Programmation orientée objet (POO)

I. Introduction: la POO

- La programmation orientée objet est un nouveau moyen de penser votre code.
- Permet une organisation plus cohérente de vos projets, une maintenance facilitée et une distribution de votre code plus aisée.
- la programmation orientée objet en PHP est très proche de celle de Java. Elle a principalement tiré son modèle objet de ce langage.
- À partir de PHP5

II. Définition s’une classe

- **Syntaxe de base:**

```
<?php
	class Personne
	{
		//déclaration des attributs
		private $nom="Tounsi";		// Le nom de la personne
		private $prenom="Yassine"; // Le prenom de la personne

		//déclaration des méthodes
		public function afficherIdentite() // Une méthode qui affiche une personne
		{
			echo "Nom: $this->nom Prénom: $this->prenom";
		}
	}
?>
```

II. Définition s'une classe

- **Les attributs:** sont les caractéristiques (propriétés) de la classe.
- Visibilité d'un attribut: Obligatoire. Elle indique à partir d'où on peut y avoir accès:
  - **public** (équivalent à **var**):accessible depuis l'intérieur(méthodes) ou l'extérieur de l'objet
  - **private**: accessible uniquement depuis l'intérieur de l'objet (méthodes).
  - **protected**
- La valeur par défaut doit être une expression scalaire statique (pas de fonctions ni variables)

```
<?php
        class Personne
        {
            public $nom="Tounsi";       // attribut publique
            var $prenom="Yassine";       // un autre attribut publique
            private $age=17; // attribut privé
            public $taille=strlen('une chaine'); //erreur
            public $pays=$_SESSION['pays']; //erreur
            public $ville='Sidi'. 'Bouzid'; //erreur pour PHP <5.6
            public $poids=100 + 15; //erreur pour PHP <5.6
        }
?>
```

II. Définition s'une classe

➢**Les méthodes:** sont les fonctions de la classe. Elles se déclarent avec le mot **function** précédé éventuellement de la visibilité.

➢Visibilité d'une méthode: Elle indique à partir d'où on peut y avoir accès:

- **public** (par défaut): accessible depuis l'intérieur ou l'extérieur de l'objet
- **private**: accessible uniquement depuis l'intérieur de l'objet
- **protected**

```
<?php
	class Personne
	{
		public function afficherIdentite(){……}  //méthode publique
		function manger(){….}    //implicitement publique
		private function seDeplacer(){…..} //méthode privée
	}
?>
```

III. Utilisation d'une classe

➢ **Création d'objet :**

- **new** : créer une instance de classe
- En PHP5, toute classe doit être déclarée avant d'être utilisée

```
<?php
	class Personne
	{
		public $nom ="Tounsi";
		public $prenom ="Yassine";
		public function afficherIdentite() // Une méthode qui affiche une personne
		 {
			echo "Nom: $this->nom Prénom: $this->prenom";
		 }
		public function manger(){....}
		public function seDeplacer(){.....}
	}
	$unePersonne=new personne(); // ou, pour ce cas, $unePersonne=new personne;
?>
```

III. Utilisation d'une classe

- **Accéder à un attribut**
  - **Syntaxe: $objet->attribut**

```
<?php
	class Personne
	{
		public $nom="Tounsi";
		public $prenom;
		private $age=17;
		……..…..
	}
	$unePersonne=new personne();
	echo $unePersonne->nom;    // affiche Tounsi
	$unePersonne->prenom="Yassine";  //modification de l'attribut $prenom
	$unePersonne->age= $unePersonne->age+1; //erreur fatale (attribut privé)
?>
```

III. Utilisation d'une classe

- **Accéder à une méthode**
  - **Syntaxe: $objet->nom_methode([paramètres])**

```
<?php
	class Personne
	{
		public $nom="Tounsi";
		public  function afficheIdentite()
		{
			return $this->nom;
		}
	}
	$unePersonne=new personne();
	echo $unePersonne-> afficheIdentite(); // appel de méthode
?>
```

III. Utilisation d'une classe

➢ **Référence à l'objet en cours: $this**

- Dans une méthode de la classe, le seul moyen pour accéder à un attribut de la classe est de passer par la référence de l'objet que nous somme en train d'utiliser: **$this->attribut**

```
<?php
	class Personne
	{
		public $nom="Tounsi";
		public  function afficheIdentite() { return $this->nom;}
		public  function changerNom($chaine)	{$this->nom=$chaine; }
	}
	$unePersonne=new personne();
	$unePersonne-> changerNom('Riahi'); // accès direct à un attribut
	$unePersonne->nom= "Ben Salah"; // accès à un attribut dans une méthode
?>
```

II. Définition s'une classe

➢ **L'opérateur de visibilité double deux points ::**

L'opérateur :: est utilisé pour appeler des éléments appartenant à telle classe et non à tel objet: (nom_classe::nom_élement)

- Les attributs et les méthodes appartenant à la classe et non à l'objet (des éléments statiques)
- les constantes de classe

➢ **Le mot clé self**

- Utilisé dans une classe (dans les méthodes)
- Permet d'accéder à des éléments appartenant à la classe et non aux objets (constantes, éléments statiques)

II. Définition s'une classe

➢ **Les constantes:**

- Elles se déclarent avec le mot clé **const**.
- Sont locales à la classe
- Par convention les noms en majuscule
- Utilisable sans instanciation d'objets
- Le nom d'une constante ne commence pas par un $

```
<?php
	class Personne
	{
		const NOMBRE_MAINS=2;
	}
	echo  Personne::NOMBRE_MAINS; // affiche 2
?>
```

III. Utilisation d'une classe

➢ **Accéder à une constante de classe**

- **nom_classe::constante** en dehors de la classe (sans création d'un objet)
- **self::constante** dans la classe. Le mot **self** référencie la classe dans laquelle il est utilisé

```
<?php
  class Personne
  {
  const NOMBRE_MAINS=2;
  public function nombre_mains() {
            return self::NOMBRE_MAINS;
            }
  }
  echo "Une personne a ". Personne::NOMBRE_MAINS . "mains<br>"; //affiche unePersonne a 2 mains

  $unePersonne=new personne(); // création d'un objet
  echo "unePersonne a ". $unePersonne->nombre_mains(). "mains"; //affiche unePersonne a 2 mains
?>
```

III. Utilisation d'une classe

➢ Méthodes statiques

- Sont faites pour agir sur une classe et non sur un objet (pas d'utilisation de $this qui désigne un objet créé)
- Déclarées avec le mot clé **static** devant function et après le type de visibilité

```
<?php
	class Personne
	{
		public static function parler()
		{
			echo "Bonjour tout le monde";
		}
	}
	Personne::parler();
	$unePersonne=new Personne();    //appel de la méthode depuis la classe
	$unePersonne-> parler(); // appel de la méthode depuis l'objet; correct aussi mais la référence
				// d'objet ($this) n'est pas passé à la méthode
?>
```

III. Utilisation d'une classe

➢ **Attributs statiques**

- même principe que les méthodes statiques.
- tous les objets auront accès à cet attribut et cet attribut aura la même valeur pour tous les objets.
- Déclarées avec le mot clé **static** devant le nom de l'attribut

```
<?php
      class Personne
      {
            private static $compteur=0; // déclaration de la variable statique privée compteur
            public function __construct(){self::$compteur++;}
            public static function getCompteur(){ return self:: $compteur; }
      }
      $unePersonne1=new Personne(); $unePersonne2=new Personne();
      $unePersonne3=new Personne(); // création de 3 objets personne
      echo Personne:: getCompteur();    //affiche 3
?>
```

III. Utilisation d'une classe

- **Les accesseurs et les mutateurs**
  - Le principe d'encapsulation nous empêche d'accéder directement aux attributs privés des objet: seule la classe peut les lire et les modifier.
  - Pour récupérer ou modifier un attribut, il faut le demander à la classe.
- **Un accesseur (ou getter)**: est une des méthodes dont le seul rôle est de retourner l'attribut. Par convention elle porte le même nom que l'attribut ou getNomAttribut()
- **Un mutateur (ou setter):** est une méthode qui permet de modifier la valeur d'un attribut. Par convention son nom est setNomAttribut(valeur)

```
<?php
	class Personne
	{
		private $nom;
		public  function nom() { return $this->nom;}	// getter ou accsseur
		public  function setNom($chaine) {$this->nom=$chaine; }  // setter ou mutateur
	}
?>
```

III. Utilisation d'une classe

➢ **Constructeur**

- c'est une méthode spéciale optionnelle nommée **__construct()** (avec 2 underscores) qui est appelée automatiquement lors de la création d'un objet
- peut accepter des paramètres
- Un constructeur doit être toujours publique

```
<?php
	class Personne
	{
		private $nom;  private $age;
		public  function __construct($n,$a) //constructeur de la classe Personne
		{
			$this->nom=$n; // ou $this->setNom($n);
			 $this->age=$a; // ou $this->setAge($a);
		}
	}
?>
```

III. Utilisation d'une classe

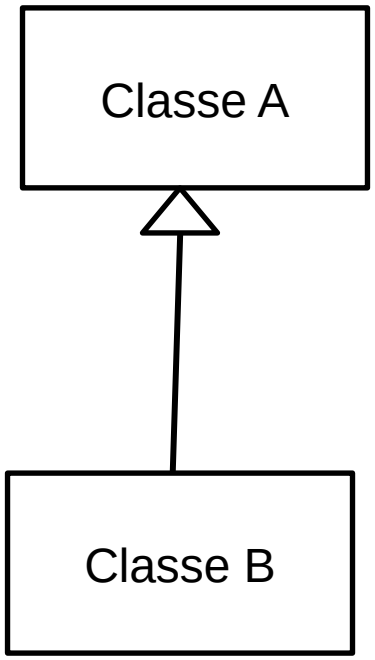
- **Destructeur** : Pour détruire un objet:
  - utiliser la méthode PHP **unset($obj)**
  - Ou utiliser une méthode **__destruct()** s'il y a un traitement à faire lors de la destruction

```
<?php
	class Personne
	{
		private $nom; private $age;
		public  function __construct($n,$a) {
			$this->nom=$n; // ou $this->setNom($n);
			 $this->age=$a; // ou $this->setAge($a);
		}
		 public  function afficher(){echo "Je suis ". $this->ali ." et j'ai ". $this->ali ."ans";

	}
	$unePersonne=new Personne('Ali',20);
	$unePersonne->afficher();
	unset($unePersonne); // destruction de l'objet
?>
```

IV. L'héritage

- L'avantage de l'héritage est de regrouper les éléments communs à plusieurs classes dans une seule classe.
- Si **une classe B hérite de la classe A**, nous avons alors:
  - La classe A est appelée classe mère, la classe B est appelée classe fille.
  - La classe B hérite de tous les attributs et méthodes **non privés** de la classe A et ils sont accessibles tout comme dans A.
  - De plus, la classe B peut avoir ses propres attributs et méthodes
- En PHP il n'y a pas d'héritage multiple
- Déclarée avec le mot clé **extends** suivi du nom de la classe mère.



IV. L'héritage

➢**Exemple:** la classe Etudiant hérite de la classe Personne. En plus des éléments hérités ($nom,$prenom,parler() et manger()), la classe Etudiant a son propre attribut $numInscription et sa propre méthode etudier()

```
<?php
	class Personne
	{
		private $nom;
		private $poids;
		public  function parler(){…….}
		 public  function manger(){…….}
	}
	class Etudiant extends Personne
	{
		private $numInscription;
		public  function etudier(){…….}
	}
?>
```

IV. L'héritage

➢Redéfinition des méthodes

- les méthodes héritées peuvent être réécrites dans la classe fille.
- Si vous redéfinissez une méthode, sa visibilité doit être la même ou plus permissive que dans la classe parente (private devient public par exemple, mais pas l'inverse).

```
<?php
        class Personne
        {
                private $nom; private $poids;
                public  function manger(){$this->poids+=1;}
        }
        class Etudiant extends Personne
        {
                 public  function manger()      //méthode redéfinie
                {
                        echo "Je mange donc je prends du poids";
                }
        }
?>
```

IV. L'héritage

➢Le mot clé parent::

- la méthode redéfinie manger() dans la classe fille ne peut pas accéder à l'attribut privé $poids de la classe mère. Pour le faire, il suffit d'appeler la méthode manger() de la classe mère avec le mot clé **parent** suivi de double deux points **::**

```
<?php
        class Personne
        {
                private $nom;  private $poids;
                public  function manger(){$this->poids+=1;}
        }
        class Etudiant extends Personne
        {
                 public  function manger()      //méthode redéfinie
                {
                        parent::manger();   //appel de la méthode de la classe mère
                        echo "Je mange donc je prends du poids";
                }
        }
?>
```

IV. L'héritage

➢Le mot clé parent::

- Accéder au constructeur de la classe parente

```
<?php
        class Personne
        {
                private $nom;
                private $poids;
                function __construct($n,$p)
                {
                        $this->nom=$n;
                        $this->poids=$p;
                }
        }
```

```
        class Etudiant extends Personne
        {
                private $numInscription;
                function __construct($n,$p,$i)
                {
                        parent::_construct($n,$p);
                        $this-> numInscription=$i);
                }
        }
?>
```

V. Portées de visibilité: private, public et protected

- **private:**
  - On ne peut accéder à l'attribut ou à la méthode que depuis l'intérieur de la classe qui l'a créé. Toute classe fille n'aura pas accès aux éléments privés.
  - Les objets instanciés n'ont pas accès aux éléments privés.
- **public:**
  - On peut accéder à l'attribut ou à la méthode de n'importe où. Toute classe fille aura accès aux éléments publics.
  - Les objets instanciés ont accès aux éléments publics.
- **protected:**
  - il a exactement les mêmes effets que private, à l'exception que toute classe fille aura accès aux éléments protégés.
  - Les objets instanciés n'ont pas accès aux éléments protégés.

VI. abstract et final

➢ **abstract:**

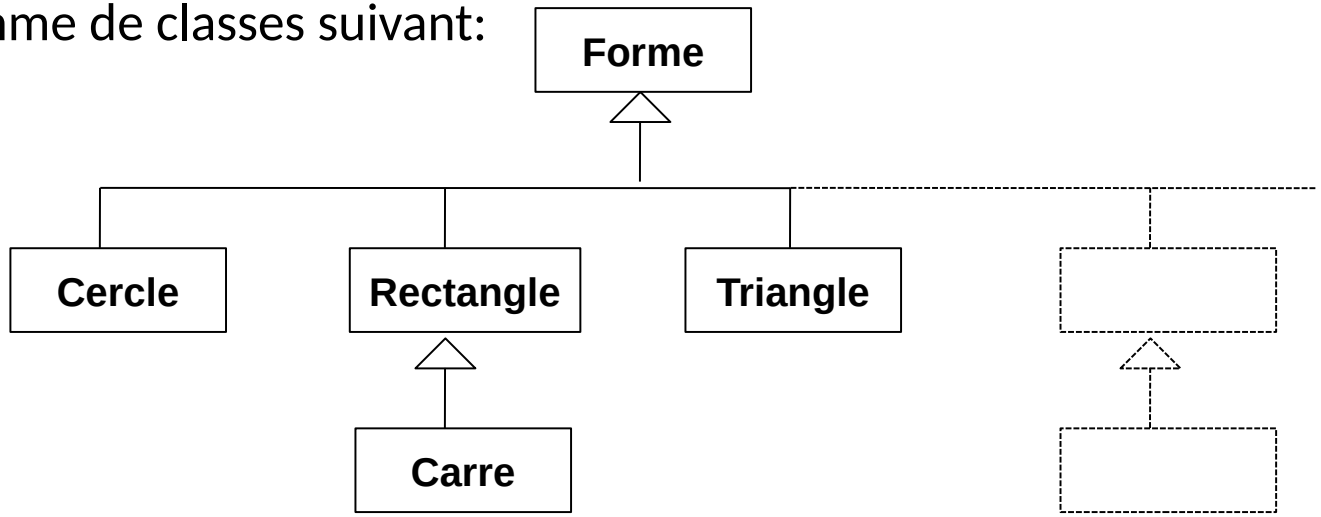
- Une classe abstraite ne peut pas être instanciée. Et sert uniquement de modèle pour ses classes dérivées (filles)
- Une méthode abstraite est une méthode qui n'est pas implémentée (n'a pas de code):

  abstract public function nom_methode();
- Une classe qui contient au moins une méthode abstraite doit être déclarée abstraite
- Dans une classe fille (qui hérite), toutes les méthodes abstraites de la classe mère doivent être définies (implémentées).
- Une classe abstraite ne peut pas être instanciée (mais elle peut avoir un constructeur).

VI. abstract et final

➢ **Exemple de classe abstraite**

- si l'on souhaite, par exemple, créer une hiérarchie de classe pour décrire des formes géométriques à deux dimensions, on pourrait envisager l'arborescence représentée par le diagramme de classes suivant:



- Sachant que toutes les formes possèdent les propriétés périmètre et surface, il est judicieux de placer les méthodes définissant ces propriétés (perimetre() et surface()) dans la classe qui est la racine de l'arborescence (Forme).

VI. abstract et final

➢ **Exemple de classe abstraite**

```
abstract class Forme
{
   abstract function perimetre();
   abstract function surface();
   public function bonjour(){echo
   "bonjour";}
```

```
class Rectangle extends Forme
{
   protected $longueur;
   protected $largeur;
   function __construct($long,$larg)
   {
         $this-> longueur =$long;
         $this-> largeur =$larg;
   }
   public function getLongueur(){return $this->longueur;}
   public function getLargueur(){return $this->largeur;}
   public function perimetre(){return 2*($this->longueur+     $this-
   >largeur);}
   public function surface(){return ($this->longueur* $this->largeur);}
```

```
class Cercle extends Forme
{
   const PI=3.14159265358979;
   private $rayon;
   function __construct($r) {$this-> rayon=$r;}
   public function getRayon(){return $this->rayon;}
   public function perimetre()
      {return 2*self::PI*$this->rayon;}
   public function surface()
     {return self::PI*$this->rayon *$this->rayon;}
}
```

VI. abstract et final

➢Final

- Le concept de classe et méthode finales est l'inverse du concept d'abstraction.
- Une **classe finale** ne peut pas être étendue

```
final class nom_classe{.....}
```

- Une **méthode finale** ne peut pas être redéfinie dans une classe fille

```
final public function nom_methode(){.....}
```

VII. Les interfaces

- Un moyen pour imposer à des classes d'implémenter certaines méthodes.
- Une interface est une classe qui ne peut pas être instanciée. Son rôle est de décrire un comportement de l'objet. C'est un mécanisme qui permet à plusieurs classes totalement indépendantes d'avoir un même comportement (mêmes méthodes).
- Une interface ne peut pas contenir d'attributs. Elle ne peut contenir que des déclarations (signatures) de méthodes (sans implémentations).
- Une interface se déclare avec le mot clé **interface**
- le méthodes d'une interface doivent être publiques et ne peuvent être ni abstraites ni finales.
- Le nom d'une interface doit être différent des noms des classes. Généralement il signifie la capacité d'avoir un comportement (se termine par able): exemple: Iterable, Nullable, Movable,..

VII. Les interfaces

- Exemple:

```
<?php
        interface Movable
        {
                public function deplacer($dx,$dy);
        }
        interface Modifiable
        {
                public function colorer($couleur);
        }
```

- Une classe peut implémenter un ou plusieurs interfaces (à condition de na pas avoir de méthodes portant le même nom) avec le mot clé **implements**

```
<?php
        class Rectangle extends Form implements Movable,Modifiable
        {
                ...
                public function deplacer($dx,$dy){.....}
                public function colorer($color){.....}
                 ...
        }
```

VII. Les interfaces

- Une interface peut hériter d'une autre interface et même de plusieurs interfaces (contrairement aux classes).

```
<?php
        interface iA { public function methode1(); }
        interface iB { public function methode2(); }
        interface iC extends iA,iB { public function methode3(); }
        class UneClasse implements iC
        {
                public function methode1() {……..
                }
                public function methode2() {……..
                }
                public function methode3() {……..
                }
                public function methode4(){……..
                }
        }
```

VIII. Résolution statique à la volée static::

- Soit l'exemple suivant: une classe Enfant qui hérite d'une classe Mere:

```
class Mere
{
  public static function lancerLeTest()
  {
    self::quiEstCe();
  }
  public static function quiEstCe()
  {
    echo 'Je suis la classe <b>Mere</b> ';
  }
}
```

```
class Enfant extends Mere
{
  public static function quiEstCe()
  {
    echo 'Je suis la classe <b>Enfant</b>';
  }
}
Enfant::lancerLeTest();
```

- Dans l'appel de la méthode lancerLeTest de la classe Enfant, comme la méthode n'a pas été réécrite, on va donc « chercher » la méthode lancerLeTest de la classe mère qui appelle la méthode quiEstCe de la classeMere (self:: fait appel à la méthode statique de la classe dans laquelle est contenu self::, donc de la classe parente).

IX. Les méthodes magiques

- Une méthode magique est une méthode qui, si elle est présente dans votre classe, sera appelée lors de tel ou tel évènement.
- Si la méthode n'existe pas et que l'évènement est exécuté, aucun effet « spécial » ne sera ajouté, l'évènement s'exécutera normalement.
- Exemple: les méthodes **__construct()** et **__destruct()** sont des méthodes magiques qui s'exécutent lorsque les événements création et destruction de l'objet aient lieu.

```
class MaClasse
{
  public function __construct()  { echo 'Construction de MaClasse';  }
  public function __destruct() { echo 'Destruction de MaClasse'; }
}
$obj = new MaClasse;
```

IX. Les méthodes magiques

- Les méthodes magiques sont des méthodes qui sont appelées automatiquement lorsqu'un certain évènement est déclenché.
- Toutes les méthodes magiques commencent par deux underscores, évitez donc d'appeler vos méthodes suivant ce même modèle.
- permettent la surcharge magique d'attributs ou méthodes qui consiste à créer dynamiquement des attributs et méthodes. Cela est possible lorsque l'on tente d'accéder à un élément qui n'existe pas ou auquel on n'a pas accès (s'il est privé et qu'on tente d'y accéder depuis l'extérieur de la classe par exemple).
- Dans cette partie nous allons présenter les méthodes magique suivantes: **__set, __get, __isset, __unset, __toString, __call , __callstatic**. Mais il en existe bien d'autres:**__set_state, __invoke, __debuginfo, ..**

IX. Les méthodes magiques

➢ **__set** :

- appelée lorsque l'on essaye d'assigner une valeur à un attribut auquel on n'a pas accès ou qui n'existe pas.
- Elle possède 2 paramètre (le nom de l'attribut et la valeur) et ne retourne rien

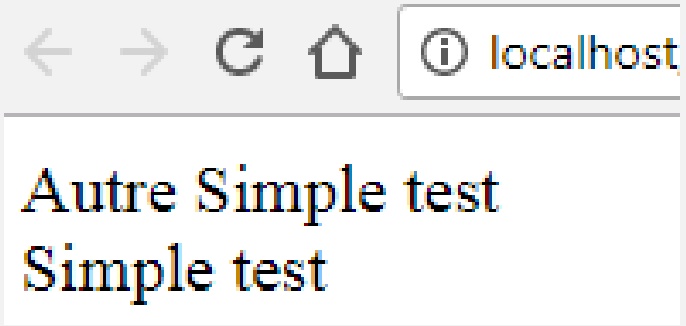
```
class MaClasse
{
  private $unAttributPrive;
  public function __set($nom, $valeur)  { $this->$nom=$valeur; }
  public function affichee(){
          echo "unAttributPrive:". $this->unAttributPrive. "<br>";
          echo "unAutreAttribut:". $this->unAutreAttribut. "<br>";}
}
$obj = new MaClasse;
$obj->unAttributPrive = 'Simple test';
$obj->unAutreAttribut = 'Autre Simple test';
$obj->affichee();
```

IX. Les méthodes magiques

➢ **__get** :

- appelée lorsque l'on essaye d'accéder à un attribut qui n'existe pas ou auquel on n'a pas accès. Elle prend un paramètre : le nom de l'attribut auquel on a essayé d'accéder.
- Cette méthode peut retourner ce qu'elle veut (par exemple la valeur de l'attribut inaccessible).

```
class MaClasse
{
  private $unAttributPrive;
  public function __set($nom, $valeur) { $this->$nom=$valeur; }
  public function __get($nom) { return $this->$nom; }
}
$obj = new MaClasse;
$obj->unAttributPrive = 'Simple test';
$obj->unAutreAttribut = 'Autre Simple test';
echo $obj->unAutreAttribut."<br>";
echo $obj->unAttributPrive;
```



IX. Les méthodes magiques

- **__toString**:
  - appelée lorsque l'objet est amené à être converti en chaîne de caractères.
  - Cette méthode doit retourner la chaîne de caractères souhaitée.

```
<?php
class MaClasse
{
  protected $texte;
  public function __construct($texte) { $this->texte = $texte; }
  public function __toString() {return $this->texte;}
}
$obj = new MaClasse('Hello world !');
// Solution 1 : le cast
$texte = (string) $obj;
var_dump($texte); // Affiche : string(13) "Hello world !".
// Solution 2 : directement dans un echo
echo $obj; // Affiche : Hello world !
```

IX. Les méthodes magiques

➢ **__isset** :

- appelée lorsque l'on appelle la fonction **isset** sur un attribut qui n'existe pas ou auquel on n'a pas accès. Retourne un booléen (TRUE ou FALSE)

```
class MaClasse
{
  private $unAttributPrive;
  public function __set($nom, $valeur) { $this->$nom=$valeur; }
  public function __get($nom) { return $this->$nom; }
  public function __isset($nom) {return isset($this->$nom);}
}
$obj = new MaClasse;
$obj->attribut = 'Simple test';
if(isset($obj->attribut))
    echo "L'attribut <b>attribut</b> existe";
else
    echo "L'attribut <b>attribut</b> n'existe pas";
```

IX. Les méthodes magiques

➢ **__unset** :

- appelée lorsque l'on tente d'appeler la fonction **unset** sur un attribut inexistant ou auquel on n'a pas accès.

```
class MaClasse
{
  private $unAttributPrive;
  public function __set($nom, $valeur) { $this->$nom=$valeur; }
  public function __get($nom) { return $this->$nom; }
  public function __isset($nom) {return isset($this->$nom);}
  public function __unset($nom) {if (isset($this->$nom)) unset($this->$nom);}
}
$obj = new MaClasse;
$obj->attribut = 'Simple test';
if(isset($obj->attribut)) echo "L'attribut <b>attribut</b> existe<br>";
else echo "L'attribut <b>attribut</b> n'existe pas<br>";
echo "<h3>Supression</h3>";
unset($obj->attribut);
if(isset($obj->attribut)) echo "L'attribut <b>attribut</b> existe<br>";
else echo "L'attribut <b>attribut</b> n'existe pas<br>";
```

IX. Les méthodes magiques

➢ **__call**:

- appelée lorsque l'on essayera d'appeler une méthode privée ou qui n'existe pas.
- prend deux arguments : le nom de la méthode et la liste des arguments (tableau).

```
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo "La méthode <b>$nom</b> a été appelée alors qu'elle n'existe pas! "
          . "Ses arguments étaient les suivants: <b>".implode($arguments, ', ')."</b>";
  }
}
$obj = new MaClasse;
$obj->methode(123, 'test');
```

La méthode **methode** a été appelée alors qu'elle n'existe pas! Ses arguments étaient les suivants: **123, test**

IX. Les méthodes magiques

- **__callstatic**: (semblable à __call)
  - appelée lorsqu'on appelle une méthode inexistante dans un contexte statique.
  - La méthode magique__callStatic doit obligatoirement être static.

```
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo "La méthode <b>$nom</b> a été appelée alors qu'elle n'existe pas! "
        . "Ses arguments étaient les suivants: <b>".implode($arguments, ', ')."</b><br>";
  }
  public static function __callStatic($nom, $arguments)
  {
    echo "La méthode <b>$nom</b> a été appelée dans un contexte statique alors qu'elle n'existe pas! "
        . "Ses arguments étaient les suivants: <b>".implode($arguments, ', ')."</b><br>";
  }
}
$obj = new MaClasse; $obj->methode(123, 'test');  MaClasse::methodeStatique(456, 'autre test');
```

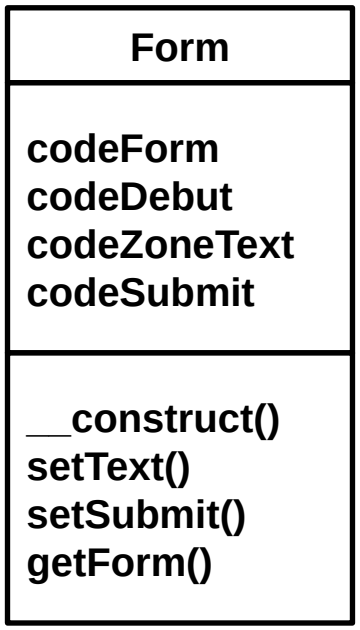
La méthode **methode** a été appelée alors qu'elle n'existe pas! Ses arguments étaient les suivants: **123, test**
La méthode **methodeStatique** a été appelée dans un contexte statique alors qu'elle n'existe pas! Ses arguments étaient les suivants: **456, autre test**

Exercice 1

1. Créer une classe nommée **Form** représentant un formulaire HTML.

Cette classe possède:

- Un constructeur pour créer le code du début du formulaire en utilisant les éléments <form> et <fieldset>.
- Une méthode setText() pour ajouter une zone de texte
- Une méthode setSubmit() pour ajouter un bouton d'envoi
- Une méthode getForm() qui retourne tout le code HTMl du formulaire



Les paramètres de ces méthodes doivent correspondre aux attributs des éléments HTML correspondants.

2.Créer 2 objets form, avec 2 zone de texte et un bouton d'envoi. Testez l'affichage obtenu.

Exercice 2

1. Créer une classe Voiture avec:
   - Les propriétés "marque" , "modele" et "puissance"
   - Un constructeur et un destructeur
   - Une méthode getInfo() qui permet de retourner les infos d'une voiture
   - Une méthode setModele() qui permet de modifier le modèle de la voiture

2.Créer un objet Voiture et utiliser l'ensemble des méthodes

3. Changer la classe Voiture en modifiant les méthodes getInfo() et setModele() par les méthodes magiques __set() et __get()

Exercice 3

1. Créer une classe abstraite Personne avec les propriétés "nom" et "prenon" une méthode abstraite getInfo() qui permet d'afficher les infos d'une personne

2.Créer 2 classes filles

- Enseignant: grade, salaire
- Etudiant: niveau, specialite

- Créer 2 objets $enseignant1 et $etudiant1 et tester la méthode getInfo()