Module de PHP 7

Support de cours complet

# Table des matières

[**Table des matières**](#_shqwg6ppqd65) **1**

**Présentation**

[**Qu’est-ce que PHP ?**](#_wju3mmneaxl4) **5**

[Définition](#_lzsig45kkc17) 5

[Petit historique](#_ysilqyyonf9v) 5

[**À quoi cela ressemble-t-il ?**](#_pepwemom6z1q) **6**

[**À quoi cela sert-il dans le monde du développement Web ?**](#_f9epq9lsf5i) **6**

**Partie 1 - Mise en place de l’environnement**

**Partie 2 - Le langage**

[**La syntaxe de base**](#_d05j9phjsfza) **10**

[Introduction](#_4v64p2pt8k5x) 10

[Commentaires](#_tgd00edg55ag) 10

[**Les instructions de base**](#_vrxkrkoz1ur6) **11**

[Déclarer / affecter une variable](#_n7bnj73q1fmn) 11

[Les opérateurs](#_39g20m1pfemh) 11

[Arithmétiques](#_ckogz4ypi2py) 12

[De comparaison](#_gn0djo56wzng) 12

[Logiques](#_4f5d2y5pigy3) 13

[Aparté sur le typage en PHP](#_1q2jmx27pmlo) 14

[Les chaînes de caractères](#_ckogz4ypi2py) 14

[Différence simple et double quote](#_dy8vim40hkyx) 14

[Concaténation](#_dy8vim40hkyx) 15

[Fonctions](#_dy8vim40hkyx) 15

[Aparté sur les fonctions](#_o21enw7mdufr) 15

[Les dates](#_shr0gw5a9ssy) 16

[Introduction](#_xslpol77q4hx) 16

[Fonctions](#_klhbat4kpkao) 16

[Aparté sur les caractères d'échappement](#_vexfs5lnso85) 17

[Les tableaux](#_b9xrtrbmn1hu) 18

[Le cas simple](#_w40x052seo9b) 18

[Le cas des tableaux associatifs](#_aydmh1mqq1ox) 19

[Le cas des tableaux multi-dimensionnels](#_fzsheui51i9n) 19

[Le cas particulier des chaînes de caractères](#_dpo0x9mzwha5) 20

[Les constantes du langage](#_egay2tbgvprz) 20

[**Les structures**](#_q4pux7cqb0mf) **21**

[Conditions](#_b2vv804mr9jd) 21

[If else](#_vmx4482oqwu5) 21

[Aparté sur l’opérateur ternaire](#_oulgec49ya24) 22

[Switch](#_z9pz2srjcgfx) 22

[Boucles](#_b2vv804mr9jd) 23

[for](#_9qgqddgzntki) 23

[Aparté sur les “abréviations”](#_e41k65q0mb2g) 24

[while](#_9qgqddgzntki) 24

[do while](#_9qgqddgzntki) 24

[foreach](#_l8wr9kbel8k8) 25

[**Les fonctions**](#_d54jy6gue3q9) **26**

[Appel de fonction](#_y82hfce51agi) 26

[Déclaration de fonction](#_y82hfce51agi) 26

[Passage d’arguments](#_y82hfce51agi) 27

[Aparté sur la portée des variables](#_gangolh1opa5) 27

[**EXERCICE**](#_wwwehti09q4) **30**

[Aparté sur les fonctions de tri](#_hkpm45z4sk2w) 30

**Partie 3 - Utiliser le langage**

[**Découper son code**](#_1k311fqip91s) **32**

[**Interagir avec la base de données**](#_6r61cim6ihho) **33**

[Aparté sur PHP Objet](#_n4l4mu5q8l1p) 33

[mysqli](#_2gwenfhhhl4r) 33

[La connexion](#_r0gobmtri27u) 34

[Les requêtes simples](#_ptls90tlnv1o) 34

[Aparté sur les Injections SQL](#_j0wo3d38ivap) 34

[Parcourir les résultats](#_vfx0no1mpvi0) 34

[Les requêtes préparées](#_67dh8j42dk6) 35

[PDO](#_7n0jjww23l1j) 36

[La connexion](#_1ci9fqgeqwsv) 36

[Les requêtes simples](#_cq193aqs93zu) 36

[Parcourir les résultats](#_gfw2bm4c46g3) 37

[Les requêtes préparées](#_y7o2gupmdzb3) 37

[**Récupérer des données de requêtes (les superglobales)**](#_emepoh1v4m2z) **39**

[Aparté sur les requêtes HTTP](#_xzlgc0s57hma) 39

[GET](#_9idfbb8az71f) 39

[Aparté sur la confiance utilisateur](#_n0ul91eft7ld) 39

[POST](#_9idfbb8az71f) 40

[FILE](#_9idfbb8az71f) 40

[SESSION](#_9idfbb8az71f) 40

[COOKIE](#_9idfbb8az71f) 40

[Les autres](#_9idfbb8az71f) 41

[**Interagir avec le système de fichiers**](#_btgjsesy1v4h) **42**

[Les fonctions file\_\*](#_muvejvlqa68u) 42

[Les fonctions f\*](#_13j6vzwcljqi) 42

[Les autres fonctions](#_ex9156d5lxk) 43

[**Échanger des informations**](#_rvczongilzda) **44**

[Formater des données](#_e7mugf6umhnu) 44

[Le JSON](#_ewf8dy8o6zob) 44

[Communiquer](#_e7mugf6umhnu) 45

[Les emails](#_ue6jidmeftvn) 45

[Aparté sur le RGPD](#_tf5wwqrjnmsu) 46

[**Liens utiles**](#_1dne7v73i2fk) **47**

[**Glossaire**](#_et628m87uiob) **48**

Présentation

# Qu’est-ce que PHP ?

## Définition

*“PHP est un langage de script exécuté côté serveur,*

*ainsi que son interpréteur.”*

**Un langage**

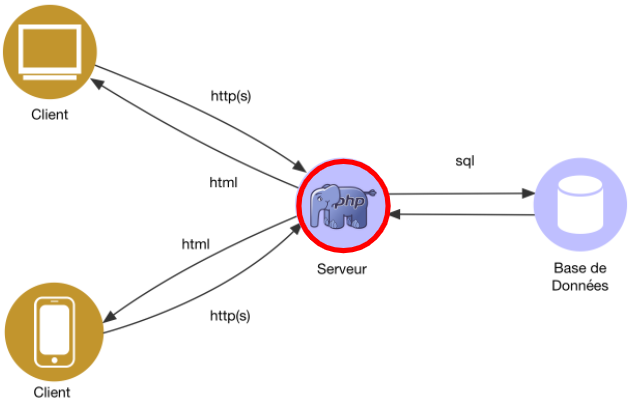
PHP est un langage, donc un outil de communication. Il sert à se faire comprendre d’une machine.

**De script**

PHP n’est pas un programme, c’est un script. La principale différence réside dans le fait qu’un programme subit ce que l’on appelle une compilation. Le programme est transformé en binaire, et le binaire est exécuté. Ce n’est pas le cas de PHP. PHP est interprété, donc “lu” à la volée par un programme qui, lui, va exécuter ce qu’on lui demande de faire.

**Exécuté côté serveur**

PHP s’exécute sur un serveur. Il ne touche donc pas à la machine du client.



La place de PHP dans une architecture client-serveur. (Source : [enib.fr](https://www.enib.fr/~choqueuse/articles/flask/introduction-a-la-programmation-web.html))

**Ainsi que son interpréteur**

PHP c’est à la fois le nom d’un langage et du programme qui sert à l’interpréter.

## Petit historique

Le langage PHP a été créé en 1994 par Rasmus Lerdorf, qui voulait l’utiliser pour son site web personnel. C’est originellement une extension d’un autre langage : le C. Il a, au fur et à mesure, fait évoluer sa petite création en lui rajoutant des fonctionnalités.

Rasmus décide, en 1995, de publier son code pour en faire profiter tout le monde. En 1997, le cœur de PHP est redéveloppé par deux étudiants, Andi Gutmans et Zeev Suraski. Après un an de travail, la version 3 de PHP est née.

Ils se sont alors attaqués à la réécriture du moteur interne de PHP, qu’ils nommeront Zend Engine (Zend est la contraction de Zeev et Andi) et qui servira de base à PHP 4.

Source pour cette partie : [Wikipédia](https://fr.wikipedia.org/wiki/PHP#Histoire).

Depuis décembre 2015, PHP en est à sa version 7. La version 8 est sortie le 26 novembre 2020, mais nous resterons sur PHP 7 dans ce cours.

# À quoi cela ressemble-t-il ?

PHP peut s’utiliser en combinaison du HTML et du CSS pour apporter de la variabilité au code. Il est inséré dans une balise qui lui est propre, <?php ?>, et s’insère donc n’importe où dans la page.

Voici à quoi cela ressemble :

| <!DOCTYPE html> <html lang="en">  <head>  <title>Hello, World!</title>  </head>   <body>  <p>  <?php  echo 'Hello, World!';  ?>  </p>  </body> </html> |
| --- |

# À quoi cela sert-il dans le monde du développement Web ?

Les utilités de PHP sont diverses. Sa mission première, comme pour tout élément constitutif d’un site web est de **renvoyer une page Web**.

Mais il est également utilisé pour tout ce qui est no statique dans un site web. Il apporte des variables et des fonctions, et permet donc pas mal de choses dont :

* L’authentification de l’utilisateur
* La communication avec des bases de données
* L’écriture / lecture de fichiers
* L’appel d’autres ressources (page web, API, …)
* L’envoi d’emails
* ...

Partie 1

Mise en place de l’environnement

Pour toute cette partie, je vous renvoie vers les slides, qui contiennent toutes les informations nécessaires : [Partie 1 - Mise en place de l'environnement](https://docs.google.com/presentation/d/1wj-Gz-Netef4v8zVN31A98J-jscegPeTC4S5Z1ghDwY/edit?usp=sharing).

Partie 2

Le langage

# 

# La syntaxe de base

## Introduction

On écrit du code dans un fichier PHP. Ce fichier peut contenir du HTML aussi, et le PHP s’intègre au sein d’une balise <?php ?>.

En PHP, on “ordonne” des choses à l’ordinateur (au serveur). Chaque “ordre” se nomme une instruction, et chaque instruction se termine par un ;.

Comme en HTML, le retour à la ligne en PHP est non syntaxique (il n’est pas interprété).

Un exemple de Hello World en PHP :

| <!DOCTYPE html> <html lang="en">  <head>  <title>Hello, World!</title> </head>  <body>  <p>  <?php echo 'Hello, World!'; ?>  </p> </body>  </html> |
| --- |

## Commentaires

Comme pour beaucoup de langages de programmation, PHP propose de commenter son code.

Il existe deux types de commentaires :

* Les commentaires en ligne, qui commencent avec // et vont jusqu’à la fin de la ligne
* Les blocs de commentaires, sur plusieurs lignes, délimités par /\* et \*/

Comme le PHP se mélange au HTML, on peut aussi commenter le HTML (en dehors des balises <?php ?> du coup) avec les <!-- -->.

# Les instructions de base

## Déclarer / affecter une variable

En PHP, comme en JavaScript, il est possible de définir des variables pour “retenir des choses”. En PHP, les variables commencent toutes par un $.

Comme en JavaScript, les variables sont faiblement typées : elles ont un type interne (nombre, chaîne de caractères, booléen, tableau, objet, …) mais peuvent implicitement passer d’un type à l’autre sans souci.

En PHP, les déclarations de variables sont implicites : une variable est créée la première fois qu’elle est appelée. Aussi, pour déclarer une variable sans rien faire d’autre, on a une instruction qui sera $variable;.

On peut donner le nom que l’on veut à une variable, en respectant quelques règles :

* Elle doit commencer par une lettre ou un *underscore* (\_)
* Elle ne doit contenir que des chiffres, lettres ou *underscore*
* Les lettres dont on parle ne doivent pas avoir d’accent

Pour affecter une variable en PHP, on utilise l’opérateur d’affectation =.

Par exemple :

| $number = 5.3; $string = 'Lorem Ipsum'; $array = [1, 2, 'three']; $boolean = true; |
| --- |

La variable peut alors avoir plusieurs types de valeurs

* Nombre : nombres entiers ou à virgule
* Chaîne de caractères : le texte
* Booléen : true ou false
* Tableau
* Objet
* Vide (null)

En PHP, contrairement au JavaScript, il n’existe pas la valeur undefined.

On va voir ensemble plus en détails les chaînes de caractères et les tableaux dans la suite de ce cours.

## Les opérateurs

Les opérateurs sont des “symboles” définis dans le langage pour exécuter certaines actions. Il en existe tout un tas, et on connaît déjà notamment l’opérateur d’affectation =.

### Arithmétiques

Les opérateurs arithmétiques sont ceux qui nous viennent des mathématiques. Ils sont au nombre de 6 :

* + : addition
* - : soustraction
* \* : multiplication
* / : division
* % : modulo
* \*\* : puissance

Il est à noter que, comme en mathématiques, on peut utiliser le + et le - sur des opérandes (ce sur quoi on applique un opérateur) uniques : +3, -2, -$a.

Il est aussi à noter que la division (/) produit un résultat à virgule, correspondant donc à la valeur exacte de la division.

Le modulo (%), quant à lui, correspond au reste d’une division euclidienne, autrement dit, si on a 18 % 5, cela fera 3, car si on retire tous les “groupes” de 5 de 18, il nous reste 3.

Comme en mathématiques, certains opérateurs sont prioritaires par rapport à d’autres. Aussi, la puissance est prioritaire sur la multiplication, la division et le modulo, eux-mêmes prioritaires sur l’addition et la soustraction.

Autrement dit, 2 \*\* 3 / 4 + 5 vaut 7, et non autre chose.

On peut toutefois forcer les priorités en encadrant ce qui est concerné par des parenthèses. Ceci s’exécutera alors avant la suite. On peut cumuler les parenthèses pour être plus précis encore.

Ainsi, 2 \*\* 3 / 4 + 5 équivaut à (((2 \*\* 3) / 4) + 5).

### De comparaison

Les opérateurs de comparaison servent, comme leur nom l’indique, à faire des comparaisons. Ils ne renvoient donc pas un nombre, comme dans le cas des opérateurs arithmétiques, mais un booléen, indiquant si la comparaison est vraie ou fausse.

Nous allons étudier 9 des 10 opérateurs de comparaison. Nous ne verrons pas l’opérateur combiné ici.

Voici les 9 opérateurs que nous verrons :

* >
* >=
* <
* <=
* ==
* !=
* <>
* ===
* !==

Comme ces opérateurs servent à comparer, ils s’utilisent chacun avec 2 opérandes. Aussi, pour plus de facilité, on va présupposer que nous avons donc 2 opérandes $a et $b, que nous utiliserons pour mieux comprendre la suite.

Pour les 4 premiers opérateurs de comparaison, on va présupposer que $a et $b sont des nombres, pour plus de facilité de compréhension. Ainsi, on va avoir :

* $a > $b : renvoie vrai si $a est strictement supérieur à $b
* $a >= $b : renvoie vrai si $a est supérieur ou égal à $b
* $a < $b : renvoie vrai si $a est strictement inférieur à $b
* $a <= $b : renvoie vrai si $a est inférieur ou égal à $b

Pour les 5 derniers opérateurs, $a et $b peuvent prendre n’importe quelle valeur. Aussi, on aura :

* $a == $b : renvoie vrai si $a a une valeur équivalente à $b
* $a != $b : renvoie vrai si $a a une valeur qui n’équivaut pas à $b
* $a <> $b : renvoie vrai si $a a une valeur qui n’équivaut pas à $b
* $a === $b : renvoie vrai si $a est égal, en type et en valeur, à $b
* $a !== $b : renvoie vrai si $a n’est pas égal, en type ou en valeur, à $b

On a, pour ces 5 derniers opérateurs, trois choses à noter :

* Les opérateurs != et <> sont équivalents.
* On obtient !== et != en remplaçant le premier “=” de (respectivement) === et ==. Ces opérateurs sont opposés entre eux (!== et === sont opposés, tout comme == et !=).
* ===, et son opposé !==, comparent le type interne de la variable en plus de la valeur, contrairement à == et son opposé !=.

Ces opérateurs aussi ont, entre eux, des priorités. Ainsi,les 4 premiers opérateurs sont prioritaires sur les 5 derniers.

Il est à noter que tous les opérateurs (ceux déjà vus et ceux à venir) ont également des priorités entre eux. Pour les connaître, je vous invite à consulter la [documentation officielle](https://www.php.net/manual/fr/language.operators.precedence.php).

### Logiques

Les opérateurs logiques servent à faire des assertions logiques (des “calculs” de logique). Ils s’utilisent sur deux opérandes, sauf pour l’opérateur de négation. Aussi, chaque opérateur s’applique sur des (ou un) booléens et renvoient un booléen.

Les opérateurs logiques sont au nombre de 6, dont deux sont en double :

* || et OR
* && et AND
* XOR
* !

Pour les expliquer, on va prendre deux opérandes de type booléen, $a et $b. Ainsi :

* $a || $b (ou $a OR $b) : renvoie vrai si $a est vrai ou si $b est vrai ; renvoie faux sinon (OR signifie “ou”)
* $a && $b (ou $a AND $b) : renvoie vrai si $a et $b sont tous les deux vrais ; renvoie faux sinon (AND signifie “et”)
* $a XOR $b : renvoie vrai si $a est vrai ou si $b est vrai, mais renvoie faux si les deux sont vrais ou si les deux sont faux (XOR veut dire “exclusive OR” → “OU exclusif”)
* !$a : renvoie vrai si $a est faux ; renvoie faux si $a est vrai (ça inverse la valeur booléenne de $a)

Vous pouvez aussi vous référer aux tables de vérité présentes dans [le support de cours](https://docs.google.com/presentation/d/1FcdYt-njXb4YMd6NoXnrMxaV5aZC4SuyLCiucqVQCPE/edit#slide=id.ga7109b8e1a_0_28).

Ces opérateurs eux aussi s’exécute selon un ordre de priorité, qui est le suivant :

* !
* &&
* ||
* AND
* XOR
* OR

Vous pouvez donc voir que, même s’ils sont équivalents, && et AND ainsi que || et OR n’ont pas la même priorité.

#### Aparté sur le typage en PHP

| PHP type faiblement ses variables : les variables ont un type intrinsèque, mais rien n’empêche le changement de type au cours de la vie d’une variable.  Lors d'opérations mêlant deux types de variables différents, PHP va, par défaut, changer le type de l’une des variables pour correspondre à celui de l’autre variable. On appelle cela le transtypage. Ce comportement peut produire des résultats assez aléatoires, et vous ne devriez pas vous y fier.  On verra plus tard que l’on peut “forcer” le transtypage pour mieux le gérer : on parle de caster une variable. |
| --- |

## Les chaînes de caractères

Une chaîne de caractères, dans les langages informatiques, ça désigne, en gros, tout ce qui est textuel. Un texte, qu’il soit petit ou gros, sera donc de type chaîne de caractères (ou *string* en anglais).

### Différence simple et double quote

En PHP, une chaîne de caractères peut se définir de deux manières :

| <?php  $str = 'Une chaîne de caractères';  // OU  $str = "Une chaîne de caractères"; ?> |
| --- |

Autrement dit, on peut utiliser des guillemets simples ou doubles (aussi appelés *simple* ou *double quotes* en anglais).

La différence réside dans l’interprétation des variables dans la chaîne de caractères. Autrement dit, les *simples quotes* ne vont pas lire les variables, tandis que les *doubles quotes*, oui. Aussi :

| <?php  $str1 = 'Une string'; // Vaut "Une string"  $str2 = "Et encore $str1"; // Vaut "Et encore Une string"  $str3 = '$str1 de plus'; /\* Vaut "$str1 de plus" et non "Une string de plus" \*/ ?> |
| --- |

A noter qu’il existe d’autres manières de définir des chaînes de caractères en PHP, mais que l’on n’abordera pas dans ce cours. Je vous invite à vous référer à la [documentation officielle](https://www.php.net/manual/fr/language.types.string.php#:~:text=Une%20cha%C3%AEne%20de%20caract%C3%A8res%20est,support%20natif%20pour%20l'Unicode.) pour de plus amples détails.

### Concaténation

La concaténation est l’équivalent de l’addition de chaînes de caractères : on met deux chaînes bout à bout l’une de l’autre.

En PHP, l’opérateur de concaténation est le .. Voici comment l’utiliser :

| <?php  $str1 = 'Hello';  $str2 = 'World';   $hello\_world = $str1 . ', ' . $str2 . '!'; // Vaut "Hello, World!" ?> |
| --- |

### Fonctions

#### Aparté sur les fonctions

| Une fonction est un bout de code auquel on donne un nom pour pouvoir le rappeler plus tard et l’exécuter à l’envi.  Une fonction peut prendre des paramètres qui lui seront utiles pour lui dicter son comportement, pour s’adapter à l’environnement dans lequel elle s’exécute, …  Un chapitre est dédié aux fonctions plus tard dans ce cours. |
| --- |

Il existe tout un tas de fonctions pouvant intervenir sur des chaînes de caractères. La liste complète se trouve sur la [documentation officielle](https://www.php.net/manual/fr/ref.strings.php).

En voici cependant quelques-unes :

* strtoupper : renvoie la chaîne de caractères mise en majuscules
* strtolower : renvoie la chaîne de caractères en minuscules
* str\_shuffle : renvoie les caractères de la chaîne mélangés

## Les dates

### Introduction

Les dates (et l’heure) sont traitées un petit peu à part en PHP. Elles peuvent se représenter de plusieurs manières. Elles peuvent être des *string*, des nombres ou encore des objets. On ne parlera pas des objets dans ce cours, puisque cela fera l’objet d’un autre module.

Aussi, rentrons un peu dans les détails.

Il y a deux manières de représenter une date :

* On peut donner le jour, le mois, l’année, l’heure, la minute, la seconde, … pour indiquer l’instant que l’on souhaite représenter
* On peut compter une certaine durée depuis un certain point de départ

Ainsi, en PHP, on va pouvoir représenter les données temporelles de deux manières :

* Avec un nombre, qui correspondra au nombre de secondes écoulées depuis le 1er janvier 1970 à minuit (Oui, ça fait un paquet de secondes). Cela correspond au type SQL TIMESTAMP.
* Avec une chaîne de caractères : on va alors simplement écrire la date sous un format que l’on aura défini et que l’on saura reconnaître. Cela correspond aux types SQL DATE, TIME et DATETIME.

Exemples :

| <?php   $timestamp = 1607963160;  $datetime = '2020-12-14 16:26:00';  $datetime2 = '14/12/20 16h26'; ?> |
| --- |

Toutes ces variables représentent le 14 décembre 2020 à 16 heures et 26 minutes.

La première, $timestamp, correspond donc au nombre de secondes écoulées entre le 1er janvier 1970 00:00 et le 14 décembre 2020 16:26. $datetime et $datetime2, quant à elles, correspondent à la même date mais avec deux formats différents, à savoir respectivement le “format MySQL” et le “format français”.

Tout ne va donc être que question de format de date et de leur manipulation.

Pour leur manipulation, justement, tout va passer par l’intermédiaire de fonction : créer une date, la formater, faire des calculs dessus, … et on voit ces fonctions dans la prochaine partie !

### Fonctions

Les deux principales fonctions avec lesquelles on interagit lorsque l’on traite avec des données temporelles en PHP, ce sont les fonctions date et time. Cependant, elles ne font pas exactement ce que l’on peut imaginer d’après leur nom :

* time() : renvoie le nombre de secondes écoulées depuis l’époque UNIX
* date($format, $time) : renvoie une date formatée correspondant à $time (si $time n’est pas fourni, il vaut l’instant présent)

Le paramètre $format de la fonction date est une chaîne de caractères qui dit comment on veut formater la date. On peut par exemple utiliser 'd/m/y' pour avoir la date sous forme '14/12/20' (“à la française”). Les formats sont tous décrits dans la [documentation officielle](https://www.php.net/manual/fr/datetime.format.php).

On peut aussi utiliser deux fonctions, date\_create et date\_create\_from\_format, pour créer des dates. Il faut cependant faire attention, car les dates retournées sont alors des objets, et non des chaînes de caractères. Si on souhaite les afficher, il faudra donc les formater grâce à la fonction date\_format.

| <?php  // On crée une date à partir du format français  $date = date\_create\_from\_format('d/m/y', '06/01/2021');   // On l'affiche au format anglais  echo date\_format($date, 'm/d/Y'); ?> |
| --- |

#### Aparté sur les caractères d'échappement

| On est en mesure de se demander comment on peut inclure des caractères “normaux” dans nos formats. On peut notamment se demander comment on fait pour afficher l’heure “à la française”, par exemple “16h26”, puisque le “h” va être interprété comme la valeur de l’heure (donc 16)...  De la même manière, on peut se demander comment stocker dans une variable de type *string* la chaîne de caractères « Maman m’a dit “va manger !” et j’ai obéi. » puisqu’on retrouve ici des *quotes*, simples comme doubles.  C’est pour ce genre de situations qu’a été pensé un caractère d'échappement. Le rôle de ce caractère, qui en PHP est l’antislash (\), est d’indiquer que le caractère qui suit *ne doit pas être interprété,* ou bien *doit être interprété différemment*.  On va ainsi pouvoir résoudre nos soucis et avoir notre code fonctionnel : |
| --- |
| <?php  $heure = date('h\hm');  // Le premier h sera remplacé par l'heure, le second non.  $phrase = 'Maman m\'a dit "va manger !" et j\'ai obéi.';  // Obligé d'échapper les '', mais pas les ""  // Ou bien  $phrase = "Maman m'a dit \"va manger !\" et j'ai obéi.";  // Obligé d'échapper les "", mais pas les '' ?> |

En plus des fonctions date et time, qui servent, donc, à créer des données temporelles, nous avons aussi pléthore de fonctions (que vous pouvez toutes retrouver sur la [documentation officielle](https://www.php.net/manual/fr/ref.datetime.php)) servant à diverses choses en rapport avec les données temporelles. Mais la majorité de ces fonctions utilisent des objets. Aussi, voici quelques fonctions qui n’en utilisent pas et qui pourraient être utiles :

* checkdate($jour, $mois, $annee) : renvoie un booléen indiquant si la date est valide ou non
* getdate($date) : renvoie un tableau (on les aborde juste après ;-)) donnant des informations sur une date UNIX
* date\_parse\_from\_format($format, $date) : fait la même chose que getdate mais avec une date formatée dont on précise le format

## Les tableaux

Dernier des types que nous verrons plus en détails pour PHP. Ils s’appellent également *array*, et on peut déclarer un tableau en PHP de deux manières :

* De manière fonctionnelle : en utilisant la fonction array()
* De manière syntaxique : en utilisant les crochets ([])

| <?php  $arr = array(1, 2, 3);  $arr2 = [4, 5, 6]; ?> |
| --- |

Un tableau, en PHP, c’est un ensemble de paires clef-valeur. On accède aux valeurs d’un tableau en utilisant sa clef avec la syntaxe suivante : $array[$clef]. Un tableau peut contenir n’importe quel type de valeur (même des tableaux, oui ;-)).

Nous allons nous attarder sur 3 cas d’utilisation des tableaux en PHP et voir quelles clefs utiliser dans ces cas.

### Le cas simple

Si on déclare un tableau à PHP, l’interpréteur va l’indexer automatiquement. Autrement dit, il va automatiquement lui donner ses clefs.

Lesquelles ? Eh bien, il va tout simplement compter, de zéro jusqu’à l’infini, et donner ce nombre comme clef pour la valeur associée. Par exemple, avec l’array $arr = ['zéro', 'un', 'deux', 'trois'];, la valeur 'un' a la clef 1. Et on peut accéder à 'trois' en écrivant $arr[3]. Du coup, il suffit de savoir compter, et de se souvenir que PHP compte à partir de zéro !

### Le cas des tableaux associatifs

Cette indexation est bien pratique, mais on peut aussi choisir de ne pas l’utiliser et d’associer nous-mêmes des clefs à nos valeurs : on crée donc un tableau associatif.

La syntaxe pour déclarer et affecter un tableau reste la même, mais on va spécifier les clefs à l’aide de la double-flèche =>. Et avec cette syntaxe, on peut utiliser des *string*, en plus des nombres, comme clefs pour nos tableaux :

| <?php  $tableau\_associatif = array(  'clef' => 'valeur',  'clef2' => 'valeur2',  46 => 72,  'truc' => 'muche',  ...  ); ?> |
| --- |

Dans cette configuration, on accèdera à 'valeur2' en écrivant $tableau\_associatif['clef2'].

En revanche attention ici, du coup, $tableau\_associatif[0], par exemple, n’existe pas.

### Le cas des tableaux multi-dimensionnels

Comme dit dans l’introduction des tableaux, un tableau peut contenir tout type de variable, y compris des tableaux. On peut donc se retrouver avec des tableaux de tableaux, et on peut aller aussi profondément que l’on veut comme ça. On appelle ces tableaux des tableaux multi-dimensionnels.

Ce qu’il faut savoir avec ces tableaux, c’est qu’ils se comportent exactement de la même manière que les autres tableaux. Autrement dit, tout ce qu’on a vu sur les clefs, automatiques ou non, reste valable et on va du coup fournir les clefs dans l’ordre pour accéder à la valeur souhaitée. Par exemple, on pourrait avoir :

| <?php  $user = array(  'prenom' => 'Alice',  'ami' => array(  'prenom' => 'Bob',  'adresse' => array(  'num' => 4,  'rue' => 'Avenue du Général De Gaulle',  )  )  ); ?> |
| --- |

Si on souhaitait, dans cette configuration, connaître la rue dans laquelle l’ami d’Alice habite, on utiliserait $user['ami']['adresse']['rue']. On doit à chaque fois utiliser la clef nécessaire pour “avancer d’un niveau” vers l'information désirée.

### Le cas particulier des chaînes de caractères

Pour l’anecdote, sachez que les chaînes de caractères, en PHP (et dans d’autres langages aussi), sont en fait des *tableaux* de caractères.

Cela signifie qu’en interne PHP considère chaque texte comme un ensemble de caractères qui sont stockés, en interne, dans un tableau.

Vous pouvez donc très bien écrire le code suivant, qui fonctionnera :

| <?php   echo 'Hello, World!'[7]; // Affichera 'W' ?> |
| --- |

Un autre petit détail qui peut avoir son importance, vous ne pouvez pas utiliser une chaîne de caractères comme un tableau pour les fonctions qui attendent un argument de type *array* car le type interne des chaînes de caractères est bel et bien *string*, et non *array*.

Sachez que vous n’utiliserez jamais cette syntaxe des tableaux sur les chaînes de caractères. C’est plus pour la culture générale.

## Les constantes du langage

Il existe des “variables” un peu particulières qui nous sont mises à disposition par PHP. Ces variables sont en fait bien souvent invariables et contiennent des valeurs qui peuvent être utiles pour l’exécution du script.

Ces “variables” sont des constantes. Elles sont souvent écrites en majuscules, et comme ce ne sont pas vraiment des variables, elles n’ont pas de $ avant leur nom.

On en a déjà vu une dans les exercices sur les opérateurs : M\_PI. Cette constante renseigne la valeur de la constante mathématique π (pi).

Mais il en existe des tas d’autres, que l’on peut retrouver dans la [documentation officielle](https://www.php.net/manual/fr/reserved.constants.php) (vous ne vous y attendiez pas, hein…).

Parmi les plus utiles, on peut citer \_\_DIR\_\_ et \_\_FILE\_\_, qui contiennent respectivement le répertoire et le fichier courant, par exemple. On a aussi PHP\_VERSION qui stocke la version utilisée de PHP.

On peut également définir des constantes à l’aide le la fonction define, qui prend en arguments le nom de la constante, ainsi que sa valeur : define('MA\_CONSTANTE', 42);.

# Les structures

Une structure est un élément du langage qui va permettre de modifier le comportement du code à un endroit donné pour lui donner une autre direction, une autre utilisation… Bref, pour le structurer.

## Conditions

### If else

Les conditions (ou structures conditionnelles), comme leur nom l’indique, conditionnent le code : si [truc] alors [machin] sinon [bidule]. Vous pouvez remplacer truc, machin et bidule par des morceaux de code dans l’exemple précédent.

En PHP, on a 3 mots-clefs du langage pour les structures conditionnelles :

* if (“si”) : laisse s’exécuter le code si une condition est remplie
* elseif (“sinon si”) : couplé à un if (et potentiellement des elseif), il laisse s’exécuter le code si la condition du if n’est pas rempli et si sa propre condition est remplie
* else (“sinon”) : couplé à un if (et potentiellement des elseif), il s’exécute si aucune des conditions précédentes n’a été remplie

N.B. : Vous pouvez aussi croiser else if, en deux mots. Cela fait exactement la même chose.

Au niveau de la syntaxe, voici à quoi cela ressemble :

| <?php  if ($assertion\_logique) {  // Si $assertion\_logique vaut true, on entre ici  } elseif ($autre\_assertion) {  // Sinon, si $autre\_assertion vaut true, on entre ici  } else {  // Sinon, en dernier recours, on entre ici  } ?> |
| --- |

Vous noterez donc que le if et le elseif prennent un “paramètre”, qui est en fait un booléen, ou une expression qui s’évalue comme un booléen (qui se “lit” comme un booléen, qui “renvoie un résultat” booléen, comme par exemple… des comparaisons ou des assertions logiques ;-)) tandis que le else non : il s’exécute si **rien** ne s’est exécuté avant dans les if et elseif qui lui sont associés.

Il peut-être bon de savoir également que les accolades ne sont pas obligatoires si un bloc d’instruction n’est composé que d’une seule instruction. Aussi, il est possible de trouver dans certains codes des écritures comme suit : if ($condition) echo 'Quelque chose.';.

#### Aparté sur l’opérateur ternaire

| Il existe un opérateur un peu particulier, appelé opérateur ternaire. Il permet, en quelque sorte, de proposer une écriture “abrégée” pour des conditions simples. Pour le comprendre, je pense qu’un exemple vaut mieux que mille mots :   | <?php  // Ecriture normale  if ($condition) {  echo 'OK';  } else {  echo 'Pas OK';  }   // Ecriture avec l'opérateur ternaire  echo $condition ? 'OK' : 'Pas OK'; ?> | | --- |   On retrouve donc notre condition, suivie d’un “?”, puis du cas vrai, d’un “:” et du cas faux. C’est une syntaxe assez difficile à lire, aussi il est préférable de les éviter, mais elles peuvent être fréquentes pour alléger le code.  A noter également que l’opérateur **renvoie** quelque chose (comme tous les opérateur). De fait, il n’**exécute** rien, et c’est pourquoi on doit sortir le echo, et on ne peut donc pas écrire : $condition ? echo 'OK' : echo 'Pas OK';. Ceci renvoie une erreur. |
| --- | --- |

### Switch

Les structures switch servent à exécuter un morceau de code en fonction de la valeur d’une expression. On va énumérer les différents cas possibles et le code à exécuter pour chaque cas. Au niveau de la syntaxe, on va retrouver le mot-clef switch suivi, entre parenthèses, de l’expression à évaluer. Puis nous allons appeler chaque cas avec le mot-clef case, suivi de la valeur désirée et du code à exécuter.

Il est à noter qu’il faut indiquer la fin de l’exécution du code par l’instruction break; car sinon le code qui suit (même s’il fait partie du cas suivant) sera exécuté lui aussi.

On peut, enfin, définir un cas particulier, default, qui sera exécuté dans tous les cas qui ne sont pas ceux énoncés.

| <?php switch ($piece) {  case 'pile':  echo 'Pierre a gagné.';  break;  case 'face':  echo 'Marie a gagné.';  break;  default:  'La pièce serait-elle tombée sur la tranche ? O.o'; } ?> |
| --- |

## Boucles

Les boucles (ou structures itératives), servent à exécuter un même code autant de fois qu’il le faut pour atteindre une certaine condition.

En PHP, il existe 4 structures itératives différentes.

### for

On répète la boucle for *pour* atteindre un état à partir d’un autre. Aussi, on va considérer, avec cette boucle, la situation de départ, celle d’arrivée, et le changement entre chaque étape, qui va amener à l’arrivée.

Au niveau de la syntaxe, ça ressemble à ça : for (situation\_de\_depart ; condition\_d\_arrivee ; changement\_d\_etape) {}

On a donc une syntaxe qui vous rappellera celle de JavaScript, avec :

* Situation de départ : une expression qui sera exécutée au début de la boucle, **quoi qu’il arrive**.
* Condition d’arrivée : une expression (une condition) qui sera évaluée au début de chaque itération. Si elle vaut true, le bloc de code entre accolades ({}) sera exécutée. Sinon, la boucle s’arrêtera.
* Changement d’étape : une expression qui sera exécutée à la fin de chaque itération.

Chacune de ces parties peut être laissée vide. Si la condition d’arrivée est vide, elle est considérée comme true. Du coup, on a une expression qui est toujours vraie et la boucle ne s’arrête jamais : on entre dans une boucle infinie.

On peut aussi mettre plusieurs expressions à chaque fois, séparées par des virgules. Elles seraient alors toutes exécutées. Dans le cas de la condition d’arrivée, toutes seraient également exécutées mais seule la première servirait de condition d’arrêt de la boucle.

On va souvent utiliser la boucle *for* pour itérer un nombre précis de fois en utilisant un compteur. On va souvent faire comme suit :

| <?php   for ($i = 0 ; $i < 40 ; $i = $i + 1) {  // Du code qui sera répété 40 fois  } ?> |
| --- |

Ici, on a une situation de départ qui affecte 0 à la variable $i : $i = 0. Puis on a comme condition d’arrêt $i < 40. Comme $i vaut 0 au début, la boucle va itérer. Puis elle va exécuter l’expression de changement d’étape : $i = $i + 1. Ainsi, $i vaut 1, la boucle va itérer à nouveau, et encore jusqu’à ce que $i valle 39. En effet, lorsque $i va valloir 39, la boucle va itérer, puis exécuter à nouveau $i = $i + 1. $i vaut donc 40. La condition d’arrêt se trouve fausse : la boucle s’arrête. On aura donc exécuté la boucle pour 40 valeurs de $i différentes : de 0 à 39.

#### Aparté sur les “abréviations”

| Comme dit précédemment, la boucle *for* va souvent être utilisée pour itérer un nombre précis de fois. On va donc souvent avoir des expressions d’itération qui ressembleront à $i = $i + un\_nombre. Il faut savoir qu’il existe une notation “abrégée” pour faire cela, qui utilise en réalité un autre opérateur d’affectation : $i += un\_nombre. Il faut savoir que ces autres opérateurs d’affectation existent pour tous les opérateurs arithmétiques, et également pour l’opérateur de concaténation. Il existe ainsi +=, -=, \*=, \*\*=, /=, %= et .=. (Il en existe encore d’autres, que nous n’avons pas vus, mais qui se trouvent dans la [documentation officielle](https://www.php.net/manual/fr/language.operators.assignment.php))  Sachez également que dans le cas de $i += 1 ou $i -= 1, nous avons deux nouvelles notations qui existent pour encore plus abréger, qui sont l’opérateur d’incrémentation et l’opérateur de décrémentation. Ainsi, $i += 1 peut s’écrire $i++ et $i -= 1 peut s’écrire $i--.  Enfin, on peut préciser que ces deux derniers opérateurs, ++ et --, peuvent s’utiliser sur la variable en les écrivant avant ou après celle-ci. La différence réside dans le fait que $i++ va d’abord être évalué (“lu”) **puis** incrémenté, tandis que ++$i va d’abord être incrémenté **puis** évalué. (Idem avec --) |
| --- |

### while

On répète la boucle while *tant que* notre condition est vraie. Cette boucle est beaucoup plus simple dans son écriture, puisqu’on ne fournit que ladite condition.

Au niveau de sa syntaxe, un while s’écrit comme suit : while (condition) {}.

La condition en question est une expression qui renvoie un booléen (le même genre de condition que pour les if).

Dans son fonctionnement, la boucle va évaluer la condition **avant** **tout** et exécuter le code si elle est évaluée à true. Du coup, il se peut que le code ne soit jamais exécuté, si la condition vaut false au départ.

A contrario, si jamais votre condition vaut **toujours** true, vous entrez à nouveau dans le cas d’une boucle infinie. Il est d’ailleurs courant de croiser while (true) {} dans certains codes qui ont besoin d’une boucle infinie. Mais si ce n’est pas votre cas, pensez à faire en sorte que la condition **devienne fausse** à un moment pour pouvoir sortir de la boucle.

### do while

La boucle do while se comporte exactement comme la boucle while, sauf qu’on passe forcément au moins une fois à l’intérieur car la condition est évaluée après avoir exécuté le code et non avant, contrairement au while.

Au niveau de la syntaxe aussi, le do while se distingue, puisque cela ressemble à ça : do {} while (condition);. Vous voyez donc qu’on met le code entre le do et le while, au lieu d’après le while. Notez aussi l’apparition du point-virgule après la condition.

### foreach

La boucle foreach est une structure itérative propre à PHP qui permet de parcourir chaque élément d’un tableau donné.

Sa syntaxe est relativement simple : foreach ($tableau as $clef => $valeur) {}. A chaque itération, $clef et $valeur vont être remplies, respectivement, par la clef et la valeur de l’entrée du tableau en train d’être parcourue.

Il est également possible de ne prendre que la valeur, si la clef ne nous est pas utile : foreach ($tableau as $valeur) {}. (Mais il n’est pas possible de ne prendre que la clef)  
Je précise également, s’il en est besoin, que les variables peuvent porter n’importe quel nom, et pas uniquement $clef et $valeur.

# Les fonctions

Les fonctions sont, comme déjà dit précédemment, des bouts de code que l’on nomme pour pouvoir les utiliser plus tard. C’est en quelque sorte une variable qui contient du code que l’on peut exécuter en l’appelant.

On peut donc créer des fonctions, les appeler, et on peut aussi leur fournir des paramètres (comme vu dans l’[aparté sur les fonctions](#_o21enw7mdufr)).

## Appel de fonction

Nous avons déjà eu l’occasion dans ce cours de travailler avec des fonctions, notamment quand nous avons vu les chaînes de caractères, ou encore les dates.

Appeler une fonction se fait très simplement : on écrit le nom de la fonction à appeler, et on met entre parenthèses, s’il y en a, les paramètres, séparés par des virgules. Évidemment, comme il s’agit d’une instruction, la ligne se termine par un point-virgule : fonction($param1, $param2, $param3);.

Si la fonction retourne quelque chose, on peut utiliser cette valeur pour n’importe quoi : la stocker dans une variable, la fournir en paramètre d’une autre fonction, etc.

## Déclaration de fonction

En PHP, on déclare une fonction à l’aide du mot-clef function. On va ensuite donner son nom, ses paramètres entre parenthèses et séparés par des virgules (s’ils existent), puis le corps de la fonction (le code qu’elle exécute). Cela donne quelque chose comme ça :

| <?php  function truc($param1, $param2, $param3) {  // Du code à exécuter  }   // Ou encore  function machin() {  // Le code d'une fonction sans paramètre  } ?> |
| --- |

A partir de là, dans le corps de truc, nous avons accès aux variables $param1, $param2 et $param3 pour pouvoir effectuer des opérations qui ont besoin de connaître leur valeur.

A la fin de la fonction, on peut utiliser l’instruction return pour renvoyer une valeur, correspondant au résultat de la fonction.

| <?php  function truc() {  return 3;  }    $variable = truc(); // $variable = 3 car `truc` renvoie 3 ! ?> |
| --- |

## Passage d’arguments

Lorsque l’on définit une fonction avec des arguments (ou des paramètres, c’est un synonyme), on peut appliquer certaines contraintes sur ces arguments. Ces contraintes sont utiles pour “forcer” certains types de valeurs, si la fonction ne doit travailler qu’avec ces dits types.

Aussi, il faut savoir que l’on peut forcer le type d’un paramètre, en précédant le nom du paramètre par le type en question dans la définition de la fonction. On écrira, pour forcer le type “chaîne de caractères” sur un paramètre nommé “nom” : string $nom. La liste des typages possibles est disponible sur la [documentation officielle](https://www.php.net/manual/fr/language.types.declarations.php).

On peut également donner des valeurs par défaut aux paramètres. Cela permet de rendre le paramètre optionnel à l’appel de la fonction, car s’il n’est pas présent, il prendra la valeur définie par la fonction : $param = une\_certaine\_valeur. Il faut évidemment bien penser à organiser les paramètres pour mettre les optionnels à la fin (sinon, on doit forcément écrire le paramètre à la fin et donc la valeur par défaut n’a pas grand intérêt).

On peut utiliser toutes les contraintes en même temps. Cela donne, par exemple :

| <?php  function bidule($param1, string $nom, $optionnel = 3) {  // Corps de la fonction  }    // Ainsi je peux faire ceci  bidule('Truc', 'Machin', 42); // $optionnel = 42   // Ou cela  bidule('Un autre truc', 'Un autre machin'); // $optionnel = 3 ?> |
| --- |

#### Aparté sur la portée des variables

| En PHP (et dans d’autres langages), les variables ne sont accessibles qu’à certains endroits du code, selon comment et où elles sont définies : on parle de portée des variables.  Ainsi, une variable définie **dans une fonction** ne sera pas accessible en dehors de celle-ci, et vice-versa : une variable définie **en dehors d’une fonction** ne sera pas accessible à l’intérieur de celle-ci.  Une page de la [documentation officielle](https://www.php.net/manual/fr/language.variables.scope.php) explique plus en détails la portée des variables en PHP. |
| --- |

Un des effets de la portée des variables, c’est qu’une fonction ne peut pas modifier une variable définie en dehors de son corps, même si celle-ci lui est passée en paramètre. On parle de passage par valeur : seule la **valeur** de la variable est transmise.

| <?php  function modification($variable) {  $variable = 3;  }    $variable = 32;  modification($variable); // $variable vaut toujours 32 ! |
| --- |

Il est toutefois possible de choisir de faire un passage par référence de la variable. Autrement dit, on ne va plus passer la valeur de la variable, mais la référence de la variable, ce qui permettra de la modifier, même dans le corps de la fonction. On utilise pour ce faire l’esperluette (&) sur la variable, qui permet d’indiquer qu’il s’agit d’une référence de variable, et non d’une variable en tant que telle :

| <?php  function modification(&$variable) { // Notez le &  $variable = 3;  }    $variable = 32;  modification($variable); // $variable vaut à présent 3 ! ?> |
| --- |

Il existe, enfin, une dernière chose à savoir qui peut être utile pour les passages d’arguments en PHP : comment passer une infinité d’arguments. Par exemple, comment peut-on écrire la signature de la fonction array ? (la “forme” de la fonction : son nom et ses paramètres, mais pas son corps)

En fait, il existe un token pour cette situation (un “jeton”, un élément du langage qui n’est pas un opérateur mais presque, c’est un “truc” en gros (plus d’infos dans la [documentation officielle](https://www.php.net/manual/fr/tokens.php))). Il s’agit du *token* .... Cela va faire en sorte que tous les éléments donnés en arguments soient “assemblés” dans un tableau. Par exemple :

| <?php // Exemple tiré de la [documentation officielle](https://www.php.net/manual/fr/functions.arguments.php)  function sum(...$numbers) {  $acc = 0;  foreach ($numbers as $n) {  $acc += $n;  }  return $acc;  }   echo sum(1, 2, 3, 4); // Affiche 10 ?> |
| --- |

# EXERCICE

Écrire une fonction de tri, permettant de trier les éléments (nombres) d’un tableau fourni en paramètre par ordre croissant.

#### Aparté sur les fonctions de tri

| Il existe tout un tas de manières de trier un tableau de manière programmatique (il n’y a qu’à voir, déjà, comment des personnes trient des cartes à jouer, et on peut déjà se faire une idée des manières de penser différentes existant.  Cet exercice (trier un tableau) représente l’exercice type d’un domaine appelé l’algorithmique. Il va s’agir d’étudier les **manières** de faire une même fonction, un même algorithme, et les incidences de chacune sur les ressources du système.  En effet, selon la manière de faire, la fonction va s’exécuter en plus ou moins de temps, en plus ou moins de calcul, en prenant plus ou moins de place en mémoire RAM, en consommant plus ou moins d’électricité, … Tout ceci découle de la manière d’écrire la fonction, et se regroupe sous le terme de complexité (on peut aussi parler d’efficacité, de rendement, ...).  Il est important d’avoir conscience de cette complexité. Elle a beaucoup d’incidence sur le rendu d’une fonction, donc du script, donc de la page. Elle influe donc sur la charge qui pèse sur le serveur, donc sur sa consommation d’énergie, la chaleur qu’il émet, la pollution qu’il produit, … |
| --- |

# 

Partie 3

Utiliser le langage

# Découper son code

L’un des principaux avantages d’avoir un langage de programmation, comme PHP, c’est de pouvoir découper son code comme on l’entend.

On va donc pouvoir avoir un système de librairie, permettant l’ajout de code extérieur, mais on va également pouvoir séparer son propre code en plusieurs fichiers, et ainsi mieux le “ranger”. On verra d’ailleurs plus tard qu’il existe des architectures de code et des design pattern, qui sont des organisations de code permettant une productivité optimisée ou un travail en équipe plus efficace, ou tout un tas d'autres choses.

Mais pour le moment, je vais me contenter de vous donner **la** fonction PHP qui vous sera utile pour couper votre code : include. Ce n’est pas à proprement parler une fonction. C’est une instruction du langage, au même titre que echo. Elle s’utilise donc de la même manière, avec ou sans parenthèses, et prend en paramètre un nom ou chemin de fichier à inclure.

Si le fichier inclus contient du HTML, ce HTML va s’écrire à l’emplacement de l’inclusion. S’il contient des fonctions, les fonctions seront alors définies dans le script qui appelle l’inclusion, etc.

On a donc la possibilité d’organiser son code en petits fichiers qui seront spécifiques à une fonctionnalité, plutôt que de faire un énorme fichier de 5000 lignes de code qui gèrent tout le site.

Il est à noter qu’il existe aussi l’instruction require qui fonctionne comme include, et require\_once ainsi que include\_once qui font aussi la même chose, mais en s’assurant que le fichier n’a été inclus qu’une seule fois.

Un des exemples d’utilisation de cette fonctionnalité aura lieu pour inclure le menu de notre site (qui ne change pas), ou encore le header et footer.

# Interagir avec la base de données

Vous avez déjà vu ce qu’était une base de données. On ne va pas revenir dessus, mais on va voir comment les utiliser avec PHP. C’est quand même mieux que de stocker les données dans des variables écrites dans le code, et plus pratique que d’ouvrir la console à chaque requête utilisateur.

#### Aparté sur PHP Objet

| Les librairies que nous allons étudier dans ce chapitre reposent sur un principe de programmation que l’on appelle “orientée objet”. Vous aurez un module complet dédié à la Programmation Orientée Objet à la suite de ce module et, de fait, je ne vais pas rentrer dans les détails de celle-ci dans ce cours.  Aussi, sachez que cela fonctionne un peu comme les objets en JavaScript : les objets PHP contiennent des attributs (des variables internes) et des méthodes (des fonctions internes), et on y accède grâce à l’opérateur -> (la flèche simple) qui est l’équivalent du . en JavaScript. |
| --- |

Le principe de base des librairies permettant de communiquer avec les bases de données est de proposer un ensemble de fonctions (ou de méthodes, du coup) permettant d’effectuer les opérations courantes :

* Se connecter à la base de données
* Exécuter une requête
* Préparer une requête avec des arguments
* “Exécuter” une requête préparée
* Récupérer les résultats d’une requête
* Parcourir les résultats d’une requête
* Compter les résultats d’une requête ou le nombre de lignes affectées par celle-ci
* Clore une connexion à une base de données
* Récupérer les erreurs intervenant à chacune de ces étapes

Ce sont les principales fonctionnalités, celles qui couvrent 95 % des besoins. Voyons à présent 2 bibliothèques différentes permettant cela.

## mysqli

MySQLi est une librairie assez ancienne, qui reprend une autre librairie, mysql, plus ancienne encore. Comme son nom le suggère, elle permet de gérer une base de données MySQL uniquement.

### La connexion

On se connecte à la base de données en instanciant la classe (en créant un objet de la classe), avec une instruction qui ressemble à ceci :

| $mysqli = new mysqli('127.0.0.1', 'votre\_utilisateur', 'votre\_mdp', 'votre\_db'); |
| --- |

Si cette instruction échoue, nous pouvons obtenir le code d’erreur ainsi qu’un texte l’expliquant avec (respectivement) les attributs $mysqli->connect\_errno et $mysqli->connect\_error.

### Les requêtes simples

Exécuter une requête se fera avec la méthode query, qui prend en paramètre la requête à exécuter. On stockera son retour dans une variable, qui contiendra un objet représentant les résultats, ou bien false en cas d’échec. Par la suite, nous pouvons connaître le nombre de lignes retournées (par exemple dans le cas de requêtes de sélections) avec l’attribut num\_rows.

| <?php  $result = $mysqli->query("SELECT truc FROM table LIMIT 10"); ?> |
| --- |

#### Aparté sur les Injections SQL

| Pour créer des requêtes, notamment de sélection, on va souvent vouloir utiliser une variable pour écrire la requête. On va donc souvent écrire quelque chose comme $query = "SELECT \* FROM table WHERE nom = $variable";, avec $variable qui est une valeur qui vient, par exemple, de l’URL ou bien d’un formulaire.  Ce qu’il faut savoir, là-dessus, c’est qu’un utilisateur mal intentionné peut utiliser cette instruction comme une opportunité. En effet, ledit utilisateur peut très bien fournir une valeur lui permettant d’écrire ce qu’il veut dans la requête, et ainsi exécuter **n’importe quelle requête** sur la base de données.  Cela s’appelle un injection SQL, car on **injecte** du SQL dans des valeurs de formulaire ou d’URL. Aussi, encore une fois, il ne faut **pas** faire confiance à l’utilisateur, et prendre quelques précautions avant d’utiliser la variable. Dans le cas présent, il convient d’échapper la variable, ce qui va avoir comme effet de permettre de ne pas interpréter les caractères qui permettent à l’imposteur de faire passer sa requête (les ;, --, etc, les caractères qui permettent de clore une requête et donc d’en écrire une autre). Il faut donc utiliser la fonction $mysqli->real\_escape\_string. |
| --- |

### Parcourir les résultats

Toujours dans le cas de requêtes de sélection, la fonction fetch\_assoc, appliquée sur le résultat de la requête, nous permet de récupérer le prochain enregistrement des résultats sous forme de tableau associatif. Cette fonction renvoie false s’il n’y a pas d’enregistrement. Aussi, on utilisera souvent une boucle while pour traiter l’ensemble des résultats (car la fonction renvoie false et permet donc d’arrêter la boucle).

| while ($actor = $result->fetch\_assoc()) {  // Affiche le nom de l'acteur actuellement parcouru  echo '<li>' . $actor['first\_name'] . ' ' . $actor['last\_name'] .  '</li>'; } |
| --- |

A la toute fin, c’est une bonne partie de libérer la mémoire occupée par les résultats, ainsi que de clore la connexion à la base de données. Nous pouvons le faire avec les méthodes free et close :

| $result->free(); $mysqli->close(); |
| --- |

### Les requêtes préparées

Les requêtes préparées sont des requêtes prévues pour être exécutées avec des paramètres. C’est, en quelque sorte, la “bonne” manière de faire des requêtes avec des variables.

Avec mysqli, les requêtes préparées utilisent la fonction prepare. On marque les “paramètres” avec des “?”. Puis on doit binder (lier) les paramètres avec la fonction bind\_param, qui prend en premier argument la description des paramètres, et ensuite tous les paramètres à lier. Cette fonction s’exécute sur le retour de la fonction prepare (un *statement*, une préparation de requête) et pas sur l’objet $mysqli directement.

Vient ensuite le moment d’exécuter la requête, grâce à la fonction execute, pour espérer récupérer des résultats. Mais pour avoir ces résultats, il faut définir des variables dans lesquelles stocker les valeurs des colonnes retour de la requête. On fait cela grâce en appliquant la fonction bind\_result sur le statement, qui prend les variables à binder. On peut ensuite aller chercher nos résultats comme on sait le faire en utilisant les variables déclarées au bind\_result.

| <?php  // On prépare la requête $stmt = $mysqli->prepare("SELECT quartier FROM ville WHERE nom = ?"); // On lie les valeurs de nos paramètres $stmt->bind\_param("s", $nom); // On exécute la requête $stmt->execute(); // On lie les paramètres de retour $stmt->bind\_result($quartier); // On peut faire ce qu'on veut faire while ($stmt->fetch()) {  echo $quartier; } // On ferme $stmt->close(); |
| --- |

Il est à noter que lors d’une requête préparée, mysqli échappe automatiquement les paramètres pour éviter une injection SQL.

## PDO

PDO est une autre librairie qui permet de gérer des bases de données. Elle offre une interface qui permet de se connecter à n’importe quel type de base de données : MySQL, mais aussi PostgreSQL, SQLite, … Pour chacune de ces bases, elle propose, globalement, les mêmes fonctionnalités.

### La connexion

La connexion avec le gestionnaire PDO se fait là aussi à l’instanciation de la classe. Seulement, PDO prend un DSN et non des paramètres séparément.

Le DSN est une chaîne de caractères qui décrit la base de données à laquelle se connecter, et cela ressemble à ça : 'mysql:host=localhost;dbname=machin'. On a pour commencer le SGBD utilisé (ici, MySQL), l’hôte (ici, localhost) puis enfin le nom de la base (ici, machin). On peut y trouver d’autres informations si nécessaire mais nous n’en aurons pas besoin.

PDO prend donc le DSN en premier paramètre, suivi du nom d’utilisateur et du mot de passe pour se connecter à la base :

| $pdo = new PDO('mysql:host=localhost;dbname=test', 'user', 'passwd'); |
| --- |

Si cette instruction échoue, PDO peut renvoyer des exceptions (des objets qui servent à indiquer des erreurs). Ces exceptions ont besoin d’être “attrapées” dans le code, sinon elles provoquent des erreurs.

Aussi, je vous mets un morceau de code ici, que je ne vais pas détailler car nous le verrons ensemble dans le module de Programmation Orientée Objet, mais qui sert, justement, à gérer les exceptions :

| <?php try {  $pdo = new PDO('mysql:host=localhost;dbname=test', 'user', 'passwd'); } catch (PDOException $e) {  echo 'Erreur : ' . $e->getMessage() . '<br/>';  die(); } ?> |
| --- |

Ici, on indique, littéralement, à PHP que s’il attrape une exception, il met fin au script en affichant une erreur.

### Les requêtes simples

Sur ce point, PDO fonctionne comme mysqli, avec une fonction query qui renvoie un jeu de données :

| <?php  $result = $pdo->query("SELECT truc FROM table LIMIT 10"); ?> |
| --- |

### Parcourir les résultats

PDO fonctionne sur le même principe que mysqli pour récupérer les résultats de requête, avec une fonction fetch qui s’exécute sur le *statement* et qui renvoie un tableau contenant les données du prochain jeu de données, ou bien false s’il n’y a pas de prochain jeu de données.

| <?php while ($row = $result->fetch()) {  echo $row['truc']; } ?> |
| --- |

On peut également, pour PDO, parcourir les résultats d’un query grâce à une boucle foreach directement sur le *statement.*

| <?php // Exemple tiré de la documentation officielle $sql = 'SELECT name, color, calories FROM fruit ORDER BY name'; foreach ($conn->query($sql) as $row) {  print $row['name'] . "\t";  print $row['color'] . "\t";  print $row['calories'] . "\n"; } ?> |
| --- |

### Les requêtes préparées

Là encore, on a un fonctionnement similaire à mysqli, avec des fonctions prepare et execute. La méthode prepare fonctionne exactement comme celle de mysqli. La seule différence notable, c’est qu’on peut noter les paramètres par des “?”, ou bien les nommer avec des “:”. Par exemple :

| <?php  $stmt = $pdo->prepare("SELECT quartier FROM ville WHERE nom = ?"); // Ou $stmt = $pdo->prepare("SELECT quartier FROM ville WHERE nom = :nom"); ?> |
| --- |

Il y a ensuite plusieurs manières d’appeler execute, que l’on va voir ensemble. Dans tous les cas, les paramètres fournis seront échappés pour éviter les injections SQL.

**Avec des bind de paramètres**

On peut, à la manière de mysqli, binder les paramètres en PDO aussi. Il y a pour cela deux fonctions, bindParam et bindValue, qui n’ont pour différence que le fait que bindValue passe le paramètre par valeur (comme son nom l’indique) tandis que bindParam le passe par référence. Pour ces deux méthodes, on passe le nom du paramètre nommé, ou la position du “?” que l’on souhaite binder, ainsi que la valeur du paramètre et on peut également préciser le type de paramètre (si ce n’est pas précisé, ce sera considéré comme *string*).

| <?php $stmt = $pdo->prepare("SELECT quartier FROM ville WHERE nom = ? AND pays = ?"); $stmt->bindParam(1, $nom, PDO::PARAM\_STR); // Le 1er "?" $stmt->bindValue(2, $pays); // Le 2nd "?" $stmt->execute();  $stmtNom = $pdo->prepare("SELECT quartier FROM ville WHERE nom = :nom AND pays = :pays"); $stmtNom->bindParam(':pays', $pays, PDO::PARAM\_STR); $stmtNom->bindValue(':nom', $nom); $stmtNom->execute(); ?> |
| --- |

**Avec un array de paramètres**

La méthode execute peut aussi prendre en paramètre un tableau, dont les éléments remplaceront les paramètres de la requête. Le tableau fourni doit **impérativement** comporter autant d’éléments que la requête ne compte de paramètres. Sinon, une erreur sera levée.

On peut choisir de fournir un tableau associatif, auquel cas les clefs devront correspondre aux noms des paramètres (il faut donc également que les paramètres de la requête soient nommés). On peut également choisir de fournir un tableau non associatif, auquel cas les paramètres seront liés un par un, du premier au dernier, dans l’ordre défini par le tableau.

Dans le cas de l’utilisation de la méthode execute avec un tableau, tous les paramètres seront bindés en utilisant PDO::PARAM\_STR (donc interprétés comme des *string*).

| <?php $stmt = $pdo->prepare("SELECT quartier FROM ville WHERE nom = :nom AND pays = :pays"); $stmt->execute(array(  'nom' => 'New York',  'pays' => 'USA' ));  // Ou encore $stmt = $pdo->prepare("SELECT quartier FROM ville WHERE nom = ? AND pays = ?"); $stmt->execute(array('New York', 'USA')); ?> |
| --- |

# 

# Récupérer des données de requêtes (les superglobales)

#### Aparté sur les requêtes HTTP

| HTTP, c’est le nom du protocole utilisé pour les requêtes entre client et serveur dans le cadre d’une requête web. Ce protocole est assez simple et ce qu’il faut savoir c’est qu’une requête est en partie composée de ce qu’on appelle des Headers (des en-têtes). Ces Headers sont en fait des paires clef-valeur qui décrivent la requête ou la réponse, afin de transmettre des informations.  L’en-tête la plus visible des yeux d’un utilisateur, c’est le Status Code, qui renvoie un code permettant de savoir comment s’est déroulé la requête. Il vaut 200 lorsque tout va bien, mais peut aussi valoir 404 lorsque la page n’est pas trouvée ou encore 500 lorsque le serveur a rencontré une erreur. Vous avez la liste des codes de réponse sur [Wikipédia](https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP). |
| --- |

Il existe des variables particulières remplies avec des valeurs de la requête HTTP émise par le client : les superglobales. Elles sont définies avant l’appel du script par le serveur, et contiennent toutes les données sous forme de tableau associatif. Elles sont toutes de la forme $\_VARIABLE.

## GET

La superglobale $\_GET contient les paramètres fournis dans l’URL (par exemple lorsqu’on soumet un formulaire avec la méthode GET). Par exemple, si notre URL est la suivante : http://monsite.fr/mapage.php?truc=1&bidule=intéressant&machin=42, alors nous aurons la variable $\_GET qui vaudra :

| <?php   // Dans notre cas  $\_GET = array(  'truc' => 1,  'bidule' => 'intéressant',  'machin' => 42  ); ?> |
| --- |

#### Aparté sur la confiance utilisateur

| Les valeurs contenues dans la variable GET, mais aussi celles de **toutes** les variables superglobales, proviennent de l’utilisateur. Aussi, il ne faut absolument **jamais** les utiliser sans vérifier au préalable qu’elles contiennent une valeur qui correspond à ce que l’on attend.  Pour le cas de la variable $\_GET, elle contient des valeurs qui proviennent de l’URL. On peut imaginer par exemple que l’on ait un paramètre correspondant au nombre d’articles que l’on doit afficher sur notre page. Imaginez à présent que l’utilisateur modifie l’URL pour mettre une valeur qui n’a pas de sens, ou encore une valeur tellement élevée que notre serveur devrait tourner 100 ans pour afficher autant d’articles.  On peut être en droit d’imaginer l’utilisateur comme quelqu’un de bienveillant. Mais ce serait oublier les quelques-uns qui ne le seront pas et qui, eux, poseront souci vis-à-vis de notre application.  Il faut donc **toujours** vérifier l’intégrité des données que nous recevons : leur type, mais également leur valeur. |
| --- |

## POST

La superglobale $\_POST contient les valeurs des champs de formulaires qui ont l’attribut method à la valeur POST. Elle se comporte exactement comme $\_GET.

## FILE

La superglobale $\_FILE contient les informations des fichiers envoyés par le client via formulaire. Dans la forme, il s’agit d’un tableau associatif ayant pour clef le nom du fichier tel que défini dans le formulaire, et comme valeur un tableau reprenant les informations du fichier en question.

Au niveau des informations, on a retrouver :

* name : Le nom original du fichier (sur la machine du client)
* type : Le type MIME du fichier
* size : La taille du fichier (en octets)
* error : Un code d’erreur lié au chargement du fichier
* tmp\_name : Le nom temporaire du fichier.

Les fichiers qui arrivent ont une adresse temporaire sur le serveur. Il faut manuellement vérifier que les fichiers nous correspondent et ont été correctement téléversés avant de déplacer les fichiers dans notre arborescence.

## SESSION

Une session est un petit fichier, stocké sur le serveur, qui contient des informations qui se transmettent de page en page. Elle peut contenir, par exemple, les informations de l’utilisateur connecté, etc. Ces données sont disponibles dans la superglobale $\_SESSION, et persistent jusqu’à la fermeture du navigateur par le client.

## COOKIE

Les cookies sont également des fichiers transmis à chaque requête par l’utilisateur. En fait, c’est la même chose qu’une session, sauf que c’est stocké sur l’ordinateur du client et non sur le serveur. La durée de vie est également différente, puisqu’un cookie peut “vivre” pendant une durée très longue. Les données des cookies sont accessibles en lecture via la superglobale $\_COOKIE. On peut également les écrire avec la fonction setcookie($nom, $valeur, $expiration, ...) qui prend en général 3 paramètres, à savoir dans l’ordre le nom, la valeur et le timestamp d’expiration du cookie.

## Les autres

Il existe d’autres superglobales, comme $\_REQUEST ou encore $\_SERVER, qui contiennent, comme leur nom l’indique, des données sur les Headers de la requête, ou encore sur le serveur qui héberge l’application. Plus d’infos dans la documentation officielle des variables [$\_SERVER](https://www.php.net/manual/fr/reserved.variables.server.php) et [$\_REQUEST](https://www.php.net/manual/fr/reserved.variables.request.php).

# Interagir avec le système de fichiers

Le système de fichiers désigne l’arborescence des fichiers sur le serveur, l’ordinateur, la machine. On va pouvoir interagir avec en PHP, pour des opérations de lecture ou d’écriture notamment.

Nous allons voir deux ensembles de fonctions permettant de le faire.

## Les fonctions file\_\*

Nous allons parler ici des fonctions file\_get\_contents et file\_put\_contents. Comme leur nom l’indique, ces fonctions servent à récupérer (get) et insérer (put) du contenu dans des fichiers.

file\_get\_contents prend en paramètre (au moins) le chemin du fichier et renvoie une chaîne de caractères qui correspond au contenu du fichier.

fil\_put\_contents prend en paramètre (au moins) le chemin du fichier ainsi que le contenu (tableau ou chaîne de caractères) à y inscrire et renvoie le nombre d’octets écrits.

| <?php   // $contenu contient tout le fichier  // sous forme de chaîne de caractères  $contenu = file\_get\_contents('chemin/vers/fichier.txt');    // On crée un nouveau fichier avec le même contenu que le premier  file\_put\_contents('chemin/vers/nouveau\_fichier.txt', $contenu); ?> |
| --- |

## Les fonctions f\*

Avec les fonctions que nous allons voir, nous n’allons pas manipuler les fichiers comme de la donnée mais plutôt… comme des fichiers.

Ainsi, nous allons les ouvrir, y écrire ou en lire le contenu, puis les fermer. Nous disposons donc des fonctions fopen, fwrite, fread et fclose… Simple comme bonjour !

On ouvre un fichier dans un certain mode avec fopen (que je vous laisse découvrir dans la [documentation](https://www.php.net/manual/fr/function.fopen.php)), et cela nous renvoie un objet que nous pouvons ensuite passer à fread et fwrite. fread prend le nombre d’octets à lire et renvoie ce qui est lu. fwrite prend la chaîne à écrire et renvoie le nombre d’octets écrits. fclose ferme le fichier. Si nous ne souhaitons pas lire le début ou écrire à la fin, il existe aussi la fonction fseek pour “déplacer le curseur virtuel” dans le fichier.

| <?php  // On ouvre le fichier en lecture et écriture  $file = fopen('chemin/vers/fichier.txt', 'r+');  // On avance de 40 octets  fseek($file, 40);  // On lit 15 octets  $contenu = fread($file, 15);  // On avance à la fin du fichier  fseek($file, filesize('chemin/vers/fichier'));  // On écrit notre contenu  fwrite($file, $contenu);  // On ferme notre fichier  fclose($file); ?> |
| --- |

Toutes ces fonctions renvoient false lorsqu’elles échouent. Ainsi, on peut vérifier leur bon fonctionnement.

## Les autres fonctions

Il est intéressant de noter qu’il existe également les fonctions du terminal de commandes Linux. On retrouve donc les mkdir, touch, unlink, chmod, … bref toutes les fonctions qui peuvent nous servir. Je vous laisse aller voir la [documentation](https://www.php.net/manual/fr/ref.filesystem.php) pour en voir l’utilité et l’utilisation.

# Échanger des informations

Un site web fait souvent partie d’un système d’information, c’est-à-dire selon [Wikipédia](https://fr.wikipedia.org/wiki/Syst%C3%A8me_d%27information) “un ensemble organisé de ressources qui permet de collecter, stocker, traiter et distribuer de l'information, en général grâce à un réseau d'ordinateurs”.

Un système d’information doit donc pouvoir **communiquer** de l’information. Il s’agit d’en recevoir, mais également d’en émettre. Cela implique d’être capable d’**écrire** de manière **conventionnée** l’information, et de l’**envoyer** en suivant un **protocole** défini.

En soi, dans ce module, nous avons déjà vu comment **stocker** et **traiter** de l’information : par les bases de données et par la logique du langage qui nous permet de les transformer à l’envi. Nous ne verrons pas comment **collecter** des données, car le souci principal dans ce cas est de trouver une **source** de données. Mais les problématiques seraient alors les mêmes que dans le cas qui nous intéresse maintenant, à savoir **distribuer** des données.

## Formater des données

Nous n’allons pas étudier **tous** les formats qui existent, il y en a bien trop. Nous allons nous intéresser à l’un d’entre eux : le JSON. Mais sachez qu’il existe peut-être des fonctions natives à PHP pour les autres formats de données et que si ce n’est pas le cas, **d’autres personnes** ont sans doute créé des librairies pour pallier le manque.

### Le JSON

Le JSON (pour JavaScript Object Notation) est **le** format de données d’aujourd’hui pour les communications web. On le retrouve partout, qu’il s’agisse de communiquer avec une API, avec un script côté client, avec des bases de données, …

La syntaxe JSON est très simple, et fonctionne sur des paires clef-valeur. La clef est une chaîne de caractères écrite entre doubles quotes. Les valeurs, quant à elles, peuvent être :

* Des nombres, écrits tels quels et dont le marqueur décimal est le point (Ex : 1.2),
* Des booléens, écrits tels quels (Ex : true),
* Des chaînes de caractères, entre doubles quotes (Ex : "Lorem ipsum."),
* La valeur null, écrite telle quelle,
* Des tableaux contenant d’autres valeurs, encadrés par des crochets et séparant les valeurs par des virgules (Ex : [1, 3.1415, "truc", false, null, 42]),
* Des objets contenant des paires clef-valeur, encadrés par des accolades et séparants les paires clef-valeur par des virgules (Ex : {"a": 3, "b": null, "c": [42, 3]})

Un exemple de JSON :

| {  "menu": {  "id": "file",  "value": "File",  "popup": {  "menuitem": [  { "value": "New", "onclick": "CreateNewDoc()" },  { "value": "Open", "onclick": "OpenDoc()" },  { "value": "Close", "onclick": "CloseDoc()" }  ]  }  } } |
| --- |

En PHP, on va principalement utiliser 2 fonctions pour traiter le JSON : json\_encode et json\_decode qui, comme leurs noms l’indiquent, servent respectivement à passer de PHP à JSON et de JSON vers PHP. Je vous renvoie vers la documentation ([celle de json\_encode](https://www.php.net/manual/fr/function.json-encode.php) et [celle de json\_decode](https://www.php.net/manual/fr/function.json-decode.php)) pour plus de détails sur les options de ces fonctions.

| <?php  $json = '{DU JSON ICI}';  $objet = json\_decode($json);  $tableau\_associatif = json\_decode($json, true);   // On modifie notre json  $tableau\_associatif['clef'] = 'Nouvelle valeur';   // On écrit un nouveau JSON  $nouveau\_json = json\_encode($tableau\_associatif); ?> |
| --- |

## Communiquer

Là encore, je ne vais pas m’attaquer à **tous** les moyens de communication existant, mais ne m’attarder que sur un seul : l’email. Cependant, encore une fois, une solution native ou additionnelle existe sans doute si vous cherchez à faire quelque chose.

### Les emails

Je ne vais pas faire la présentation des emails en détails, mais il faut savoir que PHP n’est pas le seul à intervenir, et que cela utilise au moins 2 protocoles : un pour l’envoi (SMTP) et un pour la réception (IMAP ou POP).

En PHP, nativement, tout ce qui tourne autour des emails est principalement axé sur une fonction : mail.

#### Aparté sur le RGPD

| Le RGPD (**R**èglement **G**énéral pour la **P**rotection des **D**onnées) est une réglementation à l’échelle européenne qui a pour but de protéger et réglementer l’utilisation de données personnelles des utilisateurs.  Il indique notamment que l’on doit demander le consentement de l’utilisateur avant toute utilisation de ses données à des fins commerciales ; que l’on doit indiquer l’utilisation de l’adresse email si on la récolte ; que l’on doit lui permettre de récupérer l’ensemble des données récoltées par l’application dans un format exploitable (par exemple, JSON).  Tout site, toute application qui ne respecte pas ces lois s’expose à des sanctions (notamment économiques). |
| --- |

# Liens utiles

* La documentation officielle de PHP : <https://www.php.net/manual/fr/> (en français)
* OpenClassrooms : [Concevez votre site web avec PHP et MySQL](https://openclassrooms.com/fr/courses/918836-concevez-votre-site-web-avec-php-et-mysql) (en français)
* S'entraîner à l’algorithmique : [Un Doodle de Google](https://www.google.com/doodles/celebrating-50-years-of-kids-coding?hl=fr)
* Le dossier contenant toutes les ressources de ce module : [Google Drive](https://drive.google.com/drive/folders/1D3zdeY1Za8BI8xXeHvlVrhHW7J3HN5jm?usp=sharing)
* Le dépôt contenant tous les exercices et leurs corrections : [GitHub](https://github.com/2alheure/formawave/tree/master/PHP7)

# Glossaire

Glossaire organisé par ordre de rencontre :

* langage
* script
* compilation
* interpréter
* exécuter
* serveur
* client
* variabilité
* ressource
* instruction
* faiblement typé
* déclaration
* affectation
* opérateur d’affectation
* opérateur
* opérateur arithmétique
* opérande
* opérateur de comparaison
* booléen
* opérateur logique
* assertion logique
* opérateur de négation
* chaîne de caractères
* interprétation des variables
* concaténation
* paramètre
* caractère d’échappement
* clef-valeur
* indexer
* tableau associatif
* constante
* structure
* condition (structure conditionnelle)
* évaluer
* boucle (structure itérative)
* itération
* opérateur d’incrémentation
* opérateur de décrémentation
* fonction
* corps de fonction
* argument
* contrainte
* typage
* portée des variables
* passage par valeur
* passage par référence
* signature de fonction
* token
* algorithmique
* algorithme
* complexité
* architecture de code
* design pattern
* instruction du langage
* Programmation Orientée Objet
* attribut
* méthode
* instancier
* injection SQL
* échapper une variable
* binder
* DSN
* SGBD
* exception
* Header
* intégrité
* système de fichiers
* système d’information
* RGPD