Membres de classes et membres d'instances

(Rappel cours 5)

Un membre peut être lié à l'instance de la classe (on parle alors de *membre d'instance*), c'est le cas dans tous les exemples ci-dessus, et ne peut en aucun cas être utilisé en l'absence d'une instance, ce n'est pas le cas d'un membre de classe.

Un membre peut également être lié à la classe elle-même plutôt qu'a l'instance; on parle alors de *membre de classe* ou de *membre statique*. Il est pour cela précédé du mot clé static.

Un membre statique est accessible par l'intermédiaire de l'opérateur de résolution de portée :: .

Au sein d'une classe le mot clé **self** permet d'accéder aux membres statiques, *self représente la classe en cours tout comme $this qui représente l'instance en cours* ($this n'est bien entendu pas disponible au sein d'un membre statique).

La visibilité s'applique de la même façon sur les membres d'instances et sur les membres de classes, leurs effets sont basiquement les mêmes sauf dans le cas de l'héritage de méthodes statiques que nous détaillerons plus bas.

Exemple de classe disposant de membres statiques:

```
 1 <?php
 2
 3 class MaClasse {
 4
 5    // propriété de classe
 6    public static $prefixe = "hello";
 7
 8    // propriété d'instance
 9    protected $_phrase;
10
11     // constructeur
12     public function __construct ($mot) {
13         $this->_phrase = self::$prefixe . ' ' . $mot;
14     }
15
16     // méthode d'instance
17     public function afficher () {
18         echo $this->_phrase;
19     }
20
21     // méthode de classe
22     public static function definirPrefixe ($prefixe) {
23         self::$prefixe = $prefixe;
24     }
25 }
26
27 $obj1 = new MaClasse("world");
28 $obj1->afficher(); // affiche hello world
29
```

```
30 MaClasse::definirPrefixe("strange");
31
32 $obj2 = new MaClasse("word");
33 $obj2->afficher(); // affiche strange world
34 $obj1->afficher(); // affiche strange world également
35
36 ?>
```

L'héritage

Avec l'héritage, on entre dans la vraie plus-value des langages objets par rapport aux langages impératifs, il s'agit d'un moyen simple de favoriser la *réutilisabilité* du code.

L'héritage va permettre une réutilisation verticale du code, c'est à dire que nos classes vont former une hiérarchie dans laquelle les enfants (les *classes filles*) pourront réutiliser les comportements et les données de leur(s) parent(s) (les *classes mères*).

Important: l'héritage entre deux classes caractérise la relation "*est un espèce de...*" Par exemple: une voiture est une espèce de véhicule, un chien est une espèce d'animal. L'héritage doit être sémantiquement valide afin d'éviter les dérives comme l'héritage fonctionnel.

Comme le disait un célèbre professeur: "*ce n'est pas parce que le chien pisse sur le poteau que le chien est un espèce de poteau*"... Donc ce n'est pas parce que le chien *utilise* le poteau qu'il doit *hériter* de poteau.

Concrêtement, lors qu'une classe B (la *fille*) hérite d'une classe A (la *mère*), elle dispose alors de tous les membres publics ***et protégés*** de sa mère (d'ou l'importance de cette différentiation entre membre protégé et privé). On réalise un héritage quand il y a lieu d'effectuer une *spécialisation*, c'est à dire que la classe fille va *redéfinir* (on parle aussi de *surcharge*) ou *ajouter* des comportements.

Pour que la classe B hérite de A, on utilise le mot clé **extends** comme montré ci-dessous. On peut, depuis une classe fille, faire explicitement appel au méthode de la classe parente à l'aide du mot clé parent, c'est pratique lorsqu'il s'agit de réutiliser une partie du comportement de la classe parent sans pour autant devoir le réécrire.

```
 1 <?php
 2
 3 class Vehicule {
 4
 5    protected $nombre_de_roues;
 6
 7    public function __construct ($nombre_de_roues) {
 8       $this->nombre_de_roues = $nombre_de_roues;
 9    }
10
11     public function nombreDeRoues () {
12        return $this->nombre_de_roues;
13     }
```

```
14 }
15
16 class Voiture extends Vehicule {
17
18     public function __construct () {
19         // appel du constructeur parent
20         parent::__construct(4);
21     }
22 }
23
24 class Moto extends Vehicule {
25
26     public function __construct () {
27         // appel du constructeur parent
28         parent::__construct(2);
29     }
30 }
31
32 $ma_voiture = new Voiture;
33 echo "Ma voiture a " . $ma_voiture->nombreDeRoues() . " roues"; // Ma voiture a 4 roues
34
35 $ma_moto = new Moto;
36 echo "Ma moto a " . $ma_moto->nombreDeRoues() . " roues"; // Ma moto a 2 roues
37
38 ?>
```

Important: En PHP, l'héritage multiple **n'existe pas**. Ce qui signifie que vous ne pouvez par déclarer une classe qui hérite de deux autres classes. En somme, class A extends B, C c'est interdit. Vous pouvez en revanche *réaliser* plusieurs interfaces (voir le chapitre sur le polymorphisme ci-dessous).

Exercice :

Créez une classe addition qui retourne la somme de 3 `chiffres`

Puis une autre classe moyenne qui hérite de cette classe et retourne la moyenne de ces 2 `chiffres`.

Par exemple `$chiffre_1 = 10,  $chiffre_2 = 15  et $chiffre_3 = 20`

`$somme= 45 et moyenne= 15`

Solution :

```
<?php
class addition_class
{
    function addition_class($a,$b,$c)
    {
        global $somme;
        $somme = $a +  $b + $c;
        return $somme ;
    }

}
class moyenne_class extends addition_class
{
    function moyenne($a,$b,$c)
    {
        global $somme, $maMoyenne ;
        $this -> addition_class($a,$b,$c);
        $maMoyenne = $somme/3 ;
        return $maMoyenne;
    }
}

$chiffre_1 = 10 ;
$chiffre_2 = 15 ;
$chiffre_3 = 20 ;
$maMoyenne = new moyenne_class($chiffre_1,$chiffre_2,$chiffre_3) ;
echo "<small>La somme des trois nombres $chiffre_1, $chiffre_2 et
$chiffre_3 est
    $somme <br/>";
$maMoyenne -> moyenne($chiffre_1,$chiffre_2,$chiffre_3);
echo "La moyenne des trois nombres $chiffre_1, $chiffre_2 et $chif
fre_3 est
    $maMoyenne ";
?>
```

Filliation, parents, enfants et arbre généalogique

Une classe n'est pas seulement un moyen commode de rassembler des données et des comportements au sein d'une structure, c'est surtout un excellent moyen de créer de nouveaux *types* de données, car en réalité, une classe peut être considérée comme un type au même sens qu'une chaine ou un entier.

Quand on réalise un héritage, la classe fille caractérise un nouveau type mais est toujours du type de la mère. Par exemple une pomme est toujours considérée un fruit, qui est toujours considéré comme un végétal. En programmation orientée objet c'est le même concept, on parle alors de *hiérarchie de types*.

Comme tout bon langage objet, PHP dispose de l'opérateur instanceOf qui permet de savoir si une instance est d'un type donné. Cet opérateur renvoie true si l'objet est du type (ou *super-type*) spécifié.

Exemple de hiérarchie:

```
<?php

class Vivant {
}

class Vegetal extends Vivant {
}

class Fruit extends Vegetal {
}

class Apple extends Fruit {
}

class GoldenLady extends Apple {
}

$object = new GoldenLady;

var_dump( $object instanceOf Apple );  // true
var_dump( $object instanceOf Fruit );  // true
var_dump( $object instanceOf Vegetal); // true
// ... etc.

?>
```

Il est d'ailleurs possible de spécifier sur un prototype quel type d'objet est attendu pour un paramètre donné:

```
<?php

function manger (Fruit $fruit)
{
    echo "I'm eating " . get_class($fruit);
}

$object = new Apple;
manger($object); // I'm eating Apple

?>
```

On appelle cette syntaxe de paramètres le *type-hinting*, c'est à dire qu'on demande explicitement des objets du type (ou super-type) spécifié. N'importe quoi d'autre provoquera une erreur. Si on avait passé un type natif comme un

entier ou encore une instance qui hérite de la classe Vegetal mais qui n'est pas un fruit, on se serait fait jeter.

Le type-hinting est extrèmement pratique pour sécuriser vos fonctions et vos méthodes en empêchant le développeur d'y mettre n'importe quoi.

Note: contrairement à d'autres langages objets comme Java, les objets n'héritent pas par défaut de la class Object, vous devez donc recourrir à is_object si vous voulez valider le type objet.

Le polymorphisme

Le polymorphisme (du grec *plusieurs formes*) est le mécanisme grâce auquel une même méthode peut être implémentée par plusieurs types (donc plusieurs classes) favorisant ainsi la *généricité*. Concrêtement, si une méthode est déclarée et/ou définie dans la classe parente d'une classe donnée, vous n'avez pas à vous préocuper du type précis de la fille avec laquelle vous travaillez pour l'utiliser.

Par exemple, une méthode obtenirAire est commune aux classes Cercle, Carre et Rectangle qui héritent toutes de Forme. Quand vous travaillez avec une instance de Forme, vous n'avez donc plus à vous soucier de savoir si c'est un cercle, un carré ou un rectangle pour obtenir son aire.

Comme dans la plupart des langages objet, le polymorphisme en PHP est un polymorphisme par *sous-typage* (ou *dérivation*) c'est à dire qu'on va se servir de la redéfinition de comportements prévue par l'héritage pour faire du polymorphisme. Nous avons vu plus haut avec Vehicule, Voiture et Moto qu'il est possible de redéfinir une méthode de la classe mère dans sa fille (en l'occurence, il s'agissait de son constructeur). Nous allons maintenant voir qu'il est possible d'aller plus loin.

Abstraction

En programmation objet, il est possible de déclarer des méthodes dont la définition n'est pas encore connue dans la classe mère mais le seront dans une classe fille. On parle alors de *méthodes abstraites*. Une classe qui contient au moins une méthode abstraite doit elle aussi être déclarée abstraite et ne peut plus être instanciée - vu que tout son code n'est pas implémenté. On parle alors de *classe abstraite*.

Une méthode abstraite est précédée du mot clé abstract (idem pour la classe qui la porte) et ne peut pas avoir de corp.

Exemple de classe abstraite:

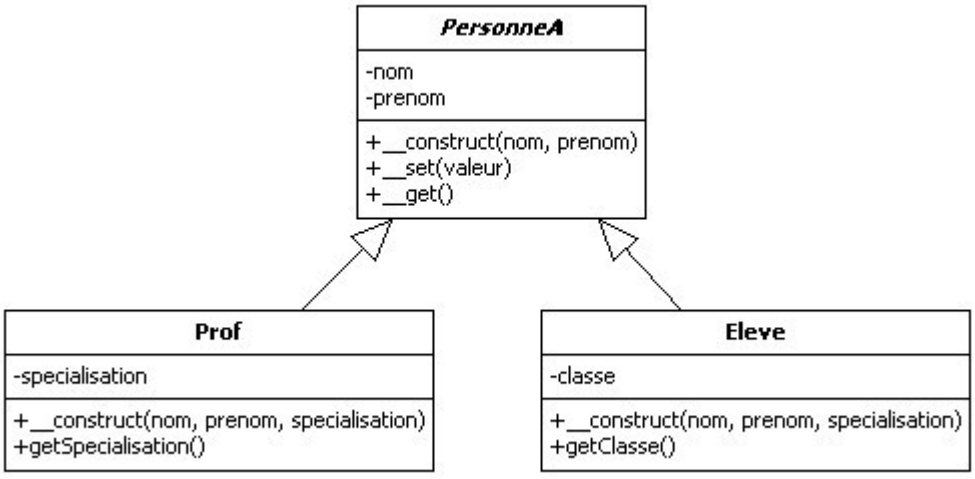
```
1 <?php
2
3 abstract class Animal {
4
```

```
 5    protected $_nom;
 6
 7    public function __construct ($nom) {
 8       $this->_nom = $nom;
 9    }
10
11     abstract public function parler ();
12 }
13
14 class Chien extends Animal {
15
16     public function parler () {
17        echo "$this->_nom: Wouf Wouf\n";
18     }
19 }
20
21 class Chat extends Animal {
22
23     public function parler () {
24        echo "$this->_nom: Miaou\n";
25     }
26 }
27
28 $chien = new Chien("Rex");
29 $chien->parler(); // affiche Rex: Wouf Wouf
30
31 $chat  = new Chat("Sac-a-puces");
32 $chat->parler();  // affichie Sac-a-puces: Miaou
33
34 ?>
```

Par opposition, une classe qui définit l'intégralité de ses comportements est souvent appellé classe concrête.

Exercice :

Développez la classe abstraite personne et ses spécialisations prof et élève



Solution :

```php
<?php
	// --- PersonneA.php
	abstract class PersonneA
	{
		private $prenom;
		private $nom;

		protected function __construct($prenom, $nom)
		{
			$this->prenom = $prenom;
			$this->nom    = $nom;
		}
		public function __set($var, $valeur) { $this->$var = $valeur; }
		public function __get($var) { return $this->$var; }
	}

	// --- Prof.php
	require_once("PersonneA.php");
	class Prof extends PersonneA
	{
		private $specialisation;

		public function __construct($prenom, $nom, $specialisation)
		{
			$this->specialisation = $specialisation;
			parent::__construct($prenom, $nom);
		}
		public function Specialisation() { return $this->specialisation; }
	}

	// --- Eleve.php
	require_once("PersonneA.php");
	class Eleve extends PersonneA
	{
		private $classe;

		public function __construct($prenom, $nom, $classe)
		{
			$this->classe = $classe;
			parent::__construct($prenom, $nom);
		}
		public function Classe() { return $this->classe; }
	}

	// --- profEleveTest.php
	require_once("Prof.php");
	require_once("Eleve.php");

	//$personne = new PersonneA("P","P");  // --- Interdit
	$prof  = new Prof("Pascal","Buguet","Maths");
	$eleve = new Eleve("Annabelle", "Buguet", "Terminale");

	echo "<br />$prof->prenom $prof->nom est specialiste en " . $prof->Specialisation();
	echo "<br />$eleve->prenom $eleve->nom est en " . $eleve->Classe();
?>
```

Classe Final

Une classe *final* est une classe qui n'est pas héritable. C'est la fin d'une chaîne d'héritage.

final class NomDeClasse { ... }

Si vous essayez d'étendre une classe finale vous obtiendriez ce message :

Fatal error: Class HeriteFinal may not inherit from final class (Finale) in C:\wamp\www\php\cours_objet\finale.php on line 12

- Méthode Final

Une méthode final n'est pas surchargeable.

final public function NomDeFonction(...) { ... }

Note : lorsque nous étudierons les Exceptions, nous verrons que dans la classe Exception de PHP il y a un certain nombre de méthodes "final".

Interfaces

Si toutes les méthodes d'une classe sont abstraites, cette classe est alors une *interface*. Dès lors, on utilise le mot clé interface en lieu et place du mot clé class. L'intérêt principal des interfaces est que plusieurs d'entre elles peuvent être *réalisées* (ou *implémentées*) par une même classe. En somme class A extend B implements C,D,E est tout à fait valide.

On croise d'ailleurs fréquement des classes qui réalisent plusieurs interfaces, c'est le cas notamment de la classe PHP ArrayIterator qui réalise Iterator, Traversable, ArrayAccess, SeekableIterator, Countable et Serializable.

En fait l'interface donne une "*forme*" a notre classe: elle va définir comment celle-ci doit se *présenter* (par là on entends quelles devront être ses méthodes) mais pas comment elle doit se *comporter*. Les iterateurs de la SPL en sont un bel exemple: tous les itérateurs partagent des prototypes de méthodes communs mais chacun les implémente différement.

Note: vu que toutes les méthodes d'une interface sont obligatoirement abstraites, il est inutile de le spécifier avec le mot clé abstract sur les méthodes et la classe.

Exemple simple :

```
<?php
interface Foo 
{
    function a();
}
interface Bar 
{
    function b();
}
class FooBar implements Foo, Bar 
{
    function a()
    {
        print "aaaaaaaa<br/>";
    }
    function b()
    {
       print "bbbbbbb<br/>";
    }
}
$aa = new FooBar;
echo "<b>Appel fonction a</b><br/>";
$aa->a();
echo "<b>Appel fonction b</b><br/>";
$aa->b();
?>
```

Exemple d'interface:

```
 1 <?php
 2
 3 interface Forme2D {
 4
 5    // toutes les formes en deux dimention ont une aire...
 6    public function obtenirAire ();
 7
 8    // ... et un périmètre
 9    public function obtenirPerimetre ();
10 }
11
12 class Carre implements Forme2D {
13
14    protected $_cote;
15
16    public function __construct ($cote) {
17       $this->_cote = $cote;
18    }
19
```

```
20     public function obtenirAire () {
21         return pow($this->_cote, 2);
22     }
23 
24     public function obtenirPerimetre () {
25         return 4 * $this->_cote;
26     }
27 }
28 
29 class Rectangle implements Forme2D {
30 
31     protected $_longueur;
32     protected $_largeur;
33 
34     public function __construct ($longueur, $largeur) {
35         $this->_longueur = $longueur;
36         $this->_largeur = $largeur;
37     }
38 
39     public function obtenirAire () {
40         return $this->_longueur * $this->_largeur;
41     }
42 
43     public function obtenirPerimetre () {
44         return 2 * ($this->_longueur + $this->_largeur);
45     }
46 }
47 
48 class Cercle implements Forme2D {
49 
50     protected $_rayon;
51 
52     public function __construct ($rayon) {
53         $this->_rayon = $rayon;
54     }
55 
56     public function obtenirAire () {
57         return M_PI * pow($this->_rayon, 2);
58     }
59 
60     public function obtenirPerimetre () {
61         return M_PI * ($this->_rayon * 2);
62     }
63 }
64 
65 ?>
```

Dans l'exemple ci-dessous, connaissant les méthodes déclarées par Forme2D, nous pouvons décrire des classes qui travaillent avec n'importe quelle instance de Forme2D. C'est ça la généricité !

Exemple d'utilisation générique d'une forme:

```
1 <?php
```

```
 2
 3 // en reprennant l'exemple précédent
 4
 5 class Figure2D {
 6
 7     protected $_formes;
 8
 9     public function ajouter (Forme2D $forme) {
10         $this->_formes[] = $forme;
11     }
12
13     public function surfaceTotale () {
14         $surface = 0;
15         foreach ($this->_formes as $forme)
16             $surface += $forme->obtenirAire();
17
18         return $surface;
19     }
20 }
21
22 $figure = new Figure;
23 $figure->ajouter(new Cercle(3));
24 $figure->ajouter(new Carre(4));
25 $figure->ajouter(new Rectangle(5,6));
26
27 echo "Ces trois figures ont une surface totale de " . $figure->surfaceTotale(); // 74.27...
28
29 ?>
```

On voit rapidement l'intérêt du polymorphisme dans ce cas: grâce à l'interface Forme2D, on peut créer autant de type de formes qu'on veut, par exemple le triangle, le losange, le polygone etc. Et tous ces objets, aussi longtemps qu'ils hériteront de Forme2D, seront utilisables avec la classe Figure2D.

Contrairement aux classes, une interface peut étendre plusieurs interfaces avec le mot clé extends. On notera également que les cas qui justifient un héritage d'interfaces sont assez rares, il est peu probable que vous en ayez besoin un jour.

Exemple d'héritage d'interfaces:

```
 1 <?php
 2
 3 interface Forme {
 4     // ...
 5 }
 6
 7 interface Forme2D extends Forme {
 8     // ...
 9 }
10
11 interface Forme3D extends Forme {
12     // ...
13 }?>
```

Conclusion

Vous l'aurez compris, la programmation orientée objet c'est aussi complexe que puissant. Une fois les base de l'OOP maîtrisées, vous pourrez commencer à aller plus loin dans l'architecture logicielle, la programmation par composants, les design-patterns etc.

De nos jours, la plupart des infrastructure logicielles importantes ou complexes sont codées en Objet, ce n'est pas un hasard. La **simplicité** grâce à l'encapsulation, la **réutilisabilité** grâce à l'héritage et la **généricité** grâce au polymorphisme expliquent à elles trois l'engoument persistant pour les langages objets et leur large domination parmi les langages de programmation.

Vous verrez, une fois habitué, penser objet vous sera naturel.