Chapitre 2 :

Programmation Orientée Objet en PHP

1. Les classes et objets

La programmation par objet (POO) a été intégrée au langage PHP dans sa version 4. Puis la versio5, ou PHP est devenu comme les autres langages objets comme Java ou C++.

Ce premier cours sur la programmation orientée objet sera une première découverte et une présentation de la syntaxe d'écriture de classes.

Nous aborderons dans un premier temps les avantages relatifs à une approche par objet. Puis nous définirons les notions de classes, d'objets, d'instance et de type avant de nous lancer pleinement dans l'écriture de nos premières classes.

- Avantages d'une approche objet

La programmation orientée objet offre de nombreux avantages. Parmi eux :

- La possibilité de réutiliser le code dans différents projets.
- Le programmeur identifie chaque élément de son programme comme un objet ayant son contexte, ses propriétés et des actions qui lui sont propres.
- Un code modulaire.
- Chaque type d'objet possède son propre contexte et ne peut agir avec d'autres suivant des interfaces bien précises. Cela permet d'isoler chaque module et d'en créer séparément de nouveaux qui viendront s'ajouter à l'application.
- Cette approche est particulièrement employée dans le cas de projets répartis entre puiseurs développeurs.
- Possibilité de s'adapter aux *design patterns (motifs de conception)* pour une meilleure structuration du code.

2. Objet

Définition

Un « objet » est une représentation d'une chose matérielle ou immatérielle du réel à laquelle on associe des propriétés et des actions.

Par exemple : une voiture, une personne, un animal, un nombre ou bien un compte bancaire peuvent être vus comme des objets.

Attributs

Les « attributs » (aussi appelés « données membres ») sont les caractères propres à un objet.

Une personne, par exemple, possèdent différents attributs qui lui sont propres comme le nom, le prénom, la couleur des yeux, le sexe, la couleur des cheveux, la taille...

Méthodes

Les « méthodes » sont les actions applicables à un objet.

Un objet personne, par exemple, dispose des actions suivantes : manger, dormir, boire, marcher, courir...

3. Classe

Une « classe » est un modèle de données définissant la structure commune à tous les objets qui seront créés à partir d'elle. Plus concrètement, nous pouvons percevoir une classe comme un moule grâce auquel nous allons créer autant d'objets de même ***type*** et de même structure qu'on le désire.

Par exemple, pour modéliser n'importe quelle personne, nous pourrions écrire une classe Personne dans laquelle nous définissons les attributs (couleurs des yeux, couleurs des cheveux, taille, sexe...) et méthodes (marcher, courir, manger, boire...) communs à tout être humain.

Instance

Une instance est une représentation particulière d'une classe.

Lorsque l'on crée un objet, on réalise ce que l'on appelle une « instance de la classe ». C'est à dire que du moule, on en extrait un nouvel objet qui dispose de ses attributs et de ses méthodes. L'objet ainsi créé aura pour type le nom de la classe.

Par exemple, les objets Hugo, Romain, Nicolas, Daniel sont des instances (objets) de la classe Personne.

Déclaration d'une classe

Nous allons déclarer une classe *Personne* qui nous permettra ensuite de créer autant d'instances (objets) de cette classe que nous le souhaitons.

Syntaxe de déclaration d'une classe

Le code suivant présente une manière de déclarer et de structurer correctement une classe.

Déclaration d'une classe PHP 5

```
<?php

 class NomDeMaClasse
 {
  // Attributs

  // Constantes

  // Méthodes
 }

?>
```

Voici par exemple ce que cela donne avec notre exemple :

Exemple de la classe Personne

```
<?php

 class Personne
 {
  // Attributs
  public $nom;
  public $prenom;
  public $dateDeNaissance;
  public $taille;
  public $sexe;

  // Constantes

  const NOMBRE_DE_YEUX = 2;
  // Méthodes
  public function __construct() { }

  public function boire()
  {
   echo 'La personne boit<br/>';
  }

  public function manger()
  {
   echo 'La personne mange<br/>';
  }
 }
```

Les attributs

Comme nous l'avons expliqué au début de ce cours, les attributs sont les caractéristiques propres d'un objet. Toute personne possède un nom, un prénom, une date de naissance, une taille, un sexe.

Par cet exemple, nous déclarons les attributs de notre classe *public.* Nous expliquerons dans un tutoriel suivant les trois niveaux de visibilité (*public, private et protected)* qui peuvent être appliqués à un attribut.

Les constantes

Il est aussi possible de déclarer des constantes propres à la classe. Contrairement au mode procédural de programmation, une constante est déclarée avec le mot-clé *const.*

Remarque : une constante doit être déclarée et initialisée avec sa valeur en même temps.

Le constructeur

Le constructeur est une méthode particulière. C'est elle qui est appelée implicitement à la création de l'objet (instanciation).

Dans notre exemple, le constructeur n'a ni paramètre ni instruction. Le programmeur est libre de définir des paramètres obligatoires à passer au constructeur ainsi qu'un groupe d'instructions à exécuter à l'instanciation de la classe. Nous nous en passerons pour simplifier notre exemple.

Remarque : en PHP, la surcharge de constructeur et de méthodes n'est pas possible. On ne peut définir qu'une seule et unique fois la même méthode.

Les méthodes

Les méthodes sont les actions que l'on peut appliquer à un objet. Il s'agit en fait de fonctions qui peuvent prendre ou non des paramètres et retourner ou non des valeurs / objets. S'agissant d'actions, nous vous conseillons de les nommer avec un verbe à l'infinitif.

Elles se déclarent de la même manière que des fonctions traditionnelles.

Au même titre que les attributs, on déclare une méthode avec un niveau de visibilité. Le mot-clé *public*
indique que l'on pourra appliquer la méthode en dehors de la classe, c'est à dire sur l'objet lui même.

4. Utilisation des classes et des objets

Instanciation d'une classe

L'instanciation d'une classe est la phase de création des objets issus de cette classe. Lorsque l'on instancie une classe, on utilise **le mot-clé** *new* suivant du nom de la classe. Cette instruction appelle la méthode constructeur (__construct()) qui construit l'objet et le place en mémoire. Voici un exemple qui illustre 4 instances différentes de la classes *Personne.*

Création d'objets de type Personne

```
<?php

 $personne1 = new Personne();
 $personne2 = new Personne();
 $personne3 = new Personne();
 $personne4 = new Personne();
```

Un objet est en fait une variable dont le type est celui de la classe qui est instanciée.

Accès aux attributs

Accès en écriture

Nous venons de créer 2 objets de même type et de même structure.

Nous affectons à présent des valeurs à chacun des attributs de chaque objet.

Utilisation des attributs d'un objet

```
<?php
```

```
// Définition des attributs de la personne 1
$personne1->nom = 'Hamon';
$personne1->prenom = 'Hugo';
$personne1->dateDeNaissance = '02-07-1987';
$personne1->taille = '180';
$personne1->sexe = 'M';


// Définition des attributs de la personne 2
$personne2->nom = 'Dubois';
$personne2->prenom = 'Michelle';
$personne2->dateDeNaissance = '18-11-1968';
$personne2->taille = '166';
$personne2->sexe = 'F';
```

Nous avons maintenant des objets ayant chacun des caractéristiques différentes.

Dans notre exemple, nous accédons directement à la valeur de l'attribut. Cela est possible car nous avons défini l'attribut comme étant *public*. Si nous avions déclaré l'attribut avec les mots-clés *private* ou *protected*, nous aurions du utiliser un autre mécanisme pour accéder à sa valeur ou bien la mettre à jour.

Accès en lecture

La lecture de la valeur d'un attribut d'un objet se fait exactement de la même manière que pour une variable traditionnelle. Le code suivant présente comment afficher le *nom* et le *prénom* de la personne 1.

Affichage du nom et du prénom de la personne 1

```
<?php

echo 'Personne 1 :<br/><br/>';
echo 'Nom : ', $personne1->nom ,'<br/>';
echo 'Prénom : ', $personne1->prenom;
```

L'exécution de ce programme produit le résultat suivant sur la sortie standard :

Résultat d'exécution du code
Personne 1 :
Nom : Hamon
Prénom : Hugo

Accès aux constantes

L'accès aux constantes ne peut se faire qu'en lecture via l'opérateur **::** . L'exemple suivant illustre la lecture d'une constante de la classe *Personne*.

Accès à une constante
<?php

echo 'Chaque personne a **', Personne::NOMBRE_DE_YEUX** ,' yeux.';
?>

L'exécution de ce code affiche la chaine suivante sur la sortie standard :

Résultat de l'exécution du code
Chaque personne a 2 yeux.

Accès aux méthodes

Appel de méthode sur des objets
<?php
$personne1->boire();
$personne2-> manger();
?>Après exécution, le résultat est le suivant :
Résulat de l'exécution du code
La personne boit
La personne mange

5. Visibilité des propriétés et des méthodes d'un objet

La visibilité des propriétés et des méthodes d'un objet constitue une des particularités élémentaires de la programmation orientée objet.

La visibilité permet de définir de quelle manière un attribut ou une méthode d'un objet sera accessible dans le programme.

PHP introduit 3 niveaux différents de visibilité applicables aux propriétés ou méthodes de l'objet.

Il s'agit des visibilités ***publiques, privées*** ou *protégées* qui sont respectivement définies dans la classe au moyen des mots-clés *public, private* et ***protected***.

L'exemple suivant illustre la syntaxe de déclaration de données membres (attributs) et de méthodes ayant des visibilités différentes.

Utilisation des mots-clés public, private et protected

```
<?php

class Vehicule
{
  // Attributs
  public $marque;
  private $_volumeCarburant;
  protected $_estRepare;

  // Méthodes
  public function __construct()
  {
    $this->_volumeCarburant = 40;
    $this->_estRepare = false;
  }

  // Démarre la voiture si le réservoir n'est pas vide
  public function demarrer()
  {
    if ($this->_controlerVolumeCarburant())
```

```
    {
      echo 'Le véhicule démarre';
      return true;
    }

    echo 'Le réservoir est vide...';
    return false;
  }

  // Vérifie s'il y'a du carburant dans le réservoir
  private function _controlerVolumeCarburant()
  {
    return ($this->_volumeCarburant > 0); // renvoi true ou false
  }

  // Met le véhicule en maintenance
  protected function reparer()
  {
    $this->_estRepare = true;
    echo 'Le véhicule est en réparation';
  }
}
```

L'accès public

C'est l'accès par défaut de PHP si l'on ne précise aucune visibilité. Tous les attributs et méthodes qui sont déclarés sans l'un de ces trois mots-clés sera considéré automatiquement par l'interpréteur comme *publique*.

Le mot-clé *public* indique que les propriétés et méthodes d'un objet seront accessibles depuis n'importe où dans le programme principal ou bien dans les classes mères héritées ou classes filles dérivées.

exemple :

Utilisation de la visibilité publique

```
<?php

  // Instanciation de l'objet : appel implicite à la méthode __construct()
  $monVehicule = new Vehicule();

  // Mise à jour de la marque du véhicule
  $monVehicule->marque = 'Peugeot';

  // Affichage de la marque du véhicule
  echo $monVehicule->marque;
?>
```

L'accès private

Le mot-clé *private* permet de déclarer des attributs et des méthodes qui ne seront visibles et accessibles directement que depuis l'intérieur même de la classe.

L'accès protected

L'accès protégé (*protected*) est un intermédiaire entre l'accès publique et l'accès privé. Il permet d'utiliser des attributs et méthodes communs dans une classe parente et ses classes dérivées (héritantes).

Utilisation des accès protégés

```
<?php

class Vehicule
{
  // Attributs
  protected $_marque;
  protected $_estRepare;

  // Méthodes
```

```
  public function __construct($marque)
  {
    $this->_marque = $marque;
    $this->_estRepare = false;
  }

  // Met le véhicule en maintenance
  public function reparer()
  {
    $this->_estRepare = true;
    echo 'Le véhicule est en réparation';
  }
}


class Voiture extends Vehicule
{
  // Attributs
  private $_volumeCarburant;

  // Constructeur
  public function __construct($marque)
  {
    // Appel du constructeur de la classe parente
    parent::__construct($marque);
    $this->_volumeCarburant = 40;
  }

  // Démarre la voiture si le réservoir
  // n'est pas vide
  public function demarrer()
  {
    if ($this->_controlerVolumeCarburant())
    {
```

```
      echo 'Le véhicule démarre';
      return true;
    }

    echo 'Le réservoir est vide...';
    return false;
  }

  // Vérifie qu'il y'a du carburant dans le réservoir
  private function _controlerVolumeCarburant()
  {
    return ($this->_volumeCarburant > 0);
  }
}
```

Nous venons de déclarer deux attributs protégés dans la classe parente *Vehicule* et nous avons redescendu l'attribut *_volumeCarburant* dans la classe *Voiture*. De même nous avons redescendu les méthodes *demarrer()* et *_controlerVolumeCarburant()* dans la classe Voiture. Ainsi, nous pouvons désormais instancier un nouvel objet de type Voiture et profiter des attributs et des méthodes de la classe *Vehicule* en plus. Enfin, notez que la méthode *reparer()* est passée du mode *protected* à *public* afin de pouvoir l'appeler depuis l'extérieur de la classe. Ce qui donne :

Utilisation des attributs et méthodes protégés

```
<?php

  $monVehicule = new Voiture('Peugeot');
  $monVehicule->demarrer();
  $monVehicule->reparer();

?>
```

Mise à jour d'un attribut privé ou protégé : rôle du mutator

Nous avons évoqué plus haut qu'il était impossible d'accéder à la valeur d'un attribut privé ou protégé en utilisant la syntaxe suivante : *$objet->attribut*. Une erreur fatale est générée. Comment faire pour mettre à jour la valeur de l'attribut dans ce cas ? C'est là qu'intervient le *mutator*.

Le mutator n'est ni plus ni mois qu'une méthode publique à laquelle on passe en paramètre la nouvelle valeur à affecter à l'attribut. Par convention, on nomme ces méthodes avec le préfixe *set*. Par exemple : *setMarque(), setVolumeCarburant()*... C'est pourquoi on entendra plus souvent parler de *setter* que de *mutator*.

Ajoutons un mutator à notre classe *Voiture* qui permet de modifier la valeur du volume de carburant. Cela donne :

Ajout du mutator (setter) setVolumeCarburant à la classe Voiture

```
<?php

class Voiture extends Vehicule
{

  // ...

  public function setVolumeCarburant($dVolume)
  {
    $this->_volumeCarburant = $dVolume;
  }
}
```

Pour respecter les conventions de développement informatique, nous utilisons dans notre exemple l'écriture au moyen du préfixe *set*. Mais ce nom est complètement arbitraire et notre méthode aurait très bien pu s'appeller *faireLePlein()* ou *remplirLeReservoir()*. Il ne tient qu' à vous de donner des noms explicites à vos méthodes.

Quant à l'utilisation de cette méthode, il s'agit tout simplement d'un appel de méthode traditionnel.

Mise à jour de la valeur de l'attribut privé $_volumeCarburant

```
<?php
  // Je remplis mon réservoir de 50 L d'essence
  $monVehicule->setVolumeCarburant(50);
?>
```

Obtenir la valeur d'un attribut privé ou protégé : rôle de l'accessor

Nous venons d'étudier comment mettre à jour (écriture) la valeur d'un attribut privé ou protégé. Il ne nous reste plus qu' à aborder le cas de la lecture de cette valeur. Comment restitue-t-on cette valeur dans un programme sachant que l'accès direct à l'attribut est interdit ? De la même manière que pour le mutator, nous devons déclarer une méthode qui va se charger de **retourner** la valeur de l'attribut. Ce n'est donc ni plus ni moins qu'une fonction avec une instruction *return.*

On appelle ce type de méthode un *accessor* ou bien plus communément un *getter.* En effet, par convention, ces méthodes sont préfixées du mot *get* qui se traduit en français par « *obtenir, récupérer* ». Au même titre que les mutators, il est possible d'appliquer n'importe quel nom à un accessor.

Reprenons notre exemple précédent. Le getter ci-après explique le principe de retour de la valeur d'un attribut privé. Nous renvoyons ici la valeur de la variable *$_volumeCarburant.*

Déclaration d'un accessor pour l'attribut privé $_volumeCarburant de la classe Voiture

```
<?php
class Voiture extends Vehicule
{
  // ...

  public function getVolumeCarburant()
  {
    return $this->_volumeCarburant;
  }
}
```

Voici un exemple d'utilisation de la méthode *getVolumeCarburant :*

Utilisation de l'accessor getVolumeCarburant() sur l'objet $monVehicule

```
<?php

echo sprintf("Il me reste %u L d'essence", $monVehicule->getVolumeCarburant());

?>
```

Caractéristiques du modèle objet de PHP

L'observation des exemples précédents nous permet de retrouver certains concepts bien connus de la POO, repris par PHP :

- une **classe** se compose d'**attributs** et de **méthodes** ;
- le mot-clé class permet de définir une classe ;
- les différents niveaux d'accessibilité sont public, protected et private ;
- le mot-clé extends permet de définir une classe dérivée (comme en Java) ;
- le mot-clé $this permet d'accéder aux membres de l'objet courant.

Spécificités du modèle objet de PHP

Même s'il est similaire à ceux de C#, Java ou C++, le modèle objet de PHP possède certaines particularités :

- PHP étant un langage à typage dynamique, on ne précise pas les types des attributs et des méthodes, mais seulement leur niveau d'accessibilité ;
- le mot-clé function permet de déclarer une méthode, quelle que soit son type de retour ;
- le mot-clé parent permet d'accéder au parent de l'objet courant. Il joue en PHP le même rôle que base en C# et super en Java ;
- le constructeur d'une classe s'écrit __construct ;
- la méthode __toString détermine comment l'objet est affiché en tant que chaîne de caractères ;
- on peut redéfinir (*override*) une méthode, comme ici __toString, sans mot-clé particulier ;
- il est possible de définir une valeur par défaut pour les paramètres d'une méthode. Elle est utilisée lorsque l'argument (paramètre effectif) n'est pas précisé au moment de l'appel. Remarques :

6. L'héritage

L'héritage en POO permet d'abstraire certaines fonctionnalités communes à plusieurs classes, tout en permettant aux classes filles d'avoir leurs propres méthodes.

```
<?php
  class voiture{
    public $roue = 4;
  }
  class Renault extends voiture{
  }
  class Peugeot extends voiture{
      public $roue = 5;
  }
  $peugeot = new Peugeot();
  $renault = new Renault();
  print_r( $peugeot->roue ); // retourne 5
  print_r( $renault->roue ); // retourne 4
?>
```

Visibilité

La visibilité d'un attribut ou d'une méthode peut être définie en prefixant sa déclaration avec un mots clé: **public**, **protected** ou **private**. Les éléments "**public**" peuvent être appelés à n'importe quelle partie du programme. Les "**protected**" ne peuvent être appelés que par la classe elle même ou les classes parents/enfants. Les "**private**" sont disponibles que pour la classe en elle même.

Exemple:

```
<?php
  class voiture{
    public $roue = 4;
    protected $prix = 5000;
    private $nom = "Batmobile";
  }
  $voiture = new voiture();
  print_r( $voiture->roue ); // retourne 4
  print_r( $voiture->prix ); // retourne erreur
  print_r( $voiture->nom ); // retourne erreur
?>
```

Mettre ce genre de protection permet d'indiquer au développeur qu'il doit récupérer les valeurs des attributs en passant par des **getter** pour des raisons de stratégies.

Exemple:

```
<?php
  class voiture{
    public $roue = 4;
    protected $prix = 5000;
    private $nom = "Batmobile";
    public function getPrix(){
        return ( $this->prix + 100 );
    }
    public function getNom(){
        return $this->nom;
    }
  }
  $voiture = new voiture();
  print_r( $voiture->roue ); // retourne 4
  print_r( $voiture->getPrix() ); // retourne 5100
  print_r( $voiture->getNom() ); // retourne Batmobile
?>
```

Pour les méthodes, c'est la même logique:

```
<?php
  class voiture{
    public $roue = 4;
    protected $prix = 5000;
    private $nom = "Batmobile";
    public function getPrix(){
      return ( $this->calcPrix() + 100 );
    }
    // methode de calcule non public
    protected function calcPrix(){
      return ( $this->prix + 10 );
    }
  }
  $voiture = new voiture();
  print_r( $voiture->getPrix() ); // retourne 5110
  print_r( $voiture->calcPrix() ); // retourne erreur
```

Exemple 1 : de La POO avec PHP

Exemple 1 : de La POO avec PHP

Le langage PHP permet d'utiliser la programmation orientée objet, avec toutefois quelques spécificités par rapport à d'autres langages comme Java ou C#. Nous allons faire un tour d'horizon des possibilités de PHP en matière de POO au travers de l'exemple classique des comptes bancaires.

```
<?php

class CompteBancaire
{
   private $devise;
   private $solde;
   private $titulaire;

   public function __construct($devise, $solde, $titulaire)
   {
      $this->devise = $devise;
      $this->solde = $solde;
      $this->titulaire = $titulaire;
   }

   public function getDevise()
   {
      return $this->devise;
   }

   public function getSolde()
   {
      return $this->solde;
   }

   protected function setSolde($solde)
   {
      $this->solde = $solde;
   }

   public function getTitulaire()
   {
      return $this->titulaire;
   }

   public function crediter($montant) {
      $this->solde += $montant;
   }
```

```
public function __toString()
    {
        return "Le solde du compte de $this->titulaire est de " .
            $this->solde . " " . $this->devise;
    }
}
```

En complément, voici la définition d'une classe CompteEpargne qui hérite de la classe CompteBancaire.

```
<?php

require_once 'CompteBancaire.php';

class CompteEpargne extends CompteBancaire
{
    private $tauxInteret;

    public function __construct($devise, $solde, $titulaire, $tauxInteret)
    {
        parent::__construct($devise, $solde, $titulaire);
        $this->tauxInteret = $tauxInteret;
    }

    public function getTauxInteret()
    {
        return $this->tauxInteret;
    }

    public function calculerInterets($ajouterAuSolde = false)
    {
        $interets = $this->getSolde() * $this->tauxInteret;
        if ($ajouterAuSolde == true)
            $this->setSolde($this->getSolde() + $interets);
        return $interets;
    }

    public function __toString()
    {
      return parent::__toString() .
        '. Son taux d\'interet est de ' . $this->tauxInteret * 100 . '%.';
    }
}
```

Voici un exemple d'utilisation de ces deux classes.

```
<?php

require 'CompteBancaire.php';
require 'CompteEpargne.php';

$compteJean = new CompteBancaire("euros", 150, "Jean");
echo $compteJean . '<br />';
$compteJean->crediter(100);
echo $compteJean . '<br />';

$comptePaul = new CompteEpargne("dollars", 200, "Paul", 0.05);
echo $comptePaul . '<br />';
echo 'Interets pour ce compte : ' . $comptePaul->calculerInterets() .
   ' ' . $comptePaul->getDevise() . '<br />';
$comptePaul->calculerInterets(true);
echo $comptePaul . '<br />';
```

Enfin, voici le résultat de son exécution.

Le solde du compte de Jean est de 150 euros

Le solde du compte de Jean est de 250 euros

Le solde du compte de Paul est de 200 dollars. Son taux d'interet est de 5%.

Interets pour ce compte : 10 dollars

Le solde du compte de Paul est de 210 dollars. Son taux d'interet est de 5%.

Exemple 2

Enoncé

L'objectif de cet exercice est de créer une classe PHP au nom de **String** qui contient les méthodes du même nom que celles de Javascript et qui font les mêmes tâches.

Pour récapituler voilà ce que font les méthodes que l'on va créer :

- **length**: Cet attribut retourne le nombre de caractères contenus dans la chaîne.
- **charAt()**: La méthode charAt(x) (avec A majuscule) permet de retourner le caractères qui se trouve à la position **x** passé en paramètre. Le paramètre est un entier qui commence de 0. La valeur 0 indexe le premier caractère de la chaîne.
- **indexOf()**: La méthode indexOf(car) (avec O en majuscule) permet de retourner la position du caractères **car** passé en paramètre. Si le caractère existe dans la chaîne, alors sa position (comprise entre 0 et la longueur de la chaîne - 1) est retournée, sinon (le caractère ne figure pas dans la chaîne) alors la valeur **-1** est retournée. La méthode **indexOf()** peut accueillir un deuxième paramètre qui est un entier qui indique à partir de quel position de la chaîne on commencera la recherche du caractère passé en premier paramètre.
- **substring()**: La méthode substring(début,fin) permet d'extraire une partie de la chaîne de caractères commençant de la position **début** et finissant à la position **fin-1**.
- **split()**: La méthode split(occurence) permet de retourner un tableau à partir des fractions de la chaîne de caractères obtenues en divisant celle-ci au niveau de l'**occurence**.
- **toLowerCase()**: La méthode toLowerCase() permet de retourner la chaîne de caractère entièrement en minuscules.
- **toUpperCase()**: La méthode toUpperCase() permet de retourner la chaîne de caractère entièrement en majuscules.

String.php

```
<?php
  class String{
    public $str;
    public $length;
    public function __construct($chaine){
      $this->str=$chaine;
      $this->length=strlen($this->str);
    }
    public function __toString(){
      return $this->str;
    }
```

Exemple 2

Enoncé

L'objectif de cet exercice est de créer une classe PHP au nom de **String** qui contient les méthodes du même nom que celles de Javascript et qui font les mêmes tâches.

Pour récapituler voilà ce que font les méthodes que l'on va créer :

- **length**: Cet attribut retourne le nombre de caractères contenus dans la chaîne.
- **charAt()**: La méthode charAt(x) (avec A majuscule) permet de retourner le caractères qui se trouve à la position **x** passé en paramètre. Le paramètre est un entier qui commence de 0. La valeur 0 indexe le premier caractère de la chaîne.
- **indexOf()**: La méthode indexOf(car) (avec O en majuscule) permet de retourner la position du caractères **car** passé en paramètre. Si le caractère existe dans la chaîne, alors sa position (comprise entre 0 et la longueur de la chaîne - 1) est retournée, sinon (le caractère ne figure pas dans la chaîne) alors la valeur **-1** est retournée. La méthode **indexOf()** peut accueillir un deuxième paramètre qui est un entier qui indique à partir de quel position de la chaîne on commencera la recherche du caractère passé en premier paramètre.
- **substring()**: La méthode substring(début,fin) permet d'extraire une partie de la chaîne de caractères commençant de la position **début** et finissant à la position **fin-1**.
- **split()**: La méthode split(occurence) permet de retourner un tableau à partir des fractions de la chaîne de caractères obtenues en divisant celle-ci au niveau de l'**occurence**.
- **toLowerCase()**: La méthode toLowerCase() permet de retourner la chaîne de caractère entièrement en minuscules.
- **toUpperCase()**: La méthode toUpperCase() permet de retourner la chaîne de caractère entièrement en majuscules.

String.php

```
<?php
  class String{
    public $str;
    public $length;
    public function __construct($chaine){
      $this->str=$chaine;
      $this->length=strlen($this->str);
    }
    public function __toString(){
      return $this->str;
    }
    public function __get($param){
```

Exemple 2

Enoncé

L'objectif de cet exercice est de créer une classe PHP au nom de **String** qui contient les méth du même nom que celles de Javascript et qui font les mêmes tâches.

Pour récapituler voilà ce que font les méthodes que l'on va créer :

- **length**: Cet attribut retourne le nombre de caractères contenus dans la chaine.
- **charAt()**: La méthode charAt(x) (avec A majuscule) permet de retourner le caractère se trouve à la position **x** passé en paramètre. Le paramètre est un entier qui commenc 0. La valeur 0 indexe le premier caractère de la chaîne.
- **indexOf()**: La méthode indexOf(car) (avec O en majuscule) permet de retourner la position du caractères **car** passé en paramètre. Si le caractère existe dans la chaine, alo position (comprise entre 0 et la longueur de la chaîne - 1) est retournée, sinon (le caractère ne figure pas dans la chaine) alors la valeur **-1** est retournée. La méthode **indexOf()** peut accueillir un deuxième paramètre qui est un entier qui indique à parti quel position de la chaîne on commencera la recherche du caractère passé en premier paramètre.
- **substring()**: La méthode substring(début,fin) permet d'extraire une partie de la chaîn caractères commençant de la position **début** et finissant à la position **fin-1**.
- **split()**: La méthode split(occurence) permet de retourner un tableau à partir des fract de la chaîne de caractères obtenues en divisant celle-ci au niveau de l'**occurence**.
- **toLowerCase()**: La méthode toLowerCase() permet de retourner la chaîne de caract entièrement en minuscules.
- **toUpperCase()**: La méthode toUpperCase() permet de retourner la chaîne de caract entièrement en majuscules.

String.php

```
<?php
  class String{
    public $str;
    public $length;
    public function __construct($chaine){
      $this->str=$chaine;
      $this->length=strlen($this->str);
    }
    public function __toString(){
      return $this->str;
    }
```

```
    return $this->$param;
  }
  public function charAt($pos){
    return substr($this->str,$pos,1);
  }
  public function indexOf($car,$pos=0){
    $existe="non";
    for($i=0;$i<$this->length;$i++){
      if(substr($this->str,$i,1)==$car && $i>=$pos){
        $existe="oui";
        return $i;
      }
    }
    if($existe=="non")
      return -1;
  }
  public function substring($deb,$fin){
    return substr($this->str,$deb,$fin-$deb);
  }
  public function split($occ){
    $tab=explode("$occ",$this->str);
    return $tab;
  }
  public function toUpperCase(){
    return strtoupper($this->str);
  }
  public function toLowerCase(){
    return strtolower($this->str);
  }
}

?>
```

testString.php

```
<?php
require 'String.php';
  $str=new String("Bonjour");
  echo $str->length; // Retourne 7
  echo $str->charAt(0); // Retourne B
  echo $str->indexOf("B"); // Retourne 0
  echo $str->indexOf("x"); // Retourne -1
  echo $str->indexOf("o",2); // Retourne 4;
  echo $str->substring(0,3); // Retourne Bon
  print_r($str->split("o")); // Retourne Array([0]=>B[1]=>nj[2]=>ur)
  echo $str->toUpperCase(); // Retourne BONJOUR
  echo $str->toLowerCase(); // Retourne bonjour
?>
```

Exemple 3 : Un Objet Simple Etudiant en PHP

Créons maintenant un objet simple en PHP. Ecrivons un objet représentant un étudiant avec s
données:

- identifiant
- nom
- date de naissance

et des méthodes pour opérer sur ces données:

- constructeur
- getters et setters
- equals()
- toString() pour affichage

ce qui donne le code suivant:

```
<?php
/** Classe Etudiant en PHP */

class Etudiant{
  /** Identification unique d'un etudiant */
  protected $etudiant_id;
  /** Nom de l'etudiant */
  protected $nom;
  /** Date de naissance de l'etudiant */
  protected $naissance;

  public function __construct($id, $nom,$naissance){
    $this->etudiant_id = (int)$id;  // cast vers integer
    $this->nom = (string)$nom;     // cast vers string
    $this->naissance= (int)$naissance;  // cast vers date(timestamp)
  }

  /**
   * Fonction de comparaison simplifiee entre etudiants
   * == comparera id, nom et naissance
   */
  public function equals(etudiant $etudiant){
    return ($this->getId() == $etudiant->getId());
  }

  public function getId(){
    return $this->etudiant_id;
  }

  public function getNom(){
```

public function getNom(){

```
    return $this->nom;
  }

  public function getNaissance(){
    return $this->naissance;
  }

  public function __toString(){
            setlocale(LC_TIME, "fr_FR");
            $ne=strftime('%A %d %B %Y',$this->naissance);
    return 'etudiant: id=' . $this->getId().', nom='.$this->getNom()." $ne";
  }
}
/* Test : */

$etu=new Etudiant(234,"Talon",time());
var_dump($etu);
echo "<br/>";
echo $etu;
?>
```

Exemple **4**

```
<?php
class Commande {
   var $prixRoyale = 6;
   var $prixCampagnarde = 8;
   var $nomClient;
   var $listePizzas;

   //Constructeur de la classe car même nom:
   function Commande() {
      $this->nomClient = "SansNom";
   }

   function ajouterRoyale($nombre) {
      $this->listePizzas[0] += $nombre;
   }

   function ajouterCampagnarde($nombre) {
      $this->listePizzas[1] += $nombre;
   }
   function calculerPrix() {
      $montant_Royale = $this->listePizzas[0] * $this->prixRoyale;
      $montant_Campagnarde = $this->listePizzas[1] * $this->prixCampagnarde;
      return $montant_Royale + $montant_Campagnarde;
   }
```

```
    function afficherCommande() {
        echo "Commande du client : ".$this->nomClient;
        echo "<BR>Pizza(s) 'Royale' : ".$this->listePizzas[0];
        echo "<BR>Pizza(s) 'Campagnarde' : ".$this->listePizzas[1];
        echo "<HR>Totale de votre commande : ".$this->calculerPrix();
        echo " Euros<BR>";
    }
}
$client1 = new Commande();
$client1->ajouterRoyale(5);
$client1->ajouterCampagnarde(2);
$client1->afficherCommande();

?>
```

Exemple 5

```php
<?php
  class voiture{
    public $nb_roues = 4;
    public $volant = 1;
    public $prix = 5000;
  }
  $voiture = new voiture();
  var_dump( $voiture );
?>
```

Résultat:

```
object(voiture)[1]
public 'nb_roues' => int 4
public 'volant'   => int 1
public 'prix'     => int 5000
```

Je peux voir la valeur d'un attibut avec la syntaxe suivante:

```php
<?php
  var_dump( $voiture->nb_roues ); // Retournera la valeur 4
?>
```

Les méthodes

Les méthodes sont des fonctions propres à la classe.

Exemple:

```
<?php
  class voiture{
    public $nb_roues = 4;
    public $volant = 1;
    public $prix = 5000;
    // retourne le prix de la voiture
    public function prix_voiture(){
      return $this->prix;
    }
  }
  $voiture = new voiture();
  print_r( $voiture->prix_voiture() ); // retourne 5000
?>
```

Les méthodes permettent entre autre de manipuler les attributs. Il est d'ailleurs d'usage de ne pas appeler un attribut directement mais de passer par une méthode.

Le constructeur

Le constructeur est une méthode qui est exécutée lors de l'instanciation de la classe.

```
<?php
  class voiture{
    public $nb_roues = 4;
    public $volant = 1;
    public $prix = 5000;
    // methode constructeur
    public function __construct(){
      $this->prix+=150;
    }
    // retourne le prix de la voiture
    public function prix_voiture(){
      return $this->prix;
    }
  }
  $voiture = new voiture();
      print_r( $voiture->prix_voiture() ); // retourne 5150
?>
```

Dans l'exemple ci-dessus on remarque que le prix à été augmenté de 150 euros. Cette action s'est déroulée dans le constructeur, qui a pour nom: **__construct()**

Méthodes statiques

La méthode statique est une méthode qui n'a pas besoin d'être appelée depuis un objet. Sa syntaxe est celle ci: **CLASSE::METHODE()**

```
<?php
  class voiture{
    public $nb_roues = 4;
    public $volant = 1;
    public $prix = 5000;
    // methode constructeur
    public function __construct(){
      $this->prix+=150;
    }
    // retourne le prix de la voiture
    public function prix_voiture(){
      return $this->prix;
    }
    public static function nom_de_la_voiture(){
      return "BATMOBILE";
    }
  }
    print_r( voiture::nom_de_la_voiture() );    // retourne BATMOBILE
?>
```

On remarque dans l'exemple ci-dessus que pour appeler une méthode statique, on appelle directement la méthode sans instancier la classe.

getter et setter

Pour modifier une propriété on peut utiliser la syntaxe suivante:

```
<?php
  class voiture{
    public $nb_roues = 4;
    public $volant = 1;
    public $prix = 5000;
  }
  $voiture = new voiture();
  $voiture->prix = 5400;
    print_r( $voiture->prix ); // retourne 5400
?>
```

Il n'est cependant pas conseillé d'utiliser cette syntaxe pour changer la valeur d'un attribut, il est préférable de passer par des methodes qui feront la modification. On appelle ce genre de méthode un **setter** ; et on récupère la valeur avec un **getter**.

```
<?php
  class voiture{
    public $nb_roues = 4;
    public $volant = 1;
    public $prix = 5000;
    // Change le prix
    public function setPrix( $prix ){
      $this->prix = $prix;
    }
    // retourne le prix
    public function getPrix( ){
      return $this->prix;
    }
  }
  $voiture = new voiture();
  // setter
  $voiture->setPrix( 5400 );
  // getter
    print_r( $voiture->getPrix() ); // retourne 5400
?>
```