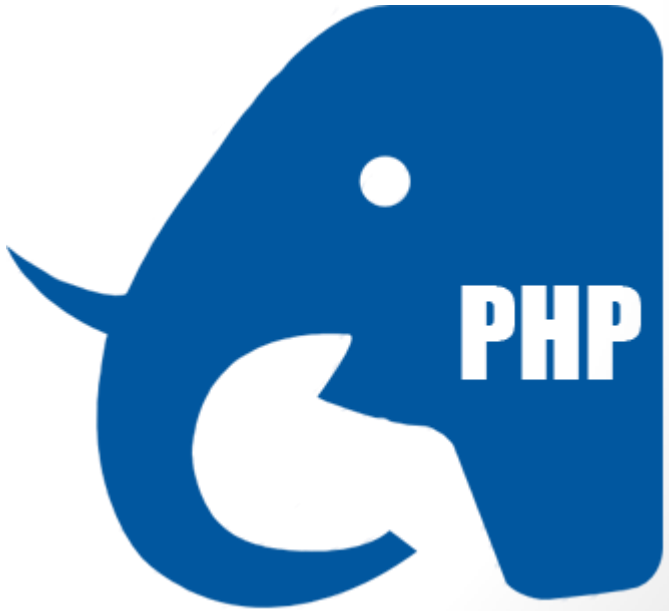
Formation Web Niveau 2 : PHP5/PHP7

Radhia Methni

Différence entre php4 et php5

PHP5, a supprimé des fonctions venant de PHP4, en a implémenté des nouvelles et apporte une nouvelle conception de codage avec la POO (programmation orientée objet) mais il n'est pas obligatoire de développer en POO avec php5, il est naturellement possible de continuer de développer en procédurale.



- La POO (programmation orientée objet) est une forme particulière de programmation destinée à faciliter la maintenance et la réutilisation / adaptation de vos scripts PHP.
- L'idéal c'est qu'un développeur (de classe) puisse transmettre son code avec une documentation et qu'un autre développeur (utilisateur de ces classes) puisse les utiliser sans avoir à replonger de manière approfondie dans le code de chacune d'entre elles.
- Un objet, dans ce modèle, est défini comme une "entité" qui a des propriétés, et un comportement.
  - ✓ La POO encourage le travail collaboratif.
  - ✓ La POO simplifie la maintenance.
  - ✓ La POO assouplit le code

Qu’est-ce qu’une classe

Les classes sont présentes pour « fabriquer » des objets. En programmation orientée objet, un objet est créé sur le modèle de la classe à laquelle il appartient.

Exemple

Prenons l'exemple le plus simple du monde : les gâteaux et leur moule.

Le moule, il est unique. Il peut produire une quantité infinie de gâteaux.

Dans ces cas-là, les gâteaux sont les objets et le moule est la classe.

La classe est présente pour produire des objets.

Elle contient le plan de fabrication d'un objet et nous pouvons nous en servir autant que nous le souhaitons afin d'obtenir une infinité d'objets.

Différence entre une classe et un objet

La classe est un plan, une description de l'objet.

Sur le plan de construction d’une voiture nous y retrouverons le **moteur** ou encore la **couleur** de la carrosserie.

L'objet est une application concrète du plan.

L'objet est la voiture.

Nous pouvons créer plusieurs voitures basées sur un plan de construction.

- Nous pouvons donc créer plusieurs objets à partir d'une classe.

Instance

Une instance de classe est le fait de créer un objet de cette classe.

Une fois l'objet créé, nous pourrons appeler les méthodes qui composent notre classe.

Représentation

Nous pouvons donc imaginer les méthodes et les attributs de nos objets.

Les méthodes pourraient être : **démarrer**(), **rouler**(), **s'arrêter**(), etc.

Les attributs pourraient être : **couleur**, **poids**, **taille**, etc.

- Le plan de construction est unique, il peut produire une quantité infinie de voitures.

➢ Créer une classe

La classe renferme l'ensemble des propriétés et de méthodes qui servent à définir l'identité de l'objet qui en découlera (l'instance de classe).

Supposons que l'on souhaite créer une classe "voiture". Celle ci aura certaines propriétés (attributs) comme: couleur, puissance et vitesse. Elle aura aussi des méthodes comme accélérer et ralentir. Notre classe "voiture" renfermera tous ces éléments (membres).

Pour créer la classe "voiture" on écrira le code suivant:

```php
<?php
    class Voiture{
        // Déclaration des membres
    }
?>
```

❖ Déclaration des attributs

Les attributs sont les variables membres de la classe. Ils constituent les propriétés ou les caractéristiques de l'objet (l'instance de classe) qui en sera né.

Pour déclarer un attribut il faut le précéder par sa **visibilité**. La visibilité d'un attribut indique à partir d'où on peut en avoir accès. Il existe trois types de visibilité:

- ✓ **public**: dans ce cas, l'attribut est accessible de partout.
- ✓ **private**: dans ce cas, l'attribut est accessible seulement de l'intérieur de la classe.
- ✓ **protected**: dans ce cas, l'attribut est accessible seulement de l'intérieur de la classe dont il est membre ainsi que de l'intérieur des classes fille qui héritent de cette classe .

```php
<?php
    class Voiture{
        private $couleur;
        private $puissance;
        private $vitesse;
    }
?>
```

❖ Déclaration des méthodes

Pour déclarer une méthode il suffit de procéder comme si on déclarait une fonction en PHP en la précédant par la visibilité qui peut être soit **public** ou **private**.

Une méthode publique est accessible de partout. Une méthode privée est accessible seulement de l'intérieur de la classe.

```php
<?php
    class Voiture{
        private $couleur;
        private $puissance;
        private $vitesse;
        public function accelerer(){
            // Corps de la méthode accelerer()
        }
        public function ralentir(){
            // Corps de la méthode ralentir()
        }
    }
?>
```

Instanciation de classes et manipulation d'objets

Instancier une classe

On appelle l'opération qui consiste à créer un objet instanciation et l'objet ainsi créé peut aussi être appelé instance de classe.

```php
<?php
    class Voiture{
        private $couleur="Rouge";
        private $puissance;
        private $vitesse;
        public function accelerer(){
            echo "Appel de la méthode accelerer()";
        }
        public function ralentir(){
            echo "Appel de la méthode ralentir()";
        }
    }
?>
```

J'ai pris le soin d'initialiser l'attribut $couleur et mettre quelque chose dans les deux méthodes accelerer() et ralentir().

Supposons qu'on veut créer un objet du nom de $citadine à partir de la classe Voiture. L'instanciation de la classe ressemblerait à ceci:

```php
<?php
    $citadine = new Voiture();
?>

```

Nous avons désormais créé l'objet **$citadine** qui dispose de ses propres attributs $couleur, $puissance, et $vitesse et les méthodes accelerer() et relentir().

Autrement dit, si nous créons un autre objet à partir de la même classe, il disposera également de ses propres attributs et méthodes du même nom.

❖ Appel d'attributs et méthodes

Après avoir créé l'objet, on peut désormais accéder à ses membres (attributs et méthodes) par le biais de l'opérateur -> (un tiret suivi d'un chevron fermant).

Voici un exemple:

```php
<?php
    class Voiture{
        private $couleur="Rouge";
        private $puissance;
        private $vitesse;
        public function accelerer(){
            echo "Appel de la méthode accelerer()";
        }
        public function ralentir(){
            echo "Appel de la méthode ralentir()";
        }
    }

    $citadine = new Voiture();

    $citadine -> accelerer();
    echo $citadine -> couleur;
?>
```

Ce qui donne:

```
Appel de la méthode accelerer()
Fatal error: Cannot access private property Voiture::$couleur in index.php on line 17
```

- La première ligne de l'exécution est logique. On a pu appeler la méthode accelerer() depuis l'extérieur de la classe car elle est publique. Ce qui affiche le message "Appel de la méthode accelerer()".
- La deuxième ligne par contre affiche l'erreur "Cannot access private property" qui signifie que l'on peut pas accéder à un attribut privé de l'extérieur de la classe, ce qui est tout à fait logique.
- Si l'attribut $couleur était public, alors le message "Rouge" serait affiché à la place de l'erreur.

❖ Constructeur et destructeur

Le constructeur d'une classe

Le constructeur d'une classe est une méthode publique (dans la plupart des cas). Elle est appelée automatiquement au moment de l'instanciation.

Elle sert généralement à initialiser les attributs et fournir à l'objet créé tout ce dont il a besoin pour fonctionner.

Le constructeur est une méthode qui a le même nom que la classe. Cependant, en PHP on préfère l'identifier par le mot clé **__construct()**(avec deux sous-tirets au début).

```php
<?php
    class Voiture{
        private $couleur;
        private $puissance;
        private $vitesse;
        public function __construct($couleur,$puissance){
            $this->couleur=$couleur;
            $this->puissance=$puissance;
            $this->vitesse=0;
        }
        public function accelerer(){
            echo "Appel de la méthode accelerer()";
        }
        public function ralentir(){
            echo "Appel de la méthode ralentir()";
        }
    }

    $citadine = new Voiture("Rouge",130);
?>
```

Le destructeur d'une classe :

Le destructeur et une méthode publique identifiée par le nom __destruct(). Il est appelé automatiquement par le compilateur lorsqu'il n'y a plus aucune référence à l'objet en cours. Autrement dit, le destructeur est appelé quand il n'y a plus aucun appel d'un membre quelconque de l'objet.

Exemple:

```php
<?php
    class Voiture{
        private $couleur;
        private $puissance;
        private $vitesse;
        public function __construct(){
            $this->couleur="Rouge";
            $this->puissance=130;
            $this->vitesse=0;
        }
        public function __destruct(){
            echo "Couleur de la voiture: ".$this->couleur;
        }
        public function accelerer(){
            echo "Appel de la méthode accelerer()";
        }
        public function ralentir(){
            echo "Appel de la méthode ralentir()";
        }
    }

    $citadine = new Voiture();
?>
```

L'héritage

Concept de l'héritage

L'héritage est un concept fondamental de la POO. C'est d'ailleurs l'un des plus importants puisqu'il permet de réutiliser le code d'une classe autant de fois que l'on souhaite tout en ayant la liberté d'en modifier certaines parties.

Supposons que nous disposons d'une classe du nom de **classe1**. Si on crée une classe du nom de **classe2** qui hérite de **classe1**, alors **classe2**hérite de tous les membres (attributs et méthodes) qui constituent **classe1**.

Autrement dit, si on instancie **classe2**, alors tous les attributs et méthodes de **classe1** peuvent être appelés à partir de l'objet créé (l'instance de **classe2**). Bien entendu, il faut que les membres appelés soient publiques. Dans ce cas on dit que **class1** est la classe **mère** et **class2** est la classe **fille**.

Mot clé extends

Pour procéder à l'héritage, on fait appel au mot clé **exends** comme ceci:

```php
<?php
    class Mere{
    }
    class Fille extends Mere{
    }
?>
```

Dans ce cas, la classe **Fille** hérite de la classe **Mère**.

Exemple:

```php
<?php
    class Mere{
        protected $attribut="Bonjour.";
        public function methode1(){
            $str=$this->attribut;
            $str.=" Je suis la classe Mère.";
            return $str;
        }
    }
    class Fille extends Mere{
        public function methode2(){
            $str=$this->attribut;
            $str.=" Je suis la classe Fille.";
            return $str;
        }
    }
    $objet=new Fille();
    echo $objet->methode1();
    echo "<br />";
    echo $objet->methode2();
?>
```

Surcharge d'une méthode

Dans l'exemple précédent où la classe Fille hérite de la classe Mere, il est possible de définir une méthode du nom de methode1() dans la classe Fille. Bien entendu, cette méthode figure déjà dans la classe Mere et elle est supposée être héritée elle aussi car elle est publique. Le fait de déclarer une méthode du même nom dans la classe Fille s'appelle **redéfinition de méthode** ou **surcharge de méthode**.

Reprenons l'exemple précédent en surchargeant la méthode methode1():

```php
<?php
    class Mere{
        protected $attribut="Bonjour.";
        public function methode1(){
            $str=$this->attribut;
            $str.=" Je suis la classe Mère.<br />";
            return $str;
        }
    }
    class Fille extends Mere{
        public function methode1(){
            $str=$this->attribut;
            $str.=" Je suis la classe Fille.<br />";
            return $str;
        }
    }
    $objet=new Fille();
    echo $objet->methode1();
?>
```

Je suis la classe Fille.

En effet, le fait de surcharger la methode1() dans la classe Fille a écrasé son code hérité de la classe Mere. Il est donc normal que seul le code de methode1() redéfinie dans la classe Fille soit exécuté.

Mais supposons que nous voulons quand même conserver le résultat retourné par methode1() dans la classe Mere et l'ajouter au résultat obtenu suite à la surcharge dans la classe Fille. Le code devrait donc ressembler à ceci:

```php
<?php
    class Mere{
        protected $attribut="Bonjour.";
        public function methode1(){
            $str=$this->attribut;
            $str.=" Je suis la classe Mère.<br />";
            return $str;
        }
    }
    class Fille extends Mere{
        public function methode1(){
            $str=Mere::methode1();
            $str.=$this->attribut;
            $str.=" Je suis la classe Fille.<br />";
            return $str;
        }
    }
    $objet=new Fille();
    echo $objet->methode1();
?>
```

```
Bonjour. Je suis la classe Mère.
Bonjour. Je suis la classe Fille.
```

Mot clé parent

Vous vous souvenez sûrement du mot clé self qui fait référence à la classe courante. Dans l'exemple précédent on a appelé la classe Mere par son nom dans la classe Fille pour appeler statiquement methode1(). Cette fois on va faire appel au mot clé parent qui désigne la classe dont on a hérité. Autrement dit, si nous sommes dans la classe Fille et on veut appeler statiquement methode1() de la classe Mere, au lieux de mettre **Mere::methode1(),** on peut mettre **parent::methode1().**

```php
<?php
    class Mere{
        protected $attribut="Bonjour.";
        public function methode1(){
            $str=$this->attribut;
            $str.=" Je suis la classe Mère.<br />";
            return $str;
        }
    }
    class Fille extends Mere{
        public function methode1(){
            $str=parent::methode1();
            $str.=$this->attribut;
            $str.=" Je suis la classe Fille.<br />";
            return $str;
        }
    }
    $objet=new Fille();
    echo $objet->methode1();
?>
```

Contraintes sur l'héritage

A priori, on peut hériter de n'importe quelle classe sans restriction. Si on dispose de la classe A, on peut en hériter pour obtenir la classe B dont on peut hériter pour avoir la classe C et ainsi de suite. Cependant, on peut appliquer certaines contraintes pour autoriser ou non l'héritage d'une classe donnée, voir même obliger le développeur à hériter d'une classe particulière.

Classes et méthodes abstraites (Mot clé abstract)

Une classe abstraite et une classe dont on doit impérativement hériter. Autrement dit, on ne peut pas l'instancier directement. Pour ce faire, on définit la classe abstraite à l'aide du mot clé abstract.

Exemple:

```php
<?php
    abstract class A{
        public function methode(){
            echo "Bonjour";
        }
    }
    class B extends A{
    }
    $objet=new B();
    $objet->methode();
?>
```

Bonjour

Dans ce cas, on a créé une classe abstraite du nom de A dont on a hérité pour créer la classe B. L'héritage dans ce cas et obligatoire car l'instanciation directe de la classe A est interdite.

Il n'y a pas que les classes qui peuvent être rendue abstraites, mais les méthodes aussi. Dans ce cas, il faut obligatoirement que la classe soit abstraite aussi.

Une méthode abstraite est une méthode qui doit être redéfinie (ou surchargée) dans les classes filles. Pour définir une méthode abstraite on déclare son prototype composé du mot clé abstract suivi de la visibilité puis le mot clé function suivi du nom de la fonction et les parenthèses. On ne définit pas le corps d'une méthode abstraite, donc pas d'accolades.

Exemple:

```php
<?php
    abstract class A{
        abstract public function methode();
    }
    class B extends A{
        public function methode(){
            echo "Bonjour";
        }
    }
    $objet=new B();
    $objet->methode();
?>
```

Bonjour

Interfaces

Une interface: une classe entièrement abstraite

Une interface se comporte comme une classe abstraite dont toutes les méthodes sont abstraites. Cependant l'ultime différence entre une classe abstraite et une interface est que les classes qui vont hériter des classes abstraites constituent des sous familles de la classe mère, par contre une interface peut être utilisées par des classes dont les finalités ne sont pas forcément semblables.

Pour mieux comprendre le principe, supposons que nous avons une classe abstraite du nom de Vehicule.

Les classes Filles qui peuvent en hériter sont par exemple Voiture, Camion et Fourgon. Ces classes sont en réalité des véhicules plus ou moins différents, donc ils constituent un sous ensemble de la classe Vehicule. Par contre les classes Vehicule et Maison ne sont pas de la même famille. Néanmoins chacune d'entre elles peut accueillir une méthode du nom de peindre(). C'est là où une interface qui renferme la méthode peindre() peut être utile.

Définir une interface

Pour définir une interface, on utilise le mot clé interface suivi du nom de celle-ci (avec une première lettre majuscule de préférence, tout comme pour une classe) suivi des accolades qui renfermeront les membres de l'interface comme ceci:

```php
<?php
    interface Entretien{
        public function peindre($c);
        public function nettoyer();
    }
?>
```

Implémenter une interface

Pour se servir de l'interface créée, il faut l'implémenter par une classe à l'aide du mot clé implements comme ceci:

```php
<?php
    class Vehicule implements Entretien{
        private $couleur;
        private $propre;
        public function peindre($c){
            $this->couleur=$c;
        }
        public function nettoyer(){
            $this->propre=true;
        }
    }
?>
```

Dans la classe Vehicule on doit obligatoirement redéfinir les méthodes peindre() et nettoyer().

Implémenter plusieurs interfaces

A l'inverse de l'héritage en PHP, on peut implémenter plusieurs interface simultanément:

```php
<?php
    interface Int1{
        public function f1();
    }
    Interface Int2{
        public function f2();
    }
    class Classe implements Int1, Int2{
        public function f1(){
        }
        public function f2(){
        }
    }
?>
```

Dans ce cas, dans la classe **Classe** on doit redéfinir les méthodes f1() et f2() dont le prototype est défini respectivement dans les interfaces **Int1** et **Int2**.

Héritage des interfaces

Comme pour les classes, une interface peut hériter d'une autre. Dans ce cas, la réécriture d'une méthode dans la classe fille n'est pas autorisée. Bien entendu, il faut réécrire les méthodes de l'interfaces mère et l'interface fille dans la classe qui implémente cette dernière:

```php
<?php
    interface Int1{
        public function f1();
    }
    Interface Int2 extends Int1{
        public function f2();
    }
    class Classe implements Int2{
        public function f1(){
        }
        public function f2(){
        }
    }
?>
```

Namespaces

Les namespaces Comme nous l'avons déjà préconisé à propos des fonctions, quand un projet pend de l'importance, et encore davantage quand il est réalisé en équipe, il y a tout intérêt à modulariser le code, chaque développeur travaillant sur un module donné. Introduits dans PHP 5.3 les namespaces (espaces de noms), qui sont les équivalents, par exemple, des packages en Java, facilitent cette modularisation. Ils nous permettent d'éviter également les conflits de noms de classe, propriétés, méthodes et fonctions. En effet, un de ces éléments peut avoir le même nom qu'un autre sans créer de conflit au moment de son appel, du moment que ces deux éléments sont définis dans des namespaces différents.

En pratique, les namespaces sont des fichiers PHP qui peuvent contenir les définitions d'un ou plusieurs espaces de noms ainsi que des définitions de classes, de constantes ou de fonctions. La première ligne du fichier doit contenir le mot-clé namespace suivi du nom choisi, et être suivie d'accolades contenant les définitions de tout ce que contient chaque espace. Tout ce qui est inclus dans les accolades appartient au namespace nommé et à lui seul

```php
<?php
    namespace Test\namespace1;
    class Personne
    {


    }
?>
<?php
    namespace Test\namespace12;
    class Personne
    {

    }
?>
<?php
    use Test\namespace1;
    $P1=new namespace1\Personne();
    use Test\namespace2;
    $P1=new namespace2\Personne();

?>
```

Objet PDO pour la connexion à une base de données

Jusqu'à la version 5 de PHP, la connexion à une base de données (en particulier MySQL) se faisait d'une manière transparente grâce à des fonctions du genre **mysql_connect**(), **mysql_select_db**(), **mysql_query**()...

Ces fonctions marchent encore sur les dernières versions PHP5.x (y compris PHP5.6), mais elles sont rendues obsolètes à partir de PHP5.5. A la sortie de PHP7 elles ont été supprimées.

Bien que ces fonctions marchaient très bien, leurs limites n'ont pas mis beaucoup de temps pour se manifester, en particulier quand on souhaite migrer vers un SGBDautre que MySQL. En effet, toutes ces fonctions préfixées par mysql_ n'auraient pas marché. Il faut par conséquent repasser en vue une grande partie du code déjà développé.

Heureusement l'objet PDO a été développé et intégré sous forme d'extension au langage PHP.

Objet PDO

PDO c'est quoi?

PDO signifie PHP Data Object. Il s'agit d'une interface qui permet au scripts PHP d’interroger une base de données via des requêtes SQL.

PDO est une extension qui s'ajoute au PHP pour que ses différentes fonctionnalités soient disponibles dans le langage.

Il constitue une interface d'abstraction de la base de données, c'est à dire qu'on peut utiliser l'ensemble de ses fonctions pour executer des requêtes SQL quelque soit le SGBD. Autrement dit, si l'application Web repose sur le SGBD MysQL, on peut migrer vers le SGBD PostgreSQL sans modifier le code source (quelques modifications mineurs sont requises).

```php
<?php
try
{
    $bdd = new PDO('mysql:host=localhost;dbname=test;charset=utf8', 'root', '');
}
catch (Exception $e)
{
        die('Erreur : ' . $e->getMessage());
}
?>

```

Récupérer les données

```php
<?php
$reponse = $bdd->query('SELECT * FROM user');
?>
```

Pour récupérer une entrée, on prend la réponse de MySQL et on y exécute **fetch(),** ce qui nous renvoie la première ligne.

```php
<?php
try
{   // On se connecte à MySQL
    $bdd = new PDO('mysql:host=localhost;dbname=test;charset=utf8', 'root', ''
}
catch(Exception $e)
{
    // En cas d'erreur, on affiche un message et on arrête tout
        die('Erreur : '.$e->getMessage());
}
// Si tout va bien, on peut continuer
// On récupère tout le contenu de la table user
$reponse = $bdd->query('SELECT * FROM user');

// On affiche chaque entrée une à une
while ($donnees = $reponse->fetch())
{
?>
    <p>
    <strong>Nom</strong> : <?php echo $donnees['nom']; ?><br />
    <strong>Prenom</strong> : <?php echo $donnees['prenom'];?>
   </p>
<?php
}
$reponse->closeCursor(); // Termine le traitement de la requête
?>
```

Écrire des données

```php
$reponse = $bdd->exec('INSERT INTO user(id, nom, prenom, adress, age)
    VALUES(\'\',\'med\', \'ali\', \'Tunis\',45)');
```

Insertion de données variables grâce à une requête préparée

Si on choisit d'utiliser une requête préparée (ce que je vous recommande si vous souhaitez insérer des variables)

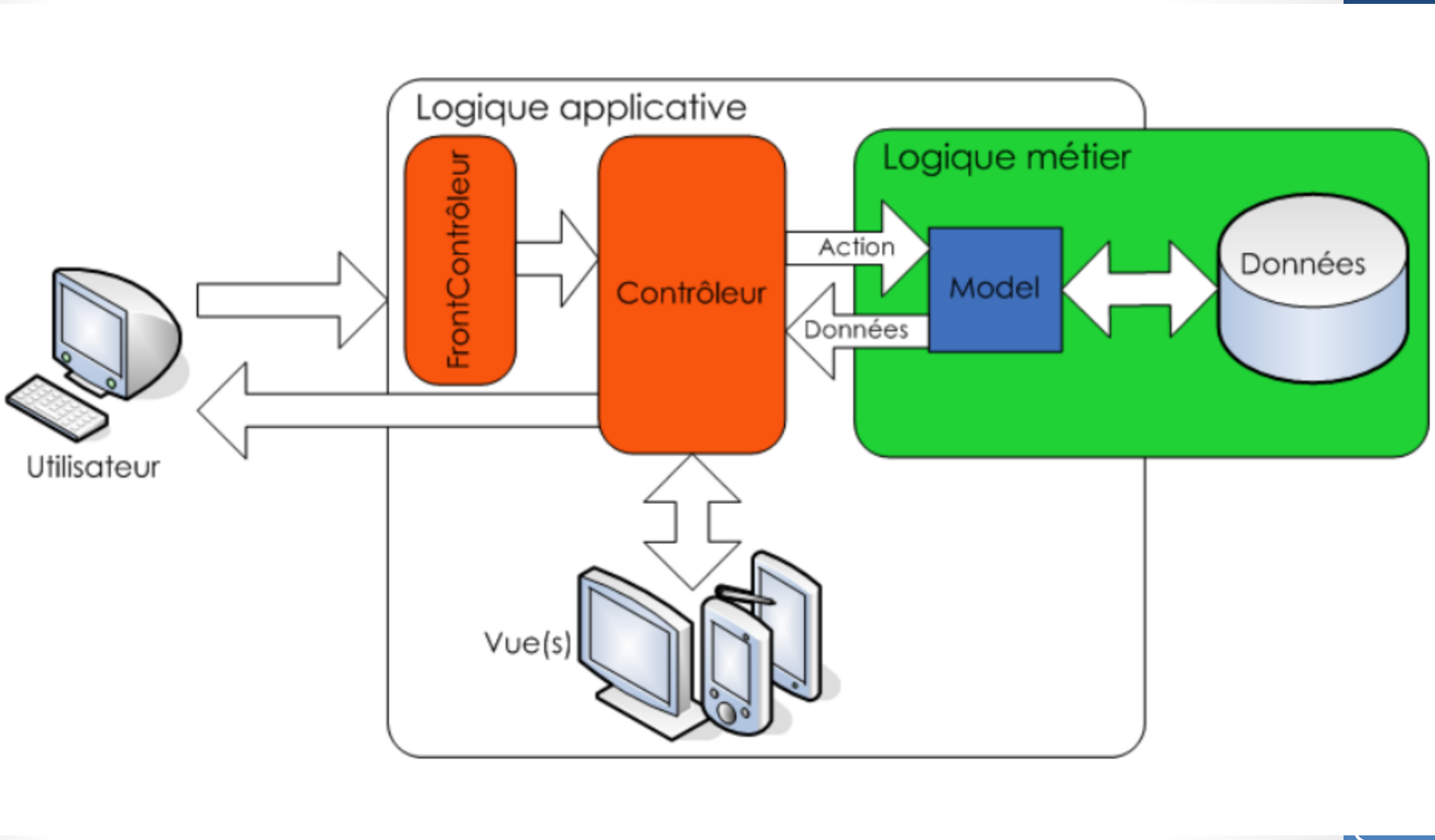
```php
<form method="post" action="insert.php">
    <input type="text" name="nom">
    <input type="text" name="prenom">
    <input type="text" name="adresse">
    <input type="submit" value="Ok">
</form>
</body>
</html>
<?php
$bdd = new PDO('mysql:host=localhost;dbname=test;charset=utf8', 'root', '');
$req = $bdd->prepare('INSERT INTO user(nom, prenom, adresse) VALUES(:nom, :prenom, :adresse)');
if (isset($_POST['nom']) && isset($_POST['prenom']) && isset($_POST['adresse']))
{
$req->execute(array(
    'nom' => $_POST['nom'],
    'prenom' => $_POST['prenom'],
    'adresse' => $_POST['adresse']
    ));
}
echo 'Le client a bien été ajouté !';
?>
```

```
<form method="post" action="insert.php">
    <input type="text" name="nom">
    <input type="text" name="prenom">
    <input type="text" name="adresse">
    <input type="submit" value="Ok">
</form>
</body>
</html>
<?php
$bdd = new PDO('mysql:host=localhost;dbname=test;charset=utf8', 'root', '');
$req = $bdd->prepare('INSERT INTO user(nom, prenom, adresse) VALUES(?,?,?)');
if (isset($_POST['nom']) && isset($_POST['prenom']) && isset($_POST['adresse']))
{
$req->execute(array($_POST['nom'],$_POST['prenom'],$_POST['adresse']));
}
echo 'Le client a bien été ajouté !';
?>
```

med : bonjour

ali : coucou

Principe de MVC



La vue : Elle gère l'aspect du formulaire : son design

Le contrôleur : Il récupère les données du formulaire pour les traiter

Le modèle : Il fait l'interface entre la base de données d'une part, le contrôleur et la vue d'autre part

La plupart des frameworks PHP utilisent MVC

Zend Framework, Symfony, ...

- MVC définit la structure globale de l'application
- La partie MVC est spécifique à l'application, contrairement aux librairies
- Les modèles et les contrôleurs sont généralement des classes
- Les vues sont généralement des templates HTML

Nom :

Prenom :

Date naissance: jj/mm/aaaa

Sexe : H F

Ville : Ariana

Competance : PHP5 HTML5 CSS3 Javascript

Ajouter