PHP POO

Introduction

L'objectif de ce chapitre n'est pas de vous expliquer toutes les subtilités de la programmation orientée objet (POO) mais simplement de voir les bases pour vous permettre de programmer un code simple ou de comprendre comment fonctionne un code objet existant.

Pour l'instant, le code vu est de type procédural, c'est-à-dire que vous créez des fonctions que vous appelez au moment où vous en avez besoin, tout cela dans l'ordre chronologique.

En POO, presque tout est objet et tous les objets interagissent entre eux.

Un objet a des caractéristiques appelées attributs et des actions appelées méthodes.

Par exemple, l'objet Animal a les attributs couleur et poids. Il a aussi comme méthodes se_deplacer et manger.

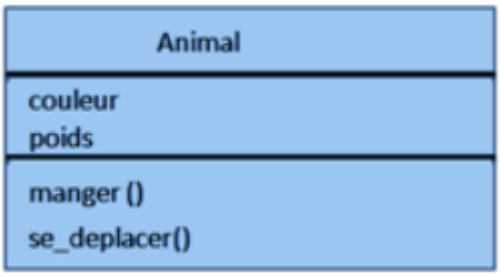
Pour construire ces objets, il faut utiliser une classe.

Les classes

1. Introduction

Une classe sert à fabriquer des objets à partir d'un modèle. Ces objets ont leurs propres attributs et certaines méthodes.

Par exemple, la classe Animal a les attributs couleur et poids et les méthodes manger ou se_deplacer.



Lorsque vous allez créer des exemplaires d'animaux à partir de la classe Animal, vous allez créer une instance de cette classe. Instancier une classe revient à créer un objet d'un certain type (Animal) avec certains attributs (couleur, poids).

Création d'une classe en PHP :

```
<?php
    class Animal // mot-clé class suivi du nom de la classe.
    {
```

```
        // Déclaration des attributs et méthodes.
    }
?>
```

Il est conseillé de mettre une classe par fichier PHP ayant le même nom que la classe.

2. L'encapsulation

En POO, tous vos attributs doivent être cachés aux autres personnes utilisant vos classes. Si vous travaillez en équipe et que vous avez créé la classe Animal, les autres développeurs ne doivent pas pouvoir changer directement les attributs de votre classe. Ainsi, les attributs couleur et poids sont cachés aux autres classes. Ils sont donc déclarés privés. La classe Animal a des méthodes pour lire ou écrire dans ces attributs. C'est le principe de l'encapsulation. Cela permet d'avoir un code plus protégé lorsque vous travaillez en équipe.

La classe Animal, ayant les propriétés couleur et poids, a une méthode pour modifier sa couleur, une méthode pour lire sa couleur, une méthode pour modifier son poids, une méthode pour lire son poids ainsi que d'autres méthodes comme manger ou se_deplacer (voir la section Mettre à jour et lire les attributs de l'instance plus loin dans ce chapitre).

3. Visibilité des attributs et des méthodes

Il existe trois types de mot-clé pour définir la visibilité d'un attribut ou méthode :

- private : seul le code de votre classe peut voir et accéder à cet attribut ou méthode.
- public : toutes les autres classes peuvent voir et accéder à cet attribut ou méthode.
- protected : seul le code de votre classe et de ses sous-classes peut voir et accéder à cet attribut ou méthode.

Les sous-classes sont abordées lors de l'héritage dans une prochaine section.

Création d'une classe avec ses attributs en PHP :

```
<?php
    class Animal // mot-clé class suivi du nom de la classe.
    {
        // Déclaration des attributs.
        private $couleur;
        private $poids;
    }
?>
```

Il est possible de définir des valeurs par défaut à vos attributs :

```
<?php
    class Animal // mot-clé class suivi du nom de la classe.
    {
```

```
        // Déclaration des attributs.
        private $couleur = "gris";
        private $poids = 10;
    }
?>
```

Pour ajouter les méthodes à votre classe, les règles de visibilité sont les mêmes que pour celles des attributs :

```
<?php
    class Animal // mot-clé class suivi du nom de la classe.
    {
        // Déclaration des attributs et méthodes.
        private $couleur = "gris";
        private $poids = 10;

        public function manger()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

        public function se_deplacer()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

}
?>
```

4. Ajout d'une méthode dans une classe

Lorsque vous ajoutez du code dans votre classe, cela s'appelle implémenter la classe. Pour accéder aux attributs de votre classe, il faut utiliser la pseudo-variable `$this` représentant l'objet dans lequel vous écrivez.

Pour accéder à l'attribut ou la méthode de l'objet, il faut utiliser l'opérateur ->.

Par exemple, pour implémenter la méthode ajouter_un_kilo() dans la classe Animal :

```
<?php
```

```
class Animal // mot-clé class suivi du nom de la classe.
    {
        // Déclaration des attributs et méthodes.
        private $couleur = "gris";
        private $poids = 10;

        public function manger()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

        public function se_deplacer()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

        public function ajouter_un_kilo()
        {
            $this->poids = $this->poids + 1;
        }

}
?>
```

Lorsque vous allez appeler la méthode `ajouter_un_kilo()`, cela ajoutera 1 au poids actuel et donc le poids final sera de 11.

Les propriétés sont déclarées avec le symbole $ mais sont appelées avec $this sans ce symbole.

5. Utilisation d'une classe

Dans un premier temps, il faut créer un fichier Animal.class.php contenant le code PHP précédent.

Pour utiliser la classe Animal, il faut l'inclure dans la page où vous souhaitez l'appeler.

Créez une page utilisation.php et tapez le code suivant :

```
<?php
```

```
include('Animal.class.php');


?>
```

Maintenant que la classe est chargée, il est possible d'instancier la classe, c'est-à-dire de créer un objet ayant comme modèle la classe Animal :

```
<?php


//chargement de la classe
include('Animal.class.php');


//instanciation de la classe Animal
$chien = new Animal();
?>
```

La variable `$chien` est une instance de la classe Animal avec les attributs couleur, poids et méthodes manger, se_deplacer, ajouter_un_kilo propres à lui-même.

6. Mettre à jour et lire les attributs de l'instance

Le principe de l'encapsulation veut que tous les attributs doivent être privés. Il faut donc créer des méthodes publiques permettant de lire ou d'écrire dans ces attributs depuis une autre page PHP.

Ces méthodes sont appelées des accesseurs.

Leurs noms sont généralement précédés du préfixe `get` pour lire la valeur de l'attribut et `set` pour écrire la valeur de l'attribut.

La classe Animal avec les accesseurs est :

```
<?php
class Animal // mot-clé class suivi du nom de la classe.
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;



        //accesseurs
        public function getCouleur()
        {
            return $this->couleur; //retourne la couleur
        }
```

```
        public function setCouleur($couleur)
        {
            $this->couleur = $couleur; //écrit dans l'attribut couleur
        }

        public function getPoids()
        {
            return $this->poids; //retourne le poids
        }
        public function setPoids($poids)
        {
            $this->poids = $poids; //écrit dans l'attribut poids
        }

        //méthodes
        public function manger()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

        public function se_deplacer()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

        public function ajouter_un_kilo()
        {
            $this->poids = $this->poids + 1;
        }

}
?>
```

Les accesseurs sont publics donc ils permettent de lire ou d'écrire dans les attributs depuis n'importe quelle autre classe ou page PHP.

Exemple avec la page utilisation.php :

```
<?php
```

```php
//chargement de la classe
include('Animal.class.php');

//instanciation de la classe Animal
$chien = new Animal();

//lire le poids
echo "Le poids du chien est : ".$chien->getPoids()." kg<br />";
//ajout d'un kilo au chien
$chien->ajouter_un_kilo();
//lire le poids
echo "Le poids du chien est : ".$chien->getPoids()." kg<br />";
//mise à jour du poids du chien
$chien->setPoids(15);
//lire le poids
echo "Le poids du chien est : ".$chien->getPoids()." kg<br />";

?>
```

Affiche :

Le poids du chien est : 10 kg

Le poids du chien est : 11 kg

Le poids du chien est : 15 kg

En effet, le poids du chien est initialisé à 10. Ensuite, la méthode `ajouter_un_kilo()` ajoute 1, donc son poids devient 11. Enfin, la méthode `setPoids(15)` fixe le poids à 15.

Il est possible de créer autant d'instances que vous voulez.

Par exemple, pour créer un chat blanc de 5 kg et un chien noir de 18 kg :

```php
<?php

//chargement de la classe
include('Animal.class.php');

//instanciation de la classe Animal

$chien = new Animal();
```

```
//mise à jour du poids du chien
$chien->setPoids(18);
//lire le poids
echo "Le poids du chien est : ".$chien->getPoids()." kg<br />";
//mise à jour de la couleur du chien
$chien->setCouleur("noir");
//lire la couleur
echo "La couleur du chien est : ".$chien->getCouleur()."<br />";

//instanciation de la classe Animal
$chat = new Animal();
//mise à jour du poids du chat
$chat->setPoids(5);
//lire le poids
echo "Le poids du chat est : ".$chat->getPoids()." kg<br />";
//mise à jour de la couleur du chat
$chat->setCouleur("blanc");
//lire la couleur
echo "La couleur du chat est : ".$chat->getCouleur()."<br />";

?>
```

Affiche :

Le poids du chien est : 18 kg

La couleur du chien est : noir

Le poids du chat est : 5 kg

La couleur du chat est : blanc

7. Passage en paramètre de type objet

Les méthodes sont comme des fonctions, elles peuvent prendre des paramètres de type différent (Integer, String…) et même de type objet.

`$chat` et `$chien` sont des objets de type Animal. Ils peuvent être passés en paramètre d'une méthode à condition que celle-ci accepte ce type d'objet.

Pour tester cet exemple, il faut changer la méthode `manger()` de la classe Animal. Elle devient `manger_animal(Animal $animal_mangé)` et prend en paramètre un objet de type Animal.

La page Animal.class.php devient :

```
<?php
```

```
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;


        //accesseurs
        public function getCouleur()
        {
            return $this->couleur; //retourne la couleur
        }
        public function setCouleur($couleur)
        {
            $this->couleur = $couleur; //écrit dans l'attribut couleur
        }

        public function getPoids()
        {
            return $this->poids; //retourne le poids
        }
        public function setPoids($poids)
        {
            $this->poids = $poids; //écrit dans l'attribut poids
        }


        //méthodes
        public function manger_animal(Animal $animal_mangé)
        {
            //l'animal mangeant augmente son poids d'autant que
            //celui de l'animal mangé
            $this->poids = $this->poids + $animal_mangé->poids;
            //le poids de l'animal mangé et sa couleur sont remis à 0
            $animal_mangé->poids = 0;

            $animal_mangé->couleur = "";
        }
```

```
        public function se_deplacer()
        {
            //méthode pouvant accéder aux propriétés
            //couleur et poids
        }

        public function ajouter_un_kilo()
        {
            $this->poids = $this->poids + 1;
        }


}
?>
```

Pour tester cette méthode, la page utilisation.php devient :

```
<?php

//chargement de la classe
include('Animal.class.php');

//instanciation de la classe Animal
$chat = new Animal();
//mise à jour du poids du chat
$chat->setPoids(8);
//lire le poids
echo "Le poids du chat est : ".$chat->getPoids()." kg<br />";
//mise à jour de la couleur du chat
$chat->setCouleur("noir");
//lire la couleur
echo "La couleur du chat est : ".$chat->getCouleur()."<br />";

//instanciation de la classe Animal
$poisson = new Animal();



//mise à jour du poids du poisson
$poisson->setPoids(1);
//lire le poids
```

```
echo "Le poids du poisson est : ".$poisson->getPoids()." kg<br />";
//mise à jour de la couleur du poisson
$poisson->setCouleur("blanc");
//lire la couleur
echo "La couleur du poisson est : ".$poisson->getCouleur()."<br /><br />";

//le chat mange le poisson
$chat->manger_animal($poisson);
//lire le poids
echo "Le nouveau poids du chat est : ".$chat->getPoids()." kg<br />";
//lire le poids
echo "Le poids du poisson est : ".$poisson->getPoids()." kg<br />";
//lire la couleur
echo "Le couleur du poisson est : ".$poisson->getCouleur()."<br /><br />";

?>
```

Affiche :

Le poids du chat est : 8 kg

La couleur du chat est : noir

Le poids du poisson est : 1 kg

La couleur du poisson est : blanc

Le nouveau poids du chat est : 9 kg

Le poids du poisson est : 0 kg

La couleur du poisson est :

L'objet `$chat` appelle la méthode `manger_animal($poisson)` en passant en paramètre l'objet `$poisson` de type Animal. C'est-à-dire que l'objet `$poisson` avec ses attributs et ses méthodes sont passés en paramètre. Cela permet de passer plusieurs valeurs en paramètre avec un seul paramètre. La méthode `manger_animal(Animal $animal_mangé)` accepte uniquement un paramètre de type Animal.

Il n'est donc pas possible d'appeler la méthode de cette façon :

```
$chat->manger_animal("Grenouille");
```

ou de cette façon :

```
$chat->manger_animal(4);
```

car les types de "Grenouille" (String) et 4 (Integer) ne sont pas de type Animal.

8. Le constructeur

Le constructeur comme son nom l'indique sert à construire un objet du type de la classe. Lorsque vous écrivez `new Animal()`, cela appelle le constructeur par défaut de la classe Animal.

Il est possible de créer vos propres constructeurs et ainsi de pouvoir lui passer en paramètre la valeur des attributs que vous souhaitez affecter à votre objet.

Le constructeur est toujours nommé `__construct` et ne possède pas de `return`.

Pour ajouter un constructeur prenant en paramètres le poids et la couleur, la page Animal.class.php devient :

```
<?php
class Animal
{
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;

        public function __construct ($couleur, $poids)
//Constructeur demandant 2 paramètres.
        {
            echo 'Appel du constructeur.<br />';
            $this->couleur = $couleur; // Initialisation de
                                       //la couleur.
            $this->poids = $poids; // Initialisation du poids.
        }
Etc.
.
.
.
}
?>
```

Appel du constructeur dans la page utilisation.php :

```
<?php

//chargement de la classe
include('Animal.class.php');
```

```
//instanciation de la classe Animal avec votre constructeur
$chien = new Animal("beige",7);
//lire le poids
echo "Le poids du chien est : ".$chien->getPoids()." kg<br />";
//lire la couleur
echo "La couleur du chien est : ".$chien->getCouleur()."<br />";

//mise à jour de la couleur du chien
$chien->setCouleur("noir");
//lire la couleur
echo "La couleur du chien est : ".$chien->getCouleur()."<br />";

?>
```

Affiche :

Appel du constructeur.

Le poids du chien est : 7 kg

La couleur du chien est : beige

La couleur du chien est : noir

Il s'affiche en premier "Appel du constructeur" car l'instruction `echo` écrite dans le constructeur `__construct` de votre classe Animal est appelée à chaque fois que vous exécutez `new Animal()`.

Le constructeur prend en paramètre les valeurs de vos attributs, cela évite d'appeler les méthodes `setCouleur()` et `setPoids()`.

Depuis PHP 7, il n'est plus possible de définir des constructeurs suivant la norme PHP 4, c'est-à-dire un constructeur avec le même nom que la classe.

En PHP, il n'est pas possible de déclarer deux constructeurs dans la même classe.

9. Le destructeur

Le destructeur sert à détruire l'objet pour le libérer de la mémoire. Il est appelé automatiquement à la fin du script PHP ou lorsque l'objet est détruit.

Pour détruire un objet, vous pouvez utiliser la fonction `unset()`. Cette fonction prend en paramètre l'objet à détruire.

Par exemple, pour détruire l'objet `$chien` :

```
<?php
```

```
//destruction de l'objet
unset($chien);


?>
```

Cela va appeler le destructeur par défaut. Vous pouvez modifier le destructeur par défaut en ajoutant la fonction `__destruct()` dans la classe.

```
<?php
class Animal
{
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;


        public function __construct ($couleur, $poids)
//Constructeur demandant 2 paramètres.
        {
            echo 'Appel du constructeur.<br />';
            $this->couleur = $couleur; // Initialisation de
                                       // la couleur.
            $this->poids = $poids; // Initialisation du poids.
        }


        public function __destruct()
        {
            echo 'Appel du destructeur';
        }


Etc.
.
.
.
}
?>
```

Généralement, il n'est pas nécessaire d'implémenter le destructeur dans la classe.

10. Exercice

Énoncé (facile)

Dans la page utilisation.php, créer deux poissons :

- poisson1, gris, 10 kg
- poisson2, rouge, 7 kg

Afficher leur poids puis le poisson1 mange le poisson2.

Réafficher leur poids.

Solution

Affiche :

Appel du constructeur.

Appel du constructeur.

Le poids du poisson1 est : 10 kg

Le poids du poisson2 est : 7 kg

Le nouveau poids du poisson1 est : 17 kg

Le nouveau poids du poisson2 est : 0 kg

Appel du destructeur

Appel du destructeur

11. Les constantes de classe

Une constante de classe est semblable à une constante normale, c'est-à-dire une valeur affectée à un nom et qui ne change jamais.

Exemple de déclaration d'une constante normale :

```
define('PI',3.1415926535);
```

Une constante de classe représente une constante mais liée à cette classe.

Lorsque vous créez un animal avec le constructeur `__construct($couleur, $poids)`, vous tapez :

```
$chien = new Animal("gris",10);
```

Lorsque vous regardez le code, vous ne savez pas immédiatement que 10 représente le poids de l'animal.

Pour cela, vous pouvez utiliser des constantes représentant chacune un certain poids :

```
const POIDS_LEGER = 5;
const POIDS_MOYEN = 10;
const POIDS_LOURD = 15;
```

Les constantes sont toujours en majuscule sans le symbole $ et précédé du mot-clé const.

La classe Animal.class.php devient :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;

        //constantes de classe
        const POIDS_LEGER = 5;
        const POIDS_MOYEN = 10;
        const POIDS_LOURD = 15;
etc.
.
.
.
?>
```

Pour faire appel à cette constante depuis la page utilisation.php, la syntaxe est un peu particulière. Il faut mettre **::** entre la classe et sa constante :

```
<?php

//chargement de la classe
include('Animal.class.php');

//instanciation de la classe Animal avec votre constructeur
$poisson1 = new Animal("gris",Animal::POIDS_MOYEN);
$poisson2 = new Animal("rouge",Animal::POIDS_LEGER);
//lire le poids
echo "Le poids du poisson1 est : ".$poisson1->getPoids()." kg<br />";

//lire le poids
echo "Le poids du poisson2 est : ".$poisson2->getPoids()." kg<br />";

//le poisson1 mange le poisson2
$poisson1->manger_animal($poisson2);
//lire le poids
echo "Le nouveau poids du poisson1 est : ".$poisson1->getPoids()." kg<br />";
//lire le nouveau poids
```

```
echo "Le nouveau poids du poisson2 est : ".$poisson2->getPoids()." kg<br />";


?>
```

Affiche :

Appel du constructeur.

Appel du constructeur.

Le poids du poisson1 est : 10 kg

Le poids du poisson2 est : 5 kg

Le nouveau poids du poisson1 est : 15 kg

Le nouveau poids du poisson2 est : 0 kg

Animal::POIDS_MOYEN vaut toujours 10 quelle que soit l'instance. La constante n'est donc pas liée à l'instance mais à la classe. C'est pour cela que sa syntaxe est particulière.

12. Les attributs et méthodes statiques

a. Méthode statique

La méthode statique est liée à la classe et non à l'objet. Dans l'exemple de la classe Animal, une méthode statique est liée à l'Animal et non aux chiens, aux chats ou aux poissons.

Pour rendre une méthode statique, il faut ajouter le mot-clé `static` devant `function`.

Par exemple, modifier la méthode `se_deplacer()` pour la rendre statique et afficher "L'animal se déplace.".

La classe Animal.class.php devient :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;


        //constantes de classe
        const POIDS_LEGER = 5;
        const POIDS_MOYEN = 10;
        const POIDS_LOURD = 15;


        public function __construct ($couleur, $poids)
// Constructeur demandant 2 paramètres.
        {
```

```
   echo 'Appel du constructeur.<br />';
    $this->couleur = $couleur; // Initialisation de la couleur.
    $this->poids = $poids; // Initialisation du poids.
}


//accesseurs
public function getCouleur()
{
    return $this->couleur; //retourne la couleur
}
public function setCouleur($couleur)
{
    $this->couleur = $couleur; //écrit dans l'attribut couleur
}

public function getPoids()
{
    return $this->poids; //retourne le poids
}
public function setPoids($poids)
{
    $this->poids = $poids; //écrit dans l'attribut poids
}


//méthodes
public function manger_animal(Animal $animal_mangé)
{

    //l'animal mangeant augmente son poids d'autant que

    //celui de l'animal mangé
    $this->poids = $this->poids + $animal_mangé->poids;
    //le poids de l'animal mangé et sa couleur sont remis à 0
    $animal_mangé->poids = 0;
    $animal_mangé->couleur = "";
}
```

```
        public static function se_deplacer()
        {
            echo "L'animal se déplace.";
        }

        public function ajouter_un_kilo()
        {
            $this->poids = $this->poids + 1;
        }


}
?>
```

Il est impossible de mettre le mot-clé `$this` dans une méthode statique car `$this` représente l'objet et la méthode statique est liée à la classe.

Pour appeler cette méthode depuis la page utilisation.php, il faut utiliser la même syntaxe que pour les constantes (liées aussi à la classe), c'est-à-dire mettre **::** entre la classe et sa méthode statique :

```
<?php

//chargement de la classe
include('Animal.class.php');

//appel de la méthode statique
Animal::se_deplacer()

?>
```

Affiche :

L'animal se déplace.

Il est possible d'appeler une méthode statique à partir d'un objet mais la méthode statique ne peut rien changer sur cet objet :

```
<?php

//chargement de la classe
include('Animal.class.php');
```

```
//instanciation de la classe Animal avec votre constructeur
$chien1 = new Animal("gris",Animal::POIDS_MOYEN);


//appel de la méthode statique
$chien1->se_deplacer();


?>
```

Affiche :

Appel du constructeur.

L'animal se déplace.

b. Attribut statique

Un attribut statique est un attribut propre à la classe et non à l'objet comme pour les méthodes statiques. C'est le même principe que pour une constante sauf que l'attribut étant une variable, il est possible de changer sa valeur.

Un attribut statique se note en ajoutant le mot-clé `static` devant son nom.

Par exemple, pour ajouter un attribut statique représentant un compteur indiquant le nombre de fois où la classe a été instanciée :

La classe Animal.class.php devient :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;



        //constantes de classe
        const POIDS_LEGER = 5;
        const POIDS_MOYEN = 10;
        const POIDS_LOURD = 15;


        // Déclaration de la variable statique $compteur
        private static $compteur = 0;


etc.
.
```

```
.
.
?>
```

Pour changer la valeur de ce compteur, vous ne pouvez pas utiliser `$this`. En effet, `$this` représente un objet (chien, chat) et non la classe Animal. Le compteur est de type static donc lié à la classe. Pour faire appel à cet attribut dans la classe, il faut utiliser le mot-clé `self` qui représente la classe.

Pour ajouter 1 au compteur à chaque fois que vous allez instancier la classe Animal, il faut modifier le constructeur. Ensuite, il faut ajouter une méthode permettant de lire cet attribut privé grâce à une méthode de type `public static` et de nom `getCompteur()`.

La classe Animal.class.php devient :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;

        //constantes de classe
        const POIDS_LEGER = 5;
        const POIDS_MOYEN = 10;
        const POIDS_LOURD = 15;



        // Déclaration de la variable statique $compteur
        private static $compteur = 0;

        public function __construct ($couleur, $poids) // Constructeur
//demandant 2 paramètres.
        {
          echo 'Appel du constructeur.<br />';
          $this->couleur = $couleur; // Initialisation de la couleur.
          $this->poids = $poids; // Initialisation du poids.

        self::$compteur = self::$compteur + 1;
         }

        // Méthode statique retournant la valeur du compteur
```

```
        public static function getCompteur()
        {
            return self::$compteur;
        }

        ...
?>
```

La page utilisation.php :

```
<?php

//chargement de la classe
include('Animal.class.php');

//instanciation de la classe Animal
$chien1 = new Animal("roux",10);
//instanciation de la classe Animal
$chien2 = new Animal("gris",5);
//instanciation de la classe Animal
$chien3 = new Animal("noir",15);
//instanciation de la classe Animal
$chien4 = new Animal("blanc",8);

//appel de la méthode statique
echo "Nombre d'animaux instanciés : ".Animal::getCompteur();

?>
```

Affiche :

Appel du constructeur.

Appel du constructeur.

Appel du constructeur.

Appel du constructeur.

Nombre d'animaux instanciés : 4

13. Les exceptions

À l'instar de Java ou C#, il est possible de gérer les exceptions à l'aide des instructions `try`, `catch` et `finally`. Le principe est qu'en cas d'erreur dans le bloc `try`, le programme passe automatiquement dans le bloc `catch`. Depuis PHP 5.5,

l'instruction `finally` permet d'exécuter du code qu'il y ait eu une exception ou pas. En cas d'erreur, le bloc `catch` récupère un objet `Exception`.

Vous avez vu dans le chapitre précédent comment gérer les erreurs de connexion à la base de données :

```
<?php
try {
   $base = new PDO('mysql:host=127.0.0.1;dbname=_test', 'root', '');
   $base->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
   echo "Connexion ok.";
}
catch (Exception $e) {
   die('Erreur : ' . $e->getMessage());
}

finally {

   $base = null; //fermeture de la connexion

}
?>
```

Il est possible de générer ses propres exceptions en cas d'erreur fonctionnelle comme un nom trop grand, une division par 0, etc. Pour cela, l'instruction **`throw`** permet de lancer une exception.

Cet exemple vous montre comment récupérer une exception en cas de division par 0 :

```
<?php
//déclaration de la fonction inverse
function inverse($x) {
    if ($x == 0) {
      //Lancement de l'exception
        throw new Exception('Division par 0.');
    }
    return 1/$x;
}

//test de l'inverse de 3
try {
    echo inverse(3)."<br />";
} catch (Exception $e) {
```

```
    //Récupération de l'exception
    echo 'Exception : ',  $e->getMessage(), "<br />";
} finally {
    echo "Fin du premier bloc.<br />";
}

//test de l'inverse de 0
try {
    echo inverse(0)."<br />";
} catch (Exception $e) {
    //Récupération de l'exception
    echo 'Exception : ',  $e->getMessage(), "<br />";
} finally {
    echo "Fin du second bloc.<br />";
}

// On continue l'exécution
echo "Suite du programme !";
?>
```

Affiche :

0.33333333333333

Fin du premier bloc.

Exception : Division par 0.

Fin du second bloc.

Suite du programme

L'héritage

1. Introduction

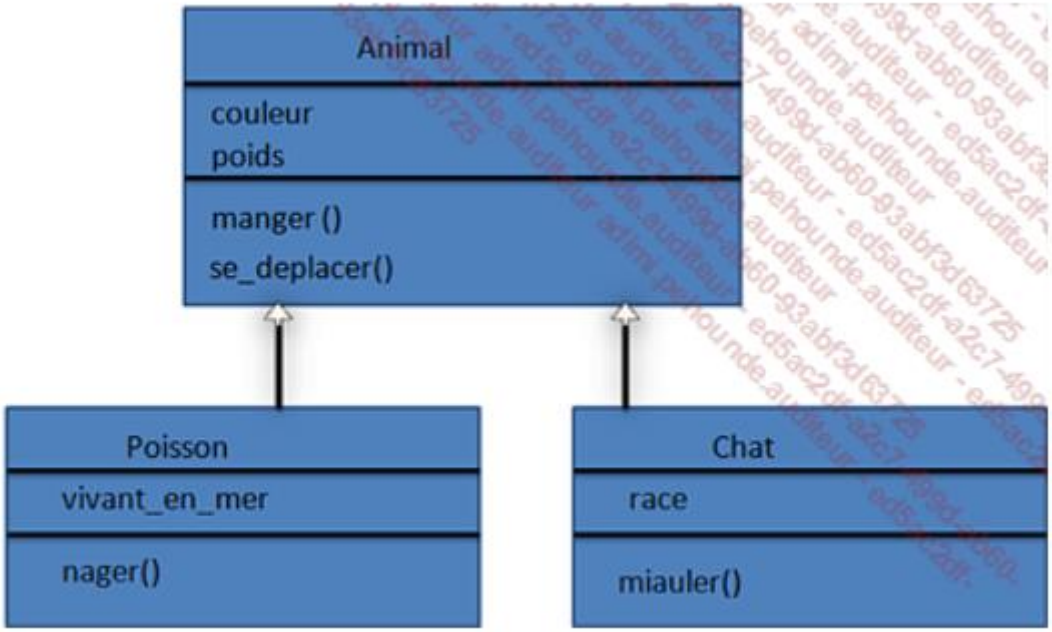
L'héritage est un concept très important en POO. Cela permet de réutiliser le code d'une classe sans le retaper.

Une classe fille hérite d'une classe mère, c'est-à-dire que la classe fille accède alors à tous les attributs et les méthodes publiques de la classe mère.

Par exemple, la classe Mammifère hérite de la classe Animal et la classe Voiture hérite de la classe Véhicule.

Si vous pouvez dire que la classe A est une sous-catégorie de la classe B, alors vous pouvez certainement faire en sorte que la classe A (Mammifère ou Voiture) hérite de la classe B (Animal ou Véhicule).

Par la suite, les classes Poisson et Chat sont prises comme exemple pour hériter de la classe Animal.



Pour créer la classe Poisson qui hérite de la classe Animal, il faut utiliser le mot-clé `extends` entre le nom de la classe fille et le nom de la classe mère.

Créer un fichier Poisson.class.php et taper le code ci-dessous :

```
<?php
class Poisson extends Animal
    {


    }
?>
```

Il faut maintenant lui ajouter un attribut privé correspondant à la variable `vivant_en_mer` puis les accesseurs `get` et `set` et enfin la méthode publique `nager()`.

```
<?php
class Poisson extends Animal
    {
        private $vivant_en_mer; //type du poisson


        //accesseurs
        public function getType()
        {
             if ($this->vivant_en_mer){
             return "vivant_en_mer";
```

```
        }
        else if ($this->vivant_en_mer===false){
             return "ne_vivant_pas_en_mer";
        }else {return "";}
        }
            public function setType($vivant_en_mer)
        {
            $this->vivant_en_mer = $vivant_en_mer;
//écrit dans l'attribut vivant_en_mer
        }
        //méthode
        public function nager()
        {
            echo "Je nage <br />";
        }
    }
?>
```

De la même façon, créer un fichier Chat.class.php :

```
<?php
class Chat extends Animal
    {
        private $race; //race du chat

        //accesseurs
        public function getRace()
        {
            return $this->race; //retourne la race
        }
        public function setRace($race)
        {
            $this->race = $race; //écrit dans l'attribut race
        }
        //méthode
        public function miauler()
        {
             echo "Miaou <br />";
        }
```

```
    }
?>
```

Les classes Chat et Poisson, héritant de la classe Animal, ont accès aux attributs publics de la classe Animal.

La page utilisation.php :

```
<?php

//chargement des classes
include('Animal.class.php');
include('Poisson.class.php');
include('Chat.class.php');

//instanciation de la classe Poisson qui appelle le constructeur de
//la classe Animal
$poisson = new Poisson("gris",8);
//instanciation de la classe Chat qui appelle le constructeur de la
//classe Animal
$chat = new Chat("blanc",4);
//lire le poids par l'accesseur de la classe mère
echo "Le poids du poisson est : ".$poisson->getPoids()." kg<br />";
//lire le poids par l'accesseur de la classe mère
echo "Le poids du chat est : ".$chat->getPoids()." kg<br />";

$poisson->setType(true);
//lire le type par l'accesseur de sa propre classe
echo "Le type du poisson est : ".$poisson->getType()."<br />";
//appel de la méthode de la classe Poisson
$poisson->nager();

$chat->setRace("Angora");
//lire la race par l'accesseur de sa propre classe
echo "La race du chat est : ".$chat->getRace()."<br />";
//appel de la méthode de la classe chat
$chat->miauler();

//appel de la méthode statique
echo "Nombre d'animaux instanciés : ".Animal::getCompteur();
```

```
?>
```

Affiche :

Appel du constructeur.

Appel du constructeur.

Le poids du poisson est : 8 kg

Le poids du chat est : 4 kg

Le type du poisson est : vivant en mer

Je nage

La race du chat est : Angora

Miaou

Nombre d'animaux instanciés : 2

La classe Poisson n'a pas accès aux attributs de la classe Chat et vice versa car l'une n'hérite pas de l'autre. Les classes Poisson et Chat n'ont pas accès directement aux attributs privés couleur et poids de la classe Animal, elles doivent passer par ses accesseurs publics.

2. Protected

Ce type de visibilité est équivalent à `private` sauf que les classes filles peuvent voir les attributs `protected` de la classe mère.

Par exemple, ajouter l'attribut `$age` de visibilité protégée (`protected`) dans la classe mère Animal :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;
        protected $age = 0;
etc.
```

Cet attribut n'a pas d'accesseur public donc aucune autre classe que les classes filles héritant d'Animal et la classe Animal ne peuvent modifier ou lire celui-ci.

Ajouter la fonction `afficherAttributs()` dans la classe Poisson :

```
public function afficherAttributs()
        {
           echo "Type : ".$this->vivant_en_mer; // ok car privé à
                                                 // cette classe
```

```
          echo "<br />";
          echo "Age : ".$this->age; // ok, car l'attribut est
                                    // protégé dans la classe mère.
          echo "<br />";
          echo "Poids : ".$this->poids; // erreur car l'attribut est
//privé dans la classe mère, l'accès est interdit. Il faut passer
//par ses accesseurs publics pour modifier ou lire sa valeur
           echo "<br />";
        }
```

Puis pour tester cette méthode, la page utilisation.php devient :

```
<?php

//chargement des classes
include('Animal.class.php');
include('Poisson.class.php');

//instanciation de la classe Poisson qui appelle le constructeur de
//la classe Animal
$poisson = new Poisson("gris",8);
//lire le poids par l'accesseur de la classe mère
echo "Le poids du poisson est : ".$poisson->getPoids()." kg<br />";

//mise à jour du type du poisson
$poisson->setType(true);
//appel de la méthode affichant les attributs de la classe Poisson
//et Animal
$poisson->afficherAttributs();

?>
```

Affiche :


Appel du constructeur.
Le poids du poisson est:8 kg
Type:1
Age:0

Notice: Undefined property: Poisson::$poids in **C:\Program Files\EasyPHP-DevServer-13.1VC11\data\localweb\Objet\Poisson.class.php** on line **35**
Poids:

En effet, la classe Poisson n'a pas accès à l'attribut poids de la classe Animal car il est privé.

En conclusion, il est généralement conseillé de mettre les attributs en visibilité `protected` car la classe elle-même, les classes filles et celles qui en héritent, ont accès à cet attribut.

3. Substitution

La substitution sert à modifier une méthode déjà existante dans une classe mère afin d'en modifier le comportement. La méthode existe donc dans deux classes différentes et c'est celle de la classe fille ou de la classe mère qui est exécutée suivant le contexte.

Par exemple, pour substituer la méthode `manger_animal(Animal $animal_mangé)` de la classe Animal afin d'initialiser le type du poisson mangé, il faut ajouter cette même méthode dans la classe Poisson et l'implémenter autrement :

Ajouter dans la classe Poisson.class.php :

```
//méthode substituée
        public function manger_animal(Animal $animal_mangé)
        {
            if(method_exists($animal_mangé, "setRace")) {
         $animal_mangé->setRace("");
        }
            if (isset($animal_mangé->vivant_en_mer)){
                $animal_mangé->vivant_en_mer=""
        }
        }
```

Le problème est que cette méthode initialise correctement l'attribut vivant_en_mer du poisson mangé mais n'initialise plus son poids et sa couleur. Vous ne pouvez pas changer son poids et sa couleur ici car ces attributs sont privés dans la classe Animal. De plus, la classe Poisson n'a pas accès à l'attribut race, il faut donc utiliser l'accesseur `setRace` pour mettre à jour la race et vérifier son existence avec la fonction `method_exists`.

La solution est d'appeler la méthode `manger_animal(Animal $animal_mangé)` de la classe Animal dans la méthode `manger_animal(Animal $animal_mangé)` de la classe Poisson :

```
//méthode substituée
      public function manger_animal(Animal $animal_mangé)
      {
          // Appelle la méthode manger_animal() de la classe parente,
          // c'est-à-dire Animal
          parent::manger_animal($animal_mangé);

          if(method_exists($animal_mangé, "setRace")) {
              $animal_mangé->setRace("");
      }
```

```
          if (isset($animal_mangé->vivant_en_mer)){
              $animal_mangé->vivant_en_mer="")
          }
      }
```

`parent` est un mot-clé désignant la classe mère, c'est-à-dire la classe Animal.

La page utilisation.php :

```
<?php

//chargement des classes
include('Animal.class.php');
include('Poisson.class.php');

//instanciation de la classe Poisson qui appelle le constructeur de
//la classe Animal
$poisson = new Poisson("gris",8);
//mise à jour du type du poisson
$poisson->setType(true);

//instanciation de la classe Poisson qui appelle le constructeur de
//la classe Animal
$autre_poisson = new Poisson("noir",5);
//mise à jour du type du poisson
$autre_poisson->setType(false);

//appel de la méthode affichant les attributs de la classe Poisson
//et Animal
$poisson->manger_animal($autre_poisson);
//lire le type par l'accesseur de sa propre classe
echo "Le type du poisson mangé est : ".$autre_poisson->getType()."<br />";
//lire le poids par l'accesseur de la classe mère
echo "Le poids du poisson mangé est : ".$autre_poisson->getPoids()."
kg<br />";

?>
```

Affiche :

Appel du constructeur.

Appel du constructeur.

Le type du poisson mangé est :

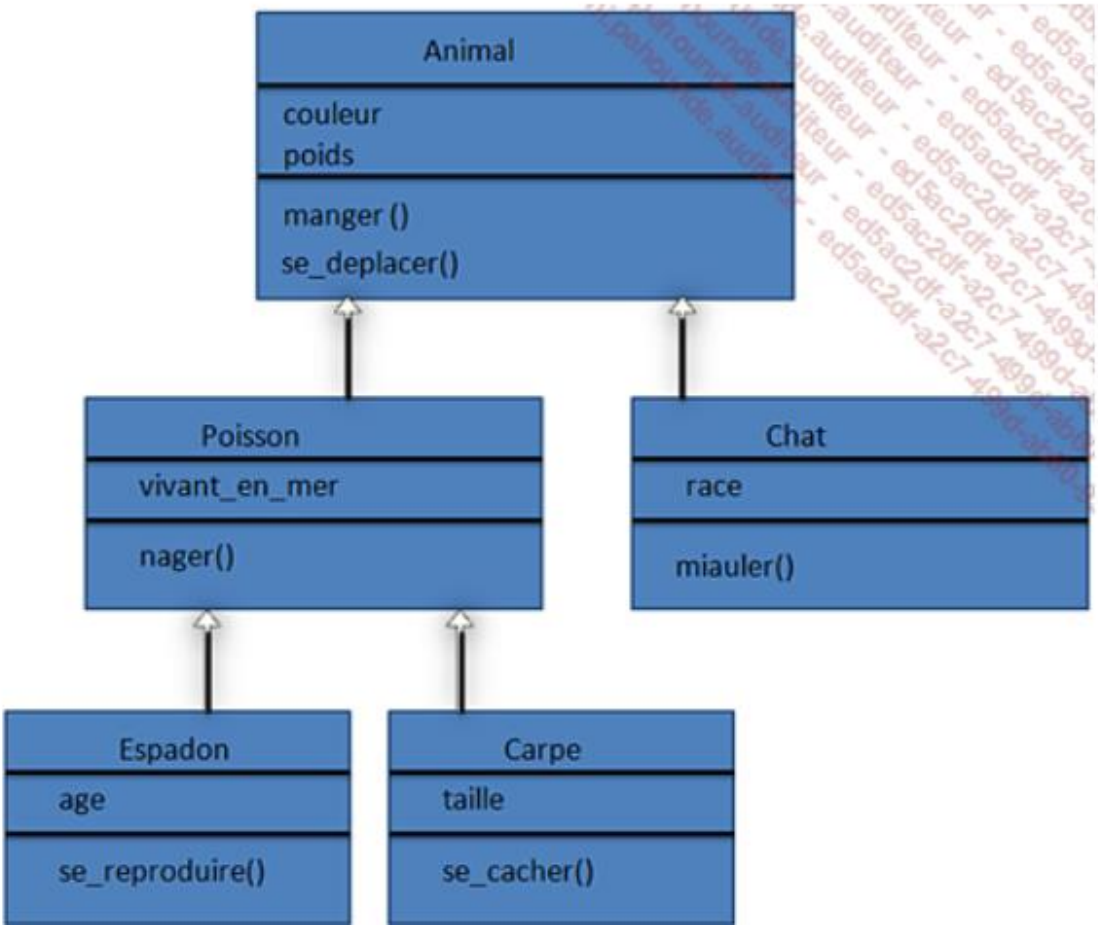
Le poids du poisson mangé est : 0 kg

Le poids a été initialisé à 0 par la méthode `manger_animal(Animal $animal_mangé)` de la classe Animal.

Dans la classe Poisson la méthode *`manger_animal(Animal $animal_mangé)`* peut changer l'attribut type sur un objet de type Animal car Poisson hérite d'Animal et dans le fichier utilisation.php, c'est un Poisson qui est passé en paramètre et non un Animal. Il s'agit du polymorphisme d'héritage.

4. Héritage en cascade

En PHP, l'héritage multiple n'existe pas, une classe ne peut hériter que d'une et une seule classe qui elle-même peut hériter d'une classe, etc.



Cet exemple montre que les classes Espadon et Carpe héritent de la classe Poisson qui hérite de la classe Animal.

La classe Espadon accède à :

- tous les attributs et méthodes privés, protégés et publics d'elle-même.

- tous les attributs et méthodes protégés et publics de la classe Poisson.
- tous les attributs et méthodes protégés et publics de la classe Animal.

Les classes abstraites

Les classes abstraites s'écrivent en ajoutant le mot-clé `abstract` devant le mot-clé `class`. Une classe abstraite ne peut pas être instanciée, c'est-à-dire qu'il n'est pas possible de créer une instance. Vous pouvez écrire des méthodes abstraites. Ce sont des méthodes dont uniquement la signature est écrite, précédée du mot-clé `abstract` : abstract visibilité function nomMéthode(attribut type_attribut...). Vous pouvez aussi implémenter des méthodes normales dans les classes abstraites. Ces classes servent donc uniquement à obliger les classes héritant de la classe abstraite à redéfinir les méthodes déclarées abstraites dans la classe abstraite et à factoriser les méthodes implémentées.

Dans l'exemple suivant, la classe Animal est abstraite car vous ne voulez pas créer (instancier) des animaux mais uniquement des poissons ou des chats.

Ajouter aussi une méthode abstraite `respire()` dans la classe Animal :

```
<?php
abstract class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        private $poids = 10;

        //constantes de classe
        const POIDS_LEGER = 5;
        const POIDS_MOYEN = 10;
        const POIDS_LOURD = 15;



        // Déclaration de la variable statique $compteur
        private static $compteur = 0;



        public function __construct($couleur, $poids) // Constructeur
//demandant 2 paramètres.
        {
           echo 'Appel du constructeur.<br />';
            $this->couleur = $couleur; // Initialisation de la couleur.
           $this->poids = $poids; // Initialisation du poids.

            self::$compteur = self::$compteur + 1;
```

```
}


//accesseurs
public function getCouleur()
{
    return $this->couleur; //retourne la couleur
}
public function setCouleur($couleur)
{
    $this->couleur = $couleur; //écrit dans l'attribut couleur
}
public function getPoids()
{
   return $this->poids; //retourne le poids
}
public function setPoids($poids)
{
    $this->poids = $poids; //écrit dans l'attribut poids
}


//méthodes publiques

public function manger_animal(Animal $animal_mangé)
{
    //l'animal mangeant augmente son poids d'autant que
    //celui de l'animal mangé
    $this->poids = $this->poids + $animal_mangé->poids;
    //le poids de l'animal mangé et sa couleur sont remis à 0
    $animal_mangé->poids = 0;
    $animal_mangé->couleur = "";
}

public static function se_deplacer()
{
    echo "L'animal se déplace.";
}
```

```
        public function ajouter_un_kilo()
        {
            $this->poids = $this->poids + 1;
        }

        // Méthode statique retournant la valeur du compteur
        public static function getCompteur()
        {
            return self::$compteur;
        }

        //code non implémenté car méthode abstraite
        abstract public function respire();

}
?>
```

Vous constatez que la méthode abstraite `respire()` n'a pas de corps, c'est-à-dire qu'il n'y a pas d'accolades avec l'implémentation de la méthode.

Comme les classes Poisson et Chat héritent de la classe Animal, vous êtes obligé de redéfinir la méthode `respire()` dans les classes Poisson et Chat.

Il faut ajouter dans la classe Poisson :

```
        public function respire()
        {
            echo "Le poisson respire.<br />";
        }
```

et dans la classe Chat :

```
        public function respire()
        {
            echo "Le chat respire.<br />";
        }
```

La page utilisation.php devient :

```
<?php

//chargement des classes
include('Animal.class.php');
```

```
include('Poisson.class.php');
include('Chat.class.php');

//instanciation de la classe Poisson qui appelle le constructeur de
//la classe Animal
$poisson = new Poisson("gris",8);
//instanciation de la classe Chat qui appelle le constructeur de
//la classe Animal
$chat = new Chat("blanc",4);
//appel de la méthode respire()
$poisson->respire();
$chat->respire();

?>
```

Affiche :

Appel du constructeur.

Appel du constructeur.

Le poisson respire.

Le chat respire.

Les interfaces

Les interfaces sont semblables aux classes abstraites. Elles ne peuvent pas être instanciées, mais vous ne pouvez pas implémenter de méthode dans les classes abstraites. Ces classes servent donc uniquement à obliger les classes héritant de l'interface à redéfinir les méthodes déclarées dans l'interface. Ces méthodes sont forcément publiques. Une interface se déclare avec le mot-clé `interface` :

Voici le code de l'interface action :

```
<?php
interface action {
  function courir();

  function manger();
}
?>
```

Pour signaler qu'une classe implémente une ou plusieurs interfaces, vous devez ajouter le mot-clé `implements` suivi du nom des interfaces.

Voici un exemple signalant que la classe Chat implemente l'interface action :

```
<?php
class Chat extends Animal implements action
   {
   ...

   //méthodes de l'interface
   function courir() {
      echo "Le chat court.<br />";
   }

   function manger() {
      echo "Le chat mange.<br />";
   }
  }
?>
```

La page utilisation.php devient :

```
<?php

//chargement des classes
include('Animal.class.php');
include('action.class.php');
include('Chat.class.php');

//instanciation de la classe Chat qui appelle le constructeur
de la classe Animal
$chat = new Chat("blanc",4);
//appelle des méthodes de l'interface action
echo $chat->courir();
echo $chat->manger();

?>
```

Affiche :

Appel du constructeur.

Le chat court.

Le chat mange.

Il est possible d'implémenter plusieurs interfaces en les séparant par une virgule.

Les classes finales

Lorsqu'une classe est finale, vous ne pouvez pas créer de classe fille héritant de cette classe. Cela n'a guère d'intérêt en pratique.

Il faut pour cela ajouter le mot-clé `final` devant le mot-clé `class`.

Par exemple, si vous ne créez pas de classe héritant de la classe Poisson, vous pouvez la mettre `final` :

```
<?php
final class Poisson extends Animal
    {
        private $vivant_en_mer; //type du poisson


        //accesseurs
        ...
        //méthode
        public function nager()
        {
            echo "Je nage <br />";
        }


        public function respire()
        {
            echo "Le poisson respire.<br />";
        }


    }
?>
```

Il est possible aussi de déclarer des méthodes finales. Ces méthodes ne pourront pas alors être substituées.

Vous avez vu dans le paragraphe sur la substitution un exemple où la méthode `manger_animal(Animal $animal_mangé)` a été substituée dans la classe Poisson. Si cette méthode était finale dans la classe Animal :

```
final public function manger_animal(Animal $animal_mangé)
        {
          //l'animal mangeant augmente son poids d'autant que
          //celui de l'animal mangé
```

```
            $this->poids = $this->poids + $animal_mangé->poids;
            //le poids de l'animal mangé et sa couleur sont remis à 0
             $animal_mangé->poids = 0;
             $animal_mangé->couleur = "";
        }
```

Il est alors impossible de substituer cette méthode dans la classe Poisson.

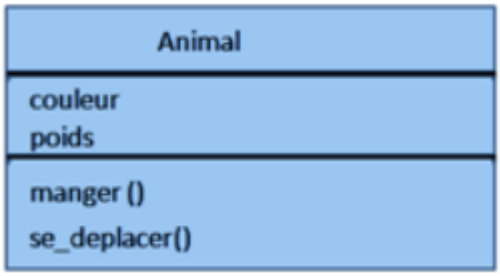
Les méthodes magiques

Une méthode magique est une méthode qui est appelée automatiquement lorsqu'un évènement se produit.

Par exemple, `__construct` est une méthode magique. Elle s'exécute automatiquement lorsque vous instanciez la classe contenant `__construct`.

Les méthodes magiques `__get` et `__set` permettent de lire ou de modifier des attributs qui n'existent pas ou dont l'accès est interdit.

Reprenez l'exemple au départ du chapitre avec la classe Animal :



Cette fois, l'attribut couleur est privé mais l'attribut poids est public :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        public $poids = 10;

        public function __construct($couleur, $poids) // Constructeur
//demandant 2 paramètres.
        {
          echo 'Appel du constructeur.<br />';
          $this->couleur = $couleur; // Initialisation de la couleur.
          $this->poids = $poids; // Initialisation du poids.
```

```
        }


        //méthodes publiques
        public function manger ()
        {

        }

        public function se_deplacer ()
        {

        }
}
?>
```

Lorsque vous créez une instance de la classe Animal, vous pouvez accéder à l'attribut poids car il est public mais pas à l'attribut couleur car il est privé.

La page utilisation.php :

```
<?php

//chargement des classes
include('Animal.class.php');

//instanciation de la classe Animal
$chien = new Animal("gris",8);

$chien->couleur = "noir";
echo $chien->couleur."<br />";
?>
```

Affiche une erreur car vous essayez d'accéder directement à l'attribut couleur qui est privé :

Appel du constructeur.

Fatal error: Cannot access private property Animal::$couleur in C:\Program Files\EasyPHP-DevServer-15.9VC11\data\localweb\Objet\utilisation.php on line 9

Maintenant, ajoutez les méthodes magiques `__get` et `__set` dans la classe Animal :

```php
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        public $poids = 10;
        private $tab_attributs = array();


        public function __construct($couleur, $poids) // Constructeur
//demandant 2 paramètres.
        {
          echo 'Appel du constructeur.<br />';
          $this->couleur = $couleur; // Initialisation de la couleur.
          $this->poids = $poids; // Initialisation du poids.

        }

        //méthodes magiques
        public function __get($nom)
        {
            echo "__get <br />";
         if (isset ($this->tab_attributs[$nom]))
            return $this->tab_attributs[$nom];
        }

        public function __set($nom, $valeur)
        {
            echo "__set <br />";
            $this->tab_attributs[$nom] = $valeur;
        }

        public function __isset($nom)
        {
            return isset ($this->tab_attributs[$nom]);
        }


        //méthodes publiques
```

```
        public function manger ()
        {
            ...
        }

           public function se_deplacer ()
        {
            ...
        }

}
?>
```

Ces méthodes se déclenchent automatiquement si l'attribut est privé ou n'existe pas. Elles stockent alors la valeur dans le tableau `$tab_attributs` avec pour index le nom de l'attribut. La méthode `__isset($attribut)` permet de savoir si l'attribut existe.

La page utilisation.php :

```
<?php

//chargement des classes
include('Animal.class.php');

//instanciation de la classe Animal
$chien = new Animal("gris",8);
if (isset($chien->couleur)) {
    echo "L'attribut couleur existe.<br />";
}
else {
    echo "L'attribut couleur n'existe pas.<br />";
}

$chien->couleur = "noir";
echo $chien->couleur."<br />";

if (isset($chien->poids)) {
    echo "L'attribut poids existe.<br />";
}
else {
    echo "L'attribut poids n'existe pas.<br />";
```

```
}
$chien->poids = 25;
echo $chien->poids."<br />";


?>
```

Affiche :

Appel du constructeur.

L'attribut couleur n'existe pas.

__set

__get

Noir

L'attribut poids existe.

25

Explication :

`$chien = new Animal("gris",8);` affiche : Appel du constructeur

`isset($chien->couleur)` retourne faux car l'attribut couleur étant privé, il est impossible d'y accéder donc affichage de : L'attribut couleur n'existe pas.

`$chien->couleur = "noir";` affiche : `__set`. En effet, l'attribut couleur est privé donc non accessible donc `__set` est automatiquement appelé puis la valeur est stockée dans le tableau `$tab_attributs`.

`echo $chien->couleur."<br />";` affiche : `__get` puis noir. En effet, l'attribut couleur est toujours privé donc `__get` est automatiquement appelé pour afficher la couleur.

`isset($chien->poids)` retourne vrai car l'attribut poids étant public, il est possible d'y accéder donc affichage de : l'attribut poids existe.

`$chien->poids = 25;` n'affiche rien car la fonction `__set` n'est pas appelée. En effet, l'attribut poids étant public, il est possible d'accéder directement à cet attribut.

`echo $chien->poids."<br />";` affiche 25. En effet, l'attribut poids est public donc il est possible d'accéder directement à cet attribut.

Pour supprimer un attribut ajouté par la méthode magique `__set`, il faut implémenter la méthode magique `__unset($attribut)` ce qui supprimera l'attribut du tableau `$tab_attributs`.

Ajoutez dans la classe Animal cette méthode :

```
public function __unset($nom)
        {
            if (isset($this->tab_attributs[$nom]))
```

```
                unset($this->tab_attributs[$nom]);
        }
```

Pour finir, les méthodes magiques `__call` et `__callStatic` permettent d'appeler des méthodes privées ou qui n'existent pas. La fonction `method exist()` vérifie si une méthode existe dans un objet. Elle prend en paramètre l'objet puis le nom de la méthode. Elle retourne `true` si la méthode existe et `false` sinon.

La classe Animal avec une méthode publique `manger()` et une méthode privée `se_deplacer()` devient :

```
<?php
class Animal
    {
        // Déclaration des attributs
        private $couleur = "gris";
        public $poids = 10;
        private $tab_attributs = array();



        public function __construct($couleur, $poids) // Constructeur
//demandant 2 paramètres.
        {
           echo 'Appel du constructeur.<br />';
           $this->couleur = $couleur; // Initialisation de la couleur.
           $this->poids = $poids; // Initialisation du poids.
        }



        //méthodes magiques
        public function __get($nom)
        {
            echo "__get <br />";
            if (isset ($this->tab_attributs[$nom]))
                return $this->tab_attributs[$nom];
        }


        public function __set($nom, $valeur)
        {
            echo "__set <br />";
            $this->tab_attributs[$nom] = $valeur;
        }
```

```
        public function __isset($nom)
        {
            return isset ($this->tab_attributs[$nom]);
        }

        public function __call($nom, $arguments)
        {
            echo "La méthode ".$nom." n'est pas accessible. Ses arguments
étaient les suivants : ".implode($arguments, ', ')."<br />";
            if(method_exists($this, $nom))
            {
                $this->$nom(implode($arguments, ', '));
            }
        }

        public static function __callStatic($nom, $arguments)
        {
             echo "La méthode static ".$nom." n'est pas accessible. Ses
arguments étaient les suivants : ".implode($arguments, ', ')."<br />";
            if(method_exists(__CLASS__, $nom))
            {
                echo __CLASS__.'::'.$nom.'<br />';
                self::$nom(implode($arguments, ', '));
            }
        }

        //méthode publique
        public function manger()
        {
            echo "Méthode publique manger() <br />";
        }

        //méthode privée
        private function se_deplacer($lieu)
        {
            echo "Méthode privée se_deplacer() <br />";
        }
}
```

```
?>
```

La page utilisation.php :

```
<?php


//chargement des classes
include('Animal.class.php');


//instanciation de la classe Animal
$chien = new Animal("gris",8);
$chien->manger();
$chien->se_deplacer("Paris");


?>
```

Affiche :

Appel du constructeur.

Méthode publique manger()

La méthode se_deplacer n'est pas accessible.
Ses arguments étaient les suivants : Paris

Méthode privée se_deplacer()

Le code `$chien->manger();` appelle effectivement la méthode publique `manger()`. Mais la méthode `se_deplacer($lieu)` étant privée, il est impossible d'y accéder directement. La méthode magique `__call` est alors appelée, elle vérifie que la méthode `se_deplacer($lieu)` existe et comme c'est le cas, la méthode `se_deplacer($lieu)` est appelée.

Si la méthode `se_deplacer($lieu)` est statique, l'appel dans la page utilisation.php est :

```
Animal::se_deplacer("Paris");
```

C'est alors la méthode magique `__callStatic` qui est appelée.

Il existe encore d'autres méthodes magiques mais elles ne sont pas expliquées dans ce support. Vous avez plus d'informations sur ces méthodes à cette adresse : http://php.net/manual/fr/language.oop5.magic.php

Les traits

Cette fonctionnalité, arrivée avec PHP 5.4, permet de réutiliser du code dans deux classes indépendantes. Sa syntaxe est :

```
trait nom_du_trait {
   //définition des propriétés et méthodes.
}
```

Prenez l'exemple de deux classes indépendantes **Facture** et **Indemnite** ayant une méthode **Calcul_taux_tva**.

```
<?php
//déclaration du trait
trait MonTrait
{
  public function Calcul_ttc($montant)
  {
    return $montant*1.2; //retourne le montant TTC
  }
}

class Facture
{
  use MonTrait;
}

class Indemnite
{
  use MonTrait;
}

$facture = new Facture;
//affichage du montant TTC de la facture
echo $facture->Calcul_ttc(10)."<br />";

$indemnite = new Indemnite;
//affichage du montant TTC de l'indemnité
echo $indemnite->Calcul_ttc(20);
?>
```

Affiche :

Cela permet aux deux classes **Facture** et **Indemnite** d'utiliser la même méthode **calcul_ttc** sans recourir à l'héritage.

Uniform Variable Syntax

Cette syntaxe de variable uniforme vous permet de résoudre certaines incohérences sur l'évaluation des expressions. Depuis PHP 5.6, il est possible d'accéder à une propriété avec cette syntaxe :

```
$obj->{$properties['name']};
```

Par exemple :

```
<?php
//Déclaration de la classe Objet
class Objet { public $couleur ="rouge"; }
//Instanciation de la classe Objet
$obj = new Objet();
$properties['name'] = "couleur";
echo $obj->{$properties['name']};
?>
```

Affiche :

rouge

En effet, $obj->{$properties['name']} revient à écrire $obj->couleur.

Depuis PHP 7, il est possible d'attacher des appels statiques.

Par exemple :

```
<?php
//Déclaration de la classe 1
class classe1 { static $nom1 = 'classe2'; }
//Déclaration de la classe 2
class classe2 { static $nom2 = 'Bonjour'; }
//Déclaration de la méthode permettant d'afficher Bonjour
classe2::$nom2 = function () { echo "Bonjour !"; };
$classe1 = 'classe1';
//Appel statique
($classe1::$nom1::$nom2)();
?>
```

Affiche :

Bonjour !

Les espaces de noms

Lorsque vous travaillez sur des projets de taille importante en équipe, il est utile de modulariser les classes et les fonctions. Ainsi, chaque développeur peut travailler sur son propre module. Depuis PHP 5.3, les namespaces (espaces de noms) permettent cette modularisation. Un namespace est une sorte de dossier virtuel dans lequel vous stockez vos objets. Il est ainsi possible de mettre des classes ou des fonctions de même nom dans des namespaces différents.

Un namespace est déclaré avec le mot-clé `namespace` suivi de son nom en début de fichier.

Par exemple :

Espace_nom.php

```
<?php
// Définition de l'espace de noms.
namespace Bibliotheque;
// Définition d'une constante.
const PI = 3.1416;
// Définition d'une fonction.
function maFonction() {
echo "Bonjour";
}
// Définition d'une classe.
class maClasse {
/*
...
*/
}

?>
```

Utilisation_espace_nom.php

```
<?php
include('espace_nom.php');
Bibliotheque\maFonction(); //Appel du namespace Bibliotheque
à la racine
?>
```

Affiche :

La constante __NAMESPACE__ retourne le nom de l'espace de noms courant.

Il est possible de créer des sous-espaces de noms en écrivant :

```
namespace Espace1\SousEspace1;
```

Les chemins pour trouver une fonction, une classe ou une constante dans un espace de noms sont relatifs si vous commencez par le namespace ou absolus si vous commencez par \.

Par exemple :

Espace_nom.php

```
<?php
// Définition de l'espace de noms.
namespace Bibliotheque;

// Définition d'une constante.
const PI = 3.1416;

// Définition d'une fonction.
function maFonction() {
        echo "Bonjour <br />";
}

// Définition d'une classe.
class Animal
    {
       // Déclaration des attributs
       private $couleur = "gris";
        //accesseurs
        public function getCouleur()
        {
            return $this->couleur; //retourne la couleur
        }
        public function setCouleur($couleur)
        {
            $this->couleur = $couleur; //écrit dans l'attribut
couleur
        }
        }
```

```
?>
```

Utilisation_espace_nom.php

```
<?php
namespace Projet;
include('espace_nom.php');
// Affichage de l'espace de noms courant.
echo 'Espace de noms courant = ', __NAMESPACE__,'<br />';
\Bibliotheque\maFonction(); //Appel du namespace Bibliotheque à la racine
echo \Bibliotheque\PI."<br />";
$chien = new \Bibliotheque\Animal();
$chien->setCouleur("noir");
echo "La couleur du chien est : ".$chien->getCouleur();


?>
```

Affiche :

Espace de noms courant = Projet

Bonjour

3.1416

La couleur du chien est : noir

Enfin, vous pouvez créer un alias sur un espace de noms ou sur un objet contenu dans cet espace de noms. Il suffit d'utiliser l'opérateur `use [namespace] as nouveau_nom`. Par exemple : `use \Bibliotheque as biblio;`

Avec les alias, la page Utilisation_espace_nom.php devient :

```
<?php
namespace Projet;
include('espace_nom.php');
// Affichage de l'espace de noms courant.
echo 'Espace de noms courant = ', __NAMESPACE__,'<br />';
\Bibliotheque\maFonction(); //Appel du namespace Bibliotheque à la racine
use \Bibliotheque as biblio; // alias d'un namespace
echo biblio\PI."<br />";
use \Bibliotheque\Animal as ani; // alias d'une classe
$chien = new ani(); //Appel de l'alias de la classe
                    //Animal$chien->setCouleur("noir");
echo "La couleur du chien est : ".$chien->getCouleur();
```

```
?>
```

Affiche :

Espace de noms courant = Projet

Bonjour

3.1416

La couleur du chien est : noir

Depuis PHP 5.6, il est aussi possible d'importer des fonctions et des constantes à l'aide de l'opérateur `use`.

Par exemple :

```
<?php
namespace Projet {
   //Déclaration d'une constante
    const PI = 3.14;
   //Déclaration d'une fonction
    function affiche() { echo __FUNCTION__."<br />"; }
}

namespace {
    use const Projet\PI;
    use function Projet\affiche;
   //Affichage de la constante
    echo PI."<br />";
   //appel de la fonction
    affiche();
}

?>
```

Affiche :

3.14

Projet\affiche

La constante `__FUNCTION__` retourne le nom de la fonction courante.