

# Concepts importants de l'orienté objet avec PHP

C. BENSARI

# Rappels sur le concept de classe

- Une classe représente un type de donnée abstrait
- Une classe précise toutes les caractéristiques (attributs et méthodes) communes d'une famille d'objets
- Une classe permet de créer (instancier) des objets grâce à sa méthode particulière appelée «constructeur» (`__construct` en PHP)
- Les conventions de nommage des attributs et méthodes préconisent d'utiliser le principe du camelCase. Exemple : `$maVariable`, `maMethode` pour les caractéristiques de la classe et `MaClasse`, `MaPrincipaleClasse` pour les noms de classes
- Dans PHP et à l'inverse de certains langages comme Java, il n'est pas possible d'avoir deux méthodes avec le même nom au sein de la même classe (pas d'**overload**)

# Exemple d'une classe en PHP

```
<?php  
  
class Voiture {  
  
    public $marque;  
    public $modele;  
    public $anneeDuModele;  
  
    public function afficherDetail() {  
        // code d'implémentation de la méthode  
    }  
  
    public function marcheEnAvant() {  
        // code d'implémentation de la méthode  
    }  
  
    public function klaxonner() {  
        // code d'implémentation de la méthode  
    }  
}  
  
?>
```

N.B : Il est préférable (et j'exige) de nommer la classe avec une majuscule au début et que le nom du fichier porte le même nom de classe

# Rappels sur le concept d'encapsulation

- L'encapsulation consiste à masquer le détail d'implémentation d'un objet
- Elle permet de donner une vue externe (interface) de l'objet
- Cette interface présentera les services et informations accessibles depuis l'extérieur (utilisateurs de l'objet)
- Le principe d'encapsulation est applicable sur les attributs, sur les méthodes et même sur les classes
- En PHP, il existe trois niveaux d'encapsulation :
  - **public** : accessible depuis l'extérieur sans restrictions
  - **private** : accessible qu'à l'intérieur de la classe où est définie la caractéristique
  - **protected** : accessible uniquement dans la classe où est définie la caractéristique et ses classes filles

# Rappels sur le concept d'encapsulation

```
<?php
class Voiture {

    public $marque;
    protected $modele;
    private $anneeDuModele;

    function MarcheEnArriere(){
        // code d'implémentation de la méthode
    }

    public function afficherDetail(){
        // code d'implémentation de la méthode
    }


    protected function marcheEnAvant(){
        // code d'implémentation de la méthode
    }

    private function klaxonner(){
        // code d'implémentation de la méthode
    }

}
?>
```

**N.B :** Pour accéder aux caractéristiques d'un objet il suffit de placer une flèche « -> » entre le nom de la variable représentant l'objet et le nom de la caractéristique sans « \$ »

L'opérateur new permet d'instancier un objet d'une classe



```
<?php

$voiture = new Voiture();
// acces attributs
$voiture->marque; // Fonctionne
$voiture->modele; // erreur fatale
$voiture->anneeDuModele; // erreur fatale

//Acces méthodes
$voiture->marcheEnArriere(); // Fonctionne
$voiture->afficherDetail(); // Fonctionne
$voiture->marcheEnAvant(); // erreur fatale
$voiture->klaxonner(); // erreur fatale

?>
```

# Encapsulation : getters & setters

- En POO et avec la notion d'encapsulation, les attributs d'une classe doivent être déclarés privés.
- Pour pouvoir accéder à la valeur des attributs (depuis l'extérieur), des méthodes publiques appelées « Accesseurs » doivent être déclarées dans la classe :
  - **getters** : permet la récupération (lecture) de la valeur d'un attribut.
  - **setters** : permet de modifier la valeur de l'attribut

N.B : le mot clé **\$this** représente l'objet actuel exécutant la fonction  
getNom/setNom

```
class Employe {  
  
    private $nom;  
  
    // Permet de lire le contenu de l'attribut $nom  
    public function getNom() {  
        return $this->nom;  
    }  
  
    // Permet de modifier le contenu de l'attribut $nom  
    public function setNom($nouveauNom) {  
        $this->nom = $nouveauNom;  
    }  
  
}
```

# Rappel sur le concept d'héritage

- L'héritage est un mécanisme de transmission (héritage) des caractéristiques d'une classe (appelée classe mère ou classe parente) à une autre classe (appelée classe fille ou sous-classe)
- Grâce à l'héritage on peut éliminer les duplications de code en utilisant une hiérarchie de classes
- L'héritage peut être simple ou multiple selon les langages de programmation
- Avec une hiérarchie de classes, les classes filles peuvent être généralisées à leurs classes mères (directes ou indirectes) et ces classes mères peuvent être spécialisées à des classes filles (directes ou indirectes)
- En PHP, l'héritage est exprimé avec le mot clé « **extends** » qui suit le nom de classe fille et précède le nom de la classe mère

```
<?php
```

```
class Forme {  
  
    private $nombreAngles;  
    private $nombreCotes;  
    private $couleur;  
  
    public function description(){  
        // implémentation de la méthode ...  
    }  
  
    // public getters et setters  
}
```

```
?>
```

```
<?php
```

```
include_once('Form.php');  
  
class Point extends Forme {  
    private $coordonneeX;  
    private $coordonneeY;  
  
    public function deplacer($deltaX, $deltaY){  
        $this->$coordonneeX+= $deltaX;  
        $this->$coordonneeY+= $deltaY;  
    }  
    // public getters et setters  
  
    public function description(){  
        // implémentation de la méthode ...  
    }  
}
```

```
?>
```

```
<?php
```

```
include_once('Point.php');
```

```
$point = new Point();  
// acces attributs  
$point->couleur; // erreur fatale car $couleur est privé  
$point->coordonneeX; // erreur fatale car $couleur est privé  
$point->coordonneeY; // erreur fatale car $couleur est privé
```

```
//Acces méthodes  
$point->description();  
$point->getCoordonneeX(); // fonctionne  
$point->getCoordonneeY(); // fonctionne  
$point->getCouleur(); //fonctionne
```

```
?>
```



# Polymorphisme :

```
<?php

include ('Form.php');
include ('Point.php');
include ('Rectangle.php');

$point = new Point();
$rectangle = new Rectangle();

// affiche: description: Je suis un Point !
afficherDescription($point);

// affiche: Je suis une forme !
$rectangle->description();

function afficherDescription(Forme $forme){

    echo "description: ";
    $forme->description();
}
```

```
<?php

class Forme {

    private $nombreAngles;
    private $nombreCotes;
    private $couleur;

    // méthode dans Forme
    public function description(){
        echo "Je suis une forme !";
    }

    // public getters et setters
}
```

```
?>

<?php

include ('Form.php');

class Rectangle extends Forme {}

?>
```

```
<?php

include_once('Form.php');

class Point extends Forme {
    private $coordonneeX;
    private $coordonneeY;

    public function deplacer($deltaX, $deltaY){
        $this->$coordonneeX+= $deltaX;
        $this->$coordonneeY+= $deltaY;
    }

    // public getters et setters
    // méthode dans Point qui redéfinit la méthode héritée de Forme
    public function description(){
        echo "Je suis un Point !";
    }
}
```

# Méthodes et attributs static

- Les méthodes et attributs statiques sont des éléments d'une classe et non pas d'instances (pas besoin d'instancier un objet de la classe pour pouvoir y accéder)
- Pour déclarer un attribut/méthode statique il suffit de placer le mot clé « **static** » juste après le mot clé définissant la visibilité de l'attribut/méthode
- Les éléments statiques sont souvent utilisé pour effectuer des traitements techniques
- Pour accéder à un élément statique de la classe il faut utiliser le nom de la classe, suivi de « **::** » et ensuite le nom de l'élément statique
- Au sein de la même classe, un élément statique est accessible avec le mot clé « **self** » suivi de « **::** » et ensuite le nom de l'élément statique

# Méthodes et attributs static

```
<?php
// --- Geometrie.php
class Geometrie
{
    const PII = 3.14; // --- Une constante
    public static $surfaceCercle; // --- Une propriété statique

    // --- Une méthode statique
    public static function calculerSurfaceCercle($rayon)
    {
        return self::$surfaceCercle = $rayon * $rayon * self::PII;
    }

    // --- Une méthode statique
    public static function calculerPerimetreCercle($rayon)
    {
        return $rayon * 2 * self::PII;
    }
}

echo Geometrie::calculerSurfaceCercle(5); // affiche 78.5
echo Geometrie::calculerPerimetreCercle(5) // affiche 31.4
?>
```

# Les opérateurs de résolution de portée **self**, **parent** et **::**

- **self** : accès aux méthodes/attributs statique au sein de la même classe
- **parent** : accès aux éléments de la classe parente
- **::** : peut être utilisé pour accéder aux éléments statiques ou avec **parent** pour accéder aux éléments de la classe parente

N.B : pour accéder à un attribut statique de la classe parente, garder le « \$ » de l'attribut après le « :: »

**Exemple :** `parent::$prenom`

# Méthodes et classes abstraites

- Une classe peut avoir une méthode « abstraite » qui n'a aucune implémentation
- Ceci peut être très utile pour la factorisation du code en utilisant les notions d'héritage et de généralisation
- En POO, une classe contenant une méthode abstraite est forcément une classe abstraite
- Une classe abstraite n'est pas instanciable (pas d'opérateur new sur cette classe)
- Pour déclarer une méthode abstraite dans une classe il suffit d'utiliser le mot clé « **abstract** » juste avant le mot clé définissant la visibilité de la méthode (public, private ou protected) si il existe ou avant le mot clé « **function** »
- Le mot clé « abstract » doit aussi précéder le mot clé « **class** »
- L'implémentation d'une méthode abstraite n'est faite que dans une classe héritant de la classe abstraite

# Méthodes et classes abstraites

```
<?php
abstract class Forme {
    private $nombreDeCotes;

    // pas de corps de la méthode
    abstract public function description();

    public function __construct($nbrCotes) {
        $this->nombreDeCotes = $nbrCotes;
    }
}

?>
```

```
<?php
include_once('Forme.php');

class Point extends Forme{
    private $coordonneeX;
    private $coordonneeY;

    public function __construct(int $nbrCotes,
                                int $coordX,
                                int $coordY){
        parent::__construct($nbrCotes);
        $this->coordonneeX = $coordX;
        $this->coordonneeY = $coordY;
    }

    // implémentation obligatoire ou mettre
    // la fonction à abstract à nouveau sinon Fatal Error
    public function description(){
        echo "Je suis un point concret !"
    }
}
```

```
<?php
include_once('Forme.php');
include_once('Point.php');

$forme = new Forme(4); // Fatal Error : cannot instantiate an abstract class
$point = new Point();

?>
```

# Les interfaces

- L'objectif des interfaces est de définir un protocole de comportements commun à plusieurs classes
- Une interface (ou contrat) est un ensemble de spécifications définies à travers des méthodes abstraites et publiques
- Ces méthodes doivent être obligatoirement implémentées par les classes implémentant cette interface
- Une interface ne doit avoir que des signatures de méthodes et des constantes (pas d'attributs comme le cas des classes)
- Une classe peut implémenter plusieurs interfaces ce qui permet de palier l'absence de l'héritage multiple en PHP

# Les Interfaces : Exemple

```
interface Forme {  
  
    public const PII = 3.14;  
  
    public function calculerSurface();  
  
    public function calculerPerimetre();  
}
```

```
class Rectangle implements Forme {  
  
    private $largeur, $hauteur;  
  
    public function calculerSurface(){  
        return $this->largeur * $this->hauteur;  
    }  
  
    public function calculerPerimetre()  
    {  
        return ($this->hauteur + $this->largeur)*2;  
    }  
}
```

```
$cercle = new Cercle();  
$cercle->setRayon(5);  
echo $cercle->calculerPerimetre();
```

```
class Carre implements Forme {  
  
    private $largeur;  
  
    public function calculerSurface(){  
        return $this->largeur * $this->largeur;  
    }  
  
    public function calculerPerimetre()  
    {  
        return $this->largeur*4;  
    }  
}
```

```
class Cercle implements Forme {  
  
    private $rayon;  
  
    public function calculerSurface(){  
        return ($this->rayon * $this->rayon) * 2;  
    }  
  
    public function calculerPerimetre()  
    {  
        return 2 * self::PII * $this->rayon;  
    }  
}
```



# Classes et méthodes « final »

- Une classe « final » est une classe qui ne peut pas être héritée
- La création d'une classe B qui hérite d'une classe « final » A envoie une erreur fatale :  
*Class B may not inherit from final class « A »*
- Pour créer une classe finale, il suffit de placer le mot clé « final » devant le mot clé « class » :

**final** class Personn {....}

- Une méthode « final » est une méthode qui ne peut pas être **redéfinie** dans une classe fille (pas d'overriding)
- Pour créer une méthode finale, il suffit de placer le mot clé « final » en tout début de la signature de la méthode

**final** public function maFonction(..){..}