[IMAC S2] Programmation Back-end

# TD 02 : Formulaires et PHP Objet

**Support**

Pascale Ho - 2018 - Ingénieur IMAC

## TL;DR

### Formulaires

- Les attributs action et method de la balise <form> permettent l'interaction via un formulaire
- PHP récupère selon la méthode HTTP les informations du formulaire dans des superglobales (tableaux associatifs) $_GET et $_POST (et $_REQUEST)
- Les clés de ces superglobales correspondent à la valeur de l'attribut name des champs du formulaire
- Un autre moyen de faire sa requête GET se fait via l'URL

### Programmation Objet

- La Programmation Orientée Objet (POO) est un concept d'organisation du code informatique, sur la base d'**objets**
- Classes :
    - Modèle à partir duquel on va définir des **attributs** (variables) et des **méthodes** (fonctions)
    - Chaque méthode et attribut possède un **niveau de visibilité** (public, private, protected)
    - On accède à un attribut ou une méthode avec la syntaxe fléchée **->**
    - Pour faire référence à l'objet courant dans la description de la classe, on utilise la variable **$this**
    - Méthodes basiques d'une classe : **constructeur, getters, setters**
    - On instancie un objet d'une classe avec le mot clé **new**
- Héritage :
    - Spécialisation d'une classe par rapport à une autre
    - On parle de **classe fille** la classe qui récupère les attributs et les méthodes de la **classe mère**

    - Une **surcharge** est une redéfinition d'une méthode de la classe fille qui existe dans la classe parente
    - Une classe mère **abstraite** sert de template pour les classes filles qui doivent surcharger les méthodes abstraites du parent
- Les **exceptions** sont des événements qui empêchent la poursuite normale de l'exécution du code et qu'on souhaite gérer

# Formulaires

Rappelez vous, on va devoir pour le projet concevoir un système d'échange de données entre le navigateur (client) et le serveur (l'application) par le protocole HTTP. Pour commencer on va voir plus en détail un moyen classique d'interaction client-serveur que sont les formulaires.

Plusieurs cas d'utilisation de formulaires :

- créer un espace sécurisé par authentification,
- uploader des fichiers,
- administrer le contenu d'un site,
- gérer un panier d'articles,
- etc.

## Rappels HTML et CM 01

https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms

Les formulaires sont encapsulés dans une balise **<form>**, qui sert de conteneur d'éléments comme des boutons, des zones de texte, etc. Ces éléments seront envoyés au serveur Web.

Il y a deux attributs à indiquer dans la balise form :

- **action :** url destinatrice des données saisies après soumission (l'envoi est déclenché grâce à un input/button de type submit ou appui sur la touche Entrée du clavier)
- **method :** méthode de requête (GET par défaut, POST)

```
<form method="POST" action="http://ingenieur-imac.fr/mail/contact.php"
accept-charset="UTF-8">
  <input placeholder="Votre e-mail*" required="required" name="email"
type="email">
  <textarea placeholder="Message*" required="required" name="message" cols="50"
  rows="10"></textarea>
  <input value="Envoyer" type="submit">
</form>
```

## Traitement du formulaire

Après soumission du formulaire, les informations sont récupérées par le serveur Web. En PHP, les données sont contenues dans des variables **superglobales** de type **tableaux associatifs** :

- $_GET : contient toutes les données envoyées via GET (ou directement via l'URL)
- $_POST : contient les données envoyées via POST
- $_REQUEST : combinaison des deux

**Qu'est-ce qu'une superglobale ?**
Ce sont des variables internes qui sont accessibles dans tout le script PHP.

On peut retrouver une valeur envoyée par HTTP grâce à une clé qui porte le même nom (attribut **name**) que le champ du formulaire.

Si on reprend l'exemple du formulaire précédent, on pourrait créer un script d'envoi de mail ultra simplifié (avec une sécurité ultra minime, alors ne faîtes pas comme cela merci). Le script **contact.php** (comme la valeur de l'attribut action du formulaire) contient ceci :

```php
<?php
// $_POST['email'] et $_POST['message'] sont-ils définis ?
if ( isset($_POST['email']) && isset($_POST['message'])) {
    // $_POST['email'] et $_POST['message'] sont-ils vides ?
    if ( !empty($_POST['email']) && !empty($_POST['message']))
        // envoi d'un mail
        sendMail($_POST['email'], $_POST['message']);
    else
      echo "Utilisation Incorrecte";
}
?>
```

Pour votre culture : il faut savoir que la balise form a un attribut **enctype** qui possède par défaut la valeur **application/x-www-form-urlencoded**. Ceci permet d'indiquer le "type" des informations issues du formulaire. Il s'agit d'une chaîne de caractères qui prend cette forme :

```
nom1=val1&nom2=val2 ...
```

Cette chaîne est encodée :

- Les caractères non alphanumériques sont remplacés par %xx, avec xx qui correspond au code ASCII du caractère
- Les espaces sont remplacés par des **+**.

## Formulaire GET

GET est de base la méthode par défaut d'un formulaire. Le formulaire va envoyer une requête **HTTP** de type **GET**. Les données envoyées sont passées directement via l'URL (et sont encodées). Les variables se récupèrent à l'aide de la superglobale **$_GET**.

**Rappel : d'après REST on va se servir de la méthode GET pour lire les informations.**

## Formulaire POST

La transmission des informations saisies depuis le formulaire se fait dans le corps de la requête **HTTP** en **POST**. Les variables se récupèrent à l'aide de la superglobale **$_POST**.

**Rappel : d'après REST on va se servir de la méthode POST pour modifier une information.**

## Cas particulier de champs de formulaire

Il existe un **cas particulier** de récupération de données de formulaire. Comment récupérer les informations lorsqu'on a un champ à choix multiple (checkbox, select avec attribut multiple) ? Il faut récupérer l'information dans un **tableau.** Dans ce cas on doit indiquer dans l'attribut name du champ qu'il s'agit d'un tableau.

Dans cet exemple, on veut sélectionner des jours de la semaine. Notre formulaire ressemble à ceci :

```
<form name="daySelection" action="script.php" method="GET">
    <li><input type="checkbox" name="day[]" value="Monday">Lundi</li>
    <li><input type="checkbox" name="day[]" value="Tuesday">Mardi</li>
    <li><input type="checkbox" name="day[]" value="Wednesday">Mercredi</li>
    <li><input type="checkbox" name="day[]" value="Thursday">Jeudimac</li>
    <li><input type="checkbox" name="day[]" value="Friday">Vendredi</li>
    <li><input type="checkbox" name="day[]" value="Saturday">Samedi</li>
    <li><input type="checkbox" name="day[]" value="Sunday">Dimanche</li>
</form>
```

Lorsqu'on envoie le formulaire avec Jeudi et Vendredi comme valeurs sélectionnées, l'URL envoyé en GET est `script.php?day[]=Jeudimac&day[]=Vendredi`

Contenu de script.php pour le traitement des champs :

```
<?php
    $texte = '';
    // La variable $_GET['day'] est-elle définie et non vide ?
    if ( isset($_GET['day']) && !empty($_GET['day']) ){
```

```
        $texte = "Vous avez choisi les journées suivantes : \n";
        // $_GET['day'] est un tableau
        foreach ($_GET['day'] as $day)
            $texte .= $day.'\n';
    }
    else
        $texte = "Vous avez choisi aucune journée";
    echo $texte;
?>
```

### Ordre de gestion d'un formulaire

1. Génération du premier formulaire
2. Envoi des données
3. Réception des données
4. Vérification de la validité des données
5. Si tout est OK, on passe au traitement (écriture bdd, exécution script, …)
6. Sinon régénération du formulaire, avec feedback au client

**Important : Ne jamais faire confiance aux données issues des formulaires ! Toujours vérifier les informations saisies (cf cours sur la sécurité qui viendra plus tard)**

## Transmission de paramètres via l'URL

On peut passer des arguments à un script directement dans l'url (comme pour la méthode GET)

syntaxe : url?var1=val1&var2=val2…

Les variables transmises dans le script PHP se retrouvent dans la superglobale **$_GET**.

Par exemple si PHP reçoit le lien `bonsoir.php?nom=Ho&prenom=Pascale`, il va remplir le tableau $_GET avec :

```
$_GET['nom'] = "Ho";
$_GET['prenom'] = "Pascale";
```

, puis exécuter le script bonsoir.php.

# Programmation objet en PHP

## Principe

La Programmation Orientée Objet (POO) est un concept d'organisation du code informatique. Elle est structurée autour des objets qui peuvent représenter un concept, une idée ou une entité. Chaque objet dispose d'une **structure de données** et comprend également les **fonctionnalités** de traitement des données.

On verra différents concepts associés à la POO : classe, héritage, exceptions, (interface, polymorphisme,) …

Ce principe de programmation s'applique dans plusieurs langages comme le PHP (normalement ce concept sera vu en détail en imac2 mais il est nécessaire que je vous le présente pour mieux réussir votre projet).

## Classe

Une classe définit un modèle, un moule à partir duquel tous les objets de la classe seront créés. Elle décrit les données internes ainsi que les fonctionnalités des objets. On appelle **instanciation** la construction d'un objet à partir d'une classe. Du coup on parle d'**instances** les objets de la classe.

### Encapsulation

On parle d'**encapsulation** le procédé qui consiste à rassembler les données/variables (**attributs**) et les fonctionnalités (**méthodes**). Les variables et les fonctions possèdent un **niveau de visibilité :**

- **public :** accessible par tous (par défaut)
- **private :** accessible uniquement qu'aux objets de la classe
- **protected :** accessible qu'aux objets de la classe et des classes dérivées (c.f. Héritage)

On accède à un attribut ou une méthode avec la syntaxe fléchée **->**. Pour faire référence à l'objet courant dans la description de la classe, on utilise la variable **$this**.

Les méthodes habituelles qu'on retrouve dans quasiment toutes les classes sont :

- constructeur / destructeur
- accesseurs / modificateurs (**getters / setters**)

### Grand exemple

Dans un fichier personne.class.php

```php
<?php
/* --- définition de la classe Personne --- */
class Personne {    // déclaration de classe "Personne"
    private $nom;    // attribut privé
    private $age = 25; // attribut avec valeur par défaut
    private $alive = true;

    // en php il ne peut y avoir qu'un seul constructeur
    public function __construct($nom) { // constructeur public
        $this->nom = (string) $nom; // this : référence à l'objet
courant
    }

    public function __destruct() { // destructeur public
        ...
    }

    public function getAge() { // getter
        return $this->age;
    }

    // un autre getter
    public function isAlive() {
        return $this->alive;
    }

    public function setAge($age){ // setter
        // le fait d'utiliser le mot-clé this permet de différencier à
        // qui appartient age
        $this->age = $age;
    }

    public function affiche() {
         echo "Nom : ".$this->nom;    // accès à l'attribut nom
    }
}
?>
```

dans un autre script .php :

```php
<?php
/* --- utilisation de la classe Personne --- */
```

```php
// inclusion de la définition de la classe Personne
require ('personne.class.php');
// instanciation : création d'un objet avec le mot-clé new
$personne = new Personne("Pascale");

// utilisation d'une méthode
$personne->affiche(); // affiche : Nom : Pascale

echo $personne->age; // impossible d'accéder ou de modifier l'attribut
                     // nom car il est privé
echo $personne->getAge(); // affiche : 25 (valeur par défaut)
$personne->setAge(1055);
echo $personne->getAge(); // affiche : 1055

// destruction de l'objet, appel du destructeur
unset($personne); // ou $personne = null;
?>
```

## Attributs et méthodes de classe (static)

On parle d'attribut et méthodes **statiques** des éléments utilisables sans besoin d'instance de la classe. Les attributs statiques **ne peuvent pas** être accédés depuis une instance ($objet->attribut ne peut pas fonctionner !) mais sont communes à toutes les instances de la classe. Par contre les méthodes statiques **peuvent** être accédés depuis une instance ( $objet->methode() ), mais $this n'est plus disponible.

```php
class MyOutilsMath {
    private static $n = 0; // attribut privé statique
    public static $pi = 3.1415926535897793; // attribut statique

    // fonction statique
    public static function addition($a, $b){
        return $a+$b;
    }

    /* utilisation d'un attribut static à l'intérieur d'une méthode de
la
    classe */
    public static function perimetreCercle($rayon){
        return 2 * $rayon * self::pi;
    }


    /* une autre façon d'accéder à un attribut statique à l'intérieur de
    la classe (mais on privilégie le mot-clé self) */
```

```
    public static function maFonction(){
        return MesOutilsMath::n;
    }
}

// Utilisation des attributs et méthodes statiques
echo MesOutilsMath::pi; // affiche : 3.141592653589793
echo MesOutilsMath::n; // impossible car attribut privé
echo MesOutilsMath::addition(2,3); // affiche 5
```

On va beaucoup de servir du mot-clé static lorsqu'on veut créer un **Singleton**, un design-pattern qui nous sera utile notamment lorsqu'on veut créer notre accès à la base de données (on verra cela plus tard).

Tiens, il est possible de déclarer des attributs **constantes** (variables à valeur constante) dans des classes ! On se servira du mot-clé **const** pour déclarer la constante et on les appelle de la même manière que pour les attributs statiques.

```
class PersonneImmortelle {
    const ALIVE = true; // un.e immortel.le ne meurt jamais !!
    ...
}

// accès à une constante de classe
echo PersonneImmortelle::ALIVE; // affiche true
$staying = new PersonneImmortelle();
$staying->ALIVE; // ne fonctionne pas
```

## Passage de paramètres

Depuis PHP5, le passage ou l'affectation d'un objet se fait par **référence** (Contrairement aux 'non-objets' qui sont passés par valeur). En gros le & est présent implicitement.

Pour pouvoir copier un objet pour ensuite manipuler la copie sans impacter l'original, il existe un moyen de **cloner** explicitement avec le mot-clé clone.

```
$obj2 = clone $obj1;
```

## Documentation

En commentant les classes il est possible de générer une documentation à l'aide d'outils comme phpDocumentor ou Doxygen (des IDE permettent également de le faire). L'intérêt de générer une documentation vient lorsqu'on (ou une autre personne utilisant votre projet)

souhaite comprendre le fonctionnement de notre code s'il est très fourni. Un code bien commenté permet une meilleure lisibilité.

```php
 <?php
/**
* Class Personne
* description de la classe
*/
class Personne {
    /**
    * @var string Nom de la personne
    */
    private $nom;
    /**
    * @var int Âge de la personne
    */
    private $age;
    /**
    * @param string $nom Nom de la personne
    * @param int $age Âge de la personne
    */
    public function __construct($nom, $age) {
        $this->nom = $nom;
        $this->age = $age;
    }
    /**
    * @return int
    */
    public function getAge() {
        return $this->age;
    }
}
```

# Héritage

## Principe

À force de créer des classes et des objets, il arrive qu'on aimerait avoir une classe qui est un type spécialisé d'une autre classe. On va prendre un cas concret (qui se souvient de la géométrie ?) : on a une classe Forme et une classe Carré, il serait utile de préciser qu'un carré est une forme et ainsi récupérer les propriétés et méthodes de la classe forme. Cela nous économiserait énormément de code surtout ! En POO, il existe un processus appelé **héritage** qui permet cela. L'héritage est un moyen pour une **classe fille** (carré) de récupérer les informations d'une **classe mère** (forme). On dit alors que la classe fille **étend** (ou

**spécialise**) la classe parent. Un carré est une forme, la classe Carré hérite de la classe Forme. Mais l'inverse ne peut pas être vraie (une forme n'est pas forcément un carré).

On parle de **surcharge** une redéfinition d'une méthode dans la classe fille, à condition que le prototype de la méthode soit compatible avec celui de la méthode de la méthode mère.

```php
// classe mère
class Forme {
    protected $nb_cotes = true; // accès à la variable dans la
                                // classe mère et les classes filles
    function __construct($var) {
        ...
    }

    function affiche() {
        echo "Je suis une forme";
    }

    function maFonction() {
         ...
    }
}

// classe fille héritant de la classe mère
class Carre extends Forme {
    protected $nb_cotes = 4;
    function __construct($var, $var2){
        parent::__construct($var); // constructeur de la classe parente
    }

    // surcharge de la méthode
    function affiche() {
        parent::affiche();  // appel de la méthode mère
        echo " mais aussi un carré";
    }
}

$forme = new Forme();
$carre = new Carre();
$forme->affiche();   // affiche : Je suis une forme
$carre->affiche();   // affiche : Je suis une forme mais aussi un carré
$carre->maFonction(); //appel de la méthode de la classe mère
```

### Classe abstraite

On peut déclarer une classe comme étant abstraite avec le mot-clé **abstract** pour empêcher la classe d'être instanciée (car … trop abstraite), en obligeant les classes filles à implémenter les méthodes abstraites. Pour que la classe soit abstraite elle doit posséder au moins une méthode abstraite. Notez bien qu'on utilise des classes abstraites dans le but de répondre à des besoins précis.

```
abstract class Forme {
    // méthode abstraite protégée
    abstract protected function nom();
    function affiche() {
        echo "Je suis {$this->nom()}";
    }
}

class Carre extends Forme {
    // implémentation de la méthode abstraite
    protected function nom() {
        return " un carré";
    }
}

$forme = new Forme(); // ne fonctionne pas
$carre = new Carre();
$carre->affiche(); // affiche : Je suis un carré
```

### Mot-clé final

Le mot-clé **final** peut servir à empêcher une méthode d'une classe à être surchargée dans ses classes filles. Si une classe est elle-même définie comme finale, cela veut dire qu'elle ne peut pas être héritée.

```
public classeMere {
    final public function maFonction(){ ... }
}
public classeFille extends classeMere {
    // tentative de surcharge impossible
    public function maFonction(){...}
}
```

## Exceptions

Des fois on souhaite gérer les erreurs qu'on pourrait rencontrer lors du déroulement de notre code. C'est le principe des **exceptions** : dans un certain contexte (base de données

inaccessible, mauvaise saisie de l'utilisateur, fichier non trouvé), on va lancer un événement qui empêche la poursuite normale de l'exécution du code.

```php
function inverse($n) {
    if (!$n) { // if $n == 0
        // lancement d'un object Exception
        throw new Exception('Division par zéro');
        /* Tout le code après cette exception ne sera jamais exécuté */
    }
    return 1/$x;
}

// Capture de l'exception
try {
    echo inverse(5); // affiche : 0.2
    echo inverse(0); // on fait une erreur volontaire
    // Tout ce bloc de code après cette exception ne sera jamais exécuté
} catch (Exception $e) {
    // bloc d'instructions lorsqu'une erreur de classe Exception se
    // produit
    echo $e->getMessage(); // affiche : Division par zéro
} catch (...) {
    // il est possible de gérer plusieurs classes différentes
d'exception
    ...
}

// poursuite normale de l'exécution
echo "Bonsoir";
```

La classe par défaut est **Exception**, mais il faut savoir qu'il existe plusieurs classes différentes (trouvable dans la documentation). Les fonctions existantes dans les modules PHP peuvent lancer des exceptions.

# Ressources Supplémentaires

**PHP :**

- Manuel PHP du site officiel : http://php.net/manual/fr/index.php