PHP - 13 : Programmation orientée objet avancée

Le but de cette séquence est de porter à votre connaissance l'existence des concepts avancés de la programmation orientée objet et de comprendre leur utilité, pas de les manipuler.

> Retenez que la plupart de ces concepts ne sont pas spécifiques au PHP, on les rencontre dans d'autres langages orientés objets.

Classes abstraites

Une classe abstraite est un début d'implémentation d'une classe. On définit certaines méthodes et attributs en obligeant les classes dérivées à les implémenter. On peut ainsi présenter un début d'implémentation et forcer les gens qui la réutilisent à l'étendre en la complétant.

Classe abstraites en PHP

Exemple de déclaration d'une classe abstraite :

```php
<?php
abstract class vehicule
{
    abstract public function avancer();
    abstract protected function freiner();
}
```

Remarques :

- Le mot-clé `abstract` précède `class` et les méthodes abstraites
- Aucun code dans les méthodes; seul leur nom est indiqué; le code sera défini dans les classes utilisant la classe abstraite
- La visibilité des méthodes doit être au moins au même niveau (ou inférieur) dans la classe abstraite et les classes filles; elle ne peut avoir la visibilité privée
- Dans une classe abstraite, toutes les méthodes ne sont pas forcèment abstraites; par contre si au moins l'une l'est, il faut déclarer la classe comme étant abstraite (c'est-à-dire ajout du mot-clé `abstract` devant `class`).
- Une classe abstraite n'est pas instanciable.

Utilisation de la classe abstraite par une classe :

```php
1  <?php
2  class voiture extends vehicule
3  {
4      function avancer()
5      {
6          echo 'on avance' ;
7      }
8  }
```

Remarques :

- Ligne 1 : la classe fille hérite de la classe abstraite via le mot-clé `extends` (pareil que pour un héritage normal)
- Ligne 4 : on définit la fonction `avancer()`

Interfaces

La notion d'interface est proche de celle de classe abstraite, mais un peu plus générique.

Les interfaces sont utiles pour forcer la présence de méthodes et de fonctionnalités appelables par l'utilisateur. Seules des méthodes publiques peuvent donc être déclarées dans une interface.

On déclare une interface de manière similaire à une classe abstraite mais avec le mot-clé `interface` . Les méthodes sont forcément publiques.

Interfaces en PHP

Exemple

- Déclaration d'une interface (nommée *crud*) :

  ```php
  interface crud
  {
      public function liste($datas);
      public function afficher($id);
      public function ajouter($datas);
      public function modifier($datas);
      public function supprimer($id);
  }
  ```

- Utilisation d'une interface : on crée une nouvelle classe (ici *Voiture*) qui "implémente" l'interface. On utilise le mot-clé `implements` :

  class voiture implements crud { public function liste($datas) { [CODE] }

  ```php
  public function afficher($id)
  {
     [ CODE ]
  }

  // etc...
  ```

  }

Documentation

Différence classe abstraite et interface

Lire la section *Interface ou classe abstraite : comment choisir ?* de cette page.

Traits

En PHP, une classe ne peut hériter que d'une seule classe mère à la fois : c'est *l'héritage simple*.

Le contraire est *l'héritage multiple*, qui existe dans certains langage (C++ par exemple).

Le problème : si une classe B n'hérite pas de la classe A, il n'est pas possible de réutiliser dans B une méthode présente dans A. Il faudrait dupliquer dans B le code de la méthode de A, or ce code est strictement identique car il fait la même chose; cela fait donc doublon.

Un trait résoud ce problème en rendant possible l'utilisation d'une méthode de la classe A dans la classe B sans que B ne soit déclarée comme fille de A. Un trait "zappe" donc la notion d'héritage.

En pratique, on va mettre dans le trait (une sorte de classe) la méthode à utiliser dans les 2 classes A et B :

Le trait :

```php
<?php
trait crud
{
   public function convertir($valeur)
   {
      // [ CODE ]
   }
}
```

Utilisation dans la classe A :

```php
<?php
class A
{
    // Appel du trait
    use crud;

     public function freiner($nbkm) {
          $nbkm = $this->convertir($nbkm);
    }
}
```

Utilisation dans la classe B :

```php
<?php
class B
{
    // Appel du trait
    use crud;

    public function avancer ($nbkm)
    {
          $nbkm = $this->convertir($nbkm);
    }
}
```

> Retenez qu'il est possible d'utiliser plusieurs traits au sein d'une même classe; dans ce cas on peut écrirer `use nom_trait_1, nom_trait_2;`

Documentation

Classes et méthodes finales

Une classe ou une méthode est dite finale lorsqu'on souhaite qu'elle ne soit plus redéfinie dans une classe fille (on veut stopper la surcharge/le polymorphisme).

- Si une classe est finale, aucune de ses méthodes ne peut être redéfinie
- Si une méthode est finale, seule cette méthode ne peut être redéfinie (les autres méthodes de la classe peuvent encore l'être, du moins si celles qui ne sont pas déclarées finales)

On indique cet état en ajoutant le mot-clé `final` .

Exemple pour une classe :

```php
final class Velo extends DeuxRoues
{
     public function avancer() {
        // [CODE]
     }

      public function freiner() {
        // [CODE]
     }
}
```

Exemple pour une méthode

```php
class Velo extends DeuxRoues
{
     public final function avancer() {
        // [CODE]
     }

     public function freiner() {
        // [CODE]
     }
}
```

Design patterns

Un *design pattern* (motif de conception, patron de conception) est un modèle de classe qui permet de résoudre un problème d'algorithmique spécifique, c'est-à-dire qui propose une solution type reconnue par l'ensemble de la profession.

Il existe de nombreux design patterns : les plus connus sont *factory*, *singleton*, *strategy*, *observer*...

design patterns PHP

Autres points

Chargement automatique de classes

Afin d'éviter de charger les classes une par une, il existe une astuce. Le préalable est d'avoir préfixé l'extension de fichier php par `class` : vos fichiers de classes doivent donc avoir le format `nomdelaclasse.class.php`

+++ TODO : exemple avec fonction +++

```php
spl_autoload_register(function($class)
{
      include "classes/".$class.".class.php";
});
```

Typage

Le typage explicite (*type hinting* en anglais) permet de s'assurer du type d'une variable en le précisant devant cette variable lorsqu'elle est passée en argument à une méthode.

Exemple

Dans la méthode `ajouter()` , on spécifie que la variable `$datas` doit être un tableau (`array`) :

```php
class Vehicule
{
     public function ajouter(array $datas) {
        // [CODE]
     }
```

Si la variable n'est pas du type attendu, une erreur fatale est levée.

Les types peuvent être un tableau, le nom d'une classe ou d'une interface, mais pas un entier ni une chaîne simple ni un trait.

Documentation

Chaînage de méthodes

Le chaînage de méthode permet d'exécuter plusieurs méthodes d'une classe à la fois.

Exemple

Au lieu d'écrire :

```php
$o = new Vehicule();
$o->avancer();
$o->freiner();
$o->arreter();
```

on peut écrit ceci :

```php
$o->avancer()->freiner()->arreter();
```

Espaces de noms

Les espaces de noms - *namespace* en anglais - permettent de regrouper plusieurs classes sous une même entité portant un nom; on obtient ainsi des composants (ou encore modules ou packages).

Pour créer un espace de nom, on ajoute le mot-clé `namespace` suivi du nom qu'on veut lui donner :

```php
   namespace ventes;
```

On place cette ligne dans un fichier ou une classe, mais au tout début : **aucune ligne de code ne doit précéder cette déclaration**.

Exemple

```php
namespace Ventes

class Vehicule
{
     // [CODE]
}
```

Pour utiliser l'espace de nom, on utilise, le mot-clé `use` suivi du nom :

```php
 use Ventes;
```

On place cette ligne là où on souhaite utiliser l'un des éléments (classes...) appartenant à l'espace de nom.

> On peut déclarer des espaces de noms sur plusieurs niveaux, alors séparés par un antislash. Exemples : `namespace Crm\Ventes` ou encore `Crm\Ventes\Vehicules` , `Crm\Ventes\Services` .

Les espaces de noms ont un autre avantage : on peut utiliser des noms de fichiers, de classes ou de variables identiques dans une application si ces derniers appartiennent à des espaces de noms distincts :

- classe A et classe B dans l'espace de nom C.
- une autre classe A et une autre classe B dans un espace de nom D.

Appel de méthodes parentes

Dans une classe fille, on fait appel à une méthode de la classe parente via le mot-clé `parent` , suivi de 2 points puis du nom de la méthode souhaitée : `parent::method()` .

> Le signe `::` est appelé opérateur de résolution de portée.

Méthode statique

Les propriétés et méthodes statiques sont des propriétés et méthodes qui appartiennent exclusivement à une classe et non pas à un objet et qui vont donc être les mêmes pour toutes les instances d'une classe. L'appel à une méthode statique se fait directement, c'est-à-dire sans instanciation (pas de `new maclasse()` . Il n'y a pas d'état/contexte et on ne peut donc pas utiliser le mot-clé `$this` .

Une méthode est déclarée statique avec le mot-clé `static` :

Pour invoquer une méthode statique dans une autre méthode de la même classe, on utilise l'opérateur `self` suivi de 2 points puis du nom de la méthode statique : `self::method()` .

Tutoriel

Classes anonymes PHP 7

Le PHP 7 introduit un nouveau concept, celui de classe anonyme.

Explications détaillées.

Introspection et débogage

Métaconstantes

Pour déboguer vos classes, il existe en PHP un type de variables appelés métaconstantes :

- `__CLASS__` : retourne le nom de la classe utilisée
- `__METHOD__` : retourne la méthode de classe
- `__NAMESPACE__` : retourne l'espace de nom utilisé (s'il y en a un de déclaré)
- `__FILE__` : indique dans quel fichier on se trouve (peut être utilisé hors des classes, dans n'importe quel fichier PHP). Dans la même veine, `__LINE__` donne le numéro de ligne.

Exemple : `echo"Fichier : ".__FILE__", ligne : ".__LINE__;`

Clonage

Il est parfois utile de devoir copier (dupliquer) un objet dans son état actuel pour, par exemple, effectuer un test tout en conservant son état (valeurs initiales des attributs). Il existe pour cela un mécanisme appelé clonage.

- Tutoriel
- Documentation

Reflexivité

PHP propose un mécanisme donnant des informations sur une classe (y-a-t-il une classe parente ? Quels sont les attributs etc.). Ce mécanisme est appelé reflexivité (*Reflection* en anglais) et permet l'introspection des classes PHP au sein du code.

- Reflexivité
- Documentation

Itérateurs

PHP fournit plusieurs classes (interfaces) d'itérateurs pour parcourir un objet. Ces itérateurs permettent de parcourir les attributs d'une instance d'objet.

- Tutoriel
- Documentation