le cnam
Grand Est

Développement web (2)

Sites dynamiques
et développement côté serveur

NFA017 (4 ECTS)

Séance 04
Introduction à l'Orienté Objet avec PHP
2022

Le Cnam en Grand Est - Centre régional
4 avenue du Docteur Heydenreich CS 65228 F 54052 Nancy Cedex
tél +33 (0)3 83 85 49 00 www.cnam-grandest.fr

Le plus grand soin a été apporté à la réalisation de ce support pédagogique afin de vous fournir une information complète et fiable. Cependant, le Cnam Grand-Est n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Le Cnam ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou autres marques cités dans ce support sont des marques déposées par leurs propriétaires respectifs.

Ce support pédagogique a été rédigé par Simon MAHIEUX, enseignant au Cnam Grand-Est.

Table des matières

1. Introduction

1.1 Présentation

La programmation orientée objet (POO) est une méthode de programmation qui consiste à définir et assembler des briques logicielles qui interagissent les unes avec les autres.

Contrairement à la programmation procédurale, dont la méthodologie est centrée sur les traitements (utilisation de procédures : fonctions contenant une série de traitements à réaliser), la programmation orientée objet se concentre sur la manipulation d'ensembles de données (les « objets »).

Un objet représente une entité matérielle ou immatérielle (une personne, une voiture, un concept, une idée, etc…). Cette entité a sa propre structure (« variables ») et son propre comportement (« fonctions »).

La programmation orientée objet permet une meilleure compréhension du code, un code plus indépendant, plus évolutif et maintenable. Le code est modulaire, il devient possible de réutiliser le même objet dans différents projets.

En revanche, la programmation orientée objet nécessite un peu plus de ressources qu'un code procédural (le temps d'exécution peut également être légèrement impacté). De plus, si l'application a été mal conceptualisée, elle risque de devenir difficilement maintenable dans le futur.

1.2 La programmation orientée objet et PHP

La POO a été introduite dans la version 4 de PHP, mais cette première version n'était pas assez aboutie. C'est seulement à partir de la version 5 du langage que le modèle objet est devenu réellement complet.

Nous allons présenter dans cette séance les concepts de base de la programmation orientée objet en PHP.

2. Concepts de base

2.1 Les objets

Un objet est une représentation abstraite d'une entité réelle ou abstraite, ayant des limites claires et un sens précis dans le contexte étudié. Un objet est constitué d'une structure de données valuée (les attributs), qui définissent son état, et des traitements associés (les méthodes), qui définissent son comportement.

Chaque objet est unique (notion d'identité).

Prenons un exemple concret : nous souhaitons manipuler des voitures. Une voiture sera caractérisée par un nom, une marque, et des dimensions : hauteur, longueur et largeur.

Nous pouvons représenter nos données avec des objets « *Voiture* ». Par convention, les noms des objets sont au singulier, avec une majuscule au début et, bien sûr, sans accent.

2.2 Les classes

Un objet est une instance d'une classe. Une classe est donc une occurrence de type abstrait, qui décrit la structure interne des données et définit les méthodes qui s'appliqueront aux objets de la même famille.

Une classe peut donc être considérée comme une sorte de « moule » dont sortirait un objet.

La structure générale d'une classe est la suivante :

```
class NomDeLaClasse {
        /* Attributs */
        ...
        /* Constructeur */
        ...
        /* Getters/Setters */
        ...
        /* Autres méthodes */
        ...
}
```

Nous allons maintenant définir notre classe *Voiture* :

```
class Voiture {
        /* Attributs */
        // ...
        /* Constructeur */
        // ...
        /* Getters/Setters */
        // ...
        /* Autres méthodes */
        // ...
}}
```

Cette classe est enregistrée dans un fichier nommé **Voiture.php**

2.3 Les attributs

Les attributs sont les variables spécifiques liées à l'objet. Ils sont définis dans la classe et permettent de stocker les données (caractéristiques) d'un objet.

Ajoutons nos attributs à notre classe *Voiture* :

```
class Voiture {
        /* Attributs */
        private $nom;
        private $marque;
        private $largeur;
        private $longueur;
        private $hauteur;

        /* Constructeur */
        // ...
        /* Getters/Setters */
        // ...
        /* Autres méthodes */
        // ...
}}
```

2.4 La visibilité

La visibilité d'un attribut (ou propriété) ou d'une méthode peut être définie en préfixant sa déclaration par un mot-clé : *public*, *protected* ou *private*.

- Mot-clé *public* : Les éléments déclarés avec le mot-clé *public* sont accessibles partout.
- Mot-clé *protected* : Les éléments déclarés avec le mot-clé *protected* sont accessibles par la classe elle-même ainsi qu'aux classes qui en héritent et les classes parentes.
- Mot-clé *private* : Les éléments déclarés avec le mot-clé *private* ne sont accessibles que par la classe elle-même.

Dans notre classe *Voiture*, nous avons définis nos attributs en *private* : seule la classe peut lire et modifier ces attributs.

2.5 Les méthodes

Les méthodes sont les fonctions spécifiques liées à l'objet. Elles sont définies dans la classe et permettent d'effectuer les différents traitements de l'objet.

Nous allons ajouter une méthode *afficher()*, qui permet d'afficher le nom et la marque de la voiture.

```
public function afficher() : void {
        echo $this->nom." (".$this->marque.")";
}
```

Nous utilisons la pseudo-variable *$this* afin d'accéder aux attributs de l'objet en cours, cela correspond à une référence de l'objet.

2.6 Les constructeurs

Les constructeurs sont des méthodes particulières très importantes : elles sont appelées automatiquement lorsqu'un objet est créé. Elles permettent généralement d'initialiser les données de l'objet.

Nous allons écrire le constructeur de notre classe. Nous faisons le choix d'initialiser les dimensions à 0 par défaut. Le développeur utilisant notre classe devra affecter les dimensions après création de l'objet.

```
public function __construct(string $nom, string $marque) {
        $this->nom = $nom;
        $this->marque = $marque;
        $this->longueur = 0;
        $this->largeur = 0;
        $this->hauteur = 0;
}
```

2.7 Les getters et setters

Les getters/setters (accesseur / mutateur) sont des méthodes particulières qui permettent d'accéder à un attribut privé.

Le getter (accesseur) permet de lire la valeur d'un attribut.
Le setter (mutateur) permet de modifier la valeur d'un attribut.

Il est également possible d'ajouter un comportement sur la modification, par exemple : si on souhaite stocker la valeur en majuscule, on peut ajouter la transformation du texte dans le setter de l'attribut en question.

Nous allons écrire maintenant les getters et setters pour chaque attribut de notre classe.

```
/* Getters */
public function getNom() : string {
        return $this->nom;
}

public function getMarque() : string {
        return $this->marque;
}

public function getLongueur() : float  {
        return $this->longueur;
}

public function getLargeur() : float  {
        return $this->largeur;
}

public function getHauteur() : float  {
        return $this->hauteur;
}

/* Getters */

public function setNom(string $nom) : void {
        $this->nom = $nom;
}

public function setMarque(string $marque) : void {
        $this->marque = $marque;
}

public function setLongueur(float $longueur) : void {
        $this->longueur = $longueur;
}

public function setLargeur(float $largeur) : void {
        $this->largeur = $largeur;
}

public function setHauteur(float $hauteur) : void {
        $this->hauteur = $hauteur;
}
```

2.8 Les méthodes magiques

Les méthodes magiques sont des méthodes spéciales qui écrase l'action par défaut de PHP quand certaines actions sont réalisées sur un objet. Toutes les méthodes commençant par « __ » sont réservées par PHP, il convient donc de ne pas préfixer ses propres méthodes par ces deux caractères.

Nous avons déjà croisé une méthode magique : __construct() qui permet de créer une nouvelle instance de l'objet.

La méthode *__toString()* est une autre méthode magique qui permet de gérer le comportement de l'objet lorsqu'il est traité comme une chaîne de caractère (avec la commande echo par exemple).

La documentation PHP présente l'ensemble des méthodes magique à l'adresse suivante : https://www.php.net/manual/fr/language.oop5.magic.php.

Nous allons définir la méthode *__toString()* et écrire son contenu, nous souhaitons afficher le nom, la marque ainsi que les dimensions de la voiture.

```
/* Méthodes magiques __toString */

public function __toString() : string {

        return $this->nom." (".$this->marque.")<br/>Dimensions : ".$this->longueur." x ".$this->largeur." x ".$this->hauteur." (L x l x h)";

}
```

2.9 Instanciation d'un objet

Pour utiliser un objet, il faut l'instancier. En PHP, il faut utiliser l'opérateur *new.*

Avec cet opérateur, PHP va appeler automatiquement le constructeur et retourner une référence sur l'objet.

Il est ensuite possible d'accéder aux attributs et aux méthodes publiques.

Nous allons maintenant utiliser notre classe Voiture en instanciant un objet.

```
// Instanciation d'un objet Voiture (appel automatique du constructeur)
$voiture = new Voiture("308", "Peugeot");


// Appel de la méthode afficher()
echo $voiture->afficher();


echo "<hr/>";


// Récupération du nom : utilisation d'un getter
echo $voiture->getNom()."<br/>";


echo "<hr/>";


// Ajout des dimensions : utilisations des setters
$voiture->setLongueur(300);
$voiture->setLargeur(200);
$voiture->setHauteur(200);


// Utilisation de l'objet comme une chaine de caractère (utilisation de echo directement sur
l'objet)
// -> appel automatique de la méthode __toString()
echo $voiture;
```

Téléchargez le fichier Voiture1.php disponible sur Moodle.

2.10 Organisation des fichiers

Il est préférable de créer un fichier pour chaque classe afin de pouvoir réutiliser la classe dans plusieurs parties de l'application.

Le nom du fichier doit respecter le format suivant : nom de la classe + extension .php

Pour inclure un script PHP dans un autre, il faut utiliser la fonction *include* (ou *require*). Toutes les fonctions et variables du script sont incluses.

Nous allons séparer le code dans le fichier *Voiture1.php* en deux fichiers :
- Voiture.php : qui va contenir seulement la classe Voiture
- manipulation.php : qui va inclure le fichier Voiture.php puis instancier des objets *Voiture*

Le fichier manipulation.php contient ainsi le code suivant :

```
<?php
	require('Voiture.php');


	// Instanciation d'un objet Voiture (appel automatique du constructeur)
	$voiture = new Voiture("308", "Peugeot");


	// Etc…
```

Téléchargez les fichiers Voiture.php et manipulation.php disponibles sur Moodle.

2.11 Chargement automatique

Il est possible d'indiquer à PHP de charger automatiquement la classe liée à l'objet que l'on souhaite instancier grâce au mécanisme de chargement automatique.

Pour l'utiliser, il faut définir une fonction de chargement, puis enregistrer cette fonction avec *spl_autoload_register.*

Nous allons définir une fonction nommée *charge()* afin d'inclure les fichiers en respectant le format : nom de la classe + extension .php, et situé dans le même répertoire que le script en cours.

```
<?php
	function charge($nomClasse) {
		include $nomClasse. '.php';
	}


	spl_autoload_register('charge');


	$v = new Voiture("308", "Peugeot");
	echo $v;
```

Téléchargez les fichiers Voiture.php et chargement_auto.php disponibles sur Moodle.

3. Principes clés de la programmation orientée objet

Cette approche de l'orienté objet introduit de nouveaux concepts et un nouveau vocabulaire.

Dans cette partie, nous allons présenter les principes clés de la POO. Nous verrons également lors de la webconférence des exemples concrets appliqués à PHP.

Nous retiendrons la définition suivante : « La conception orientée objet est une forme de construction des systèmes logiciels basée sur des collections structurées implémentant des types de données abstraites ».

3.1 La modélisation des données

La modélisation des données permet d'identifier les objets que le développeur souhaite manipuler, ainsi que leurs interactions.

L'objet ainsi modélisé sera conceptualisé dans une classe, qui aura des attributs et des méthodes.

3.2 L'encapsulation

L'encapsulation est un mécanisme consistant à masquer la structure interne d'un objet, son accès n'étant possible qu'au travers d'une interface externe.

Il s'agit donc d'occulter son implémentation réelle en empêchant d'accéder aux informations par un autre biais que les services proposés à cet effet.

Ce mécanisme est le garant de l'intégrité des données.

3.3 L'abstraction

L'abstraction est un mécanisme consistant à masquer/dissimuler le code inutile et ne proposer que les mécanismes internes pertinents pour l'utilisation de l'objet.

Ce mécanisme facilite la maintenabilité du code.

3.4 L'héritage

L'héritage est la capacité d'une sous-classe à hériter des attributs et méthodes d'une classe mère, et à les affiner : la sous-classe peut spécialiser ses propres méthodes et redéfinir celles héritées de son parent (notion de surcharge).

L'héritage peut être simple ou multiple (une ou plusieurs classes mères).

Les objectifs de l'héritage sont multiples :
- Partage du code
- Réutilisabilité
- Factorisation

Note : *En PHP, il n'y a pas d'héritage multiple.*

Pour spécifier un héritage, il faut utilise le mot-clé *extends*.

```
class ClasseFille extends ClasseMere {
...
}
```

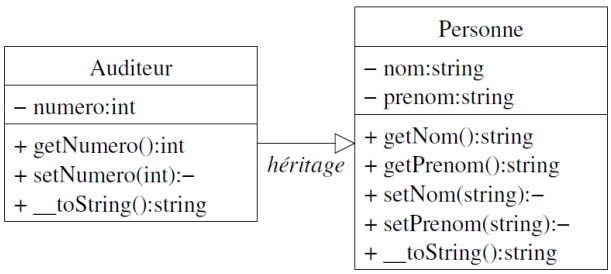
L'héritage permet de transmettre les membres de la classe mère à la classe fille, en fonction de la visibilité :
- public : accès total par la classe fille
- private : pas d'accès par la classe fille
- protected : accès avec la visibilité « public » pour la classe fille seulement

Le constructeur dans la classe fille peut appeler le constructeur de la classe mère en utilisant le mot-clé *parent ::*.

```
public function __construct(string $champ1, string $champ2, int $champ3) {
   parent::__construct($champ1, $champ2);
   $this->champ3= $champ3;
}
```

3.4.1 Exemple

Nous souhaitons écrire une classe *Personne*, contenant un nom et un prénom, ainsi qu'une classe *Auditeur*, qui hérite de *Personne* et qui contient un numéro.



Une personne a un nom et un prénom.
Un auditeur est une personne, qui possède en plus du nom et du prénom, un numéro d'auditeur.

```
class Auditeur extends Personne {
   private $numero;
   public function __construct(string $nom, string $prenom, int $numero) {
      parent::__construct($nom, $prenom);
      $this->numero = $numero;
   }
   public function getNumero() : int {
      return $this->numero;
   }
   public function setNumero(int $numero) : void {
      $this->numero = $numero;
   }
   public function __toString() : string {
      return parent::__toString()." ($this->numero)";
   }
}
```

Note : *la fonction __toString() est présente dans la classe Personne et dans la classe Auditeur, c'est donc une redéfinition de méthode. Seule la méthode de la classe Auditeur est accessible à l'extérieur de la classe.*

3.5 Le polymorphisme

Le polymorphisme est la capacité d'un objet à prendre plusieurs formes, c'est-à-dire que la même opération peut se comporter différemment sur différentes classes de la hiérarchie. Ce mécanisme permet d'augmenter la généricité du code.

Tandis que les mécanismes d'héritage favorisent la factorisation du code, le polymorphisme en accroît la généricité. La maître-mot de la Programmation Orientée Objet est donc la réutilisation, qui permet d'accroître la vitesse des développements mais facilite également la maintenance.

La programmation orientée objet comporte plusieurs avantages, notamment en termes de réutilisabilité, d'élasticité et d'efficacité. Ces principes peuvent aussi être appliqués lors de l'utilisation de microservices.

Lors de l'appel d'une méthode, la méthode appelée est celle définie dans la classe. Si elle n'existe pas, la méthode est recherchée dans la classe mère (puis fonctionnement répété pour remonter dans la hiérarchie). Si la méthode a été redéfinie, la méthode la plus spécifique à l'objet est appelée.

```
$p = new Auditeur("Bob", "Morane", 12345);
echo $p; // Affiche : Bob Morane (12345)

function compliment(Personne $p) : void {
echo "'".$p->getNom()."' est un joli nom<br/>";
}
$auditeur = new Auditeur("Bob ", "Morane", 123456);
compliment($auditeur); // Affichage : 'Morane' est un joli nom
```

Dans notre exemple, nous créons un premier objet Auditeur nommé $p. En utilisant la fonction *echo*, la méthode *__toString()* est appelée et affiche le nom, le prénom ainsi que le numéro.

Nous utilisons ensuite le *typage dynamique* en spécifiant un objet de la classe Personne comme type de paramètre pour notre fonction *compliment().*

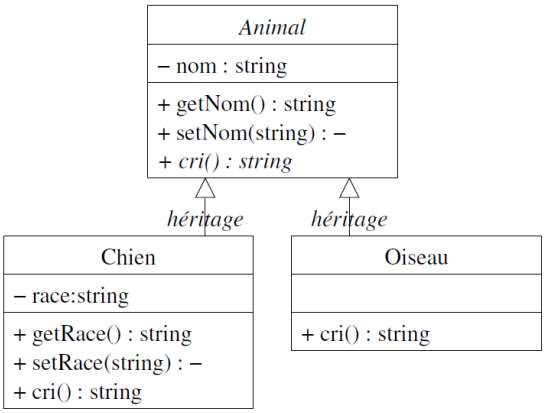
3.6 Les classes abstraites

Une classe abstraite est une classe dans laquelle les méthodes sont déclarées mais non définies. Nous utilisons dans ce cas le mot-clé *abstract*.

Une classe abstraite ne peut pas être instanciée directement. Cependant, une classe fille qui hérite d'une classe abstraite peut utiliser le constructeur de la classe mère et peut redéfinir les méthodes de la classe mère. Si elle ne redéfinies pas les méthodes, la classe fille devient également abstraite.

3.6.1 Exemple

Nous prenons un exemple assez connu pour les classes abstraites :



Une classe *Animal*, abstraite, spécifie une méthode *cri()* abstraite également (le contenu n'est pas implémenté dans la classe.)

```
abstract class Animal {
   private $nom;
   public function __construct(string $nom) {
      $this->nom = $nom;
   }
   public function getNom() : string {
      return $this->nom;
   }
   public function setNom(string $nom) : void {
      $this->nom = $nom;
   }
   public abstract function cri() : string;
}
```

Une classe *Chien*, va hériter de la classe abstraite *Animal* et va implémenter la méthode *cri()*.

```
class Chien extends Animal {
   private $race;
   public function __construct(string $nom, string $race) {
      Animal::__construct($nom);
      $this->race = $race;
   }
   public function getRace() : string { return $this->race; }
      public function setRace(string $race) : void {
      $this->race = $race;
   }
   public function cri() : string {
      return "Ouah ! Ouah !";
   }
}
```

Utilisation :

```
$animal = new Animal("Médor"); // Une erreur est affichée (instanciation impossible)

$chien = new Chien("Médor", "Caniche"); // Pas d'erreur
echo $chien->cri();
```

3.7 Les interfaces

Une interface est une classe abstraite sans donnée. Elle permet de définir un « contrat de programmation » qui liste les méthodes **qui doivent être implémentées.** Toutes ces méthodes sont publiques.

Il faut utiliser le mot clé *interface* lors de la déclaration de la classe.
Pour implémenter une interface, il faut utiliser le mot clé *implements*. Contrairement à l'héritage, il est possible d'implémenter plusieurs interfaces sur une même classe.

```
interface IPeutCrier {
       public function crier();
}
```

Notre interface (nommée par convention avec un I majuscule puis le verbe 'pouvoir' ; en anglais : commence par « can » ou termine par « able ») *IPeutCrier* indique que les objets qui vont l'implémenter doivent avoir une méthode *crier()* implémentée. Si ce n'est pas le cas, un message d'erreur sera affiché.