PHP

Guillaume Rodrigues
01/01/2024

PHP

Table des matières

PHP

PHP

PHP

PHP

1 Compétences pédagogiques :

Être capable de comprendre le fonctionnement des variables

Être capable de manipuler les opérateurs

Être capable d'utiliser les instructions conditionnelles

Être capable de manipuler un tableau

Être capable de comprendre les boucles

Être capable de créer et d'utiliser des fonctions

Être capable de comprendre le fonctionnement et l'intérêt de la programmation orienté objet

Être capable de créer et utiliser les classes

Être capable de créer et utiliser des objets

Être capable de comprendre les notions d'héritage

Être capable de comprendre les notions de polymorphisme

Être capable de créer des pages Web Dynamique

Être capable de mettre en place un système d'API

Être capable de connecter une application serveur à une base de données côté Back-end

Être capable de gérer des requêtes HTTP d'interaction côté Back-end

PHP

2 PHP c'est quoi ?

PHP (**PHP Hypertext Preprocessor**) est un langage de script libre conçu pour le développement d'applications web dynamiques.

Il s'intègre facilement dans du contenu html.

PHP est multiplateforme (Windows, Linux, Mac Os...).

Pour fonctionner PHP a besoin d'être installé sur un serveur web (Apache, NGINX, IIS sont les plus connus).

PHP est un langage qui s'exécute côté serveur et permet la génération de page web dynamique.

L'interpréteur PHP va alors générer une page web html.

De par sa polyvalence, PHP est également un langage de programmation à part entière qui peut être utilisé indépendamment d'un serveur web par exemple pour faire du scripting ou des applications graphiques (par exemple, via la librairie PHP-GTK : http://gtk.php.net/)

https://www.php.net/manual/fr/intro-whatcando.php

PHP

3 Spécificités du langage PHP

PHP est un langage « interprété »

On peut distinguer 2 grands types de langages de programmation : les langages interprétés et les langages compilés.

- **Langages interprétés** :
  - L'interpréteur lit le code et exécute les commandes ligne par ligne.
  - Le principal avantage de ces langages est leur côté multiplateforme : dès lors qu'un interpréteur existe sur la plateforme, un même code pourra y être exécuté.
  - Toutefois, cela a un coût en matière de ressources, puisqu'à chaque exécution du programme, l'interpréteur devra traduire les commandes avant qu'elles puissent être exécutées par la machine.
  - Exemples : PHP, Java, Python, JavaScript
- **Langages compilés** :
  - Les instructions sont compilées et traduites en langage machine avant de pouvoir exécuter le programme.
  - Le programme devra donc être compilé sur chaque machine sur lequel on souhaite l'exécuter.
  - Le principal avantage est l'optimisation des ressources nécessaires pour exécuter le programme, que ça soit au niveau de la mémoire nécessaire que de la vitesse d'exécution.
  - Exemples : C, C++, Pascal

Afin d'améliorer les performances des langages interprétés, la plupart de ces langages proposent en réalité une « pré-compilation ».

- **Java** utilise une **machine virtuelle**. Le compilateur Java va en réalité compiler le code en **byte code** et la machine virtuelle sera ensuite chargée d'interpréter ce byte code en langage machine.
- L'interpréteur **Python** propose (sans l'imposer) une option de précompilation des scripts en byte code.
- C'est un peu différent pour **PHP**. Le script est compilé à la volée lors de chaque appel serveur et est ensuite exécuté. Afin d'améliorer les performances et limiter les ressources, des **PHP cache code** ont été développés, permettant de mettre en cache sur le serveur le code compilé.

PHP

PHP est un langage « à typage faible » ou « non-typé »

Une autre grande différence des langages de programmation est le typage des variables, celui-ci pouvant être « fort » ou « faible ».

Un langage à typage fort vérifiera que le type de variable attendu est bien celui reçu, et renverra une erreur dans le cas contraire. De plus, les opérations sur des variables de types différents seront interdites.

A contrario, un langage faiblement typé sera beaucoup plus permissif, ce qui peut parfois entrainer des comportements surprenants.

- Exemples de langages à typage fort : C++, Java, Python
- Exemples de langages à typage faible : PHP, JavaScript, C

PHP

4 Evolution du PHP

PHP a été créé en 1994 par Rasmus Lerdorf pour la création de son site web personnel.

Son travail a été repris en 1997 par deux étudiants : Andi Gutmans et Zeev Suraski qui ont alors publié la version 3 de PHP.

Ils ont ensuite réécrit le moteur interne de PHP, aboutissant au moteur Zend Engine qui a servi de base à la version 4 de PHP. Cette version 4 est également celle qui a commencé à intégrer la programmation orientée objet au langage PHP.

La version 5 a été la version intégrant un modèle de programmation orientée objet complet.

A la suite de ça, de nombreuses évolutions et réécritures du moteur ont permis à PHP de gagner en performance et robustesse en s'inspirant des langages comme Java : support des tests unitaires, typage des méthodes et propriétés, standards de développement (PSR).

Voici une frise illustrant son évolution au fil des années : https://www.jetbrains.com/fr-fr/lp/php-25/

PHP

5 Environnement de travail :

Pour développer en PHP nous allons avoir besoin :

D'un serveur, WAMP (Windows) ou LAMP(Linux) suivant notre environnement de travail.

- Linux / Windows : Le système d'exploitation hébergeant le serveur
- Apache (serveur web pour héberger nos différents fichiers),
- MySQL (serveur de base de données, pour héberger nos bdd),
- PHP (interpréteur PHP),

Pour concevoir nos différents fichiers :

- Un éditeur de code (Visual studio code, PhpStorm, Notepad++, Bracket, Sublime Text, etc…),

Pour tester notre code :

- Un navigateur web pour afficher nos pages tester et contrôler le rendu. (Chrome, Mozilla Firefox, Edge, Safari etc…).

PHP

6 Syntaxe du langage :

Pour intégrer du code PHP nous écrivons nos scripts à l'intérieur de fichier avec l'extension **php**.

Emplacement des fichiers, Exemple :

Dans le dossier **C:\wamp64\www\exemple*index.php*** (exemple du chemin avec WAMP) du serveur apache (WAMP, XAMPP, Docker, LAMP, etc…)) nous allons créer un fichier **index.php**.

Par défaut, le serveur web renverra le contenu du fichier index.php (ou index.html) lorsque l'on visitera le site sans préciser de fichier en particulier.

Balises :

Lorsque PHP traite un fichier, il cherche les balises d'ouverture et de fermeture lui indiquant le code qu'il devra interpréter.

Nos scripts PHP devront être rédigés entre les balises :

- **< ?php** …code PHP… **?>** **:** ce sont les balises normales.
- **< ?** …code PHP… **?>** **:** ce sont les balises courtes. Toutefois ces dernières pouvant être désactivées, il est conseillé de n'utiliser que les balises normales.
- **< ?=** « du texte » **?>** : ces balises courtes permettent d'afficher du texte. Elles sont l'équivalent de
  < ?php echo « du texte » **?>**

PHP

Exemple :

La page sera accessible dans le navigateur web à l'adresse suivante :

localhost/exemple/index.php

NB : le fichier doit être exécuté et se trouver sur le serveur, si on ouvre simplement le fichier celui ne retournera rien.

*Depuis l'exemple précédent nous devons avoir le fichier à l'intérieur du répertoire WWW de Wamp ou HTDOCS de Xamp et créer un sous dossier (dans le dossier à la racine de **www** ou **htdocs** en fonction du logiciel) exemple, enfin créer un fichier index.php dans celui-ci. On saisira dans le navigateur web l'adresse suivante (url) **localhost/exemple/index.php**, pour exécuter le fichier. L'interpréteur PHP du serveur va alors lire le fichier **.php** et exécuter le code contenu dans celui-ci.*

Exemple de code d'une page :

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ma première page php</title>
</head>
<body>
    <h1>mon premier programme</h1>
    <?php
          //le script php se trouvera entre ces balises
    ?>
</body>
</html>
```

PHP

Syntaxe générale :

- Les instructions sont séparées par des « ; ». Bien que ça ne soit pas obligatoire, il est conseillé d'effectuer un retour à la ligne après chaque instruction.

```php
<?php
    //script php;
?>
```

- Les blocs de code (classes, boucles, fonctions) sont délimités par des accolades : « { », « } ».

```php
<?php
  Class MaClasse
  {
        function maFonction()
        {
            // Code de la fonction
        }
  }
?>
```

- Indentation : Bien que non obligatoire, on indente nos blocs de code pour une meilleure lisibilité (4 espaces).

PHP

Commenter des lignes de codes :

```php
<?php
    //commentaire sur une ligne

    /*
    ------------------------------------
    Commentaire sur plusieurs lignes
    ------------------------------------
    */

    /**
    * ------------------------------------
    * PHPDoc
    * ------------------------------------
    */
?>
```

PHP

Création de notre premier programme en php :

Nous allons créer un programme qui va afficher dans le navigateur internet.

Hello World

-Créer une page index.php dans votre éditeur de code et déposer là à l'intérieur de votre dossier **www/cours** du serveur apache (ou **htdocs/cours** si vous utilisez **xampp**).

-A l'intérieur de la page saisir le code ci-dessous :

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ma première page PHP</title>
</head>
<body>
    <h1>mon premier programme</h1>
    <?php
    //programme Hello Word
    //La commande echo permet d'afficher du contenu dans une page html.
    echo "Hello World";
    ?>
</body>
</html>
```

Pourquoi Hello World ?:

https://deux.io/pourquoi-hello-world/

PHP

7 Les variables :

Une variable ça sert à quoi ?

Les variables permettent de stocker des valeurs (Saisies, Résultat d'un sous-programme)

Elles vont pouvoir contenir des valeurs de types différents (texte, numérique…)

Une variable est une sorte de boite étiquetée avec un contenu.

Pour avoir accès à son contenu nous utiliserons son étiquette (son nom).

Les types de variables :

Bien que le typage en PHP soit un typage faible et dynamique (c'est-à-dire que le type est défini lorsque l'on donne une valeur à la variable), une variable en PHP possède un type de donnée.

- Le type « chaine de caractères » ou String en anglais
- Le type « nombre entier » ou Integer en anglais,
- Le type « nombre décimal » ou Float en anglais,
- Le type « booléen » ou Boolean en anglais,
- Le type « tableau » ou Array en anglais,
- Le type « objet » ou Object en anglais,
- Le type « NULL » qui se dit également NULL en anglais.
- Le type « ressource » ou Resource en anglais faisant référence à une ressource externe (ex : stream FTP, PDF document, …)

PHP

Nommer une variable :

En PHP, les variables sont représentées par un signe « $ » suivi du nom de la variable.

De plus, le nom d'une variable en PHP doit répondre à certaines contraintes.

- Le nom d'une variable doit impérativement commencer par une lettre ou le symbole underscore « _ »
- Le nom d'une variable ne peut être composé que de lettres, chiffres et de symboles underscore.
- La casse des caractères est prise en compte. Ainsi, les variables $variable et $varIABLE sont différentes.
- Par convention, une variable commencera par une lettre en minuscule, et sera écrite en camelCase. De plus, on essayera toujours de donner un nom compréhensible à la variable pour faciliter la compréhension du code.

Déclaration d'une variable :

En PHP une variable s'écrira comme ci-dessous :

$nomVariable = valeur;

Le symbole dollars **$** désignera une variable au moment de sa création et quand on l'utilisera.

Le type de la variable est défini lorsque l'on lui attribue une valeur.

Exemple d'utilisation d'une variable :

```php
<?php
    $variable = 10;
    $total = $variable + 10; //total vaut 20 (10 de variable + 10 en numérique).
?>
```

PHP

Exemple :

```php
<?php
//Définition d'une variable de type int ayant pour valeur 0
$varInt = 0;

// Définition une variable de type float (nombre à virgule flottante)
$varFloat = 3.4;

// Définition d'une variable de type bool (booléen)
$varBool1 = true;
$varBool2 = false;

/*
Définition d'une variable de type string
On peut utiliser le symbole simple quote ('') ou double quote ("") pour définir une
chaine de caractères
*/
$varString1 = 'Une chaine de caractères';
$varString2 = "Une autre chaine de caractères";

// Définition d'une variable de type array
$varArray1 = array();
$varArray2 = [];
?>
```

Afficher le contenu d'une variable :

Pour afficher le **contenu** d'une variable nous utiliseront le code ci-dessous :

```php
<?php
//initialisation d'une variable
$nbr =2 ;

//la fonction php echo permet d'afficher le contenu de la variable nbr
echo $nbr ;
?>
```

PHP

Afficher le type d'une variable :

Pour afficher le **type** d'une variable nous utiliserons la fonction « gettype($maVariable) :

```php
<?php
//initialisation d'une variable
$nbr =2;

//affichage dans la page web avec la fonction echo
echo $nbr ;

//utilisation de la fonction gettype pour afficher le type de la variable
echo gettype($nbr); //Ceci affichera "integer"

// Debugger une variable.
var_dump($nbr); // Affichera "int(2)"
?>
```

Exercice variables :

Exercice 1 :

-Créer une variable de type int avec pour valeur 5,

-Afficher le contenu de la variable (utilisation de la fonction php **echo**),

-Afficher son type (utilisation de la fonction php **gettype**),

-Créer une variable de type String avec pour valeur votre prénom,

-Afficher le contenu de la variable (utilisation de la fonction php **echo**),

-Créer une variable de type booléen avec pour valeur false,

-Afficher son type (utilisation de la fonction php **gettype**).

PHP

8 Les constantes :

Une constante est un identifiant permettant de stocker une donnée simple.

Une fois définie, cette donnée ne peut plus être modifiée.

Par convention, le nom d'une constante est toujours écrit en majuscules.

Il existe deux façons de déclarer une constante :

```php
<? php
define('MA_CONSTANTE_1', 'Hello');
const MA_CONSTANTE_2 = 'World!';

echo MA_CONSTANTE_1 . ' ' . MA_CONSTANTE_2;
//affichera "Hello World!"
?>
```

PHP

9 Les opérateurs :

Les opérateurs arithmétiques :

Pour effectuer des opérations mathématiques sur des types numériques (int, long, float, etc…)

On utilise les opérateurs mathématiques suivants :

Addition :

$a + $b

Soustraction :

$a - $b

Multiplication :

$a * $b

Division :

$a / $b

Modulo :

$a % $b (reste de la division de **$a** divisé par **$b**)

Exponentielle :

$a ** $b (Résultat de l'élévation de **$a** à la puissance **$b**)

PHP

Exercices Opérateurs arithmétiques :

Exercice 1 :

-Créer 2 variables $a et $b qui ont pour valeur 12 et 10,

-Stocker le résultat de l'addition de $a et $b dans une variable $total,

-Afficher le résultat (utilisez la fonction **echo**)

Exercice 2 :

-Créer 3 variables $a, $b et $c qui ont pour valeur $a =5, $b =3 et $c = $a+$b,

-Afficher la valeur de chaque variable (utilisez la fonction **echo**).,

-passer la valeur de $a à 2,

-Afficher la valeur de $a,

-passer la valeur de $c à $b - $a,

-Afficher la valeur de chaque variable (utilisez la fonction **echo**).

Exercice 3 :

-Créer 2 variables $a et $b qui ont pour valeur 15 et 23,

-Afficher la valeur de chaque variable (utilisez la fonction **echo**).,

-Intervertissez les valeurs de $a et $b,

-Afficher la valeur de $a et $b (utilisez la fonction **echo**).

Exercice 4 :

-Ecrire un programme qui prend le prix HT d'un article (de type float), le nombre d'articles (de type integer) et le taux de TVA (de type float), et qui fournit le prix total TTC (de type float) correspondant.

-Afficher le prix HT, le nbr d'articles et le taux de TVA (utilisez la fonction **echo**),

-Afficher le résultat (utilisez la fonction **echo**).

PHP

Les opérateurs d'incrémentation :

Pré-incrémentation :

++$a *(incrémente $a de 1 ($a = $a + 1) puis retourne la valeur de $a)*

Post-incrémentation :

$a++ *(retourne la valeur de $a puis incrémente de 1 ($a = $a+1))*

Pré-décrémentation :

--$a *(décrémente $a de 1 ($a = $a – 1) puis retourne la valeur de $a)*

Post-décrémentation :

$a-- *(retourne la valeur de $a puis décrémente de 1 ($a = $a – 1))*

PHP

Priorité des opérateurs :

La priorité des opérateurs spécifie l'ordre dans lequel les valeurs doivent être analysées. Par exemple, dans l'expression 1 + 5 * 3, le résultat est 16 et non 18, car la multiplication ("*") a une priorité supérieure par rapport à l'addition ("+"). Des parenthèses peuvent être utilisées pour forcer la priorité, si nécessaire. Par exemple : (1 + 5) * 3 donnera 18.

Lorsque les opérateurs ont une précédence équivalente, c'est leur associativité qui détermine s'ils sont évalués de droite à gauche ou inversement. Voyez les exemples ci-après.

Le tableau qui suit liste les opérateurs par ordre de précédence, avec la précédence la plus élevée en haut. Les opérateurs sur la même ligne ont une précédence équivalente (donc l'associativité décide de l'ordre de leur évaluation).

Associativité	Opérateurs	Information additionnelle
non-associative	clone new	clone et new
gauche	[	array()
droite	++ -- ~ (int) (float) (string) (array) (object) (bool) @	types et incrément/décrément
non-associatif	instanceof	types
droite	!	logique
gauche	* / %	arithmétique
gauche	+ - .	arithmétique et chaîne de caractères
gauche	<< >>	bitwise
non-associatif	< <= > >= <>	comparaison
non-associatif	== != === !==	comparaison
gauche	&	bitwise et références
gauche	^	bitwise
gauche	|	bitwise
gauche	&&	logique
gauche	||	logique
gauche	? :	ternaire
droite	= += -= *= /= .= %= &= |= ^= <<= >>= =>	affectation
gauche	and	logique
gauche	xor	logique
gauche	or	logique
gauche	,	plusieurs utilisations

PHP

10 Concaténation :

En PHP nous pouvons concaténer des valeurs entres elles. C'est à dire ajouter des chaines de caractères, des nombres, valeurs de variables au sein d'une même chaine de caractères.

Exemple :

Ecrire le nom d'une variable dans une page web :

```php
<?php
$nom = 'test';
//on va utiliser le symbole \devant le nom de la variable, ce caractère permet //d'annuler l'interprétation du caractère qui va suivre, dans ce cas il va afficher le nom de la variable et non son contenu.
echo 'affichage de la variable s'appelant \$test' ;
?>
```

Ecrire la valeur d'une variable dans une page web :

```php
<?php
$nom = 'test';
echo "affichage du contenu de la variable  \$nom : $nom ";
?>
```

Concaténer des chiffres, des chaines de caractères et les afficher dans une page web :

```php
<?php
$str = 'une chaine de caractères';
$strLength = strlen($str);
echo "<br>La chaine de caractères '$str' contient $strLength caractères.";
?>
```

PHP

Concaténer des variables dans des chaines de caractères :

```php
<?php
$var = 'du texte';

//version avec encadrement ""
$concat1 = "ma chaine $var";

//version avec encadrement ''
$concat2 = 'ma chaine ' . $var;
?>
```

Exercices :

Exercice 1 :

-Créer une variable $a qui a pour valeur ***« bonjour »,***

-Afficher le **nom de la variable et sa valeur**.

Exercice 2 :

-Créer 1 variable $a qui a pour valeur **« bon »**,

-Créer 1 variable $b qui a pour valeur **« jour »**,

-Créer 1 variable $c qui a pour valeur **10**,

-Concaténer **$a, $b et $c + 1**,

-Afficher le **résultat** de la concaténation.

Exercice 3 :

-Créer une variable $a qui a pour valeur **bonjour**,

-Afficher un paragraphe (**balise html**) et à l'intérieur la phrase suivante :

l'adrar

-Ajouter la variable $a avant la phase dans le paragraphe,

-Cela doit donner :

*** bonjour l'adrar ***

Les opérateurs d'affectation :

$a = 5 *(affecte la valeur 5 à la variable $a)*

$a += 3 *(affecte la valeur $a+3 à la variable $a : équivalent à $a = $a + 3)*

$b = 'Bonjour '; (affecte la valeur 'Bonjour ' à la variable $b)

$b .= 'le monde.'; (Concatène 'le monde.' Et l'affecte à la variable $b : équivalent à $b = $b . 'le monde.')

11 Les Fonctions :

Les fonctions permettent de générer du code qui va être exécuté chaque fois que l'on va appeler la fonction par son nom. Les fonctions sont créées pour exécuter une tâche précise et permettre de mutualiser le code afin d'améliorer la qualité et maintenabilité du code.

Nous avons déjà utilisé la fonction gettype() qui est une fonction interne à PHP.

Nous pouvons trouver la liste des fonctions internes sur la documentation officielle de PHP (https://www.php.net/manual/fr/funcref.php) ou sur le site w3school (https://www.w3schools.com/php/default.asp).

Une fonction peut prendre des paramètres en entrée et retourner un résultat.

Création d'une fonction :

Pour créer une fonction en php nous allons utiliser la syntaxe suivante :

```php
<?php
function nomDeLaFonction()
{
}
?>
```

Le nom d'une fonction devra répondre au mêmes critères de nommage que les variables (commencer par une lettre ou le symbole underscore et ne comporter que lettres, chiffres ou symboles underscore. Par convention le nom d'une fonction sera écrit en camelCase.

PHP

Appel d'une fonction :

Pour appeler une fonction on va saisir le nom de la fonction suivi de ***()***

Exemple :

```php
<?php
function maFonction()
{
}
maFonction();
?>
```

Création d'une fonction avec des paramètres :

Une fonction avec des paramètres va nous permettre d'exécuter le code de celle-ci et d'adapter son traitement en fonction de ce que l'on va passer en paramètre. Le mot clé **return** permet de retourner (int, string, boolean etc..).

Exemple :

```php
<? php
maFonction(10,5);

function maFonction($a,$b)
{
    $result= $a+$b ;

    return $result ;
}
?>
```

PHP

Exercices :

Exercice 1 :

-Créer une fonction qui soustrait à **$a** la variable **$b** (*2 paramètres en entrée*),

-la fonction doit retourner le résultat (**return**).

Exercice 2 :

-Créer une fonction qui prend en entrée un nombre à virgule (**float**),

- la fonction doit retourner l'arrondi (**return**) du nombre en entrée (utiliser une fonction interne au langage).

Exercice 3 :

-Créer une fonction qui prend en entrée **3 valeurs** et retourne **somme** des 3 valeurs.

Exercice 4 :

-Créer une fonction qui prend en entrée **3 valeurs** et retourne la **valeur moyenne** des 3 valeurs (saisies en paramètre).

Typage des paramètres :

Le version 7 de PHP introduit le typage des paramètres d'entrée et de sortie des fonctions.

Cela permet un meilleur contrôle des entrées, ce qui est impératif pour du code de qualité et devra donc être utilisé autant que possible. En effet, si le paramètre reçu par la fonction n'est pas du bon type, le serveur lancera une erreur.

Le type de sortie d'une fonction en retournant aucune valeur est « void ».

Si un paramètre peut être de plusieurs types, on peut utiliser le symbole pipe « | » pour indiquer plusieurs paramètres (exemple : int|float).

Dans l'exemple ci-dessous, on a défini que les paramètres $a et $b étaient de type « float », et que le paramètre de sortie était également de type « float ».

```php
<? php
maFonction(10,5);

function maFonction(float $a,float $b):float
{
    return $a+$b ;
}
?>
```

Exercices :

Exercice :

Modifier les fonctions précédemment écrites en typant les paramètres d'entrée et de sortie.

12 Les structures conditionnelles :

Les structures conditionnelles, ou conditions vont nous permettre de conditionner l'exécution de certains blocs de codes si certaines conditions sont remplies (ou si elles ne le sont pas.

Pour cela nous allons utiliser les instructions ***if*** (***si***), ***elseif*** (**sinon si**) ***else*** (***sinon***).

Instruction if :

Instruction fondamentale dans un programme, elle permet l'exécution d'un bloc de code si une condition est remplie.

```php
<?php
    if ($heure > 6 && $heure < 21) {
        echo 'Il fait jour';
    }
?>
```

Instruction else :

Permet d'exécuter un bloc de code dans le cas où la condition du « if » ne soit pas remplie.

```php
<?php
    if ($heure > 6 && $heure < 21) {
        echo 'Il fait jour.';
    } else {
        echo 'Il fait nuit.';
    }
?>
```

PHP

Instruction elseif :

elseif est une combinaison de if et else. Si la condition précédente n'était pas remplie, l'interpréteur va vérifier si celle-ci l'est, et exécuter le bloc de code correspondant si c'est le cas.

```php
<?php
    //$i est un nombre
    if ($i < 0) {
        echo '$i est inférieur à 0';
    } elseif ($i > 50) {
        echo '$i est supérieur à 50';
    } else {
        echo '$i est compris entre 0 et 50 inclus';
    }
?>
```

Opérateur ternaire :

Un operateur permet de réaliser certaines conditions if…else de manière abrégée.

Par exemple, voici deux façons de vérifier si une personne est majeure, l'une avec une structure if…else classique, l'autre avec un opérateur ternaire.

```php
<?php
function isAdult(int $age): bool
{
    if ($age >= 18) {
        return true;
    } else {
        return false;
    }
}

function isAdultV2(int $age) : bool
{
    return ($age >= 18) ? true : false;
}

function isAdult200IQ (int $age): bool
{
    return $age >= 18;
}
?>
```

PHP

Instruction switch :

L'instruction switch équivaut une succession d'instructions if. Elle permet de comparer une variable à des valeurs, et exécuter un (ou plusieurs) bloc(s) de code en conséquence.

Attention, le switch effectue une comparaison large (il ne compare pas le type de donnée. Equivalent à ==).

```php
<?php
    //$i est un nombre
    switch($i) {
        case 1:
            echo '$i vaut 1';
            break; //l'instruction break permet de sortir de la structure de contrôle et
donc de ne pas effectuer les instructions suivantes.
        case 2:
            echo '$i vaut 2';
            break;
        case 3:
            echo '$i vaut 3';
            break;
        default:  //Bloc exécuté dans tous les cas, sauf si on a eu un break avant
            echo '$i ne vaut ni 1, ni 2, ni 3';
    }
?>
```

PHP

Opérateurs de comparaison :

Nous allons utiliser dans nos conditions les opérateurs de comparaisons.

Exemple	Nom	Résultat
$a == $b	Egal	true si $a est égal à $b après le transtypage (comparaison large).
$a === $b	Identique	true si $a est égal à $b et qu'ils sont de même type.
$a != $b	Différent	true si $a est différent de $b après le transtypage (comparaison large).
$a <> $b	Différent	true si $a est différent de $b après le transtypage (comparaison large).
$a !== $b	Différent	true si $a est différent de $b ou bien s'ils ne sont pas du même type.
$a < $b	Strictement inférieur	true si $a est strictement inférieur à $b.
$a > $b	Strictement supérieur	true si $a est strictement supérieur à $b.
$a <= $b	Inférieur ou égal	true si $a est inférieur ou égal à $b.
$a >= $b	Supérieur ou égal	true si $a est supérieur ou égal à $b.
$a <=> $b	Combiné / Spaceship operator	Retourne -1 si $a est < à $b, 1 si $a est > à $b et 0 si $a == $b, après transtypage (comparaison large).

Opérateurs logiques :

Voici les différents opérateurs logiques qui peuvent être utilisés :

Exemple	Nom	Résultat
$a && $b	And (Et)	true si $a ET $b sont true.
$a and $b	And (Et)	true si $a ET $b valent true.
$a || $b	Or (Ou)	true si $a OU $b est true.
$a or $b	Or (Ou)	true si $a OU $b valent true.
$a xor $b	XOR	true si $a OU $b est true, mais pas les deux en même temps.
!$a	Not (Non)	true si $a n'est pas true.

PHP

Exemple :

```php
<?php
    $a = 6;

    if($a <= 3 && $a > 0) {
          //teste si $a est plus petit que 3 et est supérieur à 0
        echo "la valeur de la variable \$a est plus petite que 3";
    } elseif($a >= 3 && $a < 5) {
        //teste si $a est plus grand ou égal et 3 et inférieur à 5
        echo "la valeur de la variable \$a est comprise entre 3 et 5";
    } else {       //sinon
        echo "la valeur de la variable \$a est supérieur à 5";
    }
?>
```

Exercices :

Exercice 1 :

- Créer une fonction qui teste si un nombre est **positif** ou **négatif**

Exercice 2 :

- Créer une fonction qui prend en entrée **3 valeurs** et retourne le nombre le plus **grand**

Exercice 3 :

- Créer une fonction qui prend en entrée **3 valeurs** et retourne le nombre le plus **petit**

Exercice 4 :

- Créer une fonction calculePrixFinal qui prend en entrée un paramètre $prix de type float et retournera le prix final.

- Si le prix est > à 2000€, la ristourne sera de 20%

- Si le prix est > à 1000€, la ristourne sera de 10%

- Sinon, la ristourne sera de 0

PHP

Exercice 5 :

- Créer une fonction qui prend en entrée **1 année (entier)** et qui affiche « l'année x est une année bissextile » si l'année est bissextile ou « l'année x n'est pas une année bissextile » si ce n'est pas une année bissextile

- Pour rappel une année bissextile est définie de la façon suivante (https://fr.wikipedia.org/wiki/Ann%C3%A9e_bissextile) :

    - Les années sont en général bissextiles si elles sont multiples de quatre

    - elles ne sont pas bissextiles si elles sont multiples de cent à l'exception des années multiples de quatre cents qui le sont.

13 Les boucles :

Comme dans tous les langages de programmation, PHP gère les structures de boucle.

La boucle est un élément de base d'un langage de programmation.

Les boucles permettent de répéter plusieurs fois une ou plusieurs instructions tant qu'une condition est vérifiée ou bien jusqu'à ce qu'elle soit vérifiée.

En PHP, il existe 4 types de boucles différentes :

- La boucle « for »
- La boucle « while »
- La boucle « do…while »
- La boucle « foreach »

Pour écrire une boucle (**for**):

Boucle for :

La boucle « for » prend 3 paramètres en entrée : un paramètre initial, une condition de continuation, et une opération à effectuer en fin d'itération.

Exemple de boucle for :

Tant que $i est inférieur à 10 on répète l'opération :

```php
<?php
// for (valeur initiale ; condition ; opération)
for ($i=0; $i < 10; $i++){
    echo 'Ceci est une boucle for en PHP';
    echo '<br>';
    echo "\$i = $i";
    echo '<br>';
}
?>
```

PHP

Boucle while :

L'instruction « while » est traduite en « tant que ». Tant que la condition est remplie, la boucle continue de s'exécuter.

Exemple de boucle while :

Tant que $i est inférieur à 10 on répète l'opération :

```php
<?php
$i = 0; //variable compteur
while ($i < 10) //boucle while
{
    echo 'Ceci est une boucle while en PHP';
    echo '<br>';
    echo "\$i = $i";
    echo '<br>';
    //à chaque tour j'incrémente $i (+1)
    $i++;
}
?>
```

Boucle do…while :

L'instruction « do…while » pourrait être traduite par « faire…tant que ». Elle est dérivée de la boucle « while ». La différence est que la boucle while vérifie la condition avant que l'on rentre dans la boucle, alors que dans le cas de la boucle do…while, on entrera toujours dans le bloc « do » de la boucle, et on en sortira si la condition n'est plus remplie.

Le choix d'utiliser « while » plutôt que « do…while » dépend de la situation.

Exemple de boucle do…while :

Tant que $i est inférieur à 10 on répète l'opération :

```php
<?php
$i = 0; //variable compteur
do {
    echo 'Ceci est une boucle do...while en PHP';
    echo '<br>';
    echo "\$i = $i";
    echo '<br>';
} while (++$i < 10);
?>
```

PHP

Boucle foreach :

A la différence des autres boucles, ce type de boucle n'est pas présent dans la majorité des langages de programmation, et est pourtant d'une efficacité redoutable.

Cette boucle permet de parcourir tous les éléments d'un objet « iterable » (tableau, chaine de caractères, collection d'objets.

Exemple de boucle foreach :

```php
<?php
foreach ($tableau as $valeur)
{
    echo $valeur.'<br />';
}
?>
```

Exercices sur les boucles :

Exercice 1 :

- Choisir un nombre compris entre 0 et 999

- A l'aide d'une boucle while, effectuez des tirages aléatoires (utilisation de la fonction PHP « rand() » jusqu'à trouver le bon nombre.

- Affichez le nombre d'itérations nécessaires pour trouver le nombre

Exercice 2 :

- Choisir un nombre de lignes

- Choisir un nombre de colonnes

- A l'aide de boucles « for », obtenez le résultat suivant :

0000000000
1111111111
2222222222
3333333333
4444444444
5555555555
6666666666

PHP

Exercice 3 :

- Ecrivez des boucles qui affichent ce qui est demandé

- Le nombre de colonne à afficher dépend du n° de ligne, à la ligne i, il faut afficher i colonnes.

- Le résultat attendu est celui-ci :

1
22
333
4444
55555
666666
7777777

14 les tableaux

Les tableaux, aussi appelés **arrays** en anglais, sont des structures de données essentielles et massivement utilisées en développement PHP.

Un tableau permet de stocker plusieurs données dans une même variable. Les données stockées peuvent être de types différents.

Un tableau est une structure de type « carte ordonnée ». Ce type de structure associe une valeur à une clé et peut représenter par exemple des tableaux, des listes, des collections ou des dictionnaires.

Une clé dans un tableau est forcément unique.

Un tableau peut prendre en valeur un autre tableau, ce qui permet de créer des structures multidimensionnelles.

Par défaut, la clé est une valeur numérique directement définie par l'interpréteur PHP. On parle dans ce cas de **tableau indexé**. La structure est comparable à une liste.

Nous avons aussi la possibilité de définir nous même une clé de type chaine de caractères afin de créer un **tableau associatif**, structure qui se rapproche d'un dictionnaire.

Déclaration et instanciation d'un tableau :

Il existe différentes façons de déclarer et instancier un tableau.

Dans le cas d'un tableau indexé, la **première valeur** aura toujours pour **index 0**.

PHP

Exemple déclaration de tableaux indexé numériquement et associatif :

```php
<?php
// Déclaration d'un tableau vide
$tableau = [];
// OU
$tableau = array();

// Déclaration d'un tableau indexé (liste de valeurs)
// Ce tableau a 4 données de types différents
$tableauIndexe = ['Denis',42,199.3, array('Marc')];

// Déclaration d'un tableau associatif
$villes = [
    'Toulouse' => 31,
    'Auch' => 32,
    'Albi' => 81,
    'Montauban' => 82,
];
?>
```

Exemple ajouter une valeur à un tableau :

```php
<?php
$legumes = [];
// Ajout d'un élément a un tableau indexé numériquement il sera ajouté à la dernière position.
$legumes[] = 'salade';

// Ajout d'un élément a un tableau indexé numériquement à une position (2° position).
$legumes[1] = 'carotte';

$legumes[2] = 'cerise';

// Suppression d'une donnée d'un tableau (fonction unset)
unset($legumes[2]);

// Ajout de la taille de la personne dans le tableau associatif
$identite = [];
$identite['taille'] = 180;
$identite['prenom'] = 'Guillaume';
?>
```

PHP

Exemple parcourir un tableau :

```php
<?php
//création d'un tableau $prenoms
$prenoms[0] = 'Mathieu';
$prenoms[1] = 'Sophie';
$prenoms[2] = 'Florence';
//ou
$prenoms = ['Mathieu', 'Sophie', 'Florence'];

//parcours de tout le tableau
foreach ($prenoms as $key => $value) {
    echo '<br>';
    //Affiche le contenu du couple clé => valeur à chaque tour.
    echo "$key => $value";
}

//parcours de tout le tableau avec une boucle for
for ($i=0; $i < count($prenoms); $i++) {
    echo '<br>';
    echo "$i => " . $prenoms[$i];
}

?>
```

Tableaux imbriqués :

Il est une pratique courante d'imbriquer des tableaux dans d'autres tableaux. Voici un exemple de ce cas d'utilisation pour la gestion d'un annuaire.

```php
<?php
$annuaire = [
    [
        'nom' => 'Roger Rabbit',
        'tel' => '0504030201',
        'email' => 'roger@rabbit.com'
    ],
    [
        'nom' => 'Donald Duck',
        'tel' => '0102030405',
        'email' => 'donald@duck.com'
    ],
];

// Accéder au numéro de téléphone de Donald
$telDonald = $annuaire[1]['tel'];
?>
```

PHP

Gestion des paramètres d'une application :

L'utilisation du tableau en mode « dictionnaire » rend son usage très populaire pour la gestion des paramètres clé-valeur d'une application PHP.

Ce fichier contenant des données sensibles, il ne devrait jamais être stocké tel quel sur quelque repository Git que ce soit. Il faudra donc l'ajouter au fichier « .gitignore » de l'application. Nous versionnerons dans ce cas uniquement un fichier template (souvent suffixé par « -dist ») et l'utilisateur aura à sa charge de créer une copie de ce fichier pour y ajouter les données réelles.

Voici à quoi pourrait ressembler un fichier de config d'une application :

```php
<?php
$CONFIG = array(
    'dbtype' => 'mysql',
    'dbhost' => 'localhost',
    'dbname' => 'nom_de_la_db',
    'dbuser' => 'root',
    'dbpass' => 'password',
    'dataroot' => '/var/www/html',
);
?>
```

Exercices :

Exercice 1 :

- Générez un tableau de longueur 50 en injectant des valeurs aléatoires comprises entre -100 et 100

- Une fois les données injectées, affichez la taille du tableau

Exercice 2 :

- Créer une fonction qui affiche la valeur la plus grande du tableau (from scratch puis en utilisant une fonction interne à PHP).

Exercice 3 :

- Créer une fonction qui affiche la moyenne du tableau.

Exercice 4 :

- Créer une fonction qui affiche la valeur la plus petite du tableau (from scratch et en utilisant une fonction interne à PHP).

Exercices bonus :

https://gist.github.com/tomsihap/0ce95ee46a6b57d55144a67d68baed35

PHP

15 Fonctions internes à PHP :

Nous avons commencé à voir quelques fonctions internes à PHP. Toutefois, nous allons les détailler un peu plus.

Vous trouverez toutes les fonctions et des exemples d'utilisation dans la doc officielle de PHP : https://www.php.net/manual/fr/funcref.php

Fonctions relatives aux chaines de caractères :

empty()	Renvoie true si une chaine est vide, nulle, non définie, ou vaut 0, false, '0'
strlen()	Retourne la longueur d'une chaine de caractères
strrev()	Fonction « reverse » d'une chaine de caractères
str_repeat()	Répète une chaine de caract_res
substr()	Retourne une partie de la chaine de caractères
strcmp()	Compare 2 chaines de caractères
str_replace()	Remplace des caractères ou portions de la chaine
trim()	Supprime les espaces au début et à la fin de la chaine
strtolower()	Convertit la chaine en minuscules
strtoupper()	Convertit la chaine en majuscules
ucfirst()	Met en majuscule le 1er caractère d'une chaine
addslashes()	Echappe les caractères spéciaux d'une chaine avec des antislash
stripslashes()	Supprime les antislash d'une chaine
htmlentities()	Convertit tous les caractères éligibles en HTML
htmlspecialchars()	Convertit les caractères spéciaux en HTML
html_entity_decode()	Décode tous les caractères encodés en HTML
htmlspecialchars_decode()	Décode les caractères spéciaux encodés en HTML
strip_tags()	Supprime les balises HTML et PHP d'une chaine

PHP

Exemples :

```php
<?php
    // Longueur d'une chaine
$str = 'Bonjour le monde';
echo strlen($str);

// inversion d'une chaine
echo strrev($str);

// Utilisation des substrings
echo substr($str, 3, 4); // Retourne les 4 caractères à partir de l'index 3 de la chaine
$str -> jour

?>
```

Fonctions numériques :

ceil()	Arrondit un nombre à l'entier supérieur
floor()	Arrondit un nombre à l'entier inférieur
round()	Arrondit un nombre selon une méthode définie par l'utilisateur
abs()	Retourne la valeur absolue d'un nombre
pow()	Retourne un nombre élevé à la puissance d'un autre nombre (équivalent $nb1**$nb2)
log()	Retourne le logarithme d'un nombre
rand()	Génère un nombre aléatoire
bindec()	Convertit un nombre de sa valeur binaire à décimale
decbin()	Convertit un nombre de sa valeur décimale à binaire
dechex()	Convertit un nombre de sa valeur décimale à hexadécimale
hexdec()	Convertit un nombre de sa valeur hexadécimale à décimale

PHP

Fonctions relatives aux tableaux :

explode()	Scinde une chaine de caractères dans un tableau
implode()	Joint les éléments d'un tableau dans une chaine de caractères
range()	Crée un tableau contenant un intervalle d'éléments
min()	Retourne la valeur la plus petite d'un tableau
max()	Retourne la valeur la plus grande d'un tableau
shuffle()	Mélange un tableau de façon aléatoire
array_slice()	Extrait une portion d'un tableau
array_shift()	« Dépile » le 1er élément d'un tableau (le retourne et le supprime du tableau)
array_pop()	« Dépile » le dernier élément d'un tableau (le retourne et le supprime du tableau)
array_unique()	Supprime les doublons d'un tableau
array_merge()	Combine 2 tableaux ou plus
in_array()	Vérifie si une valeur est présente dans un tableau
array_key_exists	Vérifie si une clé est présente dans un tableau
sort()	Trie un tableau
asort()	Trie un tableau associatif par valeur
ksort()	Trie un tableau associatif par clé

Exercice :

Exercice 1 :

- A l'aide de la fonction « range » créez un tableau contenant tous les nombres de 0 à 1000.

- Parcourez le tableau et extrayez tous les nombres premiers dans un autre tableau (un nombre premier n'est divisible que par 1 et par lui-même).

- Affichez les nombres premiers ainsi obtenus dans une liste HTML (
).

PHP

Exercice 2 :

- Reprenez l'exercice précédent

- Au lieu de copier les nombres premiers dans un autre tableau il faudra les supprimer du tableau initial

- A l'aide d'une fonction PHP, comparez ce tableau avec le tableau obtenu à la fin de l'exercice 1

PHP

15 Les variables superglobales :

Le transfert de données entre des pages web est géré en PHP par le biais de variables spéciales qui s'appellent superglobales.

Les variables superglobales sont des variables qui sont directement accessibles à n'importe quel endroit de l'application, quel que soit le contexte.

Voici la liste des variables superglobales :

$GLOBALS	Référence toutes les variables accessibles dans un contexte global
$_SERVER	Tableau contenant des variables relatives au serveur (nom du serveur, adresse IP, …)
$_GET	Tableau associatif des paramètres envoyés par la méthode HTTP GET
$_POST	Tableau associatif des paramètres envoyés par la méthode HTTP POST
$_FILES	Tableau associatif des variables de téléchargement de fichiers via http
$_COOKIE	Tableau associatif des variables passées au script par les cookies HTTP
$_SESSION	Tableau associatif des variables stockées dans la session HTTP
$_REQUEST	Tableau associatif contenant les variables $_GET, $_POST et $_COOKIE
$_ENV	Tableau associatif des variables d'environnement

Dans cette partie nous allons voir les Superglobales suivantes : $_GET et $_POST.

Chacune de ces variables contient un tableau avec le contenu des différents champs html d'un formulaire.

Les formulaires html possèdent 2 méthodes d'envoi possibles GET et POST.

La méthode GET fait passer les paramètres de formulaire dans l'url de la page.

Cette méthode peut être dangereuse car elle affiche directement les paramètres de la requête HTTP dans l'URL.

Ainsi, cette méthode devra uniquement être utilisée pour faire des requêtes destinées à l'affichage de contenu.

En revanche, elle ne devra jamais être utilisée pour effectuer des requêtes visant à enregistrer des données, ou pour des requêtes faisant transiter des données sensibles (identifiants de connexion).

PHP

La méthode POST fait passer les paramètres par le body de la requête HTTP.

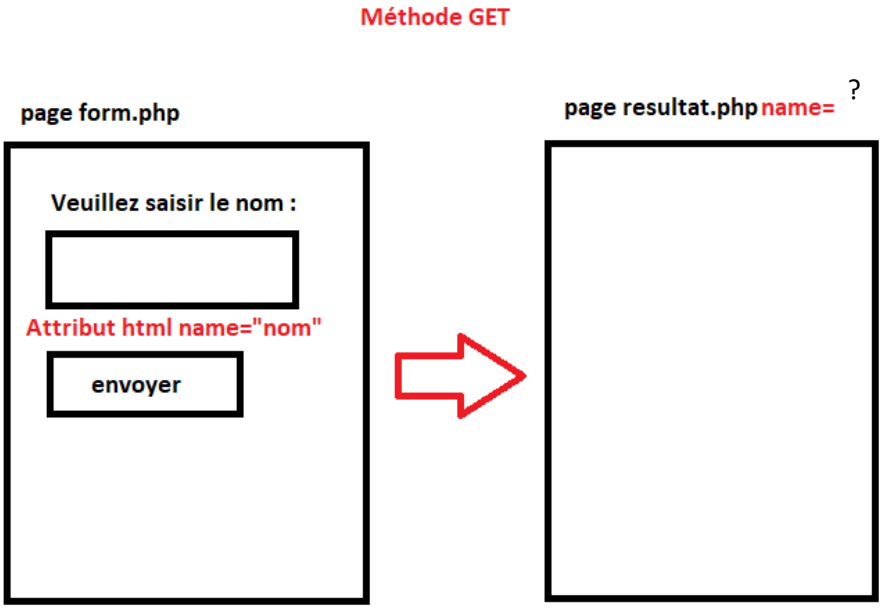
Cette méthode est à privilégier lorsque l'on fait transiter des données sensibles ou que l'on effectue des requêtes dans le but de modifier des données.

De plus elle permet de transférer des informations de taille plus importante.

Fonctionnement GET:

Le contenu des champs de formulaire va transiter dans l'url à la condition de nommer ces champs avec l'attribut html **name**.

Schéma transfert d'informations GET :



PHP

Exemple transfert de données en get :

Page form.php

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>formulaire</title>
</head>
<body>
    <form action="resultat.php" method="GET">
        <p>veuillez saisir votre nom :</p>
        <input type="text" name="nom">
        <br>
        <input type="submit" value="Envoyer">
    </form>
</body>
</html>
```

Cette page va envoyer à la page « resultat.php » le contenu du champ « nom » dans **l'url** sous la forme suivante :

http://resultat.php?nom=**valeur**.

Si l'on avait plusieurs champs dans le formulaire avec l'attribut name, ils seraient séparés du caractère **&** :

http://resultat.php?nom=valeur1&prenom=valeur2

PHP

Page resultat.php

```php
<?php
    //test de l'existence de la super globale $_GET
    if(isset($_GET['nom'])){
        $nom = $_GET['nom'];
        echo "Mon nom est : '.$nom.'";
    }
?>
```

Dans cette page nous allons afficher le contenu de la super globale **$_GET['nom']** avec la fonction **echo**.

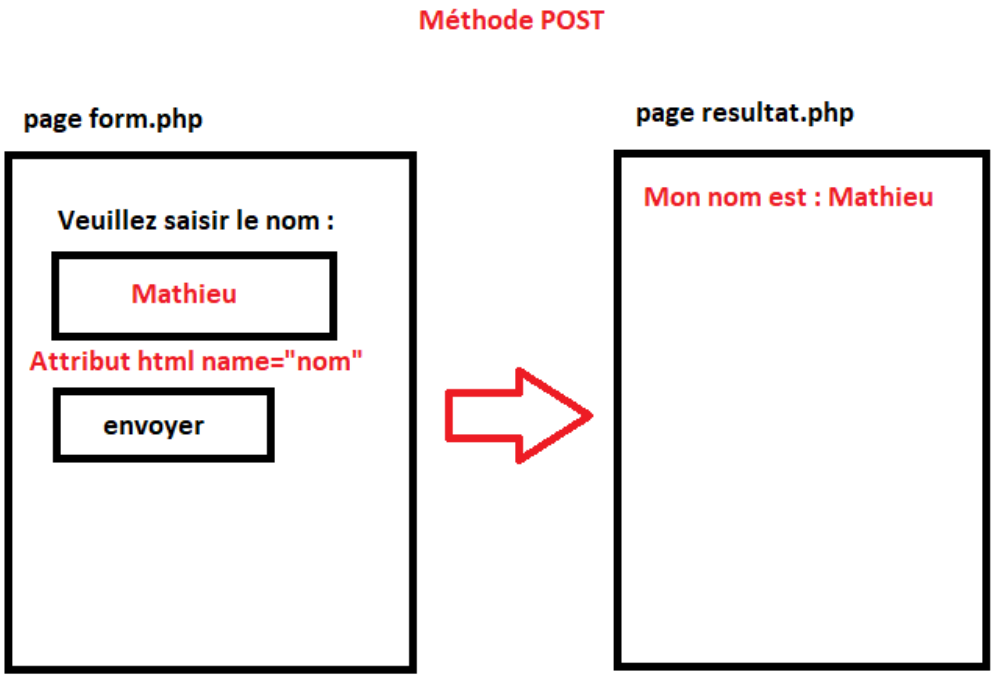
1. On vérifie l'existence de la super globale ***$_GET['nom']*** avec la fonction PHP ***isset()*** qui teste si la variable ***existe*** et si sa **valeur** n'est pas égale à **null**.
2. Ensuite on va afficher le contenu avec la méthode **echo** que l'on à vue précédemment et on concatène le résultat avec la chaine ***mon nom est :*** .

PHP

Fonctionnement POST:

Le contenu des champs de formulaire va transiter par le body de la requête HTTP à la condition de nommer ces champs avec l'attribut html **name**.

Schéma transfert d'informations POST :



PHP

Exemple transfert de données en post :

Page form.php

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>formulaire</title>
</head>
<body>
    <form action="resultat.php" method="post">
        <p>veuillez saisir votre nom :</p>
        <input type="text" name="nom">
        <br>
        <input type="submit" value="Envoyer">
    </form>
</body>
</html>
```

Cette page va envoyer à la page resultat.php le contenu du champ nom dans le **body**.

Page resultat.php

```
<?php
    //test de l'existence de la super globale $_POST
    if(isset($_POST['nom'])){
        $nom = $_POST['nom'];
        echo "mon nom est : .'$nom.'";
    }
?>
```

Dans cette page nous allons afficher le contenu de la super globale **$_POST['nom']** avec la fonction **echo**.

1. On vérifie l'existence de la super globale ***$_POST['nom']*** avec la fonction PHP ***isset()*** qui teste si la variable ***existe*** et si sa **valeur** n'est pas égal à **null**.
2. Ensuite on va afficher le contenu avec la méthode **echo** que l'on à vue précédemment et on concatène le résultat avec la chaine ***mon nom est :*** .

NB :

Si l'on souhaite traiter les données dans le même script PHP que la page de formulaire, dans la partie action (html) on laisse soit le champ **vide,** ou on saisit **#**, ou on réécrit le nom du script php (form.php dans l'exemple ci-dessus).

PHP

Exercices :

Exercice 1 :

- Créer une page de formulaire dans laquelle on aura 2 champs de formulaire de type nombre.

- Afficher dans cette même page la somme des 2 champs avec un affichage du style :

La somme est égale à : valeur

Exercice 2 :

- Créer une page de formulaire dans laquelle on aura 3 champs de formulaire de type nombre :

- 1 champ de formulaire qui demande un prix HT d'un article,
- 1 champ de formulaire qui demande le nombre d'article,
- 1 champ de formulaire qui demande le taux de TVA,

- Afficher dans cette même page le prix TTC (prix HT*taux TVA*quantité) avec un affichage du style :

Le prix TTC est égal à : valeur €.

PHP

16 Importer des fichiers :

Nous avons la possibilité en PHP d'importer des fichiers à l'intérieur du répertoire du projet

Pour ce faire, il suffit d'utiliser un champ de formulaire de type « file ».

Les informations relatives au fichier uploadé seront ensuite récupérées dans la superglobale $_FILES.

Elle va s'utiliser comme les super globales précédentes $_GET et $_POST.

Quand on importe un fichier, celui-ci va se retrouver dans un dossier temporaire (à la racine du serveur apache, dans le dossier /tmp) à moins qu'un autre dossier soit fourni avec la directive upload_tmp_dir du php.ini.

Le serveur va lui donner un nom temporaire (ex : tmp_1569565322.jpg).

Pour importer un fichier nous allons créer un formulaire HTML comme ci-dessous (***index.php***):

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Importer un fichier </title>
</head>
<body>
    <form action="index.php" method="POST" enctype="multipart/form-data">
        <h2>importer une image</h2>
        <input type="hidden" name="MAX_FILE_SIZE" value="30000" />
        <input type="file" name="file">
        <p><button type="submit">importer</button></p>
    </form>
</html>
```

Le formulaire sera soumis avec la méthode POST et le dossier sera importé dans le dossier /tmp à la racine du serveur.

Nous allons voir ci-dessous comment le traiter et le déplacer dans le bon dossier (par ex le dossier image à la racine de notre projet) :

- Créer un nouveau répertoire **import** à la racine du serveur web (*www/import ou htdocs\import*)
- Créer un fichier import.php et coller à l'intérieur le code html de la page précédente

PHP

Nous allons ajouter le code ci-dessous dans le fichier import.php pour récupérer le fichier et le déplacer dans le bon dossier (/image à la racine du projet import).

Code importation d'un fichier avec son nom.ext dans le dossier image à la racine du projet.

Pour ce faire nous allons :

- Vérifier si le fichier que l'on importe existe (utilisation de la super globale $_FILES)
- Créer différentes variables
- Déplacer le fichier dans le bon dossier avec la méthode (***move_uploaded_file***).

PHP

```php
<?php
    /*-------------------------------------------------------
                    Test (import du fichier) :
    -------------------------------------------------------*/
    //test si le fichier importé existe
    if(isset($_FILES['file'])){
        //stocke le chemin et le nom temporaire du fichier importé (ex /tmp/125423.pdf)
        $tmpName = $_FILES['file']['tmp_name'];
        //stocke le nom du fichier (nom du fichier et son extension importé ex : test.jpg)
        $name = $_FILES['file']['name'];
        //stocke la taille du fichier en octets
        $size = $_FILES['file']['size'];
        //stocke les erreurs (pb d'import, pb de droits etc...)
        $error = $_FILES['file']['error'];
        //déplacer le fichier importé dans le dossier image à la racine du projet
        $fichier = move_uploaded_file($tmpName, "../../imports/$name");
    }


/*-------------------------------------------------------
                    Formulaire HTML :
    -------------------------------------------------------*/
?>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <form action="" method="POST" enctype="multipart/form-data">
        <!-- MAX_FILE_SIZE doit précéder le champ input de type file -->
        <input type="hidden" name="MAX_FILE_SIZE" value="30000" />
        <input type="file" name="file">
        <p><button type="submit">importer</button></p>
    </form>
</html>
```

Le champ caché MAX_FILE_SIZE (mesuré en octets) doit précéder le champ input de type file et sa valeur représente la taille maximale acceptée du fichier par PHP. Cet élément de formulaire doit toujours être utilisé, car il permet d'informer l'utilisateur que le transfert désiré est trop lourd avant d'atteindre la fin du téléchargement. Toutefois, il conviendra de le vérifier côté serveur, car toutes les informations côté client peuvent facilement être modifiées par l'utilisateur.

PHP

Envoi de fichiers et sécurité :

Les sites web aux fonctionnalités complexes proposent très souvent des fonctionnalités d'upload à leurs utilisateurs (photo de profil, …).

Toutefois, il conviendra d'être très vigilant car une fonctionnalité d'envoi de fichiers mal configurée peut constituer une vulnérabilité redoutable.

En effet, imaginons un attaquant qui utiliserais le formulaire d'envoi de fichier pour y déposer un fichier php. Si ce fichier était accepté et que l'attaquant réussissait à l'appeler par l'URL, le code présent dans ce fichier serait directement exécuté par le serveur web. Un attaquant mal intentionné exploiterait cette faille pour y déposer par exemple une backdoor.

C'est la **faille upload**.

Le moyen le plus simple de s'en protéger est de n'accepter qu'un nombre limité d'extensions de fichier, répondant parfaitement à notre usage, et de rejeter tout upload de fichier qui n'y répondrait pas.

Attention toutefois, car il existe, côté attaquant diverses techniques d'évasion (fichier avec plusieurs extensions, extensions exotiques, …).

Pour plus d'informations : https://book.hacktricks.xyz/pentesting-web/file-upload

PHP

17 Interaction avec une base de données :

Le langage PHP permet d'interagir de façon simple et sécurisé (dans certains cas) avec les systèmes de gestion de bases de données les plus courants (MySQL, MariaDB, OracleDB, postgreSQL,…) .

Il existe actuellement 2 principales façons « modernes » de se connecter à une base de données SQL et d'effectuer des requêtes SQL en PHP : l'utilisation de MySQLi (https://www.php.net/manual/fr/book.mysqli.php) ou l'utilisation de PDO (https://www.php.net/manual/fr/book.pdo.php).

Ces deux méthodes présentent chacune des avantages dans des cas d'utilisation spécifiques, toutefois nous étudieront exclusivement sur la méthode PDO qui est une méthode orientée objet.

Afin de pouvoir réaliser des requêtes SQL, nous devons respecter certaines étapes :

- Etablir ou récupérer une connexion à la base de données,
- Exécuter la requête SQL
- Récupérer le résultat dans une variable (pour les requêtes de type select) et le traiter

1 - Se connecter à la base de données :

La première des actions à effectuer pour interagir avec une base de données est de se connecter à celle-ci.

Pour se faire nous devrons instancier un objet PDO.

Nous utiliserons la syntaxe suivante :

//connexion à la base de données

```php
<?php
$db = new PDO('mysql:host=localhost;dbname=nom_de_la_bdd', 'root','',
array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
$db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
?>
```

Cette ligne de code va stocker dans une variable $bdd un objet **PDO** (que vous verrons dans les chapitres prochains) qui va contenir les attributs suivants :

- mysql :host = ***localhost***; (base de données de type mysql dont le host (IP du serveur) est localhost : ici identique au serveur apache) et son nom dbname = ***nom_de_la_bdd'***
- le paramètre suivant est le nom de l'utilisateur qui se connectera à la DB dans le contexte de notre application, ici : ***'root'***
- le paramètre suivant est le mot de passe de l'utilisateur, dans l'exemple ci-dessus il est vide : ***'',***
- le paramètre array permet de spécifier certaines options. Ici, une option qui active le mode d'erreur avancé.

- L'appel de la méthode setAttribute ($db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC)) permet de définir que les requêtes SQL que nous exécuterons devront retourner un résultat sous la forme de tableau associatif, dont la clé sera le nom de la colonne récupérée.

2 - Exécution d'une requête SQL :

La façon la plus simple pour exécuter une requête avec PDO est d'utiliser la méthode « query » : par exemple :

```php
<?php
$rows = $db->query('SELECT * FROM book ORDER BY title');
foreach ($rows as $row) {
    var_dump($row);
};
?>
```

Toutefois, lorsque nous devons utiliser des paramètres de filtres (clause where notamment), nous pouvont procéder de 2 façons différentes :

-Les requêtes classiques qui ne permettront pas de se prémunir des injections SQL. Leur utilisation sera donc totalement prohibée lorsque nous n'aurons pas le contrôle total des paramètres (lorsqu'ils seront fournis par l'utilisateur). Il est donc préférable de ne jamais les utiliser pour éviter tout problème.

-Les requêtes préparées qui sont, elles sécurisées car PDO échappera directement les paramètres, limitant fortement les risques d'injections SQL.

PHP

Exemple de requête classique :

En premier lieu nous devons nous connecter à la base de données (en utilisant le code vu dans la partie 1 du chapitre 13) :

```php
<?php
//Connexion à la base de données
$db = new PDO('mysql:host=localhost;dbname=nom_de_la_bdd', 'root', '',
array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
$db->exec("set names utf8");
//Exécution de la requête SQL avec un try catch pour la gestion des exceptions (messages
d'erreurs)
try {
    //requête pour stocker le contenu de toute la table le contenu est stocké dans le
tableau $reponse
    $reponse = $db->query('SELECT * FROM utilisateur');
    //boucle pour parcourir et afficher le contenu de chaque ligne de la table
    while ($donnees = $reponse->fetch()) {
        //affichage des données d'une colonne de la bdd par son non d'attribut
        echo '<p>' . $donnees['nom_attribut'] . '</p>';
    }
} catch (Exception $e) {
    //affichage d'une exception en cas d'erreur
    die('Erreur : ' . $e->getMessage());
}

?>
```

Cette requête va stocker dans une variable **$reponse** la requête ***select***.

Dans la boucle **while** nous allons ensuite, par la méthode « fetch », récupérer chaque ligne de résultat de la requête et la stocker dans une variable **$donnees.**

Nous pouvons ensuite afficher pour chaque enregistrement le contenu d'un **attribut ($donnees['nom_attribut'])** et l'afficher avec la méthode **echo** dans un paragraphe **html** (balise **p**).

PHP

Exemple de requête préparée :

Notre requête préparée va exécuter une requête **SQL** de type **select** similaire à la requête classique ci-dessus mais dans laquelle nous allons lui passer un **paramètre** (**$nom_utilisateur**) qui contiendra un nom d'utilisateur.

```php
<?php
//Connexion à la base de données
$db = new PDO('mysql:host=localhost;dbname=nom_de_la_bdd', 'root', '',
array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
$db->exec("set names utf8");

//Préparation de la requête SQL nous stockons dans une variable $req la requête à exécuter
$req = $db->prepare('SELECT * FROM utilisateur where nom_utilisateur = :nom_utilisateur');
// On va "binder" le paramètre correspondant au "non_utilisateur" de type String
$req->bindParam('nom_utilisateur', $nomUtilisateur, PDO::PARAM_STR);
//Une fois tous les paramètres bindés, on peut exécuter la requete
$req->execute();

//boucle pour parcourir et afficher le contenu de chaque ligne de la table
while ($donnees = $reponse->fetch()) {
    //affichage des données d'une colonne du résultat de la requête par son non d'attribut
    echo '<p>' . $donnees['nom_attribut'] . '</p>';
}
//fermeture de la connexion à la bdd
$req->closeCursor();
?>
```

Cette requête effectue le même traitement que la requête classique mais injecte un paramètre de filtre de façon sécurisé.

PHP

3 - Transactions SQL

Lorsque l'on développe une application ou une fonction, nous allons souvent avoir de multiples traitements, dépendants les uns des autres, à effectuer pour parvenir à notre but.

Par exemple, il est possible que nous souhaitions enregistrer un objet X dans la DB (1), puis récupérer un objet Y (2) qui nous servira à calculer des informations nous permettant de mettre à jour un objet Z (3).

Imaginons qu'un problème survienne à l'étape (2) ou (3), par exemple un bug dans le code faisant crasher notre application, ou un problème d'accès au serveur de base de données. Nous pourrions alors nous retrouver avec des données corrompues (l'objet X aura été ajouté, mais l'objet Z n'aura pas pu être mis-à-jour), ce qui nuirait à l'intégrité des données.

Les transactions SQL vont nous permettre de s'assurer que toutes les requêtes peuvent être effectuées avec de réellement les exécuter, et vont nous permettre de revenir à l'état initial en cas de problème.

Certaines méthodes de l'objet PDO nous permettent de gérer très facilement les transactions :

- **PDO::beginTransaction** : Démarre une transaction
  PDO::beginTransaction() désactive le mode autocommit. Lorsque l'autocommit est désactivé, les modifications faites sur la base de données via les instances des objets PDO ne sont pas appliquées tant que vous ne mettez pas fin à la transaction en appelant la fonction PDO::commit(). L'appel de PDO::rollBack() annulera toutes les modifications faites à la base de données et remettra la connexion en mode autocommit.
  Pour plus d'informations, voir https://www.php.net/manual/fr/pdo.begintransaction.php

- **PDO::commit** : Valide une transaction
  PDO::commit() valide une transaction, remet la connexion en mode autocommit en attendant l'appel à la fonction PDO::beginTransaction() pour débuter une nouvelle transaction.
  Pour plus d'informations, voir https://www.php.net/manual/fr/pdo.commit.php

- **PDO::rollBack** : Annule une transaction
  Annule la transaction courante, initiée par la fonction PDO::beginTransaction().
  Si la base de données est en mode autocommit, cette fonction restaurera le mode autocommit après l'annulation de la transaction.
  Pour plus d'informations, voir https://www.php.net/manual/fr/pdo.rollback.php

PHP

Exemple d'utilisation des transactions :

```php
// Test transactions
$categorie1 = 'CAT transaction OK';
$article1 = ['Article transaction OK', "Jusqu'ici, tout va bien."];
$categorie2 = 'CAT transaction KO';
$article2 = ['Article transaction KO', "Faites vos jeux, rien ne va plus"];

// Test de transaction OK
try {
    // Ouverture de la transaction sur l'objet PDO ($conn = new PDO(…))
    $conn->beginTransaction();
    // Insertion de la catégorie en BDD
    $stt = $conn->prepare("insert into categorie(nom_categorie) values (?)");
    $stt->bindParam(1, $categorie1);
    $stt->execute();
    $catId = $conn->lastInsertId();
    // Insertion de l'article en BDD
    $stt = $conn->prepare("insert into
article(nom_article,contenu_article,id_categorie) values(?,?,?)");
    $stt->bindParam(1, $article1[0]);
    $stt->bindParam(2, $article1[1]);
    $stt->bindParam(3, $catId, PDO::PARAM_INT);
    $stt->execute();

    // Validation des résultats
    $conn->commit();
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage() . "\r\n";
}


// Test de transaction KO
try {
    // Ouverture de la transaction
    $conn->beginTransaction();
    // Insertion de la catégorie en BDD
    $stt = $conn->prepare("insert into categorie(nom_categorie) values (?)");
    $stt->bindParam(1, $categorie2);
    $stt->execute();
    $catId = $conn->lastInsertId();
    // Insertion de l'article en BDD
    $stt = $conn->prepare("insert into
article(nom_article,contenu_article,id_categorie) values(?,?,?)");
    $stt->bindParam(1, $article2[0]);
    $stt->bindParam(2, $article2[1]);
    $stt->bindParam(3, $catId, PDO::PARAM_INT);
    $stt->execute();

    // Annulation des résultats
    $conn->rollBack();
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage() . "\r\n";
}
```

A l'exécution de ce bout de code, on constatera que la catégorie1 et l'article1 auront bien été persistés en BDD mais pas la catégorie2 et l'article2.

PHP

Exemple plus réaliste d'utilisation des transactions :

Dans la pratique, nous utiliserons le plus souvent les transactions dans un bloc « try/catch ».

```php
try {
    // Ouverture de la 1ère transaction
    $conn->beginTransaction();
    // Insertion de la catégorie en BDD
    $stt = $conn->prepare("insert into categorie(nom_categorie) values
(?)");
    $stt->bindParam(1, $categorie1);
    $stt->execute();
    $catId = $conn->lastInsertId();
    // Insertion de l'article en BDD
    $stt = $conn->prepare("insert into
article(nom_article,contenu_article,id_categorie) values(?,?,?)");
    $stt->bindParam(1, $article1[0]);
    $stt->bindParam(2, $article1[1]);
    $stt->bindParam(3, $catId, PDO::PARAM_INT);
    $stt->execute();
    // Validation de la 1ère transaction
    $conn->commit();

    // Ouverture de la 2ème transaction
    $conn->beginTransaction();
    $stt = $conn->prepare("insert into
article(nom_article,contenu_article,id_categorie) values(?,?,?)");
    $stt->bindParam(1, $article2[0]);
    $stt->bindParam(2, $article2[1]);
    $stt->bindParam(3, $catId, PDO::PARAM_INT);
    $stt->execute();
    throw new Exception();  // On throw volontairement une exception.
    $conn->commit();    // Ce commit ne sera pas exécuté
} catch (Exception $e) {
    // Une exception a été levée, on annule la transaction
    $conn->rollBack();
}
```

PHP

Exercices :

TP 1ère partie :

a) Créer une base de données **MYSQL** avec les informations suivantes :
   - Nom de la bdd : « **articles** »,
   - une table nommée **article** qui va posséder les champs suivants :
     i. **id_article** (clé primaire),
     ii. **nom_article** de type varchar(50),
     iii. **contenu_article** de type varchar (255),

b) Créer une page php qui va contenir un formulaire html avec comme méthode POST (balise **form**)
   - A l'intérieur du formulaire rajouter les champs suivants :
     i. Un champ input avec comme attribut html **name = «nom_article »**
     ii. Un champ input avec comme attribut html **name = «contenu_article »**
     iii. Un champ input de type **submit** avec comme attribut html **value = «Ajouter»**

c) Ajouter le code php suivant:
   - Créer 2 variables $name, $content
   - -Importer le contenu des 2 super globales **$_POST['nom_article']**, **$_POST['contenu_article']** et tester les avec la méthode ***isset()*** dans les variables créés précédemment (**$name** et **$content**),
   - Ajouter le code de **connexion** à la base de données en vous inspirant des exemples vus dans ce chapitre,
   - Ajouter une **requête simple** qui va insérer le contenu des 2 champs dans un nouvel enregistrement (requête **SQL insert**),

d) Bonus :
   - Utiliser une requête **SQL préparée** à la place de la requête **simple.**
   - Afficher dans un paragraphe le nom et le contenu de l'article ajouté en bdd en dessous du formulaire.

PHP

TP 2ème partie :

a) Créer une page php,
b) Ajouter le script php permettant de se connecter à la base de données **articles**,
c) Ajouter le script php qui va effectuer une requête **SQL select** permettant de récupérer tous les articles,
d) Formater le résultat de la requête (dans le résultat de la boucle while) pour quelle l'affiche sous cette forme :

```
<p>numéro de  l'article : id de l'article n</p>

<br>

<p>nom de l'article : nom de l'article n</p>

<br>

<p>contenu de l'article : contenu de l'article n</p>

<br>
```

(La liste de tous les articles devra reprendre la mise en forme ci-dessus -> a l'intérieur de la boucle while).

PHP

18 PHP et JavaScript :

Interactions entre PHP et JavaScript

Comme vous avez pu le constater, le code PHP étant interprété côté serveur, il est « invisible » côté front (en utilisant l'inspecteur du navigateur par exemple).

C'est une chance car on ne souhaiterait pas que notre code serveur soit visible de tous, étant donné qu'il peut contenir des informations sensibles telles que nos informations de connexion à la base de données.

Toutefois, il est parfois nécessaire de récupérer des informations calculées côté serveur et de les traiter dans nos scripts JavaScript. Et bien, comme nous l'avons fait pour afficher des données calculées dans notre page HTML (en utilisant notamment la fonction « echo »), nous devrons ouvrir et refermer des balises php dans notre script JS lorsque nous voudrons utiliser une donnée du serveur.

Voici un exemple d'utilisation :

```
<!DOCTYPE html>
<html lang="en">
<body>
    <h1>Test d'intégration du PHP dans des scripts JavaScript

        <?php
        // Définition de variables et de calculs en php
         $a = 3;
         $b = -5;
        ?>

        <script>
            alert("Ca se passe dans la console bro");

            // On additionne, en php, les variables $a et $b
            let addition1 = <?php echo ($a + $b); ?>;

            // On récupère les variables $a et $b pour les mettres dans les variables JS a
et b
            // Puis on fait l'addition en JS cette fois
            let a = <?= $a ?>;
            let b = <?= $b ?>;
            let addition2 = a + b;

            console.log("Résultat de l'addition faite directement en PHP : " + addition1);
            console.log("Résultat de l'addition faite en 2 temps, après récupération des
variables en JS puis calcul en JS : " + addition2);
        </script>
</body>
</html>
```

PHP

AJAX : Asynchronous JavaScript And XML

Dans le web moderne, que ça soit pour des raisons ergonomiques ou pour optimiser les temps de chargement des pages web, nous ne voulons pas toujours recharger une page web au submit d'un formulaire.

Un exemple très courant est par exemple lorsque l'on est sur une page affichant une liste de résultats (par exemple une liste d'articles de blog, ou une liste de produits sur une boutique en ligne). Pour ce genre de pages, on a souvent un formulaire qui permet de filtrer ou trier les résultats, et on souhaite que le formulaire gérant ces filtres mette à jours uniquement le tableau ou la liste de résultats.

Pour ce faire, nous utilisons AJAX, qui permet de faire des requêtes au serveur et traiter la réponse de façon asynchrone (en arrière-plan). La réponse que l'on reçoit nous permettra de mettre à jour le DOM (Document Object Model) de la page web, en modifiant uniquement certaines parties de cette dernière.

Il est à noter que le XML, langage de balise autrefois utilisé comme format de réponse du serveur « favori » dans le cadre des requêtes Ajax est aujourd'hui souvent remplacé par le format JSON, format beaucoup moins verbeux.

Objet XMLHttpRequest

L'objet XMLHttpRequest est l'objet natif nous permettant de traiter les requêtes asynchrones en JS.

Cet objet est toujours largement utilisé mais progressivement délaissé au profit de l'API « Fetch » de plus haut-niveau et implémentant notamment les « promesses ».

Pour effectuer une requête Ajax, nous allons devoir suivre 4 étapes :

1. Créer un objet XMLHttpRequest
2. Initialiser et préparer la requête (URL, paramètres, méthode)
3. Envoyer la requête
4. Définir des gestionnaires d'événements nous permettant de traiter les réponses du serveur ou les cas d'erreur.

```
// Instanciation d'un nouvel objet XMLHttpRequest
let xhr = new XMLHttpRequest();

// Préparation de la requête (méthode, URL)
xhr.open("GET", "/url/de/mon/script.php");

// Définition du format de la réponse (ici JSON)
xhr.responseType = "json";

// Envoi de la requête
xhr.send();
```

PHP

Traiter la réponse renvoyée par le serveur

Pour traiter les réponses du serveur, nous allons utiliser le gestionnaire d'événements XMLHttpRequest :

- **Utilisation des gestionnaires d'événements de l'objet XMLHttpRequest**
  3 événements peuvent être déclenchés au cours du traitement de notre requête :
  - **L'événement « load »** qui se déclenche lorsque la requête a bien été effectuée et que la réponse a été reçue.
    Au sein du gestionnaire d'événement « load », nous pouvons tester la valeur du status code http de la réponse
    Voici quelques-uns des status codes les plus fréquents :
    - 100 Continue : Requête en cours
    - 200 OK : Requête correctement exécutée
    - 401 Unauthorized : Problème d'identification
    - 404 Not Found : l'url n'a pas été trouvée
    - 500 Internal Server Error : Erreur côté serveur.

    Pour s'assurer que tout s'est bien passé, on veut donc vérifier que le status code est bien 200.
    On peut donc ensuite traiter la réponse du serveur qui sera accessible par la propriété « response » de l'objet XMLHttpRequest
  - **L'événement « error »** qui se déclenche lorsque la requête n'a pas pu aboutir
  - **L'événement « progress »** qui se déclenche à intervalles réguliers et nous permet de savoir où en est l'exécution de la requête

Pour plus d'informations et des exemples d'utilisation :
https://developer.mozilla.org/fr/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest

PHP

Voici ce à quoi ça pourrait ressembler au niveau du code :

```
//On crée un objet XMLHttpRequest
let xhr = new XMLHttpRequest();

//On initialise notre requête avec open()
xhr.open("GET", "une/url");

//On veut une réponse au format JSON
xhr.responseType = "json";

//On envoie la requête
xhr.send();

//On créé une fonction anonyme qui sera exécutée au trigger de l'événement.
//En l'occurrence ici l'événement load
xhr.onload = function(){
    //Si le statut HTTP n'est pas 200, on a eu un probleme
    if (xhr.status != 200){
        //...On affiche le statut et le message correspondant
        alert("Erreur " + xhr.status + " : " + xhr.statusText);
    //Si le statut HTTP est 200, on affiche le nombre d'octets téléchargés et la réponse
    }else{
        alert(xhr.response.length + " octets  téléchargés\n" +
JSON.stringify(xhr.response));
    }
};

//On créé une fonction anonyme qui sera exécutée au trigger de l'événement.
//En l'occurrence ici l'événement error
xhr.onerror = function(){
    alert("La requête a échoué");
};

//On créé une fonction anonyme qui sera exécutée au trigger de l'événement.
//En l'occurrence ici l'événement progress
//Pendant le téléchargement.
xhr.onprogress = function(event){
    //lengthComputable = booléen; true si la requête a une length calculable
    if (event.lengthComputable){
        //loaded = contient le nombre d'octets téléchargés
        //total = contient le nombre total d'octets à télécharger
        alert(event.loaded + " octets reçus sur un total de " + event.total);
    }
};
```

PHP

API Fetch

L'utilisation de l'API Fetch est une façon plus moderne de réaliser des requêtes AJAX.

L'API Fetch définit des objets **Request**, **Response** et **Header** (permettant de créer et manipuler des requêtes, réponses, et headers HTTP) et propose également un mixin « **Body** » permettant d'interagir avec le corps de la requête.

L'API Fetch va également mettre à notre disposition la méthode « fetch() » qui permettra d'exécuter les requêtes.

La méthode fetch() attend au minimum la ressource vers laquelle on souhaite effectuer la requête (souvent le script PHP, une image, une API externe, …) et peut prendre un objet JSON de paramètres.

La méthode fetch() renvoie une promesse (objet de type Promise) qui va se résoudre avec un objet Response. La promesse sera rompue si la requête HTTP n'a pas pu être effectuée, mais le plus souvent, y compris en cas d'erreur, la promesse sera remplie. Il suffira donc de connaitre le status de la réponse HTTP. On utilisera les attributs « ok » et « status » de l'objet Response.

Pour plus d'informations et des exemples d'utilisation :
https://developer.mozilla.org/fr/docs/Web/API/Fetch_API/Using_Fetch

PHP

Voici un exemple d'utilisation :

```
//Création d'un objet Headers
var myHeaders = new Headers();

// On initialise l'objet JSON qui définira les propriétés de la requête. Ici, on définit la méthode (GET), l'objet Headers, le mode "cors" autorisant les requêtes cross-origin
var myInit = {
  method: "GET",
  headers: myHeaders,
  mode: "cors",
  cache: "default",
};

//flowers.jpg est la destination de notre requête, et myInit nos options
fetch("flowers.jpg", myInit)
  .then(function (response) {        // Definition de notre 'promesse'
    if (response.ok) {      //If response.ok, on est sur un status code HTTP compris entre 200 et 299
      response.blob().then(function (myBlob) {
        var objectURL = URL.createObjectURL(myBlob);
        myImage.src = objectURL;
      });
    } else {       //Sinon, un probleme est apparu. On peut éventuellement vérifier le status code http (response.status) pour savoir ce qu'il s'est passé
      console.log("Mauvaise réponse du réseau");
    }
  })
  .catch(function (error) {
    console.log(
      "Il y a eu un problème avec l'opération fetch : " + error.message,
    );
  });
```

PHP

Balise HTML <template>

La balise <template> est une balise qui n'est pas affichée sur la page. Elle est utilisée pour gérer l'affichage des réponses de nos requêtes Ajax.

Un usage classique sera de mettre le template d'une ligne d'un tableau de résultat à l'intérieur de cette balise, ou un champ de formulaire dont l'affichage est dynamique (exemple : ajouter plusieurs catégories d'articles dans le cas d'une relation many to many).

Exemple :

```
<table id="producttable">
  <thead>
    <tr>
      <td>UPC_Code</td>
      <td>Product_Name</td>
    </tr>
  </thead>
  <tbody>
    <!-- existing data could optionally be included here -->
  </tbody>
</table>

<template id="productrow">
  <tr>
    <td class="record"></td>
    <td></td>
  </tr>
</template>
```

Exercices

TP 3ème partie :

a) Ajouter un champ de recherche sur la page de résultats des articles (input de type text)
b) Faire en sorte que ce champ permette de filtrer les résultats de la requête pour que les articles soient retournés si le nom ou le contenu de l'article contient la valeur recherchée. La liste des articles devra être renvoyée au format JSON (**fonction PHP « json_encode() »**
c) A l'aide d'un listener sur le champ de recherche, faire en sorte qu'à chaque fois que la valeur de l'input change, une requête Ajax soit envoyée (**event « onkeyup »**).
d) A partir du JSON des articles et à l'aide d'une balise « template », mettre à jour le DOM à la réception de la réponse.

Correction

Fichier articles_ajax.php : le « contrôleur »

```php
<?php

$conn = null;
try {
    //Création de la connexion
    $conn = new PDO("mysql:host=localhost;dbname=articles", 'root', 'rootpwd');
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);     // Attribut permettant de lever une erreur en cas de problème à l'utilisation de l'objet PDO (clé : PDO::ATTR_ERRMODE, valeur :PDO::ERRMODE_EXCEPTION)
    $conn->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC); // Attribut permettant de retourner le résultat de la requête SQL sous forme de tableau associatif dont la clé est le nom de la colonne dans la BDD

    // Condition ternaire
    $searchValue = !empty($_GET['search']) ? '%'.$_GET['search'].'%' : '%%';

    $stt = $conn->prepare("SELECT * FROM article WHERE nom_article LIKE :searchvalue OR contenu_article LIKE :searchvalue2");
    $stt->bindValue('searchvalue', $searchValue);
    $stt->bindValue('searchvalue2', $searchValue);
    $stt->execute();
    $articles = $stt->fetchAll();

    $response = json_encode($articles);

    echo $response;
} catch (PDOException $e) {
    echo $e->getMessage();
}
```

PHP

Fichier list_articles_ajax.php : le code HTML / JavaScript

Code HTML

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Tous les articles</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.min.css"
rel="stylesheet"
        integrity="sha384-4bw+/aepP/YC94hEpVNVgiZdgIC5+VKNBQNGCHeKRQN+PtmoHDEXuppvnDJzQIu9"
crossorigin="anonymous">
</head>

<body>
    <input type="text" name="search" id="search" onkeyup="updateQuery()">
    <div>
        <table class="table table-striped">
            <thead>
                <tr>
                    <th scope="col">id</th>
                    <th scope="col">nom article</th>
                    <th scope="col">contenu article</th>
                </tr>
            </thead>
            <tbody id="table_tbody">

            </tbody>
        </table>
        <template id="row_table_template">
            <tr scope="row">
                <td></td>
                <td><a></a></td>
                <td></td>
            </tr>
        </template>
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/js/bootstrap.bundle.min.js"
        integrity="sha384-HwwvtgBNo3bZJJLYd8oVXjrBZt8cqVSpeBNS5n7C8IVInixGAoxmnlMuBnhbgrkm"
        crossorigin="anonymous"></script>
</body>
</html>
```

PHP

Script JS version XMLHttpRequest (à ajouter dans list_articles_ajax.php)

```
<script>
        document.addEventListener("DOMContentLoaded", () => {
        updateQuery();
        });


    function updateQuery(){
        // Récupération de la value de l'élément input
        let searchValue = document.getElementById("search").value;
    //Création de l'objet XMLHttpRequest
    let xhr = new XMLHttpRequest();
    // Création de la requête de type GET
    xhr.open("GET", "articles_ajax.php?search="+searchValue);
    xhr.responseType = "json";
    //Envoi de la requête
    xhr.send();

    // L'événement onload est déclenché lorsque la requête s'est terminée sans erreur
    xhr.onload = function() {
        document.getElementById("table_tbody").innerHTML = ""; // Purge de la table
actuelle
    const tbody = document.querySelector("tbody");  // Récupération de l'élément du DOM
tbdody
    const template = document.querySelector("#row_table_template"); // Récupération du
template de ligne

    // Boucle permettant d'itérer sur chacune des lignes retournées par le script PHP (au
format JSON)
    for(let i=0; i<xhr.response.length; i++) {
                    const clone = template.content.cloneNode(true);
    let td = clone.querySelectorAll("td");  // Sélection des éléments td
    let link = clone.querySelectorAll("a"); // Sélection des éléments a
    td[0].textContent = xhr.response[i]['id_article'];
    td[2].textContent = xhr.response[i]['contenu_article'];
    link[0].setAttribute('href', "article.php?id_article="+ xhr.response[i]['id_article']);
    link[0].textContent = xhr.response[i]['nom_article'];

    tbody.appendChild(clone);   // Ajout de notre ligne tr a la suite des autres
                }
            }

    // L'événement onerror est déclenché lorsque la requête a levé une erreur
    xhr.onerror = function() {
        console.log(xhr);
            }
        }
</script>
```

PHP

Script JS version API Fetch (à ajouter dans list_articles_ajax.php)

```
<script>
    document.addEventListener("DOMContentLoaded", () => {
        updateQuery();
    });

    function updateQuery() {
        // Récupération de la value de l'élément input
        let searchValue = document.getElementById("search").value;
    //Creation d'un objet Header
    let headers = new Headers();
    let requestParams = {
        method: "GET",
    headers: headers,
    mode: "cors",
    cache: "default",
            };
    //Utilisation d'un objet URLSearchParams pour une gestion plus propre des parametres
    let searchParams = new URLSearchParams({
        'search': searchValue,
            });
    fetch("articles_ajax.php?" + searchParams, requestParams).
    // fetch("articles_ajax.php", requestParams).
    then(function (response) {
                    if (response.ok) {
                        return response.text();
                    }
                }).then(function (articles) {
        document.getElementById("table_tbody").innerHTML = ""; // Purge de la table
actuelle
    const tbody = document.querySelector("tbody");  // Récupération de l'élément du DOM
tbdody
    const template = document.querySelector("#row_table_template"); // Récupération du
template de ligne
    JSON.parse(articles).forEach(function (article) { // loop though the response
        console.log(article);
    const clone = template.content.cloneNode(true);
    let td = clone.querySelectorAll("td");  // Sélection des éléments td
    let link = clone.querySelectorAll("a"); // Sélection des éléments a
    td[0].textContent = article['id_article'];
    td[2].textContent = article['contenu_article'];
    link[0].setAttribute('href', "article.php?id_article=" + article['id_article']);
    link[0].textContent = article['nom_article'];

    tbody.appendChild(clone);   // Ajout de notre ligne tr a la suite des autres
                    });
                });
        }
</script>
```

PHP

19 Programmation Orientée Objet en PHP

Dans ce chapitre nous allons étudier PHP sous un nouvel angle, en utilisant la **programmation orientée objet**.

Généralités

La **programmation orienté objet** est un paradigme qui permet de structurer le code autour des objets, facilitant la réutilisation, la maintenabilité et la modularité du code.

Pour faire une analogie avec la vie courante, la POO pourrait être assimilée à la manière dont nous organisons les choses dans la vie quotidienne.

En effet, nous avons des objets qui sont définis (classe) par leurs caractéristiques (attributs) et qui peuvent réaliser des actions (méthodes).

Objets et classes

- **Classe** : La classe est un modèle ou un plan qui définit les propriétés et les méthodes communes à un groupe d'objets. Par exemple, une classe Voiture pourrait avoir des propriétés comme $couleur et $marque, et des méthodes comme demarrer() ou freiner().
- **Objet** : C'est une instance d'une classe. Par exemple, une instance de la classe "Voiture" pourrait représenter une voiture spécifique avec sa couleur et sa marque.

Exemple d'une classe en PHP

```php
class Voiture {
    public $couleur;           // La visibilité de cet attribut est définie à "public"
    public $marque;

    public function demarrer() {
        echo "La voiture démarre.";
    }
}

// Instanciation d'un nouvel objet Voiture
$maVoiture = new Voiture();
$maVoiture->couleur = "Rouge"; // Modification de la valeur de la propriété/attribut 'couleur'
$maVoiture->marque = "Toyota"; // Modification de la valeur de la propriété/attribut 'marque'
$maVoiture->demarrer();  // Appel de la méthode "demarrer" sur l'objet maVoiture -> Affiche : "La voiture démarre."
```

PHP

Bonnes pratiques

Le nom d'une classe en PHP devra respecter les contraintes suivantes :

- Commencer par un caractère alphabétique ou un underscore '_'
- Ne contenir que des caractères alphanumériques et des underscores
- De plus, une bonne pratique sera de respecter la convention de nommage en PascalCase (chaque premier caractère d'un mot est en majuscule)

Dans l'exemple ci-dessus, le script PHP contient à la fois la définition de la classe Voiture, ainsi que le code du script. C'est très bien pour l'exemple et pour expérimenter.

Toutefois, une bonne pratique est que notre classe soit définie dans un fichier qui lui est propre (ex ici : Voiture.php). Ce fichier ne contiendra que le code de la classe (attributs et méthodes).

Propriétés et méthodes

- **Les propriétés** : Ce sont des variables définies au sein de notre classe et donc directement associées aux instances de nos objets. Ce sont les attributs de nos objets.
- **Les méthodes** : Ce sont les fonctions définies au sein de notre classe. Cela représente les actions que notre objet peut effectuer.

Encapsulation et visibilité

L'encapsulation est l'un des principes fondamentaux de la Programmation Orientée Objet (POO).

Il consiste à restreindre l'accès à certains éléments d'une classe (le plus souvent propriétés et méthodes). L'objectif de l'encapsulation est de ne laisser accessible/visible seulement le strict nécessaire. Cela permet d'éviter de modifier ou accéder à des ressources auxquelles on ne devrait pas, et c'est très important pour améliorer la robustesse de notre code.

Ainsi, pour mettre en place l'encapsulation, nous allons définir la visibilité de nos éléments à l'aide de mots clés :

- **public** : C'est la visibilité par défaut. Une ressource 'public' sera accessible partout, y compris en dehors de la classe.
- **private** : Un élément dont la visibilité serait définie à 'private' sera seulement accessible à l'intérieur de la classe.
- **protected** : Un élément dont la visibilité serait définie à 'protected' sera accessible au sein de la classe, et de toutes les classes enfant (donc des classes héritant de notre classe).

Une bonne pratique est de définir la visibilité par défaut de nos éléments à « private » et d'élargir (à protected ou public) la visibilité en cas de besoin.

PHP

Assesseurs et Mutateurs (Getters et Setters)

Avec l'encapsulation apparaissent aussi les concepts d'assesseurs et mutateurs (getters et setters).

Les getters et setters sont des méthodes créées pour, respectivement, récupérer la valeur d'une propriété (getter) ou modifier la valeur d'une propriété (setter).

Exemple

```php
class User {
      // Les 2 propriétés ont une visibilité 'private'. Elles sont donc uniquement accessibles dans la classe elle-même.
    private string $email;
    private string $password;

    /**  GETTERS  **/
    public function getEmail():string
    {
        return $this->email;
    }

    public function getPassword():string
    {
        return $this->password;
    }

    /**  SETTERS  **/
    public function setEmail(string $email):void
    {
        $this->email = $email;
    }

    public function setPassword(string $password):User
    {
        $this->password = $password;

        return $this;            // Ici, nous créons un setter respectant le design patter "Fluent". Ces setters retournent l'objet courant, ce qui permet ensuite de chainer l'utilisation des setters.
    }
}
// Instanciation d'un nouvel objet User
$user = new User();

// Etant donné que les propriétés sont private, nous ne pouvons plus accéder directement aux propriétés. Exemple :
// $user->email = 'mail@mail.com';        //Ce code ne fonctionnera pas car nous sommes à l'extérieur de la classe et la propriété est 'private'. Si on veut modifier l'email, il faut utiliser le setter
$user->setEmail('mail@mail.com');
```

PHP

Magic Methods PHP : Constructeur et destructeur

- **Constructeur : __construct()**
  Méthode spéciale appelée automatiquement lors de la création d'un objet. C'est ici que nous initialiserons les propriétés. Bien souvent, nous surchargerons le constructeur afin de l'adapter à nos besoins.
- **Destructeur : __destruct()**
  Méthode appelée à la destruction de l'objet (par exemple lorsqu'il n'est plus référencé).

Ces méthodes définies par PHP appartiennent aux « Magic Methods PHP ». Nous les reconnaissons par leur syntaxe particulière : leur nom commence par 2 underscores « __ ».
Ces méthodes ont parfois un fonctionnement par défaut et sont surchargeables.

Voici les magic méthods les plus courantes :

- __serialize() et __unserialize() :
  serialize() vérifie si la classe a une méthode avec le nom magique __serialize(). Si c'est le cas, cette méthode sera exécutée avant toute sérialisation. Elle doit construire et retourner un tableau associatif de paire clé/valeur qui représente la forme sérialisée de l'objet. Si aucun tableau n'est retournée une TypeError sera lancée.
  L'utilisation prévue de __serialize() est de définir une représentation arbitraire de l'objet pour le sérialiser facilement. Les éléments du tableau peuvent correspondre aux propriétés de l'objet mais ceci n'est pas requis.
  Inversement, unserialize() vérifie la présence d'une fonction avec le nom magique __unserialize(). Si elle est présente, cette fonction recevra le tableau restauré renvoyé par __serialize(). Il peut alors restaurer les propriétés de l'objet depuis ce tableau comme approprié.
- __sleep() et __wakeup() :
  serialize() vérifie si la classe a une méthode avec le nom magique __sleep(). Si c'est le cas, cette méthode sera exécutée avant toute sérialisation.
  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.
- __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.
- __invoke() :
  La méthode __invoke() est appelée lorsqu'un script tente d'appeler un objet comme une fonction.
- __clone() :
  Appelée lorsqu'un objet est cloné. Vous pouvez définir des actions à réaliser lors du clonage, comme des ajustements spécifiques pour les propriétés.
- La liste ci-dessus n'est pas exhaustive, il existe de nombreuses autres magic methods. Pour plus d'infos : https://www.php.net/manual/fr/language.oop5.magic.php

Exemple

```php
<?php
class Voiture
{
    public string $marque;
    public string $couleur;

    public function __destruct()
    {
        // TODO: Implement __destruct() method.
        echo "objet détruit : cette magic method est appelée automatiquement à la destruction de l'objet (soit par le ramasse miettes, soit par une action du develppeur)";
    }

    public function __toString()
    {
        return "La voiture est de marque $this->marque et de couleur $this->couleur";
    }
}

$car = new Voiture();
$car->couleur = 'rouge';
$car->marque = 'Peugeot';

echo $car;
echo "<br>Nous allons détruire l'objet. La méthode magique __unset sera donc appelée<br>";
unset($car);
```

Héritage

Une classe peut hériter d'une autre classe, ce qui permet à la classe enfant de réutiliser ou redéfinir les propriétés et méthodes de la classe parent.

Attention, en PHP il n'y a pas d'héritage multiple. C'est-à-dire qu'une classe ne peut hériter que d'une autre classe au maximum. Il y a toutefois des moyens d'émuler l'héritage multiple, notamment par l'utilisation de Traits.

PHP

Syntaxe de base

```php
<?php

class Vehicule
{
    protected string $marque;
    protected string $couleur;

    public function __construct(string $marque, string $couleur)
    {
        $this->marque = $marque;
        $this->couleur = $couleur;
    }
    public function rouler()
    {
        echo "Le véhicule roule.";
    }
    public function methodeASurcharger()
    {
        echo "methode a surcharger : parent";
    }
}

class Voiture extends Vehicule
{
    private int $puissanceFiscale;

    public function __construct(string $marque, string $couleur, string $puissanceFiscale)
    {
        parent::__construct($marque, $couleur);     // Le mot clé "parent' nos permet d'appeller le constructeur du parent

        $this->puissanceFiscale = $puissanceFiscale;
    }

    public function klaxonner()
    {
        echo "La voiture klaxonne.";
    }

    public function methodeASurcharger()
    {
        echo "methode a surcharger : enfant";
    }

    public function getPuissanceFiscale(): int
    {
        return $this->puissanceFiscale;
    }

    public function setPuissanceFiscale(int $puissanceFiscale): Vehicule
    {
        $this->puissanceFiscale = $puissanceFiscale;
        return $this;
    }
}
$voiture = new Voiture("Peugeot", "Grise", 8);
$vehicule = new Vehicule("John Deere", "Vert");

$voiture->rouler();     // Affichera : "Le véhicule roule."
$voiture->klaxonner();
$voiture->methodeASurcharger();     // Affichera "methode a surcharger : enfant"

// Sur le vehicule, nous n'avons pas accès à la méthode klaxonner
$vehicule->rouler();
$vehicule->methodeASurcharger();        // Affichera "methode a surcharger : parent"
```

PHP

Polymorphisme

Le **polymorphisme** est la capacité des objets à être traités de manière interchangeable, même s'ils appartiennent à des classes différentes, tant qu'ils partagent un comportement commun.

Cela se fait souvent en utilisant l'**héritage** ou l'**implémentation d'interfaces**. Le polymorphisme permet de manipuler différents types d'objets avec une interface unifiée, tout en permettant aux objets individuels de définir leur propre comportement spécifique.

Exemple

```php
class Animal {
    // Méthode générique que chaque animal va redéfinir
    public function faireDuBruit() {
        echo "Cet animal fait un bruit.<br>";
    }
}
class Chien extends Animal {
    // Le chien redéfinit la méthode faireDuBruit
    public function faireDuBruit() {
        echo "Le chien aboie : Woof Woof!<br>";
    }
}

class Chat extends Animal {
    // Le chat redéfinit la méthode faireDuBruit
    public function faireDuBruit() {
        echo "Le chat miaule : Miaou!<br>";
    }
}

class Oiseau extends Animal {
    // L'oiseau redéfinit la méthode faireDuBruit
    public function faireDuBruit() {
        echo "L'oiseau chante : Cui cui!<br>";
    }
}
```

Contexte « static »

Au sein de nos classes, il est possible de déclarer nos propriétés et méthodes comme étant « static ».

Les **méthodes ou propriétés statiques** en PHP sont des méthodes ou propriétés associées à une classe plutôt qu'à une instance particulière de cette classe.

Cela signifie que vous pouvez appeler une méthode statique sans avoir besoin de créer un objet à partir de la classe. Les méthodes statiques sont souvent utilisées lorsque vous avez des comportements qui ne dépendent pas des propriétés d'une instance spécifique, mais qui sont plutôt liés à la classe elle-même.

Syntaxe et exemples

Pour définir une méthode ou propriété statique, on utilise le mot-clé **static**.

Pour appeler une méthode statique, vous devez utiliser l'opérateur **::** (appelé opérateur de résolution de portée) avec le nom de la classe ou avec le mot-clé **self** (si vous l'appelez à l'intérieur de la classe).

PHP

Déclaration d'une méthode statique

```php
class OutilsMath {
    // Méthode statique pour calculer le carré d'un nombre
    public static function carre($nombre) {
        return $nombre * $nombre;
    }
}
```

Appel d'une méthode statique

- Depuis l'extérieur de la classe

```php
echo OutilsMath::carre(4);  // Affiche 16
```

- Depuis l'intérieur de la classe

```php
class OutilsMath {
    public static function carre($nombre) {
        return $nombre * $nombre;
    }

    public static function cube($nombre) {
        return self::carre($nombre) * $nombre;  // Appel d'une méthode statique depuis une autre
    }
}

echo OutilsMath::cube(3);  // Affiche 27
```

Cas d'utilisation des méthodes statiques

- **Les méthodes ne dépendent pas de l'état d'une instance** :
  Les méthodes statiques sont idéales lorsque l'opération ne nécessite pas de données ou de propriétés spécifiques d'un objet particulier. Par exemple, des utilitaires mathématiques, des opérations sur des chaînes, etc.
- **Méthodes utilitaires**
  Les méthodes statiques sont souvent utilisées pour des tâches utilitaires ou des fonctions qui ne dépendent pas de la création d'objets. Exemples courants : classes de validation, de manipulation de chaînes, ou d'accès à des ressources communes.

```php
class Utils {
    public static function genererMotDePasse($longueur) {
        return substr(str_shuffle("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"), 0, $longueur);
    }
}

echo Utils::genererMotDePasse(8);  // Génère un mot de passe aléatoire de 8 caractères
```

PHP

- **Design Pattern Singleton**

```php
class Singleton {
    private static $instance = null;

    // Rendre le constructeur privé pour empêcher l'instanciation directe
    private function __construct() {
        echo "Singleton initialisé.<br>";
    }

    // Méthode statique pour obtenir l'instance unique
    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Singleton();
        }
        return self::$instance;
    }
}

$obj1 = Singleton::getInstance();  // Affiche "Singleton initialisé."
$obj2 = Singleton::getInstance();  // Rien, car l'instance existe déjà
```

- **Accès à des constantes de classe**

```php
class Configuration {
    const NOM_SITE = "MonSite";

    public static function afficherNomSite() {
        return self::NOM_SITE;
    }
}

echo Configuration::afficherNomSite();  // Affiche "MonSite"
echo Configuration::NOM_SITE; //Affiche "MonSite"
```

Exercice

Codez votre classe de connexion à la DB en respectant le pattern Singleton.

Classes abstraites

Les **classes abstraites** en PHP sont des classes qui ne peuvent pas être instanciées directement. Elles servent de modèles pour d'autres classes et définissent des comportements communs que les classes dérivées (enfants) doivent implémenter.

Une classe abstraite peut contenir à la fois des méthodes abstraites (qui n'ont pas de corps) et des méthodes concrètes (avec une implémentation).

Voici les caractéristiques des classes abstraites :

- **Ne peut pas être instanciée** directement.
- Peut contenir des méthodes concrètes et abstraites.
- Les classes dérivées doivent obligatoirement implémenter toutes les méthodes abstraites (surveillez le rouge !).

PHP

Exemple de classe abstraite

```php
abstract class Animal {
    // Méthode abstraite (sans implémentation)
    abstract public function faireDuBruit();

    // Méthode concrète (avec implémentation)
    public function dormir() {
        echo "L'animal dort.<br>";
    }
}
```

Méthodes abstraites

Les **méthodes abstraites** sont des méthodes définies dans une classe abstraite, mais sans corps. Ces méthodes doivent obligatoirement être implémentées par toute classe enfant qui hérite de la classe abstraite. Elles représentent des comportements généraux que chaque classe enfant doit adapter selon son propre contexte.

Pourquoi utiliser une classe abstraite

Les classes abstraites sont utiles lorsque vous voulez fournir un **comportement commun** à un ensemble de classes, tout en forçant chaque classe dérivée à définir certains comportements spécifiques. Cela permet de structurer votre code et d'éviter les erreurs en garantissant que toutes les classes dérivées implémentent certaines fonctionnalités.

Cas d'utilisation courants :

- **Modèle de conception** : Une classe abstraite fournit un modèle général pour des entités qui ont des comportements partagés (comme dans notre exemple avec les animaux).
- **Encadrement de l'implémentation** : Une classe abstraite définit des méthodes obligatoires que les classes dérivées doivent implémenter, garantissant ainsi que toutes les classes filles respectent l'interface définie.

Interfaces

Les **interfaces** en PHP sont des structures qui définissent un ensemble de méthodes que les classes doivent implémenter, sans fournir d'implémentation concrète.

Elles permettent d'imposer un **contrat** aux classes qui les implémentent, garantissant ainsi que ces classes respectent une certaine structure.

Contrairement aux classes abstraites, les interfaces ne contiennent **que des méthodes abstraites** (sans corps) et ne peuvent pas avoir de propriétés ou de méthodes concrètes. Une interface sert à définir un comportement commun que plusieurs classes doivent respecter, tout en laissant à chaque classe la liberté de les implémenter à sa manière.

PHP

Définition d’une interface

Une interface est définie avec le mot-clé **interface**. Toutes les méthodes définies dans une interface sont abstraites par défaut (elles n'ont pas de corps) et doivent être implémentées dans les classes qui utilisent cette interface.

```
interface Oiseau {
    public function voler(int $km, string $direction):void;
    public function chanter():void;
}
```

Dans cet exemple, l'interface **Oiseau** déclare deux méthodes : **voler()** et **chanter()**. Toute classe qui implémente cette interface devra fournir des implémentations pour ces méthodes en respectant scrupuleusement leur **signature**.

Implémentation d’une inteface

Une classe qui implémente une interface utilise le mot-clé **implements**. Elle est alors obligée de définir **toutes les méthodes** déclarées dans l'interface, sous peine de provoquer une erreur.

```
class Canari implements Oiseau {
    public function voler(int $km, string $direction) {
        echo "Le canari vole $km kms en direction du $direction!<br>";
    }

    public function chanter() {
        echo "Le canari chante : Piou Piou !<br>";
    }
}
```

Pourquoi utiliser les interfaces

Les interfaces sont très utiles pour :

- **Standardiser le comportement** : Elles permettent de garantir que des classes différentes ont le même ensemble de méthodes, indépendamment de leur logique interne.
- **Favoriser le polymorphisme** : Elles permettent de traiter différents types d'objets de manière uniforme, sans se soucier de leur type, tant qu'ils respectent le même contrat.
- **Fournir des comportements communs** : Les interfaces sont souvent utilisées pour fournir des comportements communs à un ensemble de classes qui peuvent ne pas être liées entre elles hiérarchiquement.

Exemples d’utilisation que vous avez pu rencontrer :

- **Countable** : Fournit un moyen de compter des éléments. Toute classe qui implémente cette interface doit avoir une méthode **count()**.
- **Iterator** ou **Iterable** : Permet de rendre une classe itérable, c'est-à-dire utilisable dans des boucles comme **foreach**.
- **JsonSerializable** : Fournit une méthode **jsonSerialize()** permettant de spécifier comment un objet doit être converti en JSON.

PHP

```php
class Collection implements Countable {
    private $items = [];

    public function addItem($item) {
        $this->items[] = $item;
    }

    // Implémentation de la méthode de l'interface Countable
    public function count() {
        return count($this->items);
    }
}

$collection = new Collection();
$collection->addItem("Item 1");
$collection->addItem("Item 2");

echo count($collection);  // Affiche : 2
```

Différences entre classe abstraite et interface

Les **interfaces** et les **classes abstraites** ont des similitudes, mais elles diffèrent dans leur rôle et leur utilisation :

- Une interface ne contient **que des méthodes abstraites** (sans corps), tandis qu'une classe abstraite peut contenir des **méthodes abstraites et concrètes**.
- Une classe peut implémenter plusieurs **interfaces**, mais elle ne peut hériter que d'**une seule classe abstraite**.
- Les **interfaces** sont plus souvent utilisées pour définir un contrat strict (tous les objets qui implémentent l'interface doivent fournir les mêmes méthodes), tandis que les **classes abstraites** sont utilisées pour regrouper du code commun avec des comportements généraux.

Traits

Les **traits** en PHP sont une manière d'injecter des fonctionnalités partagées dans plusieurs classes, permettant ainsi de réutiliser du code sans les limitations de l'héritage simple (où une classe ne peut hériter que d'une seule autre classe). Les traits ont été introduits pour répondre à la nécessité de partager des méthodes entre différentes classes sans passer par l'héritage multiple, qui n'existe pas en PHP.

Les traits sont une solution flexible, car ils permettent à une classe d'incorporer du code provenant de plusieurs traits, en complément de l'héritage traditionnel. Cela permet d'organiser et de partager des fonctionnalités récurrentes dans un projet tout en gardant un design plus modulaire.

Définition d'un trait

Un **trait** est défini avec le mot-clé … **trait**. Il ressemble beaucoup à une classe, mais il ne peut pas être instancié tout seul. Il contient des méthodes (et éventuellement des propriétés) qui peuvent être utilisées dans différentes classes.

PHP

```php
trait Identifiable {
    private $id;

    public function setId($id) {
        $this->id = $id;
    }

    public function getId() {
        return $this->id;
    }
}
```

Dans cet exemple, le trait **Identifiable** définit une méthode **setId()** et une méthode **getId()** pour gérer une propriété **$id**. Ce trait pourra ensuite être utilisé dans n'importe quelle classe.

Import et utilisation d'un trait

Pour utiliser un trait dans une classe, on utilise le mot-clé **use** à l'intérieur de la définition de la classe. Cela permet à la classe d'incorporer les méthodes et les propriétés du trait comme si elles faisaient partie de la classe elle-même.

```php
class Utilisateur {
    use Identifiable;

    private $nom;

    public function setNom($nom) {
        $this->nom = $nom;
    }

    public function getNom() {
        return $this->nom;
    }
}

$utilisateur = new Utilisateur();
$utilisateur->setId(1);
$utilisateur->setNom("Alice");

echo $utilisateur->getId();  // Affiche : 1
echo $utilisateur->getNom(); // Affiche : Alice
```

Ici, la classe **Utilisateur** utilise le trait **Identifiable**. Cela lui permet d'utiliser les méthodes **setId()** et **getId()** définies dans le trait, en plus de ses propres méthodes **setNom()** et **getNom()**.

Cas d'utilisation des traits

- **Réutilisation de code** : Partager des fonctionnalités communes entre plusieurs classes sans passer par l'héritage classique.
- **Éviter la duplication** : Écrire du code une seule fois dans un trait, et l'utiliser dans plusieurs classes.
- **Flexibilité** : Un trait peut être utilisé dans n'importe quelle classe, peu importe son héritage. Une classe peut également utiliser plusieurs traits à la fois, ce qui permet de composer des fonctionnalités de manière modulaire.

Notions avancées

Les traits sont une alternative à l'**héritage multiple**, qui n'existe pas en PHP. Contrairement à l'héritage classique, où une classe ne peut hériter que d'une seule autre classe, une classe peut utiliser plusieurs **traits** en même temps. Cela permet de simuler certaines fonctionnalités de l'héritage multiple en PHP, en donnant à une classe la possibilité de combiner des fonctionnalités provenant de plusieurs sources.

```php
trait Horodatable {
    private $createdAt;

    public function setCreatedAt($date) {
        $this->createdAt = $date;
    }

    public function getCreatedAt() {
        return $this->createdAt;
    }
}

class Produit {
    use Identifiable, Horodatable;

    private $nom;

    public function setNom($nom) {
        $this->nom = $nom;
    }

    public function getNom() {
        return $this->nom;
    }
}

$produit = new Produit();
$produit->setId(101);
$produit->setCreatedAt('2024-10-17');
$produit->setNom("Laptop");

echo $produit->getId();          // Affiche : 101
echo $produit->getCreatedAt();   // Affiche : 2024-10-17
echo $produit->getNom();         // Affiche : Laptop
```

Dans cet exemple, la classe **Produit** utilise deux traits : **Identifiable** et **Horodatable**. Cela permet à la classe de combiner les fonctionnalités de ces deux traits.

Conflits possibles :

Si deux traits utilisés par une même classe contiennent des méthodes portant le même nom, un **conflit** se produit. PHP permet de résoudre ces conflits en spécifiant explicitement quelle méthode doit être utilisée, ou en donnant un alias à une méthode.

PHP

```php
trait A {
    public function faireAction() {
        echo "Action depuis Trait A<br>";
    }
}

trait B {
    public function faireAction() {
        echo "Action depuis Trait B<br>";
    }
}

class Test {
    use A, B {
        B::faireAction insteadof A;  // Utiliser la méthode de B
        A::faireAction as actionDeA; // Créer un alias pour la méthode de A
    }
}

$test = new Test();
$test->faireAction();  // Affiche : Action depuis Trait B
$test->actionDeA();    // Affiche : Action depuis Trait A
```

Dans cet exemple, la classe **Test** utilise deux traits **A** et **B**, qui ont tous les deux une méthode **faireAction()**. En utilisant la syntaxe **insteadof**, on résout le conflit en choisissant la méthode du trait **B**. En même temps, on crée un alias **actionDeA()** pour la méthode du trait **A**.

Limites

Les traits ne remplacent pas l'héritage et ne permettent pas de tout faire. Ils ont quelques limites :

- Les traits **ne peuvent pas être instanciés** comme des classes.
- Ils **ne permettent pas d'héritage multiple** au sens strict, car ils ne gèrent pas la hiérarchie des classes.
- Les traits ne peuvent pas avoir de **constructeurs**.

PHP

Exercices :

Exercice 1 :

Créer un fichier **test_objet.php** qui va nous servir de fichier d'exécution,

Créer une nouvelle classe Maison dans un fichier **Maison.php** qui va contenir les attributs suivants :

-nom, longueur, largeur.

Instancier une nouvelle maison dans le fichier **test_objet.php** avec les valeurs de votre choix (**nom**, **longueur** et **largeur**),

-Créer une méthode **surface** qui calcule et affiche la superficie de la maison (**longueur * largeur**) dans la **classe** Maison.

-Appeler la méthode **surface** et **afficher** sous la forme suivante le résultat :

*" la surface de **nomMaison** est égale à : **x** m2 ".*

Bonus

Ajouter un attribut **nbrEtage** à la classe Maison,

Modifier la méthode **surface** pour qu'elle prenne en compte le paramètre **nbrEtage**.

Exercice 2 :

-Créer un fichier **Vehicule.php** qui va contenir la classe,

-Dans ce fichier recréer la classe Vehicule comme dans le cours (**attributs** et **méthodes**),

-Créer un fichier **test_objet.php** au même niveau que vehicule.php,

-Appeler avec **require()** ou **include()** le fichier de la classe **Vehicule**,

-Instancier 2 nouveaux **Vehicules** dans le fichier **test_objet.php** avec les paramètres suivants :

-Objet **voiture** (nomVehicule = « Mercedes CLK », nbrRoue = 4, vitesse 250),

-Objet **moto** (nomVehicule = « Honda CBR », nbrRoue = 2, vitesse = 280),

-Créer une fonction **detect()** qui détecte si le véhicule est une moto ou une voiture (la méthode retourne une **string** ***moto*** ou ***voiture*** avec **return**) dans le fichier de classe **vehicule.php**,

-Exécuter la méthode **detect()** sur les 2 objets voiture et moto dans le fichier **test_objet.php**.

-Afficher le **type de Vehicule** dans le fichier **test_objet.php**,

-Créer une méthode **boost** qui ajoute **50** à la **vitesse** d'un objet dans le fichier de classe **Vehicule.php**,

-Appliquer la méthode **boost** a la voiture dans le fichier **test_objet.php**,

PHP

-Afficher la nouvelle **vitesse** de la voiture dans le fichier **test_objet.php**.

Bonus :

-Créer une méthode **plusRapide()** dans le fichier **vehicule.php** qui compare la vitesse des différents véhicules (*moto* et *voiture*) et retourne le Vehicule le plus rapide des 2 avec un **return**.

-Exécuter la méthode **plusRapide()** dans le fichier **test_objet.php**.

-Afficher le **Vehicule** le plus rapide dans le fichier **test_objet.php.**

PHP

20 Modèle MVC

Théorie Modèle MVC

Le MVC est un motif d'architecture logicielle visant à séparer les principales fonctionnalités d'un site web en 3 parties :

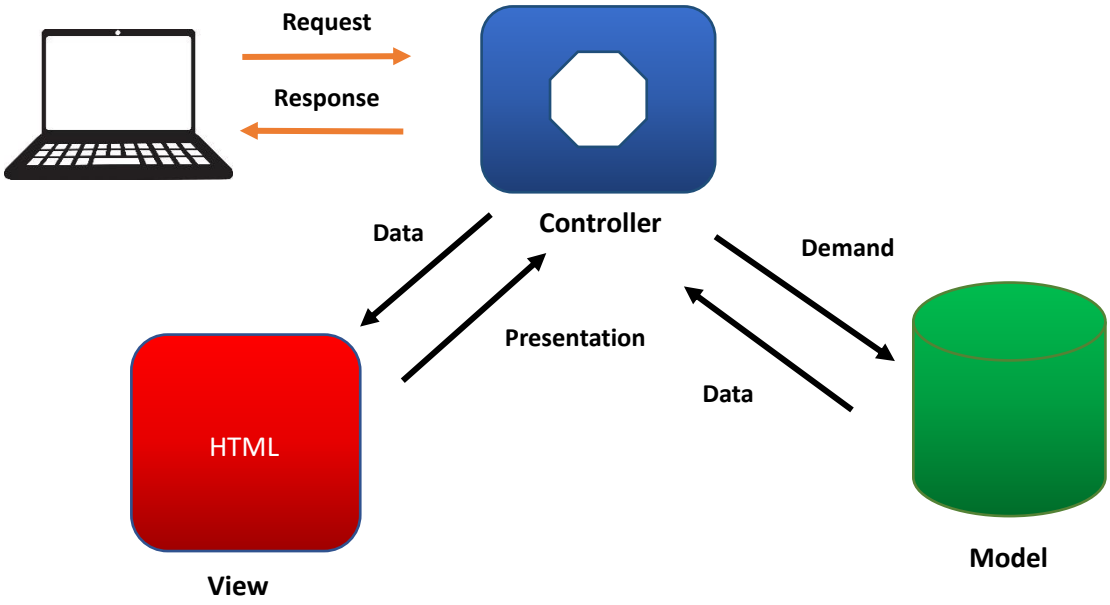
- **Modèle (Model)** : Le modèle est la partie de notre code qui aura en charge la gestion des données et de leur persistance en base de données.
  C'est cette partie qui aura notamment en charge le CRUD (Create, Read, Update, Delete) des principaux objets de notre application.
  La seule « logique » gérée par cette partie sera la logique concernant les données (hydratation des objets, validation des données, ...).
  Ce dépôt github montre ce que l'on peut attendre d'une couche Modèle : https://github.com/guirod/simple-auth-poc
  La branche la plus aboutie est la branche « oo_autoload ». Vous devrez donc la « checkout ».

- **Vue (View)** : La vue est tout ce qui concernera la partie affichage de notre application. Elle sera composée de fichiers PHP contenant principalement du code HTML. La logique dans cette partie y est minimale. Bien souvent, seulement quelques boucles pour itérer sur nos listes d'objets afin d'afficher les tables, et quelques conditions dans le cas où une portion de page ne serait visible que sous certaines conditions.

- **Contrôleur (Controller)** : Vous l'avez compris, c'est ici que se trouvera la majeure partie de la logique métier de l'application. Le contrôleur est le chef d'orchestre de l'application. Il intercepter les requêtes de l'utilisateur, interroge la couche Modèle pour récupérer les données dont il a besoin pour travailler.
  Il utilise ensuite ces données pour effectuer les traitements métier nécessaires, puis retourne à l'utilisateur la « Vue » demandée.

PHP

Echange entre les différentes couches :



Pour intégrer notre code nous allons avoir besoin d'utiliser les méthodes PHP ***include() ou require()***.

L'utilisation d'une architecture MVC présente de nombreux avantages :

- Limite la dépendance entre la gestion des données et le métier de l'application. Ainsi, si par exemple nous devions changer de Système de Gestion de Base de Données (passage de MySQL à PostgreSQL par exemple), il suffirait de mettre à jour la partie Modèle de l'application.
- Les composants développés sont plus minimalistes et dédiés à une tache précise. Cela facilite grandement la maintenance de l'application, le travail collaboratif, et la possibilité de mettre une politique de tests unitaires rigoureuse.
- Cette architecture s'intègre parfaitement à la Programmation Orientée Objet et a été reprise par les principaux frameworks.

PHP

Modèle

Les classes composant notre Modèle seront généralement calquées sur les tables de notre Base de Données.

Exemple de modèle d'un simple objet « User ».

```php
<?php

namespace Model;

use Model\ICrud;
use Model\Connexion;
use PDO;
use PDOException;

class User implements ICrud
{
    private int $id;
    private string $login;
    private string $passwordHash;
    private ?array $groups = null;


    /*-------------------------------------------------------------------------------------*/
    /*----------------Les Getters et les Setters-------------------------------------------*/
    /*------Ils nous permettent d'accéder aux propriétés de l'objet, ----------------------*/
    /*------en lecture (getters) ou écriture (setters)--------------------------------------*/
    /*-------------------------------------------------------------------------------------*/
    public function getId(): int
    {
        return $this->id;
    }

    public function setId(int $id): User
    {
        $this->id = $id;

        return $this;
    }

    public function getLogin(): string
    {
        return $this->login;
    }




    public function setLogin(string $login): User
    {
        if (!empty($login)) {
            $this->login = $login;
        }

        return $this;
    }

    public function getPasswordHash(): string
    {
        return $this->passwordHash;
    }
```

PHP

```php
    /**
     * BCrypt hash is 60 chars length
     */
    public function setPasswordHash(string $passwordHash): User
    {
        if (strlen($passwordHash) === 60) {
            $this->passwordHash = $passwordHash;
        }

        return $this;
    }

    public function addGroup(Group $group): User
    {
        $this->groups[] = $group;

        return $this;
    }

    public function removeGroup(Group $group): User
    {
        // TODO
        return $this;
    }


    /*
    Ici, on a un getter un peu plus complexe, puisqu'il doit hydrater un objet contenu dans notre objet.
    Il va donc devoir aller récupérer des informations en base de données
    On utilise ici le « lazy loading », on ne récupère la liste des groupes en DB que si on nous les demande. De plus, ça fonctionne comme un genre de cache, si on devait mettre à jour les groupes, il faudrait les repasser à null pour « rafraichir » le cache, sinon on aura toujours l'ancienne liste.
     */
    public function getGroups(): array
    {
        if ($this->groups === null) {
            $conn = Connexion::getInstance()->getConn();
            $sql = "SELECT g.* FROM `groups` g
                    INNER JOIN users_groups ug ON g.id = ug.id_group
                    INNER JOIN users u ON u.id = ug.id_user
                    WHERE u.id = ?";

            $stt = $conn->prepare($sql);
            $stt->bindParam(1, $this->id, PDO::PARAM_INT);
            $stt->execute();

            while ($arrayGroup = $stt->fetch()) {
                $this->addGroup(Group::hydrate($arrayGroup));
            }
        }

        return $this->groups;
    }
```

PHP

```php
    /**
     * Cette méthode permet de sauvegarder l'objet en base de données.
     */
    public function save()
    {
        try {
            $conn = Connexion::getInstance()->getConn();
            $stt = $conn->prepare("INSERT INTO `users` (`login`,password_hash) VALUES (?,?)");
            $stt->bindParam(1, $this->login);
            $stt->bindParam(2, $this->passwordHash);
            $stt->execute();
            $this->id = $conn->lastInsertId();

            // Add the user in the member group & insert a users_groups entry
            // Get member group :
            $memberGroup = Group::findByName('Member');
            $userGroup = new UserGroup($this->id, $memberGroup->getId());
            $userGroup->save();
        } catch (PDOException $e) {
            echo $e->getMessage();
        }
    }

    /**
     * Hydrater un objet signifie créer un objet côté PHP à partir d'un enregistrement SQL.
     */
    public static function hydrate(array $properties): User
    {
        $user = new User();
        $user
            ->setId($properties['id'])
            ->setLogin($properties['login'])
            ->setPasswordHash($properties['password_hash']);

        return $user;
    }

    /**
     * Cette méthode est une méthode "métier" qui nous permet de savoir si un utilisateur est admin
     */
    public function isAdmin(): bool
    {
        foreach ($this->getGroups() as $group) {
            if ($group->getName() === Group::GROUP_ADMIN) {
                return true;
            }
        }

        return false;
    }
}
```

PHP

Vue

Une vue est beaucoup plus simple. Elle utilisera quelques variables pour adapter la page web à la requête envoyée par l'utilisateur.

Exemple simple de Vue

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>User List</title>
</head>

<body>
    <h1>User List</h1>
    <ul>
        <?php foreach ($users as $user): ?>
            <li>
                <?= $user->name ?> (
                <?= $user->email ?>)
            </li>
        <?php endforeach; ?>
    </ul>
</body>
</html>
```

PHP

Contrôleur

En tant que chef d'orchestre, le contrôleur a la responsabilité de la majorité des fonctionnalités de notre site.

Généralement, on aura un Controller par entité (élément de notre modèle), qui sera en charge des traitements relatifs à cette entité, notamment le CRUD de l'entité.

Les méthodes du Controller sont globalement les différentes routes de l'application.

Exemple simple de Contrôleur

```php
<?php

namespace Guirod\SimpleMvc\Controllers;

use Guirod\SimpleMvc\Controller;
use Guirod\SimpleMvc\Models\User;

class UserController extends Controller
{
    //Cette fonction va afficher la liste des utilisateurs
    public function index()
    {
        // Ici, on construit une liste d'Users factice. En temps normal, on demanderai au Modèle de nous retourner la liste des utilisateurs.
        $users = [
            new User('John Doe', 'john@example.com'),
            new User('Jane Doe', 'jane@example.com')
        ];

        //La plupart des méthodes des controllers vont renvoyer la vue demandée
        $this->render('user/index', ['users' => $users]);
    }
}
```

PHP

Exemple :

Reprenons l'exercice 1 du chapitre précédent, nous allons restructurer et découper le code de cette manière :

Toute la partie **html** (*notre formulaire*) va être déplacer dans un nouveau fichier que nous allons appeler **vue_article.php** comme ci-dessous :

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ajouter un article</title>
</head>
<body>
    <form action="" method="post">
        <p>saisir le nom de l'article :</p>
        <input type="text" name="nom_article">
        <p>saisir le contenu de l'article :</p>
        <input type="text" name="contenu_article">
        <input type="submit" value="Ajouter">
    </form>
</body>
</html>
```

PHP

Toute notre **partie PHP** (*requête **SQL***) va être déplacer dans un nouveau fichier que nous allons nommer **model_article.php**. Comme ci-dessous :

```
<?php
    function addArticle($bdd, $name, $content){
        try
        {
            //Exécution de la requête SQL insert
            $req = $bdd->prepare('INSERT INTO article(nom_article, contenu_article)
            VALUES(?,?)');
            $req->bindParam(1, $name, PDO::PARAM_STR);
            $req->bindParam(2, $content, PDO::PARAM_STR);
            return "ajout de l'article : $name qui a comme contenu : $content";
        }
        catch(Exception $e)
        {
            //affichage d'une exception en cas d’erreur
            die('Erreur : '.$e->getMessage());
        }
    }
           addArticle($bdd, $name, $content) ;
?>
```

PHP

Afin de réutiliser la connexion à la base de données dans l'ensemble de notre code nous allons déplacer la connexion dans un nouveau fichier que nous allons nommer **connect.php** comme ci-dessous :

```
<?php
//connexion à la bdd
$bdd = new PDO('mysql:host=localhost;dbname=articles', 'root','',
[PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION)];
?>
```

Enfin nous allons déplacer la logique (*conditions*) dans une nouvelle page (*qui sera notre **controller***) que nous allons nommer **controller_article.php**. Comme ci-dessous :

```
<?php
//ajout de la vue
include('vue_article.php');
//connexion à la BDD
include('connect.php');
//test existence des champs nom_article et contenu article
if(isset($_POST['nom_article']) and isset($_POST['contenu_article']))
{
//création des 2 variables qui vont récupérer le contenu des super globales POST
$name = $_POST['nom_article'];
$content = $_POST['contenu_article'];
//ajout du model
include('model_article.php.');
}
else{
//affichage dans la page html de ce que l'on a enregistré en bdd
echo '<p>veuillez remplir les champs de formulaire</p>';
}
?>
```

PHP

Exercices :

Exercice 1 :

Reprendre l'exercice 1 de la partie précédente et l'adapter en MVC (se servir de l'exemple du cours) et remplacer la partie **model** par la requête **préparée**.

Intégrer la partie bonus (affichage de l'article ajouté dans un paragraphe).

Exercice 2 :

Reprendre l'exercice 2 de la partie précédente et l'adapter en MVC.

Exercice 3 :

Reprendre l'exercice 3 de la partie précédente et l'adapter en MVC.

PHP

21 Tests Unitaires

Un test unitaire (TU ou UT en anglais) est une procédure permettant de tester le bon fonctionnement d'une partie précise d'un logiciel. En PHP, cela revient très souvent à tester le bon fonctionnement d'une fonction ou méthode.

Les tests unitaires devraient idéalement être développés en même temps que vous développez votre fonctionnalité, et tester les différents cas qui sont censés survenir au cours de l'exécution de votre fonction ou méthode.

A la différence d'un test fonctionnel ou d'un test d'intégration, un test unitaire doit être le plus simple possible et il doit isoler la portion de code qu'il teste pour ne tester que le code de la fonction visée, et non les dépendances que cette fonction pourrait utiliser (services, BDD, librairies externes) par l'utilisation de bouchons.

En PHP, nous utiliserons la plupart du temps le framework PHPUnit pour développer nos tests unitaires.

Coverage (Taux de couverture)

Afin d'estimer la robustesse de nos tests unitaires, nous voulons connaitre le « coverage » de nos tests. Le coverage est le pourcentage des lignes de code de l'application qui sont exécutées par nos tests par rapport au nombre total de lignes de code.

Vous vous rendrez rapidement compte qu'avoir un taux de couverture de 100% est quasiment impossible, mais on cherchera à s'en approcher le plus (80-90% est considéré comme acceptable).

Le principal intérêt d'avoir un bon taux de couverture de nos tests unitaires est de limiter au maximum les régressions lorsque l'on va développer de nouvelles fonctionnalités.

En effet, il arrive souvent qu'en développant de nouvelles fonctionnalités, ou en corrigeant des bugs, on entraine des régressions (que l'on casse l'existant). Les tests unitaires permettent de limiter ça, car si en développant une nouvelle fonctionnalité on se rend compte que les tests ne passent plus, c'est probablement que l'on a cassé quelque chose.

PHP

Mise en place

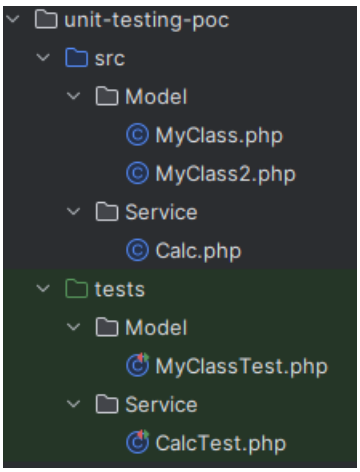
Pour une utilisation optimale de PHPUnit, il conviendra de répondre à quelques prérequis :

- Installation via Composer
- Utilisation de namespaces
- Avoir un autoload fonctionnel (le plus simple étant d'utiliser l'autoload PSR-4 fourni par Composer)

Par convention, nous placerons les tests unitaires dans un dossier « tests » à la racine du projet, dans lequel nous aurons une arborescence identique à l'arborescence de l'application.

Chaque classe de test devra hériter de la classe **TestCase** du framework PHPUnit.

Les classes de tests doivent être suffixée par « Test » et au sein de la classe, les différentes méthodes de test devront être préfixées par « test » (ex « testMaFonction() »).



Pour ensuite lancer les tests unitaires, il faut utiliser la commande suivante à la racine du projet :

```
./vendor/bin/phpunit tests
```

On peut aussi préciser un fichier particulier afin de ne pas exécuter tous les tests (ce qui peut être long en fonction du nombre de tests).

Enfin, la plupart des IDE (vscode, phpstorm) vont permettre d'exécuter les tests unitaires.

PHP

Assertions

Le cœur de nos tests unitaires sera de tester qu'une méthode retourne une valeur attendue, ou que chaque attribut d'un objet ait les bonnes valeurs.

Pour ça nous allons utiliser les assertions, qui nous permettrons de comparer les valeurs d'un objet ou d'une variable avec le résultat que nous attendons.

Il existe un grand nombre de méthodes d'assertions différentes. Je vous invite à aller voir la doc pour plus de renseignements : https://docs.phpunit.de/en/10.0/assertions.html

Quelques exemples :

- assertTrue($valeur) / assertNotTrue($valeur) / assertFalse($valeur) / assertNotFalse($valeur) : permet de comparer des valeurs booléennes
- assertSame($expected, $actual) / assertNotSame($expected, $actual) : Vérifie si des variables sont identiques (même type et même valeur).
- assertEquals($expected, $actual) / assertNotEquals($expected, $actual) : Vérifie qu'une valeur attendue est identique à la valeur reçue.

PHP

Data Providers

Lorsque nous testons une fonction ou méthode, nous allons en général vouloir tester plusieurs jeux de données pour un même test, car on va vouloir tester les cas nominaux mais aussi tous les cas limites.

Afin d'éviter de dupliquer du code et pour une meilleure lisibilité et maintenabilité, nous pouvons utiliser les dataproviders. Un dataprovider est une fonction qui va retourner un jeu de données qui seront injectées en paramètre de notre cas de test.

Le dataprovider sera déclaré par une annotation sur notre méthode de test.

Pour plus d'informations, voir la documentation : https://phpunit.de/manual/3.7/en/writing-tests-for-phpunit.html

Voici un exemple d'implémentation :

```
17      public static function providerAdd():array
18      {
19          return [
20              [0,0,0],
21              [12,10,22],
22              [-5,5,0],
23              [-53,-11,-64]
24          ];
25      }
26
27      /**
28       * @dataProvider providerAdd
29       * @return void
30       */
        guirod *
31      function testAdd($nb1,$nb2,$expected)
32      {
33          $this->assertEquals($expected, $this->calc->add($nb1,$nb2));
34      }
```

PHP

Mocks / Stubs

Comme je vous le disais au début, le test unitaire doit, comme son nom l'indique, être unitaire. C'est-à-dire qu'il ne doit tester que la méthode à laquelle il est dédié et non les méthodes ou services qui pourraient être appelés ou utilisés par cette méthode. En effet, on ne souhaite pas que notre test tombe en erreur car un service qu'il utilise a retourné une mauvaise valeur. On veut que notre test tombe en erreur uniquement quand le comportement que l'on teste lève une erreur.

Pour cela nous allons utiliser des mocks, qui sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée.

Du coup, lorsque nous testerons notre méthode faisant appel à un service externe, plutôt que tester à la fois ce service, nous définirons ce que ce service retournera à notre méthode quand il sera appelé. Nous vérifierons également que ce service a bien été appelé avec les paramètres attendus.

Voici un exemple d'implémentation de mock :

Méthode d'un modèle faisant appel à un service externe (Calc) que nous souhaiterons « mocker » :

```php
class MyClass
{
    protected int $id;
    protected string $name;

    1 usage  guirod
    public function getId(): int
    {
        return $this->id;
    }

    1 usage  guirod
    public function getName(): string{...}

    4 usages  guirod
    public function setId(int $id): MyClass{...}

    guirod
    public function setName(string $name): MyClass{...}

    guirod
    public function print(): void{...}

    1 usage  guirod
    public function calcSomething(float $nb1, float $nb2, Calc $service)
    {
        return $service->add($nb1,$nb2);
    }
}
```

Voyons maintenant comment nous allons pouvoir mocker ce service dans la méthode de test de « calcSomething » :

```php
public function testCalcSomething()
{
    $nb1 = 50;
    $nb2 = 2;

    // On créé un mock du même type que le service qu'il va remplacer
    $calcMock = $this->createMock( originalClassName: Calc::class);
    $calcMock
        ->expects($this->once())        // Le service devra être appelé 1 fois (assertion)
        ->method( constraint: 'add')        // La méthode du service qui sera appelée devra être la méthode "add" (assertion)
        ->with($this->equalTo($nb1), $this->equalTo($nb2))  // La méthode "add" devra recevoir en paramètre $nb1 et $nb2 (assertion)
        ->willReturn((float)($nb1+$nb2));   // La méthode retournera la somme de $nb1 + $nb2 (comportement simulé)

    // On appel la méthode que l'on test, les assertions sur le mock seront vérifiées à ce moment là
    $returnValue = $this->obj->calcSomething($nb1,$nb2,$calcMock);

    // On fait nos assertions standard sur la méthode que notre TU couvre
    $this->assertEquals((float)($nb1+$nb2), $returnValue);
}
```

L'utilisation des mocks est quelque chose qui peut sembler complexe au début, mais qui vous fera énormément gagner en termes de robustesse de votre code et de vos tests.

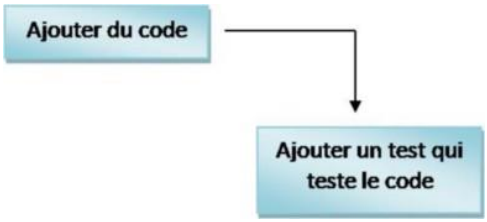
Voici la documentation concernant les mocks et stubs : https://docs.phpunit.de/en/9.6/test-doubles.html

Test Driven Development (TDD) et Extreme Programming (XP)

Enfin, je vais évoquer la méthode Test Driven Development qui repose sur un des principes Test First d'une methode de développement agile intitulée Extreme Programing appelé aussi XP.

La méthode TDD est une méthode de développement logiciel dans laquelle l'écriture des tests automatisés dirige l'écriture du code source. C'est une technique très efficace pour livrer des logiciels bien construits avec une suite de tests de non-régression.

- La méthode traditionnelle de la rédaction des tests unitaires consiste à rédiger les tests d'une portion d'un programme (appelé unité ou module) afin de vérifier la validité de l'unité implémentée.
  Voici un schéma représentant le principe de la méthode traditionnelle :

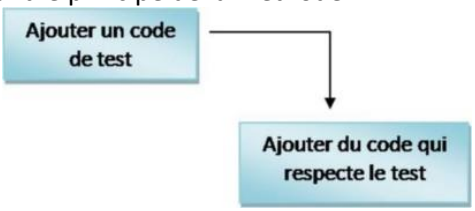


  Comme on peut le constater, le test découle du code, ce qui est contraire au principe même de TDD.

PHP

La méthode TDD quant à elle consiste à rédiger les tests unitaires avant de procéder à la phase de développement de la fonctionnalité.

Voici un schéma représentant le principe de la méthode TDD :



Ici, le développeur écrit un peu de code de test et ensuite implémente le métier.

Le code métier est soumis aux tests puis il est corrigé jusqu'a que ses tests soient validés. Le développeur procède éventuellement ensuite à une refactoriosation du code qui est un autre principe de l'Extreme Programing permettant d'améliorer la qualité interne du code en procédant à plusieurs opérations (renommage de méthodes, suppression du code inutile, ...).

Démarche à suivre pour le développement en TDD

La démarche à suivre pour mettre en place cette méthode est décomposée en trois phases appelé RGR(aussi appelé la Mantra).

Les deux premières phases sont nommées d'après la couleur de la barre de progression dans les outils de test unitaires comme JUnit(Red pour échec et Green pour réussite)

- R (Red) : écrire un code de test et les faire échouer
- G (Green) : écrire le code métier qui valide le test
- R (Refactor) : remaniement du code afin d'en améliorer la qualité

PHP

22 Patrons de conception / Design Patterns

Définition

Un design pattern est un modèle d'architecture logicielle, dont le périmètre est limité à la résolution d'une problématique d'architecture courante.

Les designs patterns servent à documenter les bonnes pratiques d'architecture basées sur l'expérience.

Les design patterns ont été formellement reconnus en 1994 à la suite de la parution du livre Design Patterns : Elements of Reusable Software, co-écrit par quatre auteurs : Gamma, Helm, Johnson et Vlissides (Gang of Four - GoF ; en français « la bande des quatre »).

Il existe 23 patrons de conception (rassurez-vous nous ne les verrons pas tous) classés dans 3 familles :

- Les patrons « créateurs » : définissent comment faire de l'instanciation et configuration de classes et objets.
- Les patrons « structuraux » : définissent comment organiser les classes d'un programme dans une structure plus large.
- Les patrons « comportementaux » : définissent comment organiser les objets pour que ceux-ci interagissent entre eux.

Dans ce cours, nous allons simplement aborder les patrons les plus couramment utilisés.

A savoir, les recruteurs aiment beaucoup vous demander de lister 2 ou 3 designs patterns.

Si vous souhaitez en savoir plus sur les design patterns : https://refactoring.guru/fr/design-patterns

Pour le code des exemples du cours, voir le repo github : https://github.com/guirod/design-patterns

N'hésitez pas à utiliser le débugguer pas-à-pas pour bien comprendre le fonctionnement du design pattern, ça vous aidera à voir comment évoluent les objets au fil du temps.

PHP

Singleton

Nous avons déjà utilisé ce patron de conception. Le singleton garantit que l'instance d'une classe n'existe qu'en un seul exemplaire tout en fournissant un point d'accès global à cette instance.

On utilise donc ce patron lorsque l'on veut contrôler l'accès à une ressource partagées (une base de données ou un fichier par exemple).

Etapes de mise en place

- Empêcher l'accès aux méthodes permettant de créer une instance de la classe que l'on souhaite passer en singleton.
  Pour ceci, nous allons simplement modifier la visibilité de ces méthodes à « private » ou d'empêcher simplement leur utilisation en les surchargeant.

  La méthode la plus évidente est bien évidemment le constructeur de notre classe, mais il faut aussi penser aux autres méthodes susceptibles de créer de nouvelles instances, telles que notamment les méthodes de clonage ainsi que les méthodes susceptibles d'altérer l'instance existante.
  En PHP, il faudra donc idéalement prendre en compte les magic méthods suivantes : __construct, __clone, __unserialize, __wakeup
- Mettre en place une méthode de création statique qui se comporte comme un constructeur. Cette méthode va en réalité vérifier si une instance de la classe existe déjà, la créer seulement si elle n'existe pas, puis la retourner.

PHP

Exemple d'implémentation (classe de connexion)

```php
<?php

namespace Guirod\DesignPatterns\Singleton;

use PDO;
use PDOException;

class Connexion
{
    const SERVER_NAME = "mysql8";
    const USERNAME = "root";
    const PASSWORD = "p@ssw0rd";
    const DB_NAME = 'mvc_tp';

    private static ?Connexion $instance = NULL;

    private ?PDO $conn = null;

    static public function getInstance(): ?Connexion
    {
        if (self::$instance === NULL) {
            try {
                self::$instance = new Connexion();
            } catch (PDOException $e) {
                echo $e->getMessage();
            }
        }

        return self::$instance;
    }

    public function getConn(): PDO
    {
        return $this->conn;
    }

    /*
     * Private Constructor
     */
    private function __construct()
    {
        $this->conn = new PDO("mysql:host=". self::SERVER_NAME .";dbname=".self::DB_NAME, self::USERNAME,
self::PASSWORD);
        $this->conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $this->conn->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
    }

    /**
     * Nous aurions simplement pu modifier la visibilité à private ici
     * @return void
     */
    public function __clone() {
        trigger_error('Cloning forbidden.', E_USER_ERROR);
    }

    // Ici, on pousse le concept du singleton jusqu'au bout. Dans un projet où il n'y a pas 50 devs qui
travaillent dessus, il ne sera pas forcément utile de surcharger ces magic methods.
    public function __wakeup(): void
    {
        trigger_error('Wakeup forbidden.', E_USER_ERROR);
    }

    public function __unserialize(array $data): void
    {
        trigger_error('Unserialize forbidden.', E_USER_ERROR);
    }
}
```

On accèdera ensuite à l'instance de notre connexion de la façon suivante :

```php
<?php
require_once 'vendor/autoload.php';

use Guirod\DesignPatterns\Singleton\Connexion;

$conn = Connexion::getInstance();
$pdo = $conn->getConn();
```

Factory / Fabrique

La fabrique est un patron de conception de création qui définit une interface pour créer des objets dans une classe mère, tout en déléguant le choix des types d'objets à créer aux sous-classes.

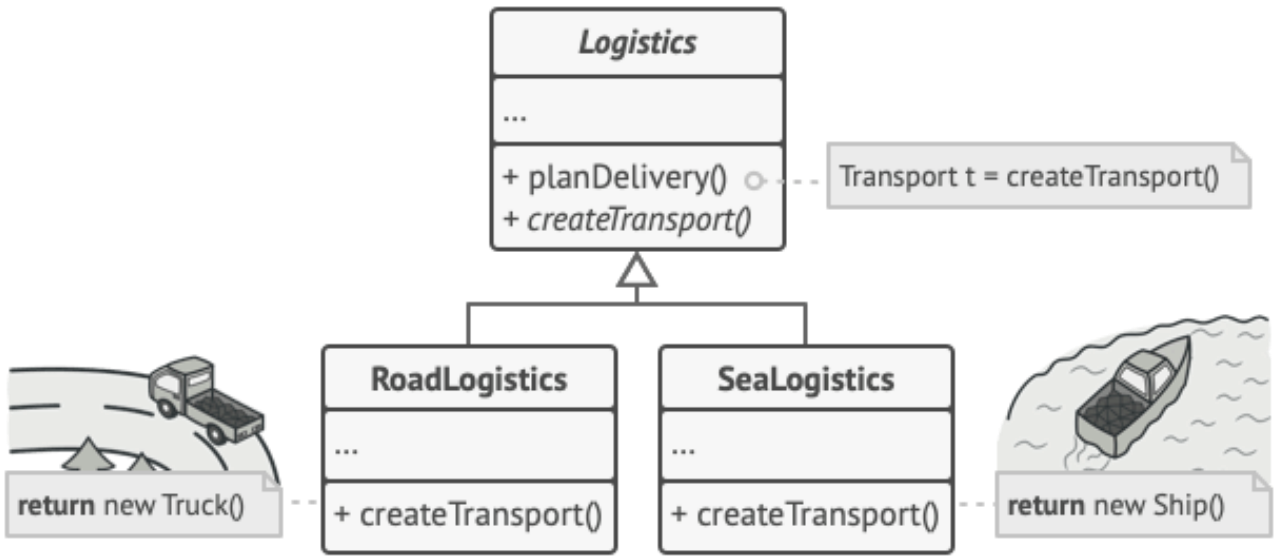
L'intérêt principal de ce patron est d'alléger aux maximum la dépendance du code par rapport aux objets créés.

Imaginons une entreprise de transport qui livre ses colis par camion. Le code métier sera donc principalement dans la classe camion et nos entités utilisées seront des camions.

Toutefois, si à terme nous ajoutons un nouveau moyen de transport (bateau, avion, …) il serait compliqué de switcher entre les différents moyens de transports du fait de la dépendance forte à la classe Camion.

La Fabrique nous propose de remplacer les appels aux constructeurs par une méthode de création d'objet en fonction d'un paramètre : ici le type de transport.

Nous pouvons donc imaginer avoir maintenant une classe mère « Logistics » qui proposera de créer le véhicule souhaité.



PHP

Exemple d'implémentation

Classe LogisticsFactory.php

```php
<?php

namespace Guirod\DesignPatterns\Factory\V1;

abstract class LogisticsFactory
{
    // Ici, nous avons 2 possibilités
    // - définir la classe et la méthode "createTransport" abstract. Auquel cas nous déléguons en totalité la création du transport aux sous-classes. Dans ce cas, le paramètre "type" sera inutile
    // - Utiliser un paramètre "type" pour permettre à la fabrique de créer l'objet correspondant.
    abstract public static function createTransport(): LogisticsFactory;

    // Ici, nous aurons les méthodes partagées par les différents moyens de transport.
}
```

Classe SeaLogistics.php

```php
<?php

namespace Guirod\DesignPatterns\Factory\V1;

class SeaLogistics extends LogisticsFactory
{
    public static function createTransport(): LogisticsFactory
    {
        return new SeaLogistics();
    }
}
```

Classe RoadLogistics.php

```php
<?php

namespace Guirod\DesignPatterns\Factory\V1;

class RoadLogistics extends LogisticsFactory
{
    public static function createTransport(): LogisticsFactory
    {
        return new RoadLogistics();
    }
}
```

Fichier factory.php

```php
<?php
require_once 'vendor/autoload.php';

use Guirod\DesignPatterns\Factory\V1\RoadLogistics as RoadLogisticsV1;
use Guirod\DesignPatterns\Factory\V1\SeaLogistics as SeaLogisticsV1;
use Guirod\DesignPatterns\Factory\V2\LogisticsFactory as LogisticsV2;

//Depending on the purpose, create Road or Sea logistics
$logistics = RoadLogisticsV1::createTransport();
$logistics = SeaLogisticsV1::createTransport();

//2nd type de factory
$logistics = LogisticsV2::createTransport(LogisticsV2::TYPE_ROAD);
$logistics = LogisticsV2::createTransport(LogisticsV2::TYPE_SEA);
```

PHP

Decorator / Décorateur

Le design pattern Décorateur est un patron de conception structurel qui permet d'affecter de nouvelles fonctionnalités à un objet existant, sans le modifier lui-même.

Pour ça, on placera, à la manière des poupées gigognes, l'objet dans un nouveau conteneur qui ajoutera des fonctionnalités.

Le décorateur devra être du même type que l'objet qu'il décore (il devra donc implémenter la même interface).

Exemple d'implémentation

Imaginons que nous ayons un composant « Notifier » qui soit chargé d'envoyer des notifications par email. Nous souhaitons que nos utilisateurs puissent aussi être notifiés par d'autre mediums (par exemple : SMS, Facebook Messenger et Slack).

Nous allons créer des Décorateurs de notre composant qui seront chargé d'envoyer les notifications.

NotifierInterface.php

```php
<?php

namespace Guirod\DesignPatterns\Decorator;

interface NotifierInterface
{
    public function sendNotification();
}
```

Notifier.php (notifier original)

```php
<?php
namespace Guirod\DesignPatterns\Decorator;

class Notifier implements NotifierInterface
{
    public function sendNotification()
    {
        echo "Send Mail<br/>";
    }
}
```

PHP

NotifierDecorator.php (Décorateur parent qui doit implémenter l'interface de l'objet à décorer)

```php
<?php
namespace Guirod\DesignPatterns\Decorator;

abstract class NotifierDecorator implements NotifierInterface
{
    // Composant que l'on décore
    protected NotifierInterface $component;

    public function __construct(NotifierInterface $component)
    {
        $this->component = $component;
    }

    public function sendNotification()
    {
        // On appelle la méthode du composant décoré (l'envoi par mail)
        $this->component->sendNotification();
    }
}
```

FacebookNotifierDecorator.php (notifier Messenger)

```php
<?php
namespace Guirod\DesignPatterns\Decorator;

class FacebookNotifierDecorator extends NotifierDecorator
{
    public function sendNotification()
    {
        // On appelle la méthode du parent
        parent::sendNotification();
        // Puis on implémente la méthode de notification propre à l'envoi par messenger
        echo "Send Messenger<br/>";
    }
}
```

PHP

SlackNotifierDecorator.php (notifier Slack)

```php
<?php
namespace Guirod\DesignPatterns\Decorator;

class SlackNotifierDecorator extends NotifierDecorator
{
    public function sendNotification()
    {
        // On appelle la méthode du parent
        parent::sendNotification();
        // Puis on implément la méthode de notification propre à l'envoi par slack
        // ici ...
        echo "Send Slack<br/>";
    }
}
```

SMSNotifierDecorator.php (notifier SMS)

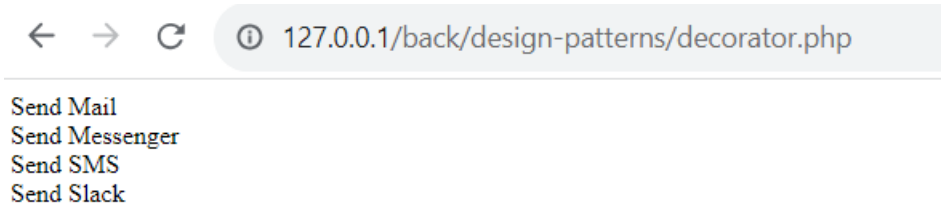
```php
<?php
namespace Guirod\DesignPatterns\Decorator;

class SMSNotifierDecorator extends NotifierDecorator
{
    public function sendNotification()
    {
        // On appelle la méthode du parent
        parent::sendNotification();
        // Puis on implément la méthode de notification propre à l'envoi par sms
        // ici ...
        echo "Send SMS<br/>";
    }
}
```

PHP

decorator.php (script de test)

```php
<?php
require_once 'vendor/autoload.php';

use Guirod\DesignPatterns\Decorator\Notifier;
use Guirod\DesignPatterns\Decorator\FacebookNotifierDecorator;
use Guirod\DesignPatterns\Decorator\SMSNotifierDecorator;
use Guirod\DesignPatterns\Decorator\SlackNotifierDecorator;

// Création de notre notifier de base
$notifier = new Notifier();

// Ici bien sur on récupèrera plutôt la config de l'utilisateur dans son compte, mais pour le test on assigne directement les variables booléennes.
$messengerNotifEnabled = true;
$smsNotifEnabled = true;
$slackEnabled = true;

if ($messengerNotifEnabled) {
    $notifier = new FacebookNotifierDecorator($notifier);
}

if ($smsNotifEnabled) {
    $notifier = new SMSNotifierDecorator($notifier);
}

if ($slackEnabled) {
    $notifier = new SlackNotifierDecorator($notifier);
}

// Envoi des messages
$notifier->sendNotification();
```

Résultat



PHP

Observer / Observateur

L'Observateur est un patron de conception comportemental qui permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.

Les objets observés sont nommés « Subjects » et les objets écoutant les événements des « Observers ».

Une classe Subject devra pouvoir gérer la liste des observers qui lui sont rattachés (méthodes pour ajouter et supprimer des observers).

En PHP, il existe des interfaces natives pour ces types d'objets : SplObserver et SplSubject.

Ce pattern est couramment utilisé dans les frameworks modernes, notamment pour gérer le logging ou les notifications à l'exécution de certains événements.

Exemple d'implémentation

Dans l'exemple suivant, nous allons justement voir comment mettre en place un système de logging et notification lorsque une classe Repository effectue des modifications en base de données.

Classe UserRepository : notre Subject

```php
<?php

namespace Guirod\DesignPatterns\Observer;

use SplObserver;
use SplSubject;

/**
 * Une classe repository est une classe gérant les ajouts/suppression/modification en BDD d'un model.
 * C'est le subject du design pattern Observer étudié.
 * En effet, de nombreux objets peuvent vouloir suivre les modifications qui peuvent avoir lieu
(exemple : logger)
 */
class UserRepository implements SplSubject
{
    /** @var User[]  : La liste des utilisateurs */
    private array $users;

    /** @var array : la liste des observers de notre subject */
    private array $observers;

    public function __construct()
    {
        // On créé un groupe d'événements pour les observers souhaitant écouter tous les évents.
        $this->observers["*"] = [];
    }

    private function initEventGroup(string $event = "*"): void
    {
        if (!isset($this->observers[$event])) {
            $this->observers[$event] = [];
        }
    }
```

PHP

```php
    private function getEventObservers(string $event = "*"): array
    {
        $this->initEventGroup($event);
        $group = $this->observers[$event];
        $all = $this->observers["*"];

        return array_merge($group, $all);
    }

    public function attach(SplObserver $observer, string $event = "*"): void
    {
        $this->initEventGroup($event);
        $this->observers[$event][] = $observer;
    }

    public function detach(SplObserver $observer, string $event = "*"): void
    {
        foreach ($this->getEventObservers($event) as $key => $s) {
            if ($s === $observer) {
                unset($this->observers[$event][$key]);
            }
        }
    }

    public function notify(string $event = "*", $data = null): void
    {
        echo "UserRepository: Broadcasting the '$event' event.<br>";
        foreach ($this->getEventObservers($event) as $observer) {
            $observer->update($this, $event, $data);
        }
    }

    // Méthodes « métier »
    public function initialize($filename): void
    {
        echo "UserRepository: Loading user records from a file.<br>";
        // ...
        $this->notify("users:init", $filename);
    }

    public function createUser(array $data): User
    {
        echo "UserRepository: Creating a user.<br>";

        $user = new User();
        $user->update($data);

        $id = bin2hex(openssl_random_pseudo_bytes(16));
        $user->update(["id" => $id]);
        $this->users[$id] = $user;

        $this->notify("users:created", $user);

        return $user;
    }
```

```php
    public function updateUser(User $user, array $data): ?User
    {
        echo "UserRepository: Updating a user.<br>";

        $id = $user->attributes["id"];

        if (!isset($this->users[$id])) {
            return null;
        }

        $user = $this->users[$id];
        $user->update($data);

        $this->notify("users:updated", $user);

        return $user;
    }

    public function deleteUser(User $user): void
    {
        echo "UserRepository: Deleting a user.<br>";

        $id = $user->attributes["id"];
        if (!isset($this->users[$id])) {
            return;
        }

        unset($this->users[$id]);

        $this->notify("users:deleted", $user);
    }
}
```

Classe User : le model (très basique ici)

```php
<?php

namespace Guirod\DesignPatterns\Observer;

/*
 * La classe User est très simple, car ce n'est pas l'objet de l'étude.
 */
class User
{
    public array $attributes = [];

    public function update($data): void
    {
        $this->attributes = array_merge($this->attributes, $data);
    }
}
```

PHP

Classe Logger : un de nos observers

```php
<?php

namespace Guirod\DesignPatterns\Observer;

use SplObserver;
use SplSubject;

class Logger implements SplObserver
{
    private string $filename;

    public function __construct(string $filename)
    {
        $this->filename = $filename;
        if (file_exists($this->filename)) {
            unlink($this->filename);
        }
    }
    public function update(SplSubject $subject, string $event = null, $data = null): void
    {
        $entry = date("Y-m-d H:i:s") . ": '$event' with data '" . json_encode($data) .
"\n";
        file_put_contents($this->filename, $entry, FILE_APPEND);

        echo "Logger: I've written '$event' entry to the log.<br/>";
    }
}
```

Classe OnboardingNotification : Un autre observer

```php
<?php

namespace Guirod\DesignPatterns\Observer;

use SplObserver;
use SplSubject;

class OnboardingNotification implements SplObserver
{
    private string $adminEmail;

    public function __construct(string $adminEmail)
    {
        $this->adminEmail = $adminEmail;
    }

    public function update(SplSubject $subject, string $event = null, $data = null): void
    {
        // mail($this->adminEmail,
        //      "Onboarding required",
        //      "We have a new user. Here's his info: " .json_encode($data));

        echo "OnboardingNotification: The notification has been emailed!<br>";
    }
}
```

Le script de test : observer.php

```php
<?php
require_once 'vendor/autoload.php';

use Guirod\DesignPatterns\Observer\Logger;
use Guirod\DesignPatterns\Observer\OnboardingNotification;
use Guirod\DesignPatterns\Observer\UserRepository;

$repository = new UserRepository();
// On inscrit le logger à tous les events du repository
$repository->attach(new Logger(__DIR__ . "/log.txt"), "*");

//On inscrit le notifier seulement aux events user:created et users:updates
$repository->attach(new OnboardingNotification("1@example.com"), "users:created");
$repository->attach(new OnboardingNotification("1@example.com"), "users:updated");

$repository->initialize(__DIR__ . "/users.csv");

$user = $repository->createUser([
    "name" => "John Smith",
    "email" => "john99@example.com",
]);

$repository->updateUser($user, [
    "name" => "John Doe",
    "email" => "john@doe.com",
]);

$repository->deleteUser($user);
```

Inversion Of Control (IOC) / Injection de dépendances

L'inversion de contrôle peut également être considérée comme un patron de conception. L'IOC est un terme générique, et sa représentation la plus connue est l'Injection de dépendance.

Le but premier de l'injection de dépendance est de découpler au maximum les dépendances entre les différentes classes d'un projet afin d'améliorer ses possibilités d'évolution et sa maintenabilité.

Ainsi, plutôt que laisser la classe ayant des dépendances avec des services ou composants initialiser elle-même ces composants, les composants seront injectés dynamiquement à l'exécution.

Imaginons par exemple un fonctionnement classique, sans injection de dépendances :

- L'application nécessite une classe Foo (par exemple un contrôleur)
- L'application créé une instance de Foo
- L'application appelle Foo
    - Foo nécessite une classe Bar (par exemple un service)
    - Foo créé une instance de Bar
    - Foo appelle Bar
        - Bar nécessite une classe Bim
        - Bar créé Bim
        - Bar appelle Bim

En utilisant l'injection de dépendances, nous procéderions de la façon suivante :

- L'application nécessite Foo, qui nécessite Bar, qui nécessite Bim
- L'application créé Bim
- L'application créé Bar et lui injecte Bim
- L'application créé Foo et lui injecte Bar
- L'application appelle Foo
    - Foo appelle Bar
        - Bar appelle Bim

Ce fonctionnement permet de remplacer les dépendances de façon plus souple et évolutive (par exemple, dans le cas d'un changement de driver de bases de données, changement de logger).

De plus, les frameworks modernes, tels que Symfony, fonctionnent avec un système de Container. C'est le container qui sera chargé de fournir les différents services à l'application. Ainsi, l'application n'aura plus à gérer elle-même l'instanciation des différents services, elle les demandera simplement au Container qui s'occupera de la création ou de la récupération des instances.