ou des mails en cas d'erreur non critique.

```
try {
    // Something
} catch (Exception $e) {
    echo 'Exception reçue : ', $e->getMessage(), "\n";
}
```

Cette structure réussit plus facilement à attraper des exceptions et les renvoie directement, une fois de plus grâce à la classe `Exception`

On peut aussi utiliser les exceptions personnalisées dans le cadre du `try...catch`

```
try {
    //something
} catch (ExceptionMessage $e){
    echo $e->message
} catch (Exception $e){
    echo 'une erreur inattendue est survenue $e->getMessage(), \n';
}
```

Grâce à ce cours, vous devriez être capables de maîtriser la POO, l'héritage et la composition.

Exercez-vous :


Vous avez été embauché(e) par un parc zoologique pour concevoir un système de gestion des oiseaux. Les oiseaux présentent une grande diversité de capacités, certains peuvent voler, d'autres peuvent nager, et certains peuvent même faire les deux. Vous devez créer un système basé sur le principe de la composition pour modéliser les différentes catégories d'oiseaux et leurs capacités.

Instructions :

1. Créez une classe de base appelée "Oiseau" qui aura les propriétés communes à tous les oiseaux, telles que le nom, l'âge, et le type (carnivore, herbivore, omnivore, etc.).
2. Ensuite, créez des traits pour différentes catégories d'oiseaux. Par exemple, "Fly" pour les oiseaux capables de voler, "Swim" pour ceux qui peuvent nager, "Walk" pour ceux qui peuvent marcher (et pas sautiller) !
3. Chaque trait aura des propriétés. Par exemple, le trait "Fly" devrait avoir une propriété pour la vitesse de vol, et le trait "Swim" devrait avoir une propriété pour la profondeur de plongée.
4. Chaque trait devrait également avoir une méthode spécifique pour décrire ses capacités. Par exemple, le trait "Fly" pourrait avoir une méthode "flying()" qui imprime un message indiquant que l'oiseau vole à une certaine vitesse.
5. Enfin, créez quelques instances d'oiseaux en utilisant les différentes classes dérivées et testez leurs capacités en appelant les méthodes appropriées. (Le manchot, le canard, le perroquet, l'autruche).

Note : Assurez-vous d'utiliser le principe de la composition pour créer des classes modulaires et réutilisables.

Bonne chance ! N'hésitez pas à m'envoyer votre travail pour une correction 😊

 Pour aller plus loin, voici d'autres design patterns à découvrir :

onion, clean, mvc, adr, DDD, hexagonal, CQRS, SOLID, DRY, KISS, modèles anémique, modèles riches.

Vous pouvez aussi vous lancer sur un Framework comme Symfony ou Laravel à partir de maintenant 😊