

Programmation orientée objet en PHP

Cyril Rabat

`cyril.rabat@univ-reims.fr`

INFO0303 - Conception Web 2

2018-2019

CM n°4

Introduction à la programmation orientée objet

Version 16 octobre 2018

Table des matières

1 Les bases de la programmation orientée objet

- Méthodes et attributs
- Utilisation des classes et objets
- *Getters* et *setters*
- Constantes et membres de classe
- Méthodes magiques
- Affectations et passage de paramètres

2 Héritage et interfaces

- L'héritage en PHP
- Les classes abstraites et les interfaces

Les classes (1/2)

- **Classe** : modèle décrivant...
 - ↪ ...des caractéristiques communes
 - ↪ ...des comportements communs d'un ensemble d'éléments
- **Objet** : instance d'une classe
 - ↪ Généré à partir de la classe
- **Membres** :
 - Attributs (données)
 - ↪ Variables propres
 - Méthodes
 - ↪ Fonctions propres

Les classes (2/2)

Structure générale d'une classe

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

- Classe Personne contenue dans le fichier "Personne.php" :
 ↪ **AVEC UNE MAJUSCULE AU DÉBUT, SANS ACCENT!!!**
- Une classe par fichier (sauf classées privées, etc.)

Attributs

- Correspondent à des variables, propres à un objet
- Définition en début de classe (de préférence)
- Syntaxe :
 - Modificateur de portée : `private` ou `public`
 - Le nom de l'attribut (avec le \$)

Exemple de définition d'attributs

```
class Personne {  
  
    private $prenom;  
    private $nom;  
    ...  
}
```

Méthodes

- Fonctions propres à une classe = méthodes
- Comme une fonction classique + modificateur de portée
- Permet d'accéder aux attributs (même privés)
- Utilisation de la pseudo-variable `$this`
↪ Correspond à une référence de l'objet

Exemple de définition d'une méthode

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

Construire un objet

- Utilisation d'une méthode spécifique appelée **constructeur**
- Permet :
 - D'instancier (de créer) un objet
 - D'initialiser les attributs
- Nom de la méthode : `__construct`

Exemple de définition d'un constructeur

```
...  
public function __construct(string $prenom, string $nom) {  
    $this->prenom = $prenom;  
    $this->nom = $nom;  
}  
...
```

Utilisation d'un objet

- Pour instancier un objet : opérateur `new`
- Appel du constructeur et retour d'une référence sur l'objet
- Accès à une méthode/attribut (publique) : `->`

Exemple d'utilisation d'un objet

```
class Personne {  
    ...  
}  
$p = new Personne("Cyril", "Rabat");  
$p->afficher();  
print_r($p);  
  
// Cyril Rabat  
// Personne Object ( [prenom:Personne:private] => Cyril [nom:...
```


Fichiers séparés

- Pour utiliser une classe :
 - Définition de la classe dans le script
 - ↪ Problème car non réutilisable
 - Utilisation d'un script spécifique
 - ↪ Nom du fichier = nom de la classe + extension `.php`
- Inclusion d'un script PHP :
 - `include` (ou `require`)
 - `include_once` : inclusion unique
- Toutes les fonctions et variables du script sont incluses

Chargement automatique

- Utilisation d'une fonction nommée `__autoload`
↪ Appelée automatiquement, personnalisable
- Autre solution (amenée à remplacer la précédente) :
 - Définition d'une fonction de chargement automatique
 - Enregistrement de cette fonction avec `spl_autoload_register`

Utilisation de `__autoload`

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

$p = new Personne("Bob", "Bob");
?>
```

Utilisation de `spl_autoload_*`

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

spl_autoload_register('charge');

$p = new Personne("Bob", "Bob");
?>
```

Retour sur les modificateurs de portée

- Permettent de protéger les attributs :
 - ↪ Évite la modification non contrôlée
- Pour récupérer les valeurs, utilisation de *getters* :
 - ↪ Méthodes retournant la valeur d'un attribut
- Pour modifier les valeurs, utilisation de *setters* :
 - ↪ Méthodes prenant en paramètre la nouvelle valeur

Les *getters*

- Retourne la valeur des attributs
- Nom : commencent par `get` suivi par une majuscule

Exemple de *getters*

```
class Personne {  
    ...  
    public function getPrenom() : string {  
        return $this->prenom;  
    }  
    public function getNom() : string {  
        return $this->nom;  
    }  
    ...  
}
```

Les *setters*

- Modifient les valeurs des attributs
- Nom : commencent par `set` suivi par une majuscule

Exemple de *setters*

```
class Personne {  
    ...  
    public function setPrenom(string $prenom) {  
        $this->prenom = $prenom;  
    }  
    public function setNom(string $nom) {  
        $this->nom = $nom;  
    }  
    ...  
}
```

Constantes

- Possible de définir des constantes dans la classe
- Syntaxe :
 - ↪ Mot-clé `const`, nom (sans \$), "=", valeur
 - ↪ Pas de modificateur de portée
- Accès avec l'opérateur `::`

Exemple de définition

```
class A {  
  
    const PI = 3.14159265359;  
  
}
```

Exemple d'utilisation

```
echo "La_valeur_de_PI_est_".  
    A::PI."<br/>";
```

Membres de classe (1/2) : définition

- Membres d'instance :
 - Nécessite d'instancier un objet pour y accéder
 - Propres à chaque objet
- Membres de classe :
 - Communs à tous les objets de la classe
 - Pas d'accès à `$this`
- Accès avec l'opérateur `::`
↪ Possible d'utiliser `self` au sein de la classe
- Mot-clé pour déclarer un membre de classe : `static`

Membres de classe (2/2) : exemple

Exemple de définition

```
class Cercle {  
    const PI = 3.1415;  
  
    public static function getPerimetre(float $rayon) : float {  
        return 2 * Cercle::PI * $rayon;  
        // ou return 2 * self::PI * $rayon;  
    }  
}
```

Exemple d'utilisation

```
echo "Périmètre_du_cercle_unité_: ".  
    Cercle::getPerimetre(1.0). "<br/>";
```


Qu'est-ce qu'une méthode magique ?

- Méthodes que l'on peut définir et qui possèdent des comportements par défaut
- Commencent toutes par "__" (deux "_")
- Exemples :
 - `__construct`, `__destruct` : constructeur et destructeur
 - `__toString` : conversion en string
 - `__clone` : copie d'un objet
 - `__set`, `__get`, `__isset` et `__unset` :
 ↪ Appelées lors de la modification, récupération, etc. de propriétés inaccessibles

Remarque

Dans ce cours, nous ne traiterons que des trois premiers points.

Méthode `__toString`

- Permet de personnaliser la conversion en chaîne de caractères
- Retourne une chaîne de caractères

Exemple d'utilisation de `__toString`

```
class Personne {  
    ...  
    public function __toString() {  
        return $this->prenom." ".$this->nom;  
    }  
    ...  
}  
$p = new Personne("Cyril", "Rabat");  
echo $p;  
  
// Sortie : Cyril Rabat
```

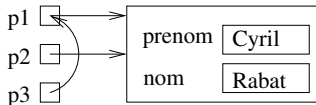
Affectations d'une variable avec un objet (1/2)

- En PHP, la variable référence l'objet
↪ Différent des types primitifs
- En cas d'affectation, seule la référence vers l'objet est copiée

Exemple d'affectations

```
$p1 = new Personne("Cyril", "Rabat");  
$p2 = $p1;  
$p3 = & $p1;
```

Illustration mémoire



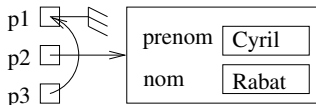
Affectations d'une variable avec un objet (2/2)

- En PHP, la variable référence l'objet
↳ Différent des types primitifs
- En cas d'affectation, seule la référence vers l'objet est copiée

Exemple d'affectations

```
$p1 = new Personne("Cyril", "Rabat");  
$p2 = $p1;  
$p3 = & $p1;  
  
$p1 = null;
```

Illustration mémoire



Cloner un objet

- Utilisation de l'opérateur `clone`
- Possible de redéfinir le comportement par défaut :
↪ Redéfinition de la méthode magique `__clone`

Exemple d'utilisation de *clone*

```
$p1 = new Personne("Cyril", "Rabat");  
$p2 = clone($p1); // Ou clone $p1  
$p1->setNom("Lignac");  
echo "$p1_et_$p2";  
  
// Sortie : Cyril Lignac et Cyril Rabat
```

Passage de paramètre

- Comme pour l'affectation : passage de la "référence"
↪ Objet modifiable dans la fonction
- Si l'on passe l'adresse :
↪ Variable modifiable dans la fonction

Exemple d'utilisation de `__toString`

```
function modifie(Personne $p) {  
    $p->setNom($p->getNom()."_(modifié)");  
}  
$p1 = new Personne("Cyril", "Rabat");  
echo "Avant_:_$p1_et_après_:_" ;  
modifie($p1);  
echo "$p1<br/>";  
  
// Sortie : Avant : Cyril Rabat et après : Cyril Rabat (modifié)
```

L'héritage (1/2)

- Objectifs multiples :
 - ↪ Partage du code
 - ↪ Réutilisabilité
 - ↪ Factorisation
- Relation de généralisation / spécialisation
- En PHP : pas d'héritage multiple
- Utilisation du mot-clé `extends`

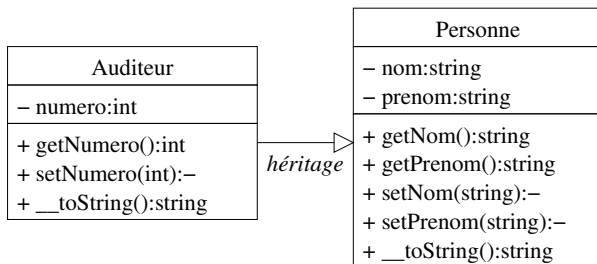
Exemple

```
class Auditeur extends Personne {  
    ...  
}
```

L'héritage (2/2)

- Transmission des membres :
 - Héritage de tous les membres
 - `public` : accès total par la classe fille
 - `private` : pas d'accès par la classe fille
 - `protected` :
 - ↪ Accès comme s'ils étaient "`public`" pour la classe fille
 - ↪ Pas d'accès pour les autres classes
- Constructeur dans la classe fille :
 - Appel du constructeur de la classe mère si nécessaire :
 - ↪ `parent::__construct(...)` :
 - ↪ Première instruction du constructeur (de préférence)
 - Initialisation des attributs de la classe fille

Exemple (1/2)



Explications

- Un auditeur **est** une personne
- Il possède en plus des nom et prénom un numéro d'auditeur

Exemple (2/2)

Extrait du code de la classe 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) {  
        $this->numero = $numero;  
    }  
    ...  
}
```

Redéfinition de méthodes

- Possible de redéfinir une méthode existante dans la classe mère :
↪ Sauf si la méthode est `final`
- Même signature que dans la classe mère
- Depuis la classe fille, possible d'appeler celle de la classe mère :
↪ Instruction : `parent::nomDeLaMethode(...)`
- De l'extérieur : seule la méthode redéfinie est accessible

Extrait du code de la classe `Etudiant`

```
class Auditeur extends Personne {  
    ...  
    public function __toString() : string {  
        return parent::__toString()."_"($this->numero)";  
    }  
    ...  
}
```

Polymorphisme

- Lors d'un appel de méthode :
 - ↪ La méthode est définie dans la classe...
 - ↪ ...ou dans la classe mère (voire plus "haut" dans la hiérarchie)
 - ↪ Soit les deux : redéfinition
- En cas de redéfinition, appel à la méthode la plus spécifique

Exemple de polymorphisme

```
$p = new Auditeur("Cyril", "Rabat", 12345);  
echo $p;  
  
// Affichage : Cyril Rabat (12345)
```

Typage dynamique

- Quand une classe est spécifiée comme type de paramètre (ou retour) :
 - Possibilité de spécifier `null`
 - Un objet de cette classe...
 - ...ou de toute classe qui en hérite

Exemple de typage dynamique

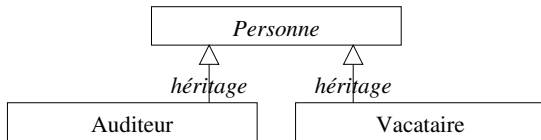
```
function compliment(Personne $p) {  
    echo "". $p->getNom(). " est un joli nom<br/>";  
}  
$auditeur = new Auditeur("Cyril", "Rabat", 123456);  
compliment($auditeur);
```

Attention

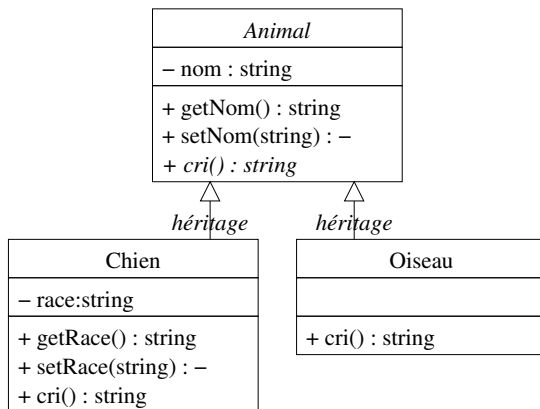
Si `null` est passé à la fonction `compliment`, cela entraîne une erreur fatale.

Une classe abstraite

- Classes dans lesquelles des méthodes sont déclarées mais non définies
↔ Utilisation du mot-clé `abstract`
- Une classe abstraite ne peut être instanciée
- Une classe fille qui hérite d'une classe abstraite :
 - Peut utiliser le constructeur de la classe mère
 - Est abstraite si les méthodes abstraites ne sont pas définies



Exemple complet (1/2)



Exemple complet (2/4)

Classe abstraite Animal

```
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) {  
        $this->nom = $nom;  
    }  
  
    public abstract function cri() : string;  
}
```


Exemple complet (3/4)

Classe Chien

```
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) { $this->race = $race; }  
  
    public function cri() : string {  
        return "Ouah_!_Ouah_!";  
    }  
  
}
```

Exemple complet (4/4)

Exemple d'utilisation : erreur

```
$animal = new Animal("Médor");  
  
// Classe Animal abstraite => pas d'instanciation !
```

Exemple d'utilisation : pas d'erreur

```
$chien = new Chien("Médor", "Caniche");  
echo $chien->cri();
```

Les interfaces

- Une interface est une classe abstraite sans donnée
- Intérêt : définit un contrat de programmation
 - ↪ Liste de méthodes qui doivent être implémentées
 - ↪ Toutes publiques !
- Peut contenir des constantes
- Mot clé : `interface`
- Pour implémenter une interface : `implements`
 - ↪ Possible d'implémenter plusieurs interfaces
- Une interface peut hériter d'une autre :
 - ↪ Une classe qui implémente l'interface "fille" implémentera les méthodes des deux interfaces