Support de cours PHP Orienté Objet

Jérôme AMBROISE

Sommaire

Sommaire

1. Introduction

Qu’est-ce qu’un objet ?

Qu’est-ce qu’une classe ?

Autres exemples

Différences entre une classe et un objet

Introduction

Qu’est-ce qu’un objet ?

Un objet est un conteneur symbolique, qui possède sa propre existence et englobe des caractéristiques, états et comportements et qui, par métaphore, représente quelque chose de tangible du monde réel manipulé par informatique.

En programmation orientée objet, un objet est créé à partir d'un modèle appelé classe, duquel il hérite les comportements et les caractéristiques. Les comportements et les caractéristiques sont typiquement basés sur celles propres aux choses qui ont inspiré l'objet : une personne, un dossier, un produit, une lampe, une chaise, un bureau. Tout peut être objet.

Introduction

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.

Introduction

Autre exemple

Etant donné qu'une classe est le modèle de quelque chose que nous voudrions construire.

- ➔ Le plan de construction d'une maison qui réunit les instructions destinées à la construction est donc une classe. Cependant le plan n'est pas une maison.
- ➔ La maison est un objet qui a été « fabriqué » à partir de la classe (le plan).

A partir du plan (la classe) nous pouvons construire une ou plusieurs maisons (l'objet).

Le vocabulaire diffère légèrement en Programmation Orientée Objet, nous ne parlerons plus de fonctions mais de méthodes, nous ne parlerons plus de variables mais d'attributs (ou propriétés).

Une classe est une entité regroupant un certain nombre d'attributs et de méthodes que tout objet issu de cette classe possède.

Introduction

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.

2.

Créer des classes et des objets

Créer une classe

Présentation de “$this”

Instancier une classe

Visualiser son objet

Accéder aux propriétés et méthodes

Accéder aux constantes de classes

Inférence

Transformation

Classes par défaut

Référence

Clônage

Comparer des objets

Appartenance à une classe

Créer des classes et des objets

Créer une classe

Une définition de classe basique commence par le mot-clé **class**, suivi du nom de la classe.

Suit une paire **d'accolades** contenant la définition des propriétés et des méthodes appartenant à la classe.

Créer des classes et des objets

Créer une classe

- Le nom de la classe peut être quelconque à condition que ce ne soit pas un mot réservé en PHP.
- Un nom de classe valide commence par une lettre ou un underscore.
- Le nom d'une classe peut contenir n'importe quel nombre de chiffres, ou de lettres, ou d'underscores.

Créer des classes et des objets

Présentation de “$this”

La pseudo-variable $this est disponible lorsqu'une méthode est appelée depuis un contexte objet.

$this est une référence à l'objet appelant

Créer des classes et des objets

Créer une classe

Exemple :

```
<?php
class SimpleClass
{
   // déclaration d'une propriété
   public $var = 'une valeur par défaut';

   // déclaration d'une méthode
   public function displayVar() {
      echo $this->var;                    // $this permet d'accéder à l'objet courant
   }
}
?>
```

Créer des classes et des objets

Instancier une classe

Pour utiliser une classe, il faut l'instancier. C'est à dire créer un objet que nous allons utiliser respectant la structure de la classe.

La façon la plus classique est d'utiliser "new"

Exemple :

```
$instanceSimpleClass = new SimpleClass();
```

Créer des classes et des objets

Instancier une classe

Pour utiliser votre classe, il faut l'inclure à votre fichier d'extension ".php"

Exemple :

```
require('Classes/SimpleClass.php');

$instanceSimpleClass = new SimpleClass();
```

Créer des classes et des objets

Instancier une classe

Une solution existe pour éviter les appels répétitifs de classes.

Exemple :

```
function my_autoloader($class) {
    include 'Classes/' . $class . '.php';        // A modifier
}

spl_autoload_register('my_autoloader');

$instanceSimpleClass = new SimpleClass();
```

Créer des classes et des objets

Visualiser son objet

Pour visualiser votre objet :

```
var_dump($instanceSimpleClass);
print_r($instanceSimpleClass);
```

Créer des classes et des objets

Accéder aux propriétés et aux méthodes

Accéder aux propriétés et aux méthodes se fait via l'utiliser de " -> ".

Exemple :

```
// Propriété
echo $instanceSimpleClass->varClass;
// Méthode
$instanceSimpleClass->displayVar();
```

Créer des classes et des objets

Accéder aux constantes de classe

- ➔ Les constantes de classes se définissent avec le mot-clef “const”
- ➔ Pour récupérer la valeur des constantes de classes, il faut utiliser “ :: ”.

Exemple :

```
class SacADos{
        const CONST_MAX = 14;
}
echo "Constante max : " . $monSac->CONST_MAX ; // Undefined property
echo "Constante max : " . $monSac::CONST_MAX ;
```

Créer des classes et des objets

Inférence

Une autre manière de générer l'objet est par inférence, c'est-à-dire par déduction à partir du contexte.

```
$obj->var1 = 'Valeur 1';
$obj->var2 = 'Valeur 2';
var_dump($obj);

echo "Variable : " . $obj->var1 . "<br/>";
echo "Variable2 : " . $obj->var2 ;
```

Remarque : pour fixer le warning “if (!isset($res)){ $obj = new stdClass();} ”

Créer des classes et des objets

Transformation

On peut générer un objet à partir d'un tableau associatif grâce à la transformation (cast)

```
$fruits = array (
        'var1' => 'Valeur 1',
        'var2' => 'Valeur 2',
   "numbers" => array(1, 2, 3, 4, 5, 6),
   "fruits"  => array("a" => "orange", "b" => "banana", "c" => "apple"),
);

$objetFruits = (object) $fruits;
var_dump($objetFruits);     // $objectFruits est un objet
```

Créer des classes et des objets

Classe par défaut

Les objets créés à la volée n'ont pas de classes définies.

Par défaut ils appartiennent donc à la classe mère de PHP "stdClass".

Créer des classes et des objets

Références

Exemple :

```
$instanceSimpleClass = new SimpleClass();          // Instanciation
$instance2 = $instanceSimpleClass;                 // Référence

echo "Valeur de l'instance 1 : " . $instanceSimpleClass->varClass . "<br/>" ;
echo "Valeur de l'instance 2 : " . $instance2->varClass . "<br/>" ;

$instance2->varClass = "Nouvelle valeur";          // Changement de valeur

echo "Valeur de l'instance 1 : " . $instanceSimpleClass->varClass . "<br/>" ;
echo "Valeur de l'instance 2 : " . $instance2->varClass . "<br/>" ;
```

Créer des classes et des objets

Clonage

Si vous voulez créer une copie d'un objet et que ces deux objets soient distincts, il faut le cloner.

Pour cloner, il faut utiliser le mot-clef "clone"

#nouvel objet# = clone #objet existant# ;

Remarque : J'utilise une convention de nommage pour les exemples : #variable# . Lorsque vous implémentez les exemples il faut **remplacer la valeur** à l'intérieur des dièses **ET enlever les dièses**.

Créer des classes et des objets

Clonage

Exemple :

```
$instanceSimpleClass = new SimpleClass();
$clone1 = clone $instanceSimpleClass;

echo "Valeur de l'instance 1 : " . $instanceSimpleClass->varClass . "<br/>" ;
echo "Valeur de l'instance 2 : " . $clone1->varClass . "<br/>" ;

$clone1->varClass = "Nouvelle valeur";

echo "Valeur de l'instance 1 : " . $instanceSimpleClass->varClass . "<br/>" ;
echo "Valeur de l'instance 2 : " . $clone1->varClass . "<br/>" ;
```

Créer des classes et des objets

Comparer des objets

Il est possible de comparer des objets entre eux avec les opérateurs d'égalité (==, ===) :

- ➔ Opérateur "==" : teste si toutes les propriétés sont égales et que les 2 objets sont issus de la même classe
- ➔ Opérateur "===" : teste si toutes les propriétés sont égales et que les 2 objets pointent vers le même objet

Créer des classes et des objets

Comparer des objets

Attribution	==	===
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now")); $developpeur2 = new SimpleDeveloppeur("John", "Doe", new DateTime("now"));	TRUE	FALSE
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now")); $developpeur2 = $developpeur;	TRUE	TRUE
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now")); $developpeur2 = &$developpeur;	TRUE	TRUE
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now")); $developpeur2 = clone $developpeur;	TRUE	FALSE
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now")); $developpeur2 = new OtherDeveloppeur("John", "Doe", new DateTime("now"));	FALSE	FALSE

Créer des classes et des objets

Appartenance à une classe

Il est possible de tester si un objet appartient à une certaine classe avec “instanceof”.

Exemple :

```
Class SimpleDeveloppeur{ /* … */ }
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now"));
var_dump($developpeur instanceof SimpleDeveloppeur); // true
```

3.

L’encapsulation

Présentation

Visibilité

Accesseurs et mutateurs

L’encapsulation
Présentation

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.

➔ L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

L’encapsulation
Présentation

Bien généralement, les créateurs de classes et les utilisateurs de classes sont 2 développeurs différents.

- ➔ Au sein d’une même équipe : certains développeurs créés les classes, d’autres les utilisent.
- ➔ Récupération d’un plugin : lors de la récupération d’un plugin externe, nous ne définissons pas les classes mais les utilisons.

L’encapsulation permet aux créateurs des classes de protéger certaines parties de leurs classes pour empêcher les utilisateurs de toucher à des “zones” (attributs, …) et de dégrader le fonctionnement normale de la classe.

L’encapsulation

Présentation

De plus, l'utilisateur d'une classe n'a pas forcément à savoir de quelle façon sont structurées les données dans l'objet, cela signifie qu'un utilisateur n'a pas à connaître l'implémentation.

Ainsi, en interdisant l'utilisateur de modifier directement les attributs, et en l'obligeant à utiliser les fonctions définies pour les modifier (appelées interfaces), on est capable de s'assurer de l'intégrité des données.

Exemples :

- ➔ S'assurer que le type des données fournies est conforme à nos attentes.
- ➔ S’assurer que les données se trouvent bien dans l'intervalle attendu.

L’encapsulation
Visibilité

Différentes visibilités sont disponibles, elles permettent de restreindre leurs utilisations. Les mots-clefs de visibilité se placent devant un attribute ou une méthode.

#mot-clef de visibilité# #nom attribut/méthode#

- ➔ Visibilité publique (**public**) : restriction la plus faible. Les éléments déclarés comme publics peuvent être utilisés par n'importe quelle partie du programme.
- ➔ Visiblité privée (**private**) : restriction le plus forte. L'accès aux éléments privés est uniquement réservé à la classe qui les a défini.
- ➔ Visibilité protégées (**protected**) : L'accès aux éléments protégés est limité à la classe elle-même, ainsi qu'aux classes qui en héritent, et à ses classes parentes.

L’encapsulation
Visibilité

Exemple : Définition de la classe

```
class ClassVisibility
{
   public      $attrPublic = 'Attribut public';
   protected   $attrProtected = 'Attribut protégé';
   private     $attrPrivate = 'Attribut privé';

   function printAttr()
   {
      echo "<br/>Méthode 'printAttr() '<br/> - ".
                        $this->attrPublic . "<br/> - " .
                        $this->attrProtected . "<br/> - " .
                        $this->attrPrivate . "<br/>";
   }
}
```

L’encapsulation
Visibilité

Exemple : Utilisation de la classe

```
$obj = new ClassVisibility();

echo $obj->attrPublic;          // Fonctionne
//echo $obj->attrProtected;     // Erreur fatale
//echo $obj->attrPrivate;       // Erreur fatale

$obj->printAttr();              // Affiche les attributs
```

L’encapsulation
Accesseurs et mutateurs

L’encapsulation nous impose de déclarer des attributs privés ou protégés. Comment l’utilisateur de la classe peut-il alors accéder et modifier ces attributs ?

Grâce aux accesseurs et aux mutateurs :

- ➔ Accesseur : permet d’accéder aux attributs (privés ou protégés) depuis l’extérieur de la classe.

- ➔ Mutateur : permet de modifier les attributs (privés ou protégés) depuis l’extérieur. L’objectif du mutateur est de tester si la nouvelle valeur de l’attribut respecte bien les règles que nous avons définis.

Les accesseurs et mutateurs sont à créer dans vos classes si nécessaire. Les différents contrôles sont aussi à définir seulement la logique métier de votre projet.

L’encapsulation
Accesseurs et mutateurs

Par convention, les attributs privés ou protégés sont préfixés de “_”.
Par convention, les accesseurs sont préfixés de “get” et les mutateurs de “set” :

Exemple : Définition de la classe

```
class SacADos{
        private $_couleur = "marron";

   public function getCouleur(){
      return $this->_couleur;
   }
   public function setCouleur($newCouleur){
      $this->_couleur = $newCouleur;
   }
```

L’encapsulation
Accesseurs et mutateurs

Exemple : Utilisation de la classe

```
$monSac = new SacADos();

//echo "Couleur du sac : " . $monSac->_couleur ;                // Erreur fatale

echo "Couleur du sac : " . $monSac->getCouleur() . "<br/>";    // Première valeur

$monSac->setCouleur("Rouge");                                  // Changement de la valeur

echo "Couleur du sac : " . $monSac->getCouleur() . "<br/>";    // Deuxième valeur
```

4.

Méthodes prédéfinies

Méthodes prédéfinies

Présentation

Les méthodes prédéfinies (ou dites “magiques”) sont des outils très pratiques pour simplifier le code et automatiser certaines tâches.

Elles sont appelées automatiquement lorsque certaines actions sont effectuées et permettent d’éviter certaines erreurs.

Méthodes prédéfinies

Le constructeur

Les classes qui possèdent une méthode constructeur appellent cette méthode à chaque création d'une nouvelle instance de l'objet, ce qui est intéressant pour toutes les initialisations dont l'objet a besoin avant d'être utilisé.

Le constructeur se place dans la classe et porte le nom “ __construct()”

Une méthode ayant le même nom que la classe est considérée comme étant le constructeur (précédente convention de PHP 4). Toutefois il est préférable d’utiliser la méthode prédéfinie “__construct”.

Méthodes prédéfinies
Le constructeur

Exemple : Définition de la classe

```
class Moto{
        private $marque;

        function __construct(){
                echo "Appel du constructeur <br/>" ;
                $this->marque = "Renault";
        }

        public function getMarque(){
                return $this->marque;
        }
}
```

Méthodes prédéfinies
Le constructeur

Exemple : **Utilisation de la classe**

```
$maMoto = new Moto();

echo "Marque de ma moto " . $maMoto->getMarque() ;
```

Méthodes prédéfinies

Le constructeur

Pour initialiser ses attributs, il faut passer au constructeur des arguments.

Exemple : Définition de la classe

```
class Moto{
   private $marque;

        function __construct($marque){
                echo "Appel du constructeur <br/>" ;
                $this->marque = $marque;
        }

        public function getMarque(){
                return $this->marque;
        }
}
```

Méthodes prédéfinies
Le constructeur

Exemple : Utilisation de la classe

```
$maMoto = new Moto("Triumph");

echo "Marque de ma moto : " . $maMoto->getMarque() ;
```

Méthodes prédéfinies

Le destructeur

PHP 5 introduit un concept de destructeur similaire à celui d'autres langages orientés objet, comme le C++.

La méthode destructeur est appelée dès qu'il n'y a plus de référence sur un objet donné, ou dans n'importe quel ordre pendant la séquence d'arrêt.

Le destructeur se place dans la classe et porte le nom “ __destruct()”

Méthodes prédéfinies

Le destructeur

Exemple : Définition de la classe

```
class Moto{
   private $marque;

        function __construct($marque){
                echo "Appel du constructeur <br/>" ;
                $this->marque = $marque;
        }

        function __destruct(){
                echo "<br/>Appel du destructeur <br/>" ;
        }

        public function getMarque(){
                return $this->marque;
        }
}
```

Méthodes prédéfinies

Méthodes “__GET” et “__SET”

- ➔ __set() : est sollicitée lors de l'écriture de données vers des propriétés inaccessibles.
- ➔ __get() : est appelée pour lire des données depuis des propriétés inaccessibles.

Selon les besoins métiers il est donc possible de définir des comportements spécifiques à effectuer lors de l’appel (accès/modification) de propriétés inexistantes ou inaccessibles.

Exemple d’application : création d’un traitement dynamique d’accès à une base de données (getMyClassById(), getMyClassByName(), …). On “split” le nom de la méthode appelée pour obtenir le champs voulu (id, name, …) pour ensuite faire un appel à la base de données.

Méthodes prédéfinies
Méthodes “__GET” et “__SET”

Exemple :

```
public function __get($property) {
        echo "Entrée dans \"__get\" <br/>";
        echo "Récupération de '$property' <br/>";
        /* traitement à définir */
}

public function __set($property, $value) {
        echo "Entrée dans \"__set\" <br/>";
        echo "Définition de '$property' à la valeur '$value'<br/>";
        /* traitement à définir */
}
```

Méthodes prédéfinies
Méthodes “__CALL”

➔ __call() : est appelée lorsque l'on invoque des méthodes inaccessibles dans un contexte objet.

Exemple :

```
public function __call($methode, $arguments) {
        echo "Entrée dans \"__call\" <br/>";
        echo "La méthode '$methode' n'existe pas.";
}
```

Méthodes prédéfinies
Méthodes “__ISSET”

➔ __isset() : est sollicitée lorsque isset() ou la fonction empty() sont appelées sur des propriétés inaccessibles.

Exemple :

```
public function __isset($property){
        echo "Entrée dans \"__isset\" <br/>";
        echo "La propriété '$property' n'existe pas.<br/>";
}
```

Méthodes prédéfinies
Méthodes “__UNSET”

➔ __unset() : est invoquée lorsque unset() est appelée sur des propriétés inaccessibles.

Exemple :

```
public function __unset($property){
        echo "Entrée dans \"__unset\" <br/>";
        echo "La propriété '$property' est inaccessible.<br/>";
}
```

Méthodes prédéfinies

Méthodes “__SLEEP”

- La fonction serialize() vérifie si votre classe a une méthode avec le nom magique __sleep(). Si c'est le cas, cette méthode sera exécutée avant toute linéarisation.

- Le but avoué de __sleep() est de valider des données en attente ou d'effectuer des opérations de nettoyage. De plus, cette fonction est utile si vous avez de très gros objets qui n'ont pas besoin d'être sauvegardés en totalité.

Exemple :

```
public function __sleep(){
        return array('couleur', 'marque');
}
```

Méthodes prédéfinies

Méthodes “__WAKEUP”

➔ Réciproquement, la fonction unserialize() vérifie la présence d'une méthode dont le nom est le nom magique __wakeup(). Si elle est présente, cette fonction peut reconstruire toute ressource que l'objet pourrait posséder.

➔ Le but avoué de __wakeup() est de rétablir toute connexion de base de données qui aurait été perdue durant la linéarisation et d'effectuer des tâches de réinitialisation.

Exemple :

```
public function __wakeup(){
        if(empty($this->poidsAVide)){
                $this->poidsAVide = 1000;
        }
}
```

Méthodes prédéfinies

Exemples “__SLEEP” et “__WAKEUP”

```php
<?php
class Connection
{
    protected $link;
    private $dsn, $username, $password;

    public function __construct($dsn, $username, $password)
    {
        $this->dsn = $dsn;
        $this->username = $username;
        $this->password = $password;
        $this->connect();
    }

    private function connect()    {
        $this->link = new PDO($this->dsn, $this->username, $this->password);    }

    public function __sleep()           {       return array('dsn', 'username', 'password');    }
    public function __wakeup()          {       $this->connect();    }
}
?>
```

Méthodes prédéfinies
Méthodes “__toString”

➔ La méthode __toString() détermine comment l'objet doit réagir lorsqu'il est traité comme une chaîne de caractères. Par exemple, ce que “echo $obj;” affichera.

Exemple :

```
public function __toString(){
        return $this->marque . ' (' . $this->couleur . ' , ' . $this->poidsAVide . ')';
}
```

Méthodes prédéfinies
Méthodes “__INVOKE”

➔ La méthode __invoke() est appelée lorsqu'un script tente d'appeler un objet comme une fonction.

Exemple :

```php
<?php
        class CallableClass
        {
            public function __invoke($x)
            {
                var_dump($x);
            }
        }
        $obj = new CallableClass;
        $obj(5);
        var_dump(is_callable($obj));
?>
```

Méthodes prédéfinies
Méthodes “__SET_STATE”

➔ Cette méthode statique est appelée pour les classes exportées par la fonction var_export() depuis PHP 5.1.0.
Le seul paramètre de cette méthode est un tableau contenant les propriétés exportées sous la forme array('propriété' => valeur, ...).

Méthodes prédéfinies

Méthodes “__SET_STATE”

Exemple :

```
// La définition dans la classe

   public static function __set_state($an_array) // Depuis PHP 5.1.0
   {
        echo "Entrée dans \"__set_state\" <br/>";
        $obj = new Moto($an_array['marque'],'1000');
        //$obj->couleur = $an_array['couleur'];
        return $obj;
   }

// Pour l’appel

eval('$bb = ' . var_export($motoo, true) . ';');
var_dump($bb);
```

Méthodes prédéfinies
Méthodes “__CLONE”

➔ Une copie d'objet est créée en utilisant le mot-clé clone (qui fait appel à la méthode __clone() de l'objet, si elle a été définie).

Exemple :

```
public function __clone(){
        echo "Entrée dans \"__clone\" <br/>";
        $this->marque = "Renault";
}
```

Méthodes prédéfinies

Méthodes “__CLONE”

➔ Cette méthode est appelée par var_dump() lors du traitement d'un objet pour récupérer les propriétés qui doivent être affichées. Si la méthode n'est pas définie dans un objet, alors toutes les propriétés publiques, protégées et privées seront affichées.

Cette fonctionnalité a été ajoutée en PHP 5.6.0.

Exemple :

```
public function __debugInfo() {
    return [
        'Marque' => $this->marque,
        'Couleur' => $this->couleur,
        'Poids à vide' => $this->poidsAVide
    ];
}
```

5.

Héritage

Présentation

Exemple d’héritage simple

Surcharge de méthodes

Eléments statiques

Eléments statiques “self::” et “parent::”

Méthodes statiques

Abstraction de classe

Les interfaces

Les constantes d’interface

Implémentation de plusieurs interfaces

Mot-clé “final”

Les traits

Héritage
Présentation

L'héritage est un des grands principes de la programmation orientée objet (POO), et PHP l'implémente dans son modèle objet. Ce principe va affecter la manière dont de nombreuses classes sont en relation les unes avec les autres.

- ➔ Par exemple, lorsque vous étendez une classe, la classe fille hérite de toutes les méthodes publiques et protégées de la classe parente. Tant qu'une classe n'écrase pas ces méthodes, elles conservent leur fonctionnalité d'origine.

- ➔ L'héritage est très utile pour définir et abstraire certaines fonctionnalités communes à plusieurs classes, tout en permettant la mise en place de fonctionnalités supplémentaires dans les classes enfants, sans avoir à réimplémenter en leur sein toutes les fonctionnalités communes.

Héritage
Présentation

- ➔ Il faut définir une classe appelée “classe mère” qui définira des propriétés et/ou des méthodes.
- ➔ Une classe appelée “classe fille” peut alors étendre ses différents symboles en utilisant le mot-clé “extends”
- ➔ L’héritage peut-être multiple

Héritage

Exemple d'héritage simple

Exemple :

Classe mère :

```
class SimpleHumain {

        private $prenom ;
        private $nom ;
        private $dateDeNaissance;

        public function __construct($prenom, $nom, $dateDeNaissance){
                $this->prenom = $prenom;
                $this->nom = $nom;
                $this->dateDeNaissance = $dateDeNaissance;
        }

        public function getPrenom(){            return $this->prenom;     }
        public function getNom(){               return $this->nom;        }
        public function getDateDeNaissance(){
                return $this->dateDeNaissance->format("d M Y");
        }
}
```

Héritage
Exemple d'héritage simple

Exemple :

Classe fille :

```
class SimpleDeveloppeur extends SimpleHumain {
        private $langages = [];

        public function getLangages(){
                return implode(" ," , $this->langages);
        }
        public function addLangage($item){
                array_push($this->langages, $item);
        }
}
```

Héritage
Exemple d'héritage simple

Exemple :

Appel :

```
$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now"));

echo $developpeur->getPrenom() .
        ' ' . $developpeur->getNom() .
        ' (' . $developpeur->getDateDeNaissance() . ')';            // John Doe (11 May 2017)

$developpeur->addLangage("HTML5");
$developpeur->addLangage("CSS3");

echo '<br/>Langages : ' . $developpeur->getLangages();        // Langages : HTML5 ,CSS3
```

Héritage
Surcharge de méthodes

Lorsqu'un héritage est fait, il est possible de redéfinir des méthodes héritées dans la classe enfant pour changer son comportement.

➔ Il est possible d'accéder à la méthode de la classe Mère depuis la classe Fille (pas obligatoire) avec "parent::" suivi par le nom de la méthode.

Exemple :

Définition d'une méthode dans la classe Mère :

```
public function getIdentity() {
        echo $this->getPrenom() .
        ' ' . $this->getNom() .
        ' (' . $this->getDateDeNaissance() . ')<br/>';
}
```

Surcharge de la méthode dans la classe Fille :

```
public function getIdentity() {
        parent::getIdentity();          // On accède au comportement initial
        echo 'Langages : ' . $this->getLangages() . '<br/>';
}
```

Héritage
Surcharge de méthodes

Exemple :

Appel :

```
$humain = new SimpleHumain("David", "Smith", new DateTime("now"));

$developpeur = new SimpleDeveloppeur("John", "Doe", new DateTime("now"));
$developpeur->addLangage("HTML5");
$developpeur->addLangage("CSS3");

$humain->getIdentity();                  // David Smith (11 May 2017)
$developpeur->getIdentity();             // John Doe (11 May 2017) Langages : HTML5 ,CSS3
```

Héritage
Eléments statiques

Le fait de déclarer des propriétés ou des méthodes comme statiques vous permet d'y accéder sans avoir besoin d'instancier la classe.

- ➔ On ne peut accéder à une propriété déclarée comme statique avec l'objet instancié d'une classe (bien que ce soit possible pour une méthode statique).

- ➔ Comme les méthodes statiques peuvent être appelées sans qu'une instance d'objet n'ai été créée, la pseudo-variable $this n'est pas disponible dans les méthodes déclarées comme statiques.

- ➔ On ne peut pas accéder à des propriétés statiques à travers l'objet en utilisant l'opérateur ->.

Héritage
Eléments statiques : self et parent

- L’accès aux variables et aux méthodes statiques se fait de la manière suivante avec l’opérateur de portée “ :: ”
- Les mots clefs “self” et “parent” permettent d’accéder aux variables et aux méthodes statiques à l’intérieur d’une classe.

Héritage
Eléments statiques : self et parent

Exemple :

Dans la classe “Humain” :

```
public static $varStatiqueHumain = "valeur de la variable statique (côté parent)";

public function getVarStatiqueHumain(){
        return self::$varStatiqueHumain;
}
```

Lors de l’appel :

```
echo Humain::$varStatiqueHumain . '<br/>';

$humain = new Humain("John", "Doe", new DateTime("now"));
echo $humain::$varStatiqueHumain . '<br/>';
echo $humain->getVarStatiqueHumain() . '<br/>';
```

Héritage
Eléments statiques : self et parent

Exemple :

Dans la classe “Developpeur” :

```
public function getVarStatiqueDeveloppeur(){
        return parent::getVarStatiqueHumain();
}
```

Lors de l’appel :

```
echo Developpeur::$varStatiqueHumain . '<br/>';

$developpeur1 = new Developpeur("John", "Smith", new DateTime("now"));
echo $developpeur1::$varStatiqueHumain . '<br/>';
echo $developpeur1->getVarStatiqueHumain() . '<br/>';
echo $developpeur1->getVarStatiqueDeveloppeur() . '<br/>';
```

Héritage
Méthode statique

Exemple :

Dans la classe “Humain” :

```
public static function getInfoMethodStatic(){
        echo "Accès à la méthode statique parent";
}
```

Lors de l’appel :

```
echo Humain::getInfoMethodStatic() . '<br/>';
$classname = 'Humain';
$classname::getInfoMethodStatic(); // Depuis PHP 5.3.0

echo Developpeur::getInfoMethodStatic() . '<br/>';
$classname = 'Developpeur';
$classname::getInfoMethodStatic(); // Depuis PHP 5.3.0
```

Héritage
Eléments statiques

➔ Le mot-clé “static” permet aussi de faire une résolution statique à la volée. Il peut être utilisé au même titre que “self” et “parent”. Cela permet de référencer la classe qui a été appelée durant l'exécution et non la classe où il a été défini.

Exemple :

```
<?php
class A {
    public static function qui() {                 echo __CLASS__;             }
    public static function testSelf() {             self::qui();                }
    public static function testStatic() {           static::qui(); // Ici, résolution à la volée
    }
}

class B extends A {    public static function qui() {        echo __CLASS__;    }    }

B::testSelf();          // A
B::testStatic();        // B
?>
```

Héritage
Abstraction de classes

PHP 5 introduit les concepts de classes et de méthodes abstraites. Les classes définies comme abstraites ne peuvent pas être instanciées, et toute classe contenant au moins une méthode abstraite doit elle-aussi être abstraite. Les méthodes définies comme abstraites déclarent simplement la signature de la méthode - elles ne peuvent définir son implémentation.

- ➔ Toutes les méthodes marquées comme abstraites dans la déclaration de la classe parente doivent être définies par l'enfant
- ➔ Ces méthodes doivent être définies avec la même visibilité, ou une visibilité moins restreinte
- ➔ Les signatures de ces méthodes doivent correspondre, ce qui signifie que les types des paramètres et le nombre d'arguments requis doivent être les mêmes.

Héritage
Abstraction de classes

Exemple :

Classe “AbstractHumain” :

```
abstract class AbstractHumain{

        abstract public function getIdentity();
        /* … */
}
```

Héritage

Abstraction de classes

Exemple :

Classe “ConcretDeveloppeur” :

```
class ConcretDeveloppeur extends AbstractHumain{

        private $langages = [];

        public function getIdentity(){
                $langages = "";

                if(sizeof($this->langages) > 0){
                        $langages = implode(" ,", $this->getLangages());
                }
                return $this->getPrenom() .
                                ' ' . $this->getNom() .
                                ' ( ' . $this->getDateDeNaissance()->format('Y-m-d H:i') .
                                ' )<br/>' . $langages ;
        }
        /* … */
}
```

Héritage
Abstraction de classes

Exemple :

Lors de l'appel :

```
$concretDeveloppeur = new ConcretDeveloppeur("Tom", "Smith", new DateTime("now"));
echo '<p>' . $concretDeveloppeur->getIdentity() . '</p>' ;

$concretDeveloppeur->addLangages("HTML5");
$concretDeveloppeur->addLangages("CSS3");

echo '<p>' . $concretDeveloppeur->getIdentity() . '</p>' ;
```

Héritage
Les interfaces

Les interfaces objet vous permettent de créer du code qui spécifie quelles méthodes une classe doit implémenter, sans avoir à définir comment ces méthodes fonctionneront.

- ➔ Les interfaces sont définies en utilisant le mot-clé interface, de la même façon que pour une classe standard, mais sans qu'aucune des méthodes n'ait son contenu de spécifié.

- ➔ De par la nature même d'une interface, toutes les méthodes déclarées dans une interface doivent être publiques.

Héritage
Les interfaces

Pour implémenter une interface, l'opérateur “implements” est utilisé. Toutes les méthodes de l'interface doivent être implémentées dans une classe ; si ce n'est pas le cas, une erreur fatale sera émise. Les classes peuvent implémenter plus d'une interface, en séparant chaque interface par une virgule.

➔ La classe implémentant l'interface doit utiliser exactement les mêmes signatures de méthodes que celles définies dans l'interface. Dans le cas contraire, une erreur fatale sera émise.

Héritage
Les interfaces

Exemple :

Interface iHumain :

```
interface iHumain{

        public function getIdentity();
}
```

Classe Humain :

```
class Humain implements iHumain{

        public function getIdentity(){
                return $this->getPrenom() .
                        ' ' . $this->getNom() .
                        ' ( ' . $this->getDateDeNaissance()->format('Y-m-d H:i') . ' )';
        }
        /* ... */
}
```

Héritage
Les interfaces

- ➔ Une interface peut définir des constantes. Ces constantes ne peuvent pas être redéfinies par la suite.
- ➔ Une classe peut implémenter (mot clé “implements”) plusieurs interfaces
- ➔ Une interface peut étendre (mot clé “extends”) une autre interface
- ➔ Contrairement aux classes, les interfaces peuvent hériter (mot clé “extends”) de plusieurs interfaces à la fois.

Héritage
Constantes d'interfaces

Exemple :

Définitions des classes :

```
interface iInterfaceConstante{     const NB_MAX = 42;          }
class ConcretClass2 implements iInterfaceConstante{        }
/* class ConcretClassToFail implements iInterfaceConstante{    const NB_MAX = 136;    }*/
```

Lors de l'appel :

```
echo '<p> Constante de l\'interface : ' . iInterfaceConstante::NB_MAX . '</p>';
echo '<p> Constante de ConcretClass : ' . ConcretClass2::NB_MAX . '</p>';
//echo '<p> Constante de ConcretClassToFail : ' . ConcretClassToFail::NB_MAX . '</p>';
```

Héritage

Implémentation de plusieurs interfaces

Exemple :

Définitions des classes :

```
interface iInterface1{      public function maFonction1();          }
interface iInterface2{      public function maFonction2();          }

class ClassImplementsMultipleInterfaces implements iInterface1, iInterface2{
        public function maFonction1(){          echo 'Appel de la fonction 1<br/>';     }
        public function maFonction2(){          echo 'Appel de la fonction 2<br/>';     }
}
```

Lors de l'appel :

```
$myClass2 = new ClassInterfaceMereFille;
$myClass2->maFonction1();
$myClass2->maFonction2();
```

Héritage

Implémentation de plusieurs interfaces

Exemple :

Définitions des classes :

```
interface iInterface1{        public function maFonction1();          }
interface iInterface2{        public function maFonction2();          }
interface iInterface3 extends iInterface1, iInterface2{
        public function maFonction3();
}
class ClassInterfaceExtendsMultipleInterfaces implements iInterface3{
        public function maFonction1(){          echo 'Appel de la fonction 1<br/>';         }
        public function maFonction2(){          echo 'Appel de la fonction 2<br/>';         }
        public function maFonction3(){          echo 'Appel de la fonction 3<br/>';         }
}
```

Lors de l'appel :

```
$myClass3 = new ClassInterfaceExtendsMultipleInterfaces;
$myClass3->maFonction1();
$myClass3->maFonction2();
$myClass3->maFonction3();
```

Héritage
Mot-clé “final”

PHP 5 dispose du mot-clé "final", qui empêche les classes filles de surcharger une méthode, lorsque la définition de cette dernière est préfixée par le mot-clé "final". Si la classe elle-même est définie comme finale, elle ne pourra pas être étendue.

```
<?php
class BaseClass {
   public function test() {                    echo "BaseClass::test() appelée\n";
         }

   final public function moreTesting() {       echo "BaseClass::moreTesting() appelée\n";   }
}

class ChildClass extends BaseClass {
   public function moreTesting() {             echo "ChildClass::moreTesting() appelée\n";  }
}
// Résultat : Fatal error: Cannot override final method BaseClass::moreTesting()
?>
```

Héritage
Mot-clé “final”

Une classe peut utiliser aussi le mot-clef “final”

```
<?php
final class BaseClass {
   public function test() {                    echo "BaseClass::test() appelée\n";          }

   // Ici, peu importe si vous spécifiez la fonction en finale ou pas
   final public function moreTesting() {       echo "BaseClass::moreTesting() appelée\n";   }
}

class ChildClass extends BaseClass {
}
// Résultat : Fatal error: Class ChildClass may not inherit from final class (BaseClass)
?>
```

Remarque : Les propriétés ne peuvent être déclarées comme finales

Héritage
Les traits

Les traits sont un mécanisme de réutilisation de code dans un langage à héritage simple tel que PHP.
Un trait tente de réduire certaines limites de l'héritage simple, en autorisant le développeur à réutiliser un certain nombre de méthodes dans des classes indépendantes.

- ➔ Un trait est semblable à une classe, mais il ne sert qu'à grouper des fonctionnalités d'une manière intéressante.
- ➔ Il n'est pas possible d'instancier un Trait en lui-même.
- ➔ C'est un ajout à l'héritage traditionnel, qui autorise la composition horizontale de comportements, c'est à dire l'utilisation de méthodes de classe sans besoin d'héritage.

Héritage
Les traits

Exemple :

Définitions :

```
trait GetInfoTrait{
        public function getTitre(){             return $this->titre;        }
        public function getDescription(){       return $this->description;  }
}
class Product{
        private $titre;
        private $description;
        use GetInfoTrait;
        public function __construct($titre, $description){
                $this->titre = $titre;
                $this->description = $description;
        }
}
```

Héritage
Les traits

Exemple :

Lors de l'appel :

```
$product = new Product("Boîte", "Ceci est une boîte très spéciale.");
$titre = $product->getTitre();
$description = $product->getDescription();
echo "<p><b> $titre </b> : $description </p> ";
```

Héritage
Les traits

Comment sont gérés les conflits de noms entre la définition **d'une méthode de Classe et d'une méthode de Trait** ?

- ➔ Une méthode héritée depuis la classe de base est écrasée par celle provenant du Trait.
- ➔ Ce n'est pas le cas des méthodes réelles, écrites dans la classe de base.

Héritage
Les traits

Exemple : Surcharge du parent (le trait l'emporte)

Définitions des classes :

```
class BaseClass {
   public function fonctionPrint() {      echo 'fonctionPrint (parent)';   }
}
trait MyTrait {
   public function fonctionPrint() {      echo 'fonctionPrint (trait)';      }
}
class MyClass extends BaseClass {
   use MyTrait;
}
```

Lors de l'appel :

```
$instance = new MyClass();
$instance->fonctionPrint(); // fonctionPrint (trait)
```

Héritage
Les traits

Exemple : Surcharge de l'enfant (l'enfant l'emporte)

Définitions des classes :

```
trait MyTrait {
    public function fonctionPrint() {            echo 'fonctionPrint (trait)';    }
}
class MyClass {
    use MyTrait;
        public function fonctionPrint() {       echo 'fonctionPrint (enfant)';    }
}
```

Lors de l'appel :

```
$instance = new MyClass();
$instance->fonctionPrint(); // fonctionPrint (enfant)
```

Héritage
Les traits

Utilisation de plusieurs traits :

➔ Une classe peut utiliser de multiples Traits en les déclarant avec le mot-clé use, séparés par des virgules.

```
trait Hello {     public function sayHello()  {        echo 'Hello ';     }  }

trait World {     public function sayWorld() {        echo 'World';    }   }

class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {        echo '!';    }
}

$o = new MyHelloWorld();
$o->sayHello();                    // Hello
$o->sayWorld();                    // World
$o->sayExclamationMark();          // !
```

Héritage
Les traits

Comment sont gérés les conflits de noms entre la définition **de plusieurs méthodes de Traits** ?

➔ Mot-clé “insteadof” : permet de choisir une méthode en conflit plutôt qu’une autre.

➔ Mot-clé “as” : peut être utilisé pour permettre l'inclusion d'une des méthodes conflictuelles sous un autre nom.

Héritage
Les traits

Exemple :

```php
<?php
trait A {
    public function smallTalk() {
        echo 'a';
    }
    public function bigTalk() {
        echo 'A';
    }
}

trait B {
    public function smallTalk() {
        echo 'b';
    }
    public function bigTalk() {
        echo 'B';
    }
}

class Talker {
    use A, B {
        B::smallTalk insteadof A;    // b
        A::bigTalk insteadof B;      // A
    }
}

class Aliased_Talker {
    use A, B {
        B::smallTalk insteadof A;    // b
        A::bigTalk insteadof B;      // A
        B::bigTalk as talk;          // B
    }
}

$talker = new Talker();
$talker->smallTalk();       // b
$talker->bigTalk();         // A

$aliasTalker = new Aliased_Talker();
$aliasTalker->smallTalk(); // b
$aliasTalker->bigTalk();    // A
$aliasTalker->talk();               // B

?>
```

Héritage
Les traits

En utilisant la syntaxe as, vous pouvez aussi ajuster la visibilité de la méthode dans la classe qui l'utilise.

```php
<?php
trait HelloWorld {
    public function sayHello() {        echo 'Hello World!';    }
}

// Modification de la visibilité de la méthode sayHello
class MyClass1 {
    use HelloWorld { sayHello as protected; }
}

// Utilisation d'un alias lors de la modification de la visibilité
// La visibilité de la méthode sayHello n'est pas modifiée
class MyClass2 {
    use HelloWorld { sayHello as private myPrivateHello; }
}
?>
```

```php
$myClass = new MyClass1();
$myClass->sayHello();               // protected

$myClass = new MyClass2();
$myClass->sayHello();               // public
$myClass->myPrivateHello();         // private
```

Héritage

Les traits composés de traits

Tout comme les classes peuvent utiliser des traits, d'autres traits le peuvent aussi. Un trait peut donc utiliser d'autres traits et hériter de tout ou d'une partie de ceux-ci.

```php
<?php
trait Hello {
    public function sayHello() {        echo 'Hello ';    }
}
trait World {
    public function sayWorld() {        echo 'World!';    }
}
trait HelloWorld {    use Hello, World;    }

class MyHelloWorld {    use HelloWorld; }

$o = new MyHelloWorld();
$o->sayHello();      // Hello
$o->sayWorld();      // World!
?>
```

Héritage

Méthodes abstraites dans les Traits

Les traits supportent l'utilisation de méthodes abstraites afin d'imposer des contraintes aux classes sous-jacentes.

```
<?php
trait Hello {
    public function sayHelloWorld() {        echo 'Hello'.$this->getWorld();    }
    abstract public function getWorld();
}

class MyHelloWorld {
    private $world;
    use Hello;
    public function getWorld() {             return $this->world;    }
    public function setWorld($val) {         $this->world = $val;    }
}

$instance = new MyHelloWorld();
$instance->setWorld(" World!");
$instance->sayHelloWorld(); // Hello World!
?>
```

Héritage
Propriétés dans les Traits

Les traits peuvent aussi définir des propriétés.

```php
<?php
trait PropertiesTrait {
    public $x = 1;
}

class PropertiesExample {
    use PropertiesTrait;
}

$example = new PropertiesExample;
$example->x;

?>
```

6.

Namespace

Présentation

Déclaration d’un namespace

Déclaration d’un sous-namespace

Déclarer plusieurs namespaces

Utiliser les namespaces

Alias

Namespace

Présentation

Que sont les espaces de noms ? Dans leur définition la plus large, ils représentent un moyen d'encapsuler des éléments.

Dans le monde PHP, les espaces de noms sont conçus pour résoudre deux problèmes que rencontrent les auteurs de bibliothèques et d'applications lors de la réutilisation d'éléments tels que des classes ou des bibliothèques de fonctions :

- ➔ Collisions de noms entre le code que vous créez, les classes, fonctions ou constantes internes de PHP, ou celle de bibliothèques tierces.
- ➔ La capacité de faire des alias ou de raccourcir des Noms_Extrememement_Long pour aider à la résolution du premier problème, et améliorer la lisibilité du code.

Les espaces de noms PHP fournissent un moyen pour regrouper des classes, interfaces, fonctions ou constantes.

Namespace
Déclaration d'un namespace

Les espaces de noms sont déclarés avec le mot-clé namespace.
Un fichier contenant un espace de noms doit déclarer l'espace au début du fichier, avant tout autre code.

Exemple :

```php
<?php
namespace MonProjet;

const CONNEXION_OK = 1;
class Connexion { /* ... */ }
function connecte() { /* ... */  }

?>
```

Namespace
Déclaration d'un sous-namespace

Comme pour les fichiers et les dossiers, les espaces de noms sont aussi capables de spécifier une hiérarchie d'espaces de noms. Ainsi, un espace de noms peut être défini avec ses sous-niveaux :

Exemple :

```php
<?php
namespace MonProjet\Sous\Niveau;

const CONNEXION_OK = 1;
class Connexion { /* ... */ }
function connecte() { /* ... */  }

?>
```

Namespace

Déclarer plusieurs namespaces

Plusieurs espaces de noms peuvent aussi être déclarés dans le même fichier. Il y a deux syntaxes autorisées.

Syntaxe 1 :

```php
<?php
namespace MonProjet;

const CONNEXION_OK = 1;
class Connexion { /* ... */ }
function connecte() { /* ... */  }

namespace AutreProjet;

const CONNEXION_OK = 1;
class Connexion { /* ... */ }
function connecte() { /* ... */  }
?>
```

Syntaxe 2 :

```php
<?php
namespace MonProjet {
        const CONNEXION_OK = 1;
        class Connexion { /* ... */ }
        function connecte() { /* ... */  }
}

namespace AutreProjet {
        const CONNEXION_OK = 1;
        class Connexion { /* ... */ }
        function connecte() { /* ... */  }
}
?>
```

Namespace
Déclarer plusieurs namespaces

Il est fortement recommandé, en tant que pratique de codage, de ne pas mélanger plusieurs espaces de noms dans le même fichier.

Pour combiner plusieurs codes sans espace de noms dans du code avec espace de noms, seule la syntaxe à accolades est supportée. Le code global doit être encadré par un espace de noms sans nom

Exemple :

```php
<?php
namespace MonProjet {
	const CONNEXION_OK = 1;
	class Connexion { /* ... */ }
	function connecte() { /* ... */  }
}
namespace { // code global
	session_start();
	$a = MonProjet\connecte();
	echo MonProjet\Connexion::start();
} ?>
```

Namespace
Utiliser les namespaces

Pour utiliser un symbole du namespace :

- ➔ Il faut importer le fichier contenant la définition des namespaces
- ➔ Il faut préfixer les appels (variables, constantes, fonctions, classes) avec le nom du namespace

Exemple :

Définition du namespace dans le fichier "MonProjetNamespace.php"

```
<?php
namespace MonProjet ;

class Connexion {
        public function __construct(){          echo 'Classe "Connexion" créée<br/>';     }

        public function start(){            echo 'Connexion en cours<br/>';   }
}
function connecte() {       echo 'Vous êtes connecté !<br/>';}
?>
```

Namespace

Utiliser les namespaces

Exemple :

Utilisation du namespace dans le fichier “index.php”

```
<!doctype html>
<html>
<head>
        <meta charset="utf-8" />
</head>
<body>
<?php

require_once('MonProjetNamespace.php');

$connexion = new MonProjet\Connexion();          // Classe "Connexion" créée
$connexion->start();                             // Connexion en cours
MonProjet\connecte();                            // Vous êtes connecté !
?>
</body>
```

Namespace

Utiliser les namespaces

Pour éviter de préfixer chaque appel de symbole par le nom du namespace, on peut utiliser le namespace en le déclarant au début du fichier PHP.

Exemple :

Utilisation du namespace dans le fichier "index.php"

```
<?php
        namespace MonProjet;
        require_once('MonProjetNamespace.php');
?>
<!doctype html>
<html>
<head>        <meta charset="utf-8" />   </head>
<body>
<?php
        connecte();                          // Appel de MonProjet\connecte()
        $connexion = new Connexion();   // Appel de la classe MonProjet\Connexion
        $connexion->start();
?>
</body>
```

Namespace
Alias

Dans le cas où le nom d'un sous-namespace est très long, il est possible d'utiliser un alias.

Exemple :
Définition d'un sous-namespace dans le fichier "SubNamespace.php"

```
<?php

namespace MonProjet\Nested\Sub\Sub\Sub ;

$maVariable = "Ma variable";
const CONSTANTE = "Constante";

?>
```

Namespace
Alias

Exemple :
Utilisation d'un alias pour un sous-namespace dans le fichier "index.php"

```
<?php
	namespace MonProjet;
	require_once('MonProjetNamespace.php');
	require_once('SubNamespace.php');
	use MonProjet\Nested\Sub\Sub\Sub as SubNamespace;
?>
<!doctype html>
<html>
<head>
	<meta charset="utf-8" />
</head>
<body>
<?php  echo SubNamespace\CONSTANTE; ?> // Appelle MonProjet\Nested\Sub\Sub\Sub\CONSTANTE
</body>
```

7.

Gestion des erreurs et exceptions

Présentation

La classe “Exception”

Etendre la classe “Exception”

Gestion des erreurs et exceptions

Présentation

PHP 7 change la façon dont la plupart des erreurs sont signalées par PHP. Au lieu de signaler des erreurs au moyen du mécanisme de déclaration d'erreur traditionnel utilisé par PHP 5, la plupart des erreurs sont maintenant signalées en lançant des exceptions d'erreur .

Les exceptions sont provoqués par l'utilisation d'un code source dans une condition "exceptionnelle".

Gestion des erreurs et exceptions

Présentation

Les exceptions peuvent être développés dans le cas où certaines informations requises ne dépendent pas uniquement du code source :

- ➔ Données saisies par l'utilisateur
- ➔ Données provenant d'une base de données

Le gestion des exceptions peut permettre éventuellement de ne pas provoquer une erreur fatale et de continuer le déroulement du script si cela est possible (en informant l'utilisateur et re-demander une saisie par exemple).

Gestion des erreurs et exceptions

Présentation

Une exception peut être lancée ("throw") et attrapée ("catch") dans PHP.

- ➔ Le code devra être entouré d'un bloc try pour faciliter la saisie d'une exception potentielle.
- ➔ Chaque try doit avoir au moins un bloc catch ou finally correspondant.
- ➔ L'objet lancé doit être une instance de la classe Exception ou une sous-classe de la classe Exception.

Gestion des erreurs et exceptions

Présentation

- ➔ Bloc “try” : permet d’entourer l’appel d’instructions
- ➔ Blocs “catch” : permet d’attraper les exceptions (il peut en avoir plusieurs)
- ➔ Bloc “finally” : peut aussi être spécifié après des blocs catch. Le code à l'intérieur du bloc finally sera toujours exécuté après les blocs try et catch, indépendamment du fait qu'une exception ait été lancée, avant de continuer l'exécution normale.

Gestion des erreurs et exceptions
Présentation

Exemple :

```
<?php
echo "<h1>Bienvenue dans notre calculatrice !</h1>";
function inverse($x) {
    if (!$x) {       throw new Exception('La division par zéro est interdite !<br/>');    }
    return 1/$x;
}
try {
    echo inverse(5) . "<br/>";
    echo inverse(0) . "<br/>";
} catch (Exception $e) {
    echo '<br/>Exception reçue : ',  $e->getMessage(), "<br/>";
}
// l'execution continue
echo "<blockquote>Merci d'avoir utilisé notre calculatrice !</blockquote>";
?>
```

Gestion des erreurs et exceptions

La classe Exception

Pour arriver à gérer une exception, il faut s'intéresser de plus près à la classe Exception de PHP.

```
Exception {

   /* Propriétés */
   protected string $message ;
   protected int $code ;
   protected string $file ;
   protected int $line ;

   /* Méthodes */
   public __construct ([ string $message = "" [, int $code = 0 [, Throwable $previous = NULL ]]] )
   final public string getMessage ( void )
   final public Exception getPrevious ( void )
   final public mixed getCode ( void )
   final public string getFile ( void )
   final public int getLine ( void )
   final public array getTrace ( void )
   final public string getTraceAsString ( void )
   public string __toString ( void )
   final private void __clone ( void )
}
```

Lien du site officiel : http://php.net/manual/fr/class.exception.php

Gestion des erreurs et exceptions

La classe Exception

Nous pouvons donc accéder à ses méthodes pour obtenir des informations sur l'exception. Par exemple :

- ➔ Exception::**__construct** — Construit l'exception
- ➔ Exception::**getMessage** — Récupère le message de l'exception
- ➔ Exception::**getCode** — Récupère le code de l'exception
- ➔ Exception::**getFile** — Récupère le fichier dans lequel l'exception est survenue
- ➔ Exception::**getLine** — Récupère la ligne dans laquelle l'exception est survenue

- ➔ Exception::**__toString** — Représente l'exception sous la forme d'une chaîne

Gestion des erreurs et exceptions

Etendre la classe Exception

Si vous voulez prendre la main sur la gestion des exceptions de la classe Exception, cela est possible. Nous pouvons étendre la classe Exception et définir de nouvelles méthodes personnalisées.

- ➔ La plupart des méthodes de la classe Exception sont finales et ne peuvent donc pas être surchargées.
- ➔ La méthode “__toString()” peut être surchargée pour modifier l’affichage de l’exception.
- ➔ Le méthode “__construct()” peut être surchargée mais il est conseillé d’utiliser “parent::__construct();” pour pouvoir enregistrer les informations de base de la classe.

Gestion des erreurs et exceptions
Etendre la classe Exception

Exemple :

Définition de la classe personallisée d'exception

```
class MyException extends Exception{
        public function __toString(){
                return "<b>Exception : </b>{$this->getMessage()}
                        <b>Dans le fichier : </b>{$this->getFile()} <br/>
                        <b>A la ligne : </b>{$this->getLine()} <br/>
                        <b>Trace : </b>{$this->getTraceAsString()} <br/>";
        }
}
```

Gestion des erreurs et exceptions
Etendre la classe Exception

Exemple :

Modification de la provocation d'exception

```
<?php
/* … */
   if (!$x) {       throw new MyException('La division par zéro est interdite !<br/>');    }
/* … */
catch (Exception $e) { // Le polymorphisme nous permet de conserver "Exception"
   echo $e;
}
/* … */

?>
// L'exception s'affiche alors tel que nous l'avons défini dans la méthode
"__toString()" de la classe MyException.
```

8.

PHP Data Objects (PDO)

Présentation

Prérequis

Connexion

Connexion persistante

Opération de récupération

Opération de modification

Transaction

Requêtes préparées

PHP Data Objects (PDO)

Présentation

L'extension PHP Data Objects (PDO) définit une excellente interface pour accéder à une base de données depuis PHP.

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

PHP Data Objects (PDO)

Prérequis

PDO ainsi que tous les pilotes principaux interagissent avec PHP en tant qu'extensions partagées, et ont tout simplement besoin d'être activés en éditant le fichier php.ini :

extension=php_pdo.dll

Remarque : Si l'extension est commentée, il faut la décommenter en enlevant le dièse (#) préfixant la ligne.

PHP Data Objects (PDO)

Connexion

Les connexions sont établies en créant des instances de la classe de base de PDO. Peu importe quel pilote vous voulez utiliser ; vous utilisez toujours le nom de la classe PDO.

Le constructeur accepte des paramètres :

- ➔ La source de la base de données (connue en tant que DSN)
- ➔ Le nom d'utilisateur
- ➔ Le mot de passe (s'il y en a un)

PHP Data Objects (PDO)
Connexion

Exemple :

```php
<?php

$user = '...';
$pass = '...';


        $dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);

?>
```

PHP Data Objects (PDO)
Connexion

Il est conseillé d'utiliser les blocs "try" et "catch" au cas où la connexion n'a pas pu être établie. Une exception de type "PDOException" est générée en cas de problème.

Le constructeur accepte des paramètres :

- ➔ La source de la base de données (connue en tant que DSN)
- ➔ Le nom d'utilisateur
- ➔ Le mot de passe (s'il y en a un).

PHP Data Objects (PDO)

Connexion

Exemple :

```php
<?php

$user = '...';
$pass = '...';

try {

    $dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
    $dbh = null;       // Fermeture de la connexion

} catch (PDOException $e) {
    print "Erreur !: " . $e->getMessage() . "<br/>";
    die();             // Arrêt du script
}

?>
```

PHP Data Objects (PDO)

Connexion

Lorsque la connexion à la base de données a réussi, une instance de la classe PDO est retournée à votre script. La connexion est active tant que l'objet PDO l'est.

Pour clore la connexion, vous devez détruire l'objet en vous assurant que toutes ses références sont effacées. Vous pouvez faire cela en assignant NULL à la variable gérant l'objet.

Si vous ne le faites pas explicitement, PHP fermera automatiquement la connexion lorsque le script arrivera à la fin.

PHP Data Objects (PDO)
Connexion

Exemple :

```php
<?php

$user = '...';
$pass = '...';

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
// utiliser la connexion ici


// et maintenant, fermez-la !
$dbh = null;

?>
```

PHP Data Objects (PDO)

Connexion persistantes

Beaucoup d'applications web utilisent des connexions persistantes aux serveurs de base de données.

Les connexions persistantes ne sont pas fermées à la fin du script, mais sont mises en cache et réutilisées lorsqu'un autre script demande une connexion en utilisant les mêmes paramètres.

Le cache des connexions persistantes vous permet d'éviter d'établir une nouvelle connexion à chaque fois qu'un script doit accéder à une base de données, rendant l'application web plus rapide.

PHP Data Objects (PDO)
Connexion

Exemple :

```php
<?php

$user = '...';
$pass = '...';

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass, array(
    PDO::ATTR_PERSISTENT => true
));

?>
```

PHP Data Objects (PDO)
Opérations de récupérations

La façon la plus simple d'effectuer des opérations de récupérations avec la classe PDO est d'utiliser sa méthode "query()". Elle prend en argument une requête SQL classique.

Exemple :

Fonction de récupération de produits :

```
function getProduit($conn) {
        $sql =  'SELECT * FROM materiel';
        return $conn->query($sql);
}
```

PHP Data Objects (PDO)
Opérations de récupérations

Exemple :

Affichage des produits dans une liste à puces :

```
$dbh = new PDO("…");
$result = getProduit($dbh);
foreach  ($result as $row) {        ?>
        <ul>
                <li><?php echo $row['libelle']; ?></li>
                <li><?php echo $row['description']; ?></li>
                <li><?php echo $row['poids']; ?>g</li>
                <li><?php echo $row['prix']; ?>€</li>
        </ul>
<?php
}                       // Fermeture du foreach
$dbh = null;            // Fermeture de la connexion
?>
```

PHP Data Objects (PDO)
Opérations de modifications

Pour effectuer des modifications (création, modification, suppresion), on utilise la méthode “exec()”.

Exemple :

Fonction d’ajout d’un produit “Chaussure2” :

```
function addChaussure($connexion) {
        return $connexion->exec('INSERT INTO materiel (libelle, description, poids, prix)
                VALUES (\'Chaussure2\', \'Ceci est une chaussure\', 200, 100)');
}
```

Appel de la fonction :

```
$nbColumns = addChaussure($dbh); // Retourne le nombre de lignes affectées
```

PHP Data Objects (PDO)

Transactions

Il est aussi possible d'utiliser les transactions pour gérer ses requêtes SQL.

4 fonctionnalités majeures :
Atomicité, Consistance, Isolation et Durabilité (ACID).

- ➔ Les transactions sont typiquement implémentées pour appliquer toutes vos modifications en une seule fois ; ceci a le bel effet d'éprouver radicalement l'efficacité de vos mises à jour. En d'autres termes, les transactions rendent vos scripts plus rapides et potentiellement plus robustes.
- ➔ Le travail mené à bien dans une transaction, même s'il est effectué par étapes, est garanti d'être appliqué à la base de données sans risque, et sans interférence pour les autres connexions, quand il est validé.

PHP Data Objects (PDO)

Transactions

Exemple :

Fonction d'une transaction d'ajout :

```
function addTransaction($dbh){
        try {
          $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
          $dbh->beginTransaction();
          $dbh->exec("INSERT INTO materiel (libelle, description, poids, prix)
                VALUES ('Botte5', 'Ceci est une botte', 200, 100)");
          $dbh->exec("INSERT INTO materiel (libelle, description, poids, prix)
                VALUES ('Chaussette5', 'Ceci est une chaussette', 200, 100)");
          $dbh->commit();
        } catch (Exception $e) {
                $dbh->rollBack();    echo "Failed: " . $e->getMessage();
        }
}
```

Appel de la fonction :

```
addTransaction($dbh);
```

PHP Data Objects (PDO)

Requêtes préparées

La plupart des bases de données supportent le concept des requêtes préparées. Qu'est-ce donc ? Vous pouvez les voir comme une sorte de modèle compilé pour le SQL que vous voulez exécuter, qui peut être personnalisé en utilisant des variables en guise de paramètres. Les requêtes préparées offrent deux fonctionnalités essentielles :

- ➔ La requête ne doit être analysée (ou préparée) qu'une seule fois, mais peut être exécutée plusieurs fois avec des paramètres identiques ou différents.
- ➔ Les paramètres pour préparer les requêtes n'ont pas besoin d'être entre guillemets ; le pilote gère cela pour vous.

PHP Data Objects (PDO)
Requêtes préparées

Les requêtes préparées proposent 2 marqueurs pour les paramètres :

- ➔ Les marqueurs nommés : on nomme les paramètres
- ➔ Les marqueurs “ ? ” : les paramètres sont identifiés par ordre

PHP Data Objects (PDO)

Requêtes préparées

Exemple : Requête préparée avec marqueurs nommés

```php
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// insertion d'une ligne
$name = 'one';
$value = 1;
$stmt->execute();

// insertion d'une autre ligne avec des valeurs différentes
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

PHP Data Objects (PDO)

Requêtes préparées

Exemple : Requête préparée avec marqueurs “ ? ”

```php
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (?, ?)");
$stmt->bindParam(1, $name);
$stmt->bindParam(2, $value);

// insertion d'une ligne
$name = 'one';
$value = 1;
$stmt->execute();

// insertion d'une autre ligne avec différentes valeurs
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

9.

Design pattern

Présentation

Familles

Création

Structuraux

Comportement

Exemple avec le singleton

Design pattern
Présentation

En informatique, et plus particulièrement en développement logiciel, un patron de conception (souvent appelé design pattern) est un arrangement caractéristique de modules, reconnu comme **bonne pratique en réponse à un problème de conception** d'un logiciel.

Il décrit une solution standard,
utilisable dans la conception de différents logiciels.

Design pattern
Présentation

- Un patron de conception est issu de l'expérience des concepteurs de logiciels. Il décrit un arrangement récurrent de rôles et d'actions joués par des modules d'un logiciel, et le nom du patron sert de vocabulaire commun entre le concepteur et le programmeur.

- D'une manière analogue à un motif de conception en architecture, le patron de conception décrit les grandes lignes d'une solution, qui peuvent ensuite être modifiées et adaptées en fonction des besoins.

- Les patrons de conception décrivent des procédés de conception généraux et permettent en conséquence de capitaliser l'expérience appliquée à la conception de logiciel. Ils ont une influence sur l'architecture logicielle d'un système informatique.

Design pattern
Familles

On distingue trois familles de patrons de conception selon leur utilisation :

- ➔ **Construction** : ils définissent comment faire l'instanciation et la configuration des classes et des objets.

- ➔ **Structuraux** : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).

- ➔ **Comportementaux** : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

Design pattern
Création

- ➔ Fabrique abstraite (Abstract Factory)
- ➔ Monteur (Builder)
- ➔ Fabrique (Factory Method)
- ➔ Prototype (Prototype)
- ➔ Singleton (Singleton)

Design pattern
Structuraux

- ➔ Adaptateur (Adapter)
- ➔ Pont (Bridge)
- ➔ Objet composite (Composite)
- ➔ Décorateur (Decorator)
- ➔ Façade (Facade)
- ➔ Poids-mouche ou poids-plume (Flyweight)
- ➔ Proxy (Proxy)

Design pattern
Comportement

- ➔ Chaîne de responsabilité (Chain of responsibility)
- ➔ Commande (Command)
- ➔ Interpréteur (Interpreter)
- ➔ Itérateur (Iterator)
- ➔ Médiateur (Mediator)
- ➔ Mémento (Memento)
- ➔ Observateur (Observer)
- ➔ État (State)
- ➔ Stratégie (Strategy)
- ➔ Patron de méthode (Template Method)
- ➔ Visiteur (Visitor)
- ➔ Fonction de rappel (Callback)

Design pattern
Singleton

Le Singleton, en programmation orientée objet, répond à la problématique de n'avoir qu'une seule et unique instance d'une même classe dans un programme.

Par exemple, dans le cadre d'une application web dynamique, la connexion au serveur de bases de données est unique. Afin de préserver cette unicité, il est judicieux d'avoir recours à un objet qui adopte la forme d'un singleton.

Design pattern

Singleton

Exemple : Classe SingletonConfig

```php
<?php

class SingletonConfig
{

    private $settings = [];
    private static $_instance; // L'attribut qui stockera l'instance unique

    /*
     * La méthode statique qui permet d'instancier ou de récupérer l'instance unique
     */
    public static function getInstance($file)
    {

    /*
     * Le constructeur avec sa logique est privé pour empêcher l'instanciation en dehors de la classe
     */
    private function __construct($file)
    {

    /*
     *  Permet d'obtenir la valeur de la configuration
     */
    public function get($key)
    {

    /*
     *  Permet d'empêcher le clonage
     */
     public function   clone(){

}
```

Design pattern
Singleton

Exemple : Fichier de configuration “config.php”

```
<?php

        return array(
                "db_user" => "root",
                "db_pass" => "",
                "db_host" => "localhost",
                "db_name" => "testpoo"
        );

?>
```

Design pattern
Singleton

Exemple : Constructeur du singleton

```
private function __construct($file)
{
        echo '<p>Entrée dans le constructeur</p>';
        $this->settings = require($file); // Retourne un tableau
}
```

Design pattern
Singleton

Exemple : Retourne l'instance unique du singleton

```
public static function getInstance($file)
{
    if (is_null(self::$_instance)) {
        self::$_instance = new SingletonConfig($file);
    }
    return self::$_instance;
}
```

Design pattern
Singleton

Exemple : Retourne une info du tableau

```
public function get($key)
{
        if (!isset($this->settings[$key])) {
                return null;
        }
        return $this->settings[$key];
}
```

Exemple : Empêche le clônage

```
public function __clone(){
        throw new Exception('Cet objet ne peut pas être cloné');
}
```

Design pattern
Singleton

Exemple : Utilisation du Singleton

```
<?php

require_once('Singleton.php');

// Création
$settings = SingletonConfig::getInstance("config.php");
var_dump($settings);

// Récupération
$settings = SingletonConfig::getInstance("config.php");
var_dump($settings);
```

Design pattern
Singleton

Exemple : Test de clônage du Singleton

```
// Test de clonage
$cloneSettings = clone $settings; // Exception
var_dump($cloneSettings);
```

Design pattern
Singleton

Exemple : Utilisation des informations de la configuration pour une connexion à la BDD

```
// Connexion BDD
try{
        $db_user = $settings->get("db_user");
        $db_pass = $settings->get("db_pass");
        $db_host = $settings->get("db_host");
        $db_name = $settings->get("db_name");

        $dbh = new PDO('mysql:host='.$db_host.';dbname='.$db_name, $db_user, $db_pass);
        echo 'Connexion ok';
} catch (PDOException $e) {
        echo 'Connexion ko';
        echo '<h2>Erreur lors de la connexion à la base de données</h2>
        <p>Motif : '. $e->getMessage() .'</p>';
        die();
}
```

Merci de votre attention !

=)