

# Technologies du Web

- POO en PHP
- PDO en PHP (PHP Data Objects)
- PHP en MVC

**Pr. J. ANTARI**

**Département: Mathématiques et Informatique**

# Technologies du Web

## POO en PHP

**Pr. J. ANTARI**

**Département: Mathématiques et Informatique**

1 Introduction

2 Classe et Objet

3 Setter & Getter

4 Méthode et attribut

5 Encapsulation

6 Constructeur & Destructeur

7 Héritage

## **Introduction**

Avant la version 5, PHP était loin d'être un langage de programmation orientée objet (POO), en comparaison de Java ou de C++.

Les concepteurs de PHP 5 ont dû effectuer une refonte totale du modèle objet très sommaire de PHP 4 pour le rendre plus proche de celui de Java.

PHP 5 fournit néanmoins désormais les outils nécessaires à ceux qui souhaitent choisir cette orientation. La manipulation d'objets n'est pas une obligation dans la plupart des cas mais pourrait devenir une nécessité pour de gros projets.

### **Pour la POO, elle y a :**

- La modularité ;
- Encapsulation
- Héritage
- Polymorphisme

## La Classe

Une classe est une description des caractéristiques d'un ou de plusieurs objets.

**Elle est composé de :**

**Attribut :** Une donnée propre à une classe.

**Méthode :** Une fonction propre à une classe

La syntaxe de base pour définir une classe ressemble à ceci :

```
class Nomclass{  
.....  
}
```

## Objet

Un objet est un représentant de la classe à partir de laquelle il est créé. On dit qu'il est une instance de cette classe. L'objet créé a des propriétés correspondant aux variables de la classe et des méthodes qui correspondent aux fonctions de la classe.

Pour créer de **nouvelles instances** de cette classe, la syntaxe suivante est utilisée :

```
$objet_1= new Nomclass() ;
```

## Exemple

//Fichier **Article.php**

**<?php**

class Article

{

    //@var string \$ reference reference de produit

    private \$reference ;

    //@var string \$ designation nom de produit

    private \$designation ;

    //@var string \$ description description de produit

    private \$description ;

}

**?>**

## Exemple

//Fichier index.php

<?php

//Inclusion de fichier classe

require\_once "../src/App/Entity/Article.php" ;

//Création d'un objet Instanciation

//instance =objet

\$article\_1= new Article() ;

\$article\_2= new Article() ;

var\_dump(\$article\_1) ;

var\_dump(\$article\_2) ;

?>

## Visibilité des propriétés et méthodes

Pour appeler une propriétés Un membre doit être déclaré soit :

**public** :visible par tous, donc de l'extérieur

**protected** :visible par la classe et les héritiers

**private** :visible uniquement par la classe

### Remarque :

Le faite de déclarer un attribut en public on peut lui aecter une valeur sans aucun contrôle.

Pour appeler un attribut ou une méthode on utilise le : ->



## Visibilité des propriétés et méthodes: Exemple

<?php

```
//Inclusion de fichier classe  
require_once "../src/App/Entity/Article.php" ;  
//Création d'un objet Instanciation  
//instance =objet  
$article_1= new Article() ;  
$article_1->reference='GFD123' ;  
echo $article_1->reference ;  
echo "<br>" ;  
var_dump($article_1) ;
```

?>

## Mutateur :Setter

Il permet d'affecter une valeur à une propriété d'un objet

### Exemple :

```
// dans la classe Article.php  
public function setReference($reference)  
{  
    $this->reference = $reference ;  
}
```

**Dans index.php on fait l'appelle de la manière suivante :**

```
$article_1->setReference('AAAAAA') ;
```

**\$this représente l'objet courant**

## Exemple

En faisant un contrôle sur la référence :elle ne doit pas dépasser 4 caractère.

```
public function setReference($reference)
{
    if (strlen($reference) > 4)
    {
        echo 'La référence' . $reference . 'dépasse 4 caractères' ;
    }
    else
    {
        $this->reference = $reference ;
    }
}
```

## Accesseur :Getter

Il permet de lire la valeur d'une propriété d'un objet.

### **Exemple :**

```
// dans la classe Article.php  
public function getReference()  
{  
    return $this->reference ;  
}
```

**Dans index.php on fait l'appelle de la manière suivante :**

```
echo $article_1->getReference() ;
```

## Méthodes d'objet

Les méthodes (c-à-d les fonctions associées à un objet) ont la syntaxe suivante :

```
public function MaMethode($Paramètre){  
    //instruction  
}
```

Appeler une méthode d'un objet déjà instancié est similaire à l'accès à une propriété de l'objet, il sut d'ajouter les parenthèses :

**\$objet->MaMethode(\$Paramètre) ;**

## Méthodes d'objet: Exemple

### dans Article.php

```
public function augmentePrix($quotient)
{
    $this->prix=$this->prix * (1 + $quotient/100) ;
}
public function reduitPrix($quotient)
{
    $this->prix=$this->prix * (1 - $quotient/100) ;
}
```

### dans index.php

```
$article_1->setPrix(100) ;
$article_1->augmentePrix(5) ;
```

# Méthodes et attribut

## Variable de Classe

Une variable /attribut ou propriété de classe désigne une caractéristique commune à tous les objets. Elle existe même si aucun objet n'a été créé. Pour la créer on utilise le mot clé **static**.

Exemple :

```
public static $remise ;
```

On peut l'appeler dans index.php par :

```
echo Article : :$remise ;
```

Si on affiche un objet avec `var_dump()` cette variable ne sera pas affichée.

## Méthode de Classe

C'est une méthode qui peut être appelée même si on a pas instancié un objet, pour la déclarer on utilise **static**.

### Exemple :

```
public static function getRemise()  
{  
    return self : :$remise ;  
}  
public static function setRemise($remise)  
{  
    self : :$remise = $remise ;  
}
```



## Méthode de Classe: Exemple

Dans le fichier **Article.php**

```
public static function isPositive($price){  
    if($price >=0){  
        return true ;  
    }  
    return false ;  
}
```

Dans le fichier **index.php**

```
//test de la méthode de classe  
$prixPourArticle_1 = -250 ;  
$prixOk = Article : :isPositive($prixPourArticle_1) ;  
if ($prixOk){  
    $article_1->setPrix($prixPourArticle_1) ;  
} else { echo '<br>' ; echo 'le prix proposé n'est pas positif ' ; }
```

## Les Constantes

Une constante est comme une propriété de classe accessible en lecture seule, sa valeur ne sera jamais modifiée ni à l'extérieur, ni à l'intérieur de la classe.

Une constante est toujours public et elle est toujours **static**.

### Exemple :

```
// déclaration d'une constante
```

```
const REMISE_MAX=30 ;
```

```
//Appel de la classe
```

```
echo Article : :REMISE_MAX ;
```

# Encapsulation

L'encapsulation est le procédé qui permet de séparer clairement l'interface (partie publique) de l'objet de son implémentation (partie privée).

L'idée est de masquer tout ce qui pourrait rendre les autres objets dépendants (principe de faible couplage entre objets).

# Constructeur

Un Constructeur est une méthode spéciale qui sera automatiquement appelée au moment de l'instanciation d'un Objet, il a comme nom le mot **\_\_construct()**.

Un constructeur est méthode qui doit être publique.

## Exemple :

```
public function __construct()  
{  
    $this->datecreation =new DateTime('now') ;  
}
```

## Constructeur: Exemple

Dans le fichier Client.php

```
public function __construct($nom,$prenom)
{
    $this->nom = $nom ;
    $this->prenom = $prenom ;
    $this->datecreation =new DateTime('now') ;
}
```

**Dans le fichier index.php**

```
$clientA = new Client('Jawahri','Adil') ;
```

## Destrueteur

Un destruteur est une méthode spéciale qui sera appelée automatiquement à la fin du script, il a comme nom `__destruct()`.  
Un destruteur peut être appelé avec la méthode `unset()`

### Exemple:

**Dans le fichier Client.php**

```
public function __destruct()  
{  
    echo "<p> L'objet de type Client '. $this->getNom().' a été  
    détruit" ;  
}
```

**Dans le fichier index.php**

```
$clientA = new Client('Jawahri','Adil') ;  
unset(clientA) ;  
echo "ceci est la fin du script" ;
```

# Les méthodes chaînées

C'est un concept de la programmation Orientée Objet qui consiste à faire un appel successive des méthodes d'un Objet an de créer un code très lisible est structuré. Pour permettre l'enchainement des méthodes, elles doivent retourner un **\$this** .

## Exemple :

```
$Client->setnom('Jawhari')  
->setPrenom('Adil')  
->setCode('J2526') ;
```

# Héritage

L'héritage est une technique puissante en POO qui permet à une classe d'utiliser les propriétés et les méthodes d'une autre classe dite mère.

Quand on parle d'héritage, c'est qu'on dit qu'une classe B hérite d'une classe A. La classe A est donc considérée comme la classe mère et la classe B est considérée comme la classe fille.

## Exemple en PHP

Pour mettre en pratique l'héritage , il sut d'utiliser le mot-clé **extends**. Vous déclarez votre classe comme d'habitude (**class** MaClasse) en ajoutant extends NomDeLaClasseAHeriter comme ceci :

```
<?php class Personne // Création d'une classe simple.  
{  
}  
  
class Etudiant extends Personne // Notre classe Etudiant hérite  
//des attributs et méthodes de Personne.  
{  
}
```



# Technologies du Web

## PDO en PHP

PHP Data Objects

**Pr. J. ANTARI**

**Département: Mathématiques et Informatique**

1 Introduction

2 La classe PDO

3 La classe PDOStatement

4 Exemple

## Qu'est ce que PDO

- **PDO : PHP Data Objects**
- Extension PHP fournissant une interface pour accéder à une base de données
- Fournit une interface d'abstraction pour l'accès aux données
- Ne fournit PAS une abstraction de base de données
  - SQL spécifique au moteur
  - Fonctionnalités présentes / absentes
- Interface orientée objet

## Pourquoi PDO?

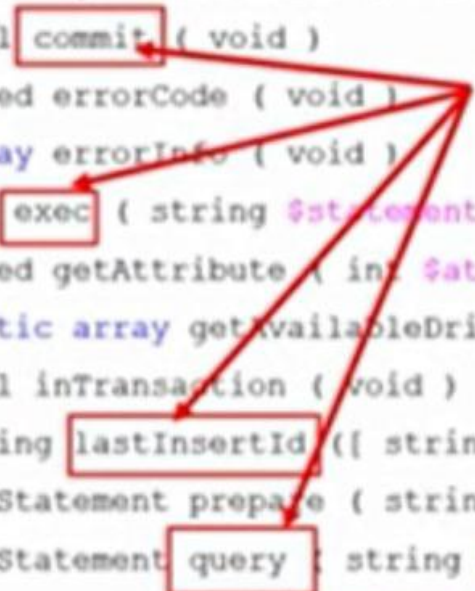
PDO fournit une interface d'abstraction à l'accès de données, ce qui signifie que vous utilisez les mêmes fonctions pour exécuter ou récupérer les données quelle que soit la base de données utilisée.

# La classe PDO

## La classe PDO <http://fr2.php.net/manual/fr/class.pdo.php>

### Synopsis de la classe :

```
PDO {  
    public __construct ( string $dsn [, string $username [, string $password [, array  
$driver_options ]]] )  
    public bool beginTransaction ( void )  
    public bool commit ( void )  
    public mixed errorCode ( void )  
    public array errorInfo ( void )  
    public int exec ( string $statement )  
    public mixed getAttribute ( int $attribute )  
    public static array getAvailableDrivers ( void )  
    public bool inTransaction ( void )  
    public string lastInsertId ([ string $name = NULL ] )  
    public PDOStatement prepare ( string $statement [, array $driver_options = array() ] )  
    public PDOStatement query ( string $statement )  
    public string quote ( string $string [, int $parameter_type = PDO::PARAM_STR ] )  
    public bool rollBack ( void )  
    public bool setAttribute ( int $attribute , mixed $value )  
}
```



Nom des méthodes utilisables

Le rôle de chacune de ces méthodes est expliqué dans la documentation :  
<http://php.net/manual/fr/class.pdo.php>

- [PDO::beginTransaction](#) — Démarre une transaction
- [PDO::commit](#) — Valide une transaction
- [PDO::\\_\\_construct](#) — Crée une instance PDO qui représente une connexion à la base
- [PDO::errorCode](#) — Retourne le SQLSTATE associé avec la dernière opération sur la base de données
- [PDO::errorInfo](#) — Retourne les informations associées à l'erreur lors de la dernière opération sur la base de données
- [PDO::exec](#) — Exécute une requête SQL et retourne le nombre de lignes affectées
- [PDO::getAttribute](#) — Récupère un attribut d'une connexion à une base de données
- [PDO::getAvailableDrivers](#) — Retourne la liste des pilotes PDO disponibles
- [PDO::inTransaction](#) — Vérifie si nous sommes dans une transaction
- [PDO::lastInsertId](#) — Retourne l'identifiant de la dernière ligne insérée ou la valeur d'une séquence
- [PDO::prepare](#) — Prépare une requête à l'exécution et retourne un objet
- [PDO::query](#) — Exécute une requête SQL, retourne un jeu de résultats en tant qu'objet PDOStatement
- [PDO::quote](#) — Protège une chaîne pour l'utiliser dans une requête SQL PDO
- [PDO::rollBack](#) — Annule une transaction
- [PDO::setAttribute](#) — Configure un attribut PDO

## La classe PDO: méthode query

Lorsque l'on souhaite exécuter une requête de type SELECT, il faut utiliser la méthode "query"

**public PDOStatement query ( string \$statement )**

Si l'on utilise la méthode "query", nous aurons pour valeur de retour un objet **PDOStatement** (donc il ne s'agit pas du résultat de la requête).

# Préparation d'une requête

## **PDOStatement PDO ::prepare(string statement [, array driver\_options])**

- statement : la requête à préparer. Peut contenir des paramètres anonymes ( ? ) ou nommés ( :nom )
- driver\_options : tableau d'options du driver
- retourne un objet PDOStatement qui effectuera l'association des paramètres et exécutera la requête

### **Exemple :**

```
$pdo=new PDO("mysql :host=localhost ;dbname=mysql") ;  
$pdostat = $pdo->prepare( "SELECT * FROM user WHERE User= ?" ) ;
```

## Intérêt des requêtes préparées

- Optimisation des performances pour des requêtes appelées plusieurs fois
- Émulation faite par PDO si le driver ne les supporte pas nativement
- Protection automatique des valeurs des paramètres pour interdire les attaques par injection de code SQL



# La classe PDOStatement

## La classe PDOStatement

### Synopsis de la classe :

```
PDOStatement implements Traversable {  
    /* Propriétés */  
    readonly string $queryString;  
    /* Méthodes */  
    public bool bindColumn ( mixed $column , mixed &$amp;param [, int $type [, int $maxlen [, mixed $driverdata ]]] )  
    public bool bindParam ( mixed $parameter , mixed &$amp;variable [, int $data_type = PDO::PARAM_STR [, int $length [, mixed $driver_options ]]] )  
    public bool bindValue ( mixed $parameter , mixed $value [, int $data_type = PDO::PARAM_STR ] )  
    public bool closeCursor ( void )  
    public int columnCount ( void )  
    public void debugDumpParams ( void )  
    public string errorCode ( void )  
    public array errorInfo ( void )  
    public bool execute ([ array $input_parameters ] )  
    public mixed fetch ([ int $fetch_style [, int $cursor_orientation = PDO::FETCH_ORI_NEXT [, int $cursor_offset = 0 ]]] )  
    public array fetchAll ([ int $fetch_style [, mixed $fetch_argument [, array $ctor_args = array() ]]] )  
    public string fetchColumn ([ int $column_number = 0 ] )  
    public mixed fetchObject ([ string $class_name = "stdClass" [, array $ctor_args ] ] )  
    public mixed getAttribute ( int $attribute )  
    public array getColumnMeta ( int $column )  
    public bool nextRowset ( void )  
    public int rowCount ( void )  
    public bool setAttribute ( int $attribute , mixed $value )  
    public bool setFetchMode ( int $mode )  
}
```

A partir d'un objet de type PDOStatement, il est possible d'appeler toutes les méthodes publiques de la classe PDOStatement

Par exemple, si nous souhaitons récupérer tous les résultats de la requête SELECT en une fois, nous pouvons utiliser la méthode « fetchAll »

Ou encore connaître le nombre de résultats de la requête avec « rowCount »

# La classe PDOStatement: Documentation

Le rôle de chacune de ces méthodes est expliqué dans la documentation :  
<http://fr2.php.net/manual/fr/class.pdostatement.php>

le d'appeler

- [PDOStatement::bindColumn](#) — Lie une colonne à une variable PHP
- [PDOStatement::bindParam](#) — Lie un paramètre à un nom de variable spécifique
- [PDOStatement::bindValue](#) — Associe une valeur à un paramètre
- [PDOStatement::closeCursor](#) — Ferme le curseur, permettant à la requête d'être de nouveau exécutée
- [PDOStatement::columnCount](#) — Retourne le nombre de colonnes dans le jeu de résultats
- [PDOStatement::debugDumpParams](#) — Détaille une commande préparée SQL
- [PDOStatement::errorCode](#) — Récupère les informations sur l'erreur associée lors de la dernière opération sur la requête
- [PDOStatement::errorInfo](#) — Récupère les informations sur l'erreur associée lors de la dernière opération sur la requête
- [PDOStatement::execute](#) — Exécute une requête préparée
- [PDOStatement::fetch](#) — Récupère la ligne suivante d'un jeu de résultats PDO
- [PDOStatement::fetchAll](#) — Retourne un tableau contenant toutes les lignes du jeu d'enregistrements
- [PDOStatement::fetchColumn](#) — Retourne une colonne depuis la ligne suivante d'un jeu de résultats
- [PDOStatement::fetchObject](#) — Récupère la prochaine ligne et la retourne en tant qu'objet
- [PDOStatement::getAttribute](#) — Récupère un attribut de requête
- [PDOStatement::getColumnMeta](#) — Retourne les métadonnées pour une colonne d'un jeu de résultats
- [PDOStatement::nextRowset](#) — Avance à la prochaine ligne de résultats d'un gestionnaire de lignes de résultats multiples
- [PDOStatement::rowCount](#) — Retourne le nombre de lignes affectées par le dernier appel à la fonction `PDOStatement::execute()`
- [PDOStatement::setAttribute](#) — Définit un attribut de requête
- [PDOStatement::setFetchMode](#) — Définit le mode de récupération par défaut pour cette requête

# Association des paramètres d'une requête

**bool PDOStatement : :bindValue(mixed parameter, mixed value [, int data\_type])**

- parameter : le paramètre (nom ou position [1. . . n])
- value : sa valeur
- data\_type : le type de la valeur
  - PDO : :PARAM\_BOOL booléen.
  - PDO : :PARAM\_NULL NULL SQL.
  - PDO : :PARAM\_INT INTEGER SQL.
  - PDO : :PARAM\_STR CHAR, VARCHAR ou autre chaîne.
  - PDO : :PARAM\_LOB "objet large" SQL.

**bool PDOStatement : :execute([array parameters])**

parameters : tableau associatif ou indexé des valeurs

## Exemple de selection

-> On commence par instancié un objet PDO :

```
$objetPDO = new PDO
```

```
('mysql:host=localhost ;dbname=FPT','root','') ;
```

->On utilise une méthode de la classe PDO :

```
$pdostat = $objetPDO->query('SELECT * from etudiant') ;
```

->Récupérer les résultats de la requête :

```
$toutesleslignes=$pdostat->fetchall() ;
```

Avec \$toutesleslignes est un tableau array.

Acher le nom de l'étudiant de la première ligne :

```
echo $toutesleslignes[0]->nom. ;
```

# **PHP en MVC**

1 Introduction

2 Modèle

3 La Vue

4 Contrôleur

5 Schéma MVC

6 Exemple

# 1 Introduction

Il y a des problèmes en programmation qui reviennent tellement souvent qu'on a créé toute une série de bonnes pratiques que l'on a réunies sous le nom de design patterns. Un des plus célèbres design patterns s'appelle MVC (Modèle - Vue - Contrôleur). Le pattern MVC permet de bien organiser son code source. Il va nous aider à savoir quels fichiers créer, mais surtout à définir leur rôle. Le but de MVC est justement de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts



## Principe

Principe de conception d'applications (WEB ou pas) qui sépare les fonctions nécessaires en trois composants distincts :

**Modèle** -> Gère les données

**Vue Gère** -> la présentation (UI)

**Contrôleur** -> Agit

Il existe de nombreux framework PHP basés sur cette architecture.

