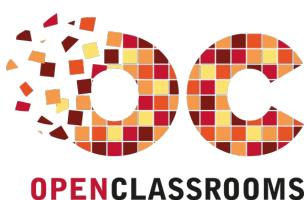


# Amusons-nous avec le PHP

Par `Haku



[www.openclassrooms.com](http://www.openclassrooms.com)

*Licence Creative Commons 2.2.0  
Dernière mise à jour le 29/10/2012*

## Sommaire

Sommaire .....	2
Lire aussi .....	3
Amusons-nous avec le PHP .....	5
Partie 1 : Les bases du PHP .....	6
Introduction .....	6
PHP, le Web, qu'est-ce donc ? .....	6
La programmation, c'est quoi ? .....	8
Pour apprendre, quelle attitude adopter ? .....	9
La boîte à outils du programmeur .....	10
L'invité surprise : l'éditeur de code .....	11
Apache et PHP sous Windows .....	13
Apache et PHP sous GNU/Linux, Mac OS X et les autres .....	21
Configurer pour mieux régner .....	23
Configuration de PHP .....	23
Configuration d'Apache .....	24
Premiers pas... .....	24
Les balises PHP .....	25
Les commentaires .....	26
Les variables, le cœur du PHP .....	28
Première approche .....	29
Les variables à la sauce PHP .....	29
Valeurs et opérateur d'affectation .....	30
Une histoire de typage .....	31
Les types primitifs .....	32
Les entiers, alias INT (ou INTEGER) .....	32
Les nombres à virgules, alias FLOAT .....	32
Les vrais ou faux, alias BOOLEEN .....	33
Les chaînes de caractères, alias STRING .....	33
Un type à part : NULL .....	34
Quel est ton type ? .....	34
Les opérateurs : partie I .....	36
Retour de l'opérateur d'affectation et introduction des expressions .....	37
Les opérateurs arithmétiques .....	38
Opérateur arithmétique d'addition .....	38
Opérateur arithmétique de soustraction .....	39
Opérateur arithmétique de multiplication .....	39
Opérateur arithmétique de division .....	39
Calculs complexes et modulo .....	39
Un cinquième opérateur arithmétique : le modulo .....	41
Les expressions, un peu de pratique ! .....	41
Le vrai visage de var_dump() .....	43
Les opérateurs : partie II .....	45
Opérateurs arithmétiques d'affectation et opérateurs d'incrémentation .....	45
Opérateurs de comparaison .....	47
Opérateurs logiques .....	48
Les structures de contrôle .....	50
Structures conditionnelles .....	51
Évaluation .....	54
Autre structure conditionnelle .....	54
Structure de boucle : partie I .....	57
Structure de boucle : partie II .....	58
Les fonctions .....	61
Première approche .....	61
Votre première fonction .....	61
Souvenir, souvenir .....	63
Les fonctions, c'est trop fort ! .....	65
Les chaînes de caractères, un casse-tête .....	68
Bref rappel et nouvelle structure .....	69
Différence entre guillemets et apostrophes .....	70
Nouvel opérateur : concaténation .....	72
Le type manquant : les arrays .....	75
Faisons connaissance .....	75
Qui es-tu vraiment ? .....	75
Tu commences à me plaire.....	77
Des fonctions et structures bien pratiques .....	79
Es-tu là, petite valeur ? .....	79
Es-tu là, petite clé ? .....	80
Quelle est ta clé ? .....	80
Dis-moi tout.....	81
Je veux des arrays ! .....	81
Découpage .....	81
Collage .....	82
Derniers mots et structure de contrôle .....	82
Les faux jumeaux .....	85

Bons plans .....	87
Présenter son code, tout un art .....	87
La bible du codeur PHP .....	90
Le transtypage, c'est pratique .....	92
Transtypage en booléen .....	94
Transtypage en entier .....	95
Transtypage en nombre à virgule .....	95
Transtypage en chaîne de caractères .....	95
Transtypage en array .....	96
Transmettre des variables .....	96
35, rue machin .....	97
Gardons la forme ! .....	99
Les zones de texte .....	100
Les zones de texte multi-lignes .....	100
Les cases à cocher .....	100
Les listes .....	101
Les boutons .....	101
Le champ caché .....	101
Le bouton de validation .....	102
Un bouton plus coloré ! .....	102
Aspiration ! .....	102
Never Trust User Input .....	102
L'enfer des guillemets magiques .....	104
Je suis perdu ! .....	106
Premier TP : une calculatrice .....	108
Objectifs .....	109
Quelques conseils .....	109
Correction ! .....	110
Peut mieux faire ! .....	112
<b>Partie 2 : MySQL, PHP et MySQL .....</b>	<b>115</b>
MySQL, qui es-tu ? .....	116
Parle-moi de toi... .....	116
On s'organise ? .....	116
La console à tout âge .....	118
Créons une base de données et une table pas à pas .....	119
La base avant tout .....	120
On passe à table ? .....	123
Le plat de résistance ! .....	126
Gérons nos tables ! .....	129
Une histoire d'entiers .....	130
Des types en pagaille .....	133
Et la gestion ? .....	136
SELECT, ou comment récupérer des données .....	140
MySQL et les types .....	141
Les données venant d'une table .....	146
Tri, ordre et limitation .....	149
Mettons de l'ordre .....	149
Filtrons tout ça .....	153
Vertical Limit ! .....	154
Supprimer, modifier et insérer des données .....	157
DELETE s'exhibe .....	158
UPDATE en folie .....	160
INSERT sous les feux de la rampe .....	160
PHP et MySQL : des alliés de poids .....	162
Retour en arrière .....	163
Dis bonjour à MySQL ! .....	163
Exploiter les ressources .....	167
Second TP : un visionneur de bases de données et de leurs tables .....	173
Objectifs et cycle de développement .....	173
L'étape d'algorithmie .....	174
Quelques outils .....	175
Correction .....	177
Fonctions, conditions et quelques opérateurs SQL .....	180
Les fonctions, de bons souvenirs .....	181
Un cas particulier : NULL .....	182
Les conditions .....	184
Un peu de repos avec les opérateurs .....	186
La gestion des dates .....	188
Qui peut me donner la date ? .....	189
Formatage et opérateurs simples .....	189
Des fonctions, en veux-tu en voilà ! .....	192
Des fonctions d'extraction .....	192
Des fonctions de calcul .....	194
La magie des intervalles .....	195
Bataillons, à vos rangs... fixe ! .....	196
Les intérêts et les limitations du regroupement .....	197
La clause magique .....	200
Les fonctions d'agrégation .....	201
Calculer une somme avec SUM(Expr) .....	201
Calculer une moyenne avec AVG(Expr) .....	202

Trouver le maximum et le minimum avec MAX(Expr) et MIN(Expr) .....	202
La dernière des fonctions : COUNT() .....	203
ROLLUP et HAVING .....	203
L'ordre d'exécution des clauses .....	206
<b>Troisième TP : un livre d'or .....</b>	<b>208</b>
Les spécifications .....	209
La boîte à outils ! .....	210
Correction .....	213
Discussion .....	218
<b>Les index et sous-requêtes .....</b>	<b>219</b>
Un index, pourquoi ? .....	220
Les différents index et leur gestion .....	220
Où placer des index ? .....	222
Surveiller les index .....	223
Les sous-requêtes .....	225
<b>Partie 3 : PHP, MySQL, allons plus loin .....</b>	<b>227</b>
<b>Retour sur les fonctions et les tableaux .....</b>	<b>228</b>
Des fonctions à nombre de paramètres variable .....	228
Les fonctions, des valeurs .....	230
Les tableaux .....	234
Fonctions utiles .....	239
Un pointeur interne .....	241
Tableaux et chaîne de caractères .....	242
Au cœur des variables .....	243
Des noms à dormir dehors .....	244
Les références .....	245
La portée des variables .....	249
Des variables non-déclarées ou non-initialisées .....	251
Au cœur du PHP .....	252
Les tableaux .....	256
La pile .....	257
Pour terminer .....	259
Dépasser la portée .....	259
Des types à part .....	260
foreach et les références .....	261
Les variables statiques .....	264
Ne pas abuser des références .....	265
Fragmenter son code .....	266
Des scripts dans le script .....	267
Une boucle infinie .....	268
Retourner une valeur .....	269
L'include_path .....	269
Des fichiers critiques .....	270
Les dangers de l'inclusion .....	271
\$_GET et les inclusions .....	271
Les espaces de noms .....	272
Créer des espaces de noms .....	274
Utiliser les espaces de noms .....	276
Les nombres .....	280
Comment écrire un nombre ? .....	280
Des nombres équivalents .....	281
Les opérateurs bits à bits .....	282
Les flags .....	284
Des bases plus pratiques .....	286
Fonctions de changement de base .....	287
PHP et les erreurs .....	289
Les erreurs .....	289
Gérer les erreurs .....	291
Affiner la gestion .....	293
debug_backtrace() .....	295
<b>Partie 4 : Annexes .....</b>	<b>299</b>
<b>Histoire de ce tutoriel .....</b>	<b>299</b>
Remerciements .....	299
Historique des mises à jour .....	299
Suite du tutoriel .....	300



## Amusons-nous avec le PHP

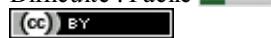


Par

'Haku

Mise à jour : [29/10/2012](#)

Difficulté : Facile



Vous avez cinq minutes à perdre ?

Vous voulez lire ?

Vous voulez apprendre le PHP ?

Si la réponse à au moins une de ces questions est positive, vous êtes au bon endroit 😊.

Ce tutoriel n'a pas la prétention de faire de vous des pros du PHP, il n'est là que comme base, comme un point de départ et un soutien qui vous permettra, je l'espère, d'évoluer ensuite par vous-mêmes.

Mais on ne se limitera pas à des choses simples pour autant, et je ne vous ménagerai pas, il faudra s'accrocher pour suivre sans quoi vous serez totalement largués 😱.

Un petit détail qui vaut son pesant d'or : ce tutoriel n'engage que moi ; je vous montrerai ma vision du PHP, elle n'est pas universelle mais personnelle. Par conséquent, certaines parties de ce tutoriel pourraient s'opposer à d'autres tutoriels, articles ou autres textes.

Il est très fortement conseillé d'avoir des bases en (X)HTML avant de s'engager dans ce tutoriel ; si vous n'en avez pas, je vous conseille de lire le [tutoriel de M@teo21](#) qui en parle.

## Partie 1 : Les bases du PHP

Avant de se lancer dans des scripts de folie, il faut commencer par le début : la syntaxe du PHP.

Cette partie est **indispensable**, si vous ne connaissez pas les bases du langage sur le bout des doigts, vous perdrez un temps fou par la suite.

### Introduction

Avant d'entamer une lutte sans merci, il est de bon usage de s'informer et de se préparer. Comme l'a probablement dit un très grand homme dont j'ignore le nom et dont je ne peux prouver l'existence, « la moitié du combat se joue pendant sa préparation ». Au menu du jour, vous trouverez une courte introduction sur PHP et surtout sur le fonctionnement global de celui-ci. Le tout sera agrémenté de quelques lignes sur ce qu'est la programmation et en ce qui concerne dessert, je vous laisse le découvrir.

#### PHP, le Web, qu'est-ce donc ?

Le PHP, pour *PHP : Hypertext Preprocessor* est un langage de programmation. Il y en a beaucoup d'autres, comme le C, le Java, l'OCaml,... mais on peut dire du PHP que c'est un langage de programmation orienté pour le Web, ou encore pour les sites internet.

Pour la petite histoire, le PHP est né de la main de Rasmus Lerdorf en 1994. Au départ très basique, il n'a été rendu public qu'en 1995 sous le nom de PHP/FI. Par la suite, le développement passa des mains de Rasmus Lerdorf à celles de deux étudiants — Zeev Suraski et Andi Gutmans — qui fondèrent par la suite *Zend Technologies* pour promouvoir PHP. Si vous développez des applications avec PHP, vous entendrez sans aucun doute parler de Zend, que ce soit via leur *framework*, leur environnement de développement ou encore leur serveur, respectivement *Zend Framework*, *Zend Studio* et *Zend Server*.

Non content d'être un langage de programmation, PHP est un langage **interprété**. Quand vous utilisez une application, sous Windows par exemple, vous double-cliquez sur le programme pour qu'il se lance ; le programme s'exécute alors, votre ordinateur peut directement exécuter le programme. Pour le PHP, c'est un peu différent. En effet, votre ordinateur ne comprend pas le PHP, il ne sait pas l'exécuter comme vous pouvez le voir à la figure 1.1. Pour que le PHP soit exécuté, il faut que le fichier qui contient le code PHP soit interprété par... l'interpréteur PHP.

Concrètement, l'interpréteur, c'est quoi ? Comme je l'ai dit, votre ordinateur ne peut pas exécuter de code PHP comme il exécute une application classique. Pour que votre code prenne vie, l'interpréteur — qui lui est une application exécutable — va le lire, le traduire en un dialecte intermédiaire — l'*Opcode* — et finalement demander à votre ordinateur d'exécuter les instructions correspondant à l'*Opcode* généré comme c'est illustré à la figure 1.2. Vous entendrez à nouveau parler d'*Opcode* dans la suite de ce cours, ne vous arrêtez donc pas là-dessus pour le moment, sachez juste que ça existe.

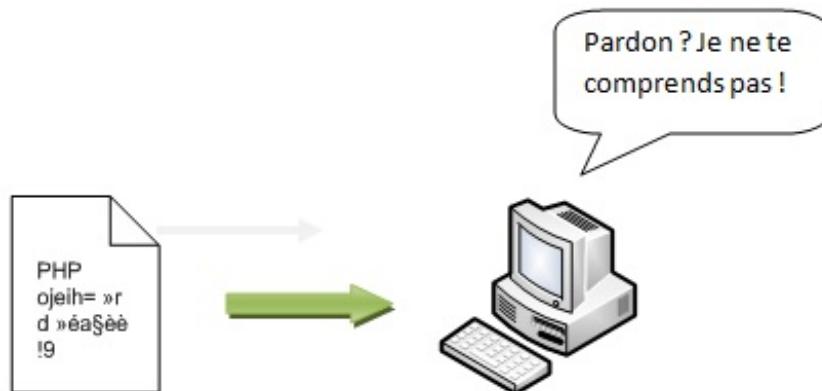


Figure 1.1 — Sans interpréteur, le code ne peut être exécuté

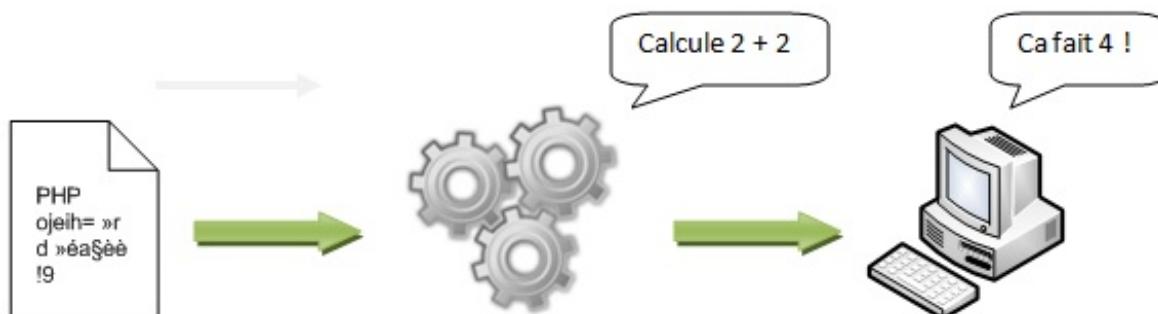


Figure 1.2 — Avec l'interpréteur, tout va bien !

### Le Web, comment est-ce que ça fonctionne ?

Avant tout, qu'est-ce que le Web ? Le Web — ou *World Wide Web* de son nom complet —, c'est le système qui vous permet de lire ce cours, qui vous permet d'accéder à du contenu, des pages d'un site, généralement accessible via Internet. Le Web est souvent confondu avec Internet, mais Internet ne se résume définitivement pas au Web : bon nombre d'autres systèmes, applications et protocoles utilisent Internet. Il est d'ailleurs assez triste que certaines entités — des entreprises par exemple — se basent sur cette confusion pour faire des offres plus attractives en nous martelant de slogans similaires à « Internet illimité » alors qu'en fait, ils ne nous offrent qu'un accès au Web, et encore, pas toujours très « illimité »... Mais je m'égare, ça n'est pas le sujet.

Le Web fonctionne sur un modèle dans lequel se trouvent deux intervenants : le **client** et le **serveur**. Vous pouvez parfaitement faire une comparaison avec un restaurant, c'est tout à fait pertinent et ça donne du bon goût. Quand vous êtes dans un restaurant, tranquillement assis, vous choisissez un plat, sifflez le serveur puis lui dites ce que vous désirez. Une fois la requête du client reçue, le serveur va faire sa popote interne pour finalement délivrer ce que vous avez commandé. Il y a donc deux messages, d'abord une requête du client vers le serveur, puis une réponse du serveur vers le client. Pour un site Web, c'est pareil. Votre navigateur — il joue le rôle de client — va envoyer une requête à un serveur — et plus exactement un serveur Web — qui y répondra après l'avoir traitée. Vous avez une illustration de ce dialogue à la figure 1.3.

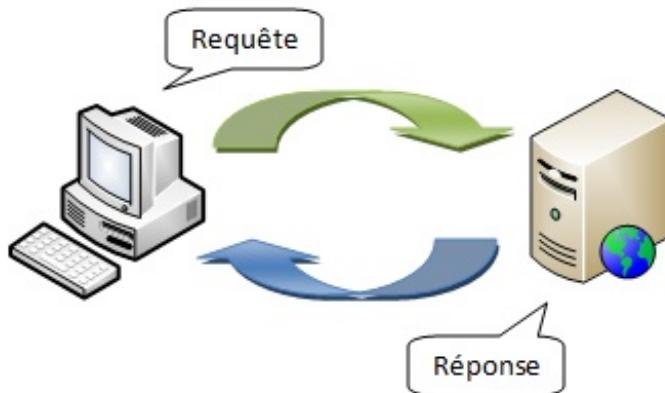


Figure 1.3 — Le modèle client-serveur

### Le serveur Web et l'interpréteur PHP, une histoire d'amour ?

Comme nous l'avons dit précédemment, pour que le PHP puisse vivre, il a besoin de son interpréteur. Un serveur Web n'étant pas un interpréteur PHP, ces deux éléments doivent entrer en relation à un moment donné. Mais non, malheureusement pour nos deux amis, ce n'est pas vraiment une relation amoureuse qu'ils entretiennent, ça serait plutôt une relation maître-esclave, l'esclave étant le pauvre interpréteur.

En fait, le serveur Web peut gérer différentes requêtes. Vous pouvez par exemple lui demander une image, une page Web statique — c'est-à-dire ne contenant par exemple que du XHTML et du CSS —, une page Web dynamique — contenant du PHP, du Python, etc. —, un document PDF et bien d'autres choses encore. Lorsque vous lui demanderez ce qui nous intéresse, une page Web contenant du PHP, le serveur Web le saura, il déterminera le type de contenu demandé à partir de la requête. Lorsque le serveur Web recevra une requête faisant intervenir PHP, il appellera l'interpréteur en lui donnant le fichier à interpréter et attendra la réponse de celui-ci qui sera ensuite transmise au client. Tout le monde est ainsi content à la figure 1.4 : le client a reçu ce qu'il attendait, le serveur a délivré le contenu, et l'interpréteur a été sauvagement abusé par le serveur. 😊

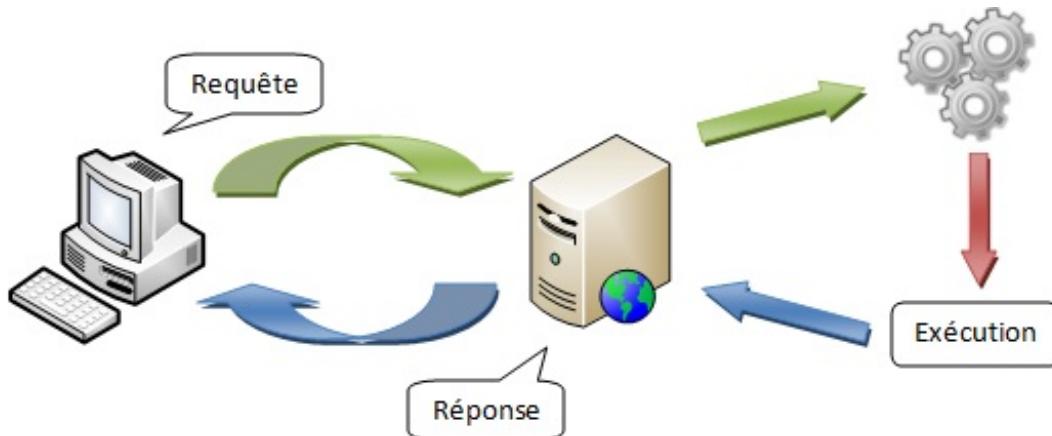


Figure 1.4 — Les interactions entre le client, le serveur et l'interpréteur

De ces interactions, nous pouvons déduire quelque chose qui a son importance : le PHP s'exécute du côté du **serveur**, il ne travaille que quand le serveur doit répondre à la requête, et jamais quand le serveur a déjà répondu à la requête. Si vous désirez afficher une horloge sur votre site Web, horloge qui se mettrait à jour toutes les secondes, serait-ce possible avec PHP ? Non, car l'horloge ne serait visible par le visiteur — l'utilisateur du navigateur Web — que quand le serveur a déjà envoyé la réponse, PHP ne pourrait donc plus intervenir. Pour réaliser cela, et plus généralement pour toutes les interactions entre le visiteur et une page Web déjà envoyée à celui-ci, on passera par un autre langage, **le Javascript**.

Après ce petit tour d'horizon de ce qu'est le PHP, du fonctionnement du Web et de l'interaction entre le serveur Web et l'interpréteur PHP, passons à ce dont nous aurons besoin pour travailler. Que chacun d'entre vous possède un serveur dédié en antarctique est un peu utopique, nous allons donc installer le serveur Web et PHP sur **votre** ordinateur. Oui, ça fonctionne, ne vous en faites pas.

Comme serveur Web, nous allons choisir le plus répandu et connu actuellement : *Apache HTTP Server* ou encore *Apache HTTPD*. Ce logiciel, ce serveur Web est édité par l'organisation à but non-lucratif éponyme, *l'Apache Software Foundation*. Cette organisation est également connue pour d'autres projets ainsi que pour leur licence, la licence Apache qui est libre et *open-source*. Mais avant de procéder à l'installation de ces logiciels, prenons le temps de faire connaissance avec la programmation.

## La programmation, c'est quoi ?

La programmation, tantôt qualifiée de mystique, tantôt qualifiée d'incroyablement compliquée, qu'est-ce donc vraiment ? Pour ma part, la programmation, c'est le quotidien. Non pas parce que j'ai le nez collé à mon PC huit heures par jour, mais bien parce que de la programmation, vous en faites au quotidien. Prenons un exemple qui devrait parler à tout le monde : la conduite d'une voiture. Avec une voiture, vous pouvez effectuer quelques opérations de bases, notamment :

- freiner,
- avancer
- et tourner

Il y a d'autres opérations, mais celles-ci suffiront pour l'exemple. Bien heureux dans votre voiture, vous arrivez au boulot, à l'école ou ailleurs si vous le désirez, dans tous les cas l'heure de se garer est arrivée. Par chance, vous trouvez une petite place entre deux voitures comme illustré à la figure 1.5.

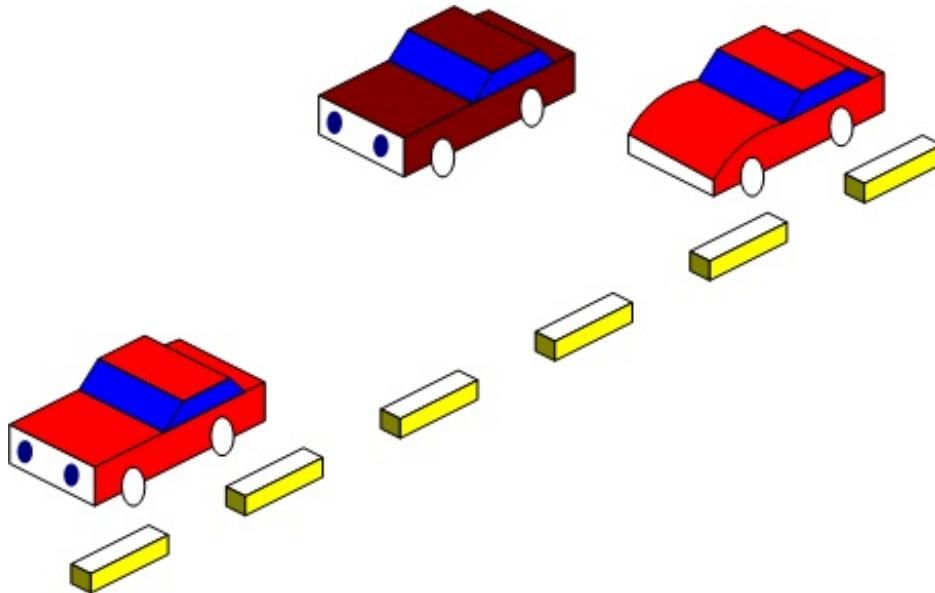


Figure 1.5 — Une jolie place où se garer

Vous voilà face à un problème, **comment** rentrer élégamment dans cet emplacement ? Résoudre un problème, y apporter une solution, c'est de la programmation. Et résoudre ce genre de problème, vous le faites tous les jours consciemment ou pas. C'est pour cela que j'aime dire que la programmation, c'est notre quotidien à tous. Pour résoudre ce problème, qu'avons-nous à notre disposition ? À vrai dire, pas grand chose, mais c'est amplement suffisant. Nous disposons en effet de quelques opérations de bases nous permettant de nous déplacer et nous possédons également quelques informations sur le problème, la principale étant que nous avons effectivement la place pour nous garer. Le tout est donc de parvenir à utiliser, à manipuler nos quelques opérations pour résoudre le problème. Le but n'étant pas de faire de vous des as du volant, voici les étapes de la solution représentée à la figure 1.6 :

1. nous avançons jusqu'à la hauteur de la voiture puis freinons ;
2. nous tournons les roues vers la gauche ;
3. nous avançons à l'envers, c'est-à-dire que nous reculons ;
4. lorsque nous formons un bel angle, nous mettons les roues vers la droite ;

5. et enfin nous freinons au bon moment pour terminer la manœuvre.

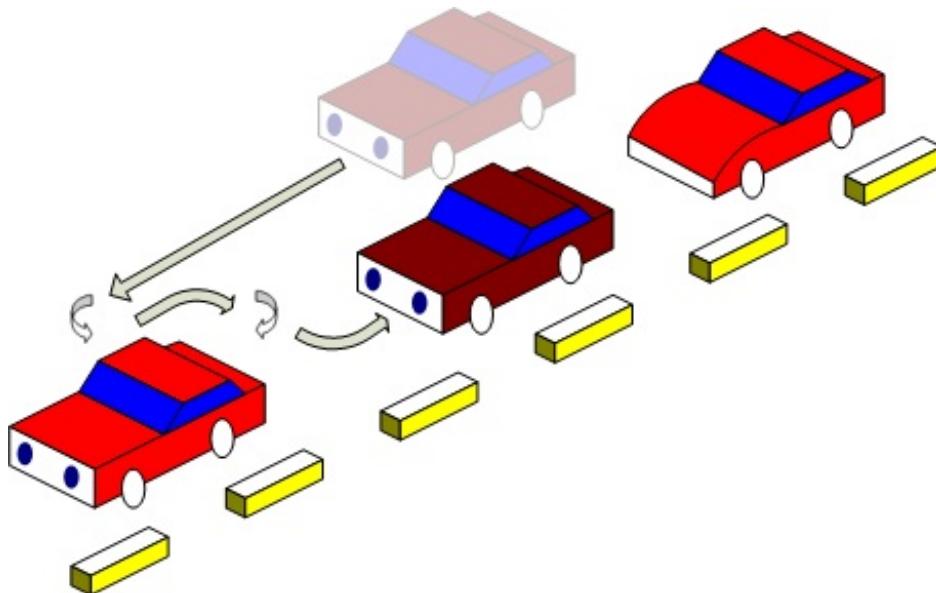


Figure 1.6 — La voiture garée !

Ce que nous avons fait ici, c'est **décrire, énoncer, exprimer** la solution à notre problème, et là nous sommes au cœur de ce qu'est la programmation : notre voiture nous offre certains **concepts** tels que l'accélération de la voiture ou son changement de direction et nous permet d'utiliser ces concepts via des **opérations élémentaires** telles que *avancer* ou *tourner*, ce qui nous permet **d'exprimer une solution à un problème**. Ces quelques opérations n'ont peut être l'air de rien, mais elles suffisent à exprimer un grand nombre de manœuvres que vous pourriez être amenés à réaliser avec une voiture. Une fois la solution exprimée, il ne reste plus qu'à la mettre en application en utilisant les pédales, volants et autres outils propre à votre voiture.

Un langage informatique, comme le PHP par exemple, fait exactement la même chose que cette voiture : il nous offre certains concepts nous permettant d'exprimer des solutions ainsi que des outils pour traduire ces solutions en code que l'interpréteur pourra faire exécuter. Il faut donc bien comprendre que la programmation n'est pas une seule et unique chose mais bien deux étapes distinctes : la résolution du problème et l'application de la solution. La résolution du problème, c'est la **conception du programme**, tandis que l'application de la solution, c'est son **implémentation**, la traduction en code donc.

Nous reviendrons plus tard sur la **conception** et l'**implémentation**, l'important c'est que vous reteniez que la programmation se déroule en deux étapes.

## Pour apprendre, quelle attitude adopter ?

Si vous décidez de poursuivre ce cours — et j'espère que vous serez nombreux dans ce cas —, quelle attitude adopter ? Comme mon introduction l'a sous-entendu, je ne suis pas ici pour faire du *fast-food* version programmation. Mon but, ce n'est pas de faire de vous des *roxxors pgm de la prog' en 24 heures chrono* mais bien d'apprendre, de comprendre. Vous devrez donc parfois être un peu patient avant de voir le fruit de vos efforts.

En ce qui concerne mes explications, je ne vais pas vous prendre pour des abrutis. Bien que nous débutions, nous ne sommes pas des bébés, nous avons la capacité de réfléchir et de nous débrouiller par nous-même. C'est pour cela qu'il m'arrivera souvent de vous lancer un « RTFM » — *Read The Fucking Manual* — ou la version plus délicate : cherchez par vous-même. Je ne veux pas faire de vous des assistés. Je vous aiderai, je vous guiderai, mais je ne vous donnerai pas tout tout cuit au fond du bec. Vous serez donc sans le moindre doute contraint à un moment ou à un autre de **chercher par vous-même**. Cette capacité, cette débrouillardise est une qualité très importante, je veux donc que vous l'utilisiez, et pour cela je ne dois évidemment pas faire de vous des professionnels de l'assistanat.

Bien sûr, il arrivera parfois que même en cherchant un certain temps, vous ne trouviez pas de solution. Dans ce cas, vous serez amenés à demander de l'aide, sur les forums du site du zéro par exemple. Si vous le faites, ou plutôt, quand vous le ferez, n'oubliez pas certaines choses.

### Le titre du sujet

Imaginons que je rencontre un problème et que ma recherche ne m'a rien apporté. Je me rends dans un forum traitant du PHP et je crée un nouveau sujet.

La première chose à laquelle vous devez faire attention, c'est le titre. Il a pour but de décrire votre sujet ; ainsi, les personnes visitant le forum sauront si elles sont susceptibles de vous aider sans avoir à lire le message.

Il est donc impératif de choisir un titre qui décrit votre sujet. Si vous mettez quelque chose comme « Gros problème », « Besoin d'aide urgente », etc., on ne sait absolument pas quel est l'objet du sujet.

Petit détail plutôt amusant, certaines personnes mettent un titre du style « Problème avec PHP ». On ne peut pas dire que ça ne

décrit pas le sujet, mais ne trouvez-vous pas logique que dans un forum consacré au PHP on parle de... PHP ? C'est juste pour dire qu'il est inutile de spécifier que vous parlez de PHP.

On a vu ce qu'il ne fallait pas faire, mais comment choisir un bon titre, finalement ?

Eh bien il n'y a pas de méthode magique, il faut réfléchir. Si j'ai un problème en voulant afficher un texte, qu'est-ce que je pourrais mettre comme titre ?

Si je mets « Problème PHP », ça ne sert à rien. Par contre, un titre comme « Problème pour afficher un texte » est déjà plus explicite, non ?

Vous devez essayer de résumer l'objet du sujet en quelques mots. Si vous faites cela, vous aurez souvent un bon titre. Si vous ne parvenez pas à résumer votre sujet, mettez simplement les mots-clés qui vous apparaissent comme étant les plus pertinents.

### Le contenu du sujet

Après le titre vient le contenu du sujet. Le titre est le résumé, le contenu est le corps du sujet.

La première chose qu'on doit voir dans votre sujet, c'est une formule de politesse. Eh oui : moi, quand je ne vois pas de « Bonjour » ou autre, je n'aime pas ça et je ne réponds pas au sujet. La politesse tient en quelques mots, quelques secondes de votre temps, mais beaucoup de gens la négligent. Si je viens chez vous demander du sucre de cette façon : « Donne-moi du sucre », vous allez m'en donner ? Je pense que non.

Maintenant, si je dis : « Bonjour, est-ce que vous pourriez me prêter un peu de sucre ? Merci beaucoup », j'ai déjà plus de chance de pouvoir finir mon gâteau. C'est la même chose quand vous faites un sujet dans un forum. Un brin de politesse vous assurera un accueil plus chaleureux de la part de ceux qui vous aideront.

Le contenu du sujet vient ensuite. Il est indispensable de parler français. Vous devez faire des phrases qui ont du sens, vous devez faire attention à votre orthographe (pas de SMS, pas de *kikoulol attitude*) et vous devez vous exprimer clairement.

Parfois, les gens rédigent tellement mal leur sujet qu'on ne comprend absolument rien à la demande. Et dans ce genre de cas, c'est plutôt difficile d'aider.

Si vous devez mettre du code PHP ou autre dans votre sujet, faites-y également attention ! En effet, certains forums fournissent des outils pour rendre la lecture de celui-ci plus agréable et il est plus aisément d'apporter de l'aide quand le code est plus lisible.

Deuxième chose importante : ne mettez pas un pâtit de code...

Il arrive que des gens aient un petit problème, mais la flemme de chercher un peu d'où ça vient : ils nous mettent des centaines de lignes de code. Vous pensez qu'on va les lire ? Si c'est le cas, vous rêvez.

Les visiteurs du forum ne vont pas passer deux heures à essayer de trouver d'où vient l'erreur et cinq minutes à vous répondre. C'est pourquoi vous devez extraire les parties de votre code qui sont responsables de l'erreur. Pour le moment, vous ne savez pas le faire, mais le PHP est bien conçu et il est relativement facile de trouver d'où viennent les erreurs, nous verrons ça plus tard.

### Le respect

Dernière chose qui me tient à cœur : le respect envers ceux qui vous aident. Ces personnes utilisent leur temps libre pour vous aider, alors respectez-les. On ne demande pas de faire des courbettes, mais un simple « Merci » quand le problème est résolu fait tellement plaisir. Aussi, n'attendez pas qu'on vous serve la solution sur un plateau d'argent. Dans l'idéal, ceux qui vous aident ne feront que pointer la cause de l'erreur, ils vous donneront des pistes pour la corriger, mais c'est à vous de vous corriger. Ne revenez pas cinq minutes après parce que vous ne comprenez pas ce qu'ils vous disent. Cherchez par vous-mêmes à comprendre ce qu'ils vous signalent, sinon vous ne serez jamais capables de vous en sortir seuls.

Maintenant que vous savez qu'un serveur Web est nécessaire pour obtenir une page Web, que le PHP nécessite d'être interprété par l'interpréteur PHP et que vous avez une idée des interactions entre le serveur Web et l'interpréteur PHP, vous êtes presque paré. Embrassez-vous de lire le prochain chapitre pour terminer votre préformation, vous serez alors fin prêt pour attaquer l'apprentissage du PHP à proprement parler !

## La boîte à outils du programmeur

Dans ce chapitre, nous allons parler de l'installation des outils dont vous aurez besoin pour faire du PHP : nous allons équiper votre machine pour en faire un serveur Web capable de traiter du PHP. Et pour cela nous avons besoin d'installer certains outils, d'où ce chapitre. Vous connaissez déjà deux de ces outils : PHP évidemment et le serveur Web Apache. Il manque toutefois un outil pour avoir une parfaite panoplie : l'éditeur de code.

### L'invité surprise : l'éditeur de code

Avant de se lancer dans l'installation d'Apache et PHP, présentons l'outil surprise : un bon éditeur de code. Bon, soyons clair, ce choix n'a pas grande importance actuellement. Puisque nous débutons, nous ne ferons pas vraiment de « projet », tout ce que nous ferons se limiteront à un ou deux fichier PHP. Sortir un éditeur lourd est donc pour le moment inutile. C'est pour cela qu'actuellement, au niveau de l'éditeur de code, le choix importe peu. La seule fonction vraiment indispensable à notre niveau, c'est la coloration syntaxique.

C'est outil, cette coloration, est un outil certes très basique mais ô combien utile, c'est presque l'incarnation de Chuck Norris.

Petit exemple :

#### Code : PHP

```
<?php

$toChar = function() use($collation) {
    return $collation += 42;
};
```

#### Code : Console

```
<?php

$toChar = function() use($collation) {
    return $collation += 42;
};
```

Ces codes ne servent à rien et sont moches mais illustrent bien l'apport de la coloration syntaxique : une plus grande lisibilité du code. En colorant les différents éléments du langages dans des couleurs différentes, il est plus aisé de s'y retrouver. Un autre avantage parfois plus difficile à percevoir de cette coloration est à sa tendance à nous faire découvrir des erreurs dans du code, exemple :

#### Code : Console

```
<?php

$string = 'Bonjour, il y avait l'ancien troll ici, où est-il ?';
```

#### Code : PHP

```
<?php

$string = 'Bonjour, il y avait l'ancien troll ici, où est-il ?';
```

Ces deux codes présentent une erreur de syntaxe, et si elle passe relativement inaperçue sans coloration syntaxique, elle est presque impossible à rater avec.

#### Sous Windows

Pour ce système, je vous conseille [notepad++](#), dont vous avez une illustration à la figure 2.1. Ce programme est assez léger et facile à prendre en main, mais dispose malgré tout d'un bon nombres d'outils qui vous faciliteront la vie. Je ne l'utilise plus qu'occasionnellement, mais c'est souvent vers lui que je me tourne quand je dois éditer un petit code PHP.

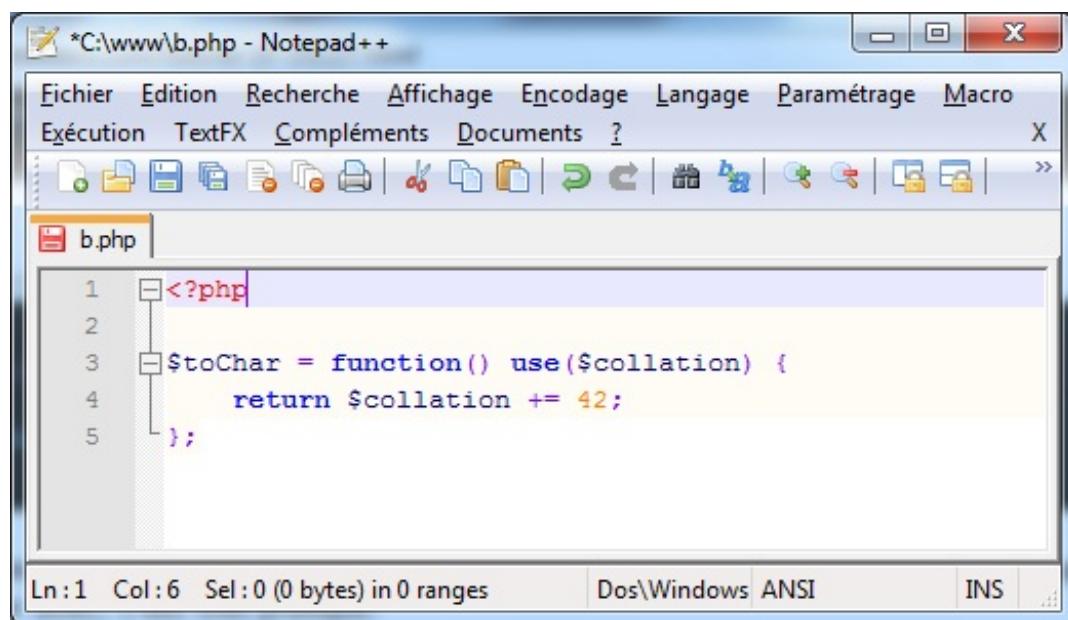


Figure 2.1 — Notepad++ : un éditeur de code idéal pour débuter

## Sous GNU/Linux

Si vous êtes féru de la console, Vim et Emacs vous proposeront coloration syntaxique et un nombre incalculables d'outils. Ces outils sont tellement puissant qu'il est difficile de les appréhender complètement, mais s'il s'agit juste d'avoir un peu de coloration syntaxique, ça n'est pas très complexe.

Si vous préférez les interfaces graphiques, j'ai entendu beaucoup de bien de GEdit et de Kate — que vous pouvez admirer à la figure 2.2 —, respectivement pour les environnements de bureaux Gnome et KDE.

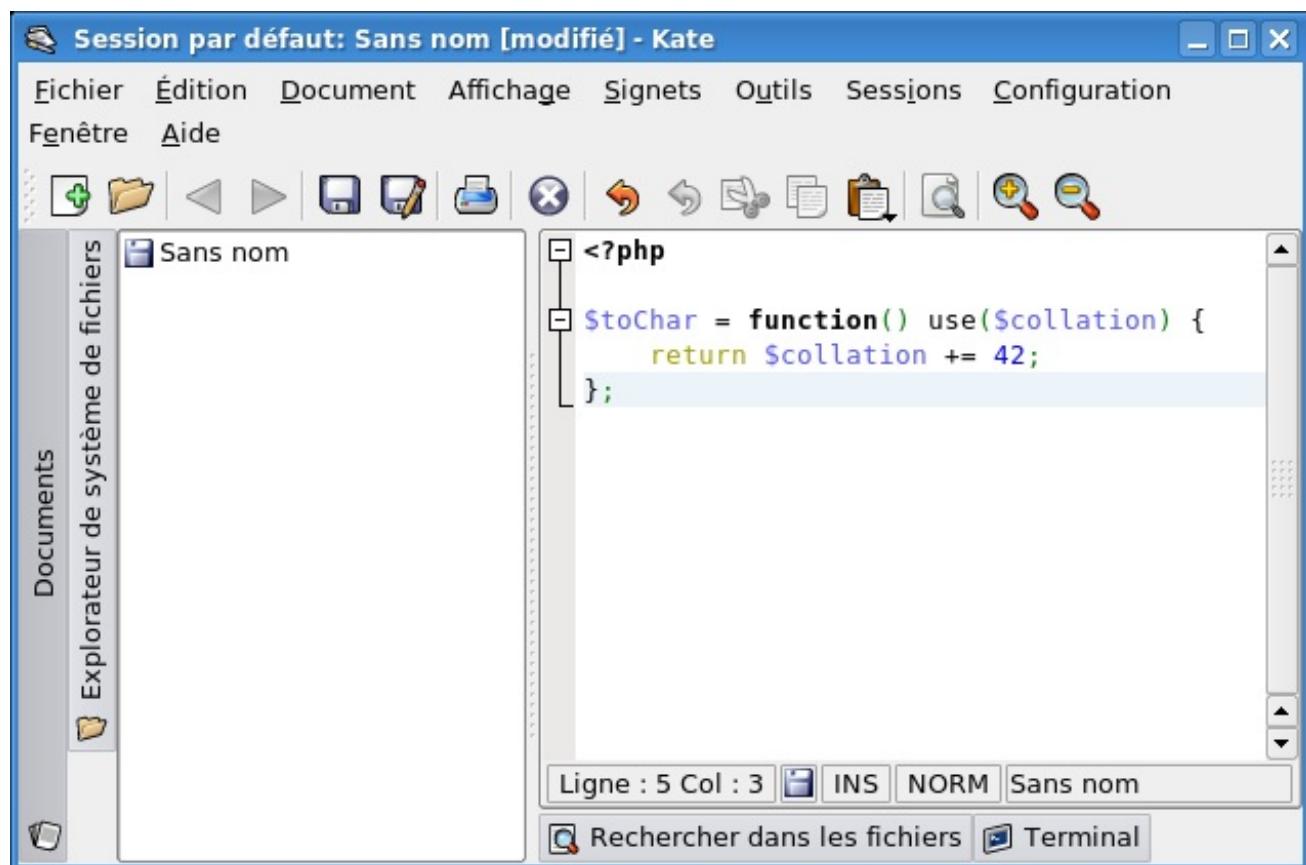


Figure 2.2 — Kate en action avec du PHP

## Sous Mac OSX et les autres

N'ayant pas de Mac et n'ayant jamais vraiment touché à d'autres systèmes que Windows et GNU/Linux, je suis malheureusement incapable de vous aider pour ces systèmes. Si vous avez des bons petits éditeurs de code facile à prendre en main pour ces systèmes, merci de m'en faire part, je les intégrerai aussitôt !

### À propos de l'encodage

Avant d'aller plus loin, assurez-vous que votre éditeur de texte utilise par défaut l'encodage UTF-8 sans BOM. Ce réglage se trouve souvent dans les préférences de votre éditeur de code. Sous notepad++ par exemple, vous devez vous rendre dans l'onglet « Paramétrage », cliquer sur « Préférences » et vous rendre dans l'onglet « Nouveau document/dossier » de la fenêtre qui vient de s'ouvrir. Vous pouvez alors choisir UTF-8 sans BOM comme encodage. Nous verrons par la suite pourquoi cela est important. Si jamais vous ne trouvez pas UTF-8 sans BOM, mettez simplement de l'UTF-8.

Maintenant que nous sommes équipés pour l'édition de codes, passons à l'installation d'Apache et PHP !

### Apache et PHP sous Windows

 Vous devez avoir les droits d'administration sur votre machine pour procéder à l'installation.

Rendez-vous sur le site de [Apache HTTPD](#), vous y trouverez un lien « from a mirror » dans le menu de gauche près de « download ». Actuellement, la page sur laquelle vous arrivez propose 4 releases :

- 2.3.8-alpha,
- 2.2.16,
- 2.0.63 et
- 1.3.42.

Cela peut bien évidemment évoluer, prenez simplement la version la plus récente qui ne soit ni en alpha, ni en bêta. Dans mon cas, je prendrai donc la release 2.2.16. Vous avez alors le choix entre diverses possibilité telles que *Unix Source*, *Win32 Source* ou encore *Win32 Binary*. Puisque vous êtes sous Windows, prenez un des *Win32 Binary*, le *Win32 Binary including OpenSSL*. Une fois le fichier téléchargé, exécutez-le.

Au début de l'installation, il vous sera demandé d'accepter la licence, faites donc cela. Une fois arrivé à l'écran de sélection du domaines et de certaines autres informations, remplissez comme indiqué dans la figure 2.3.

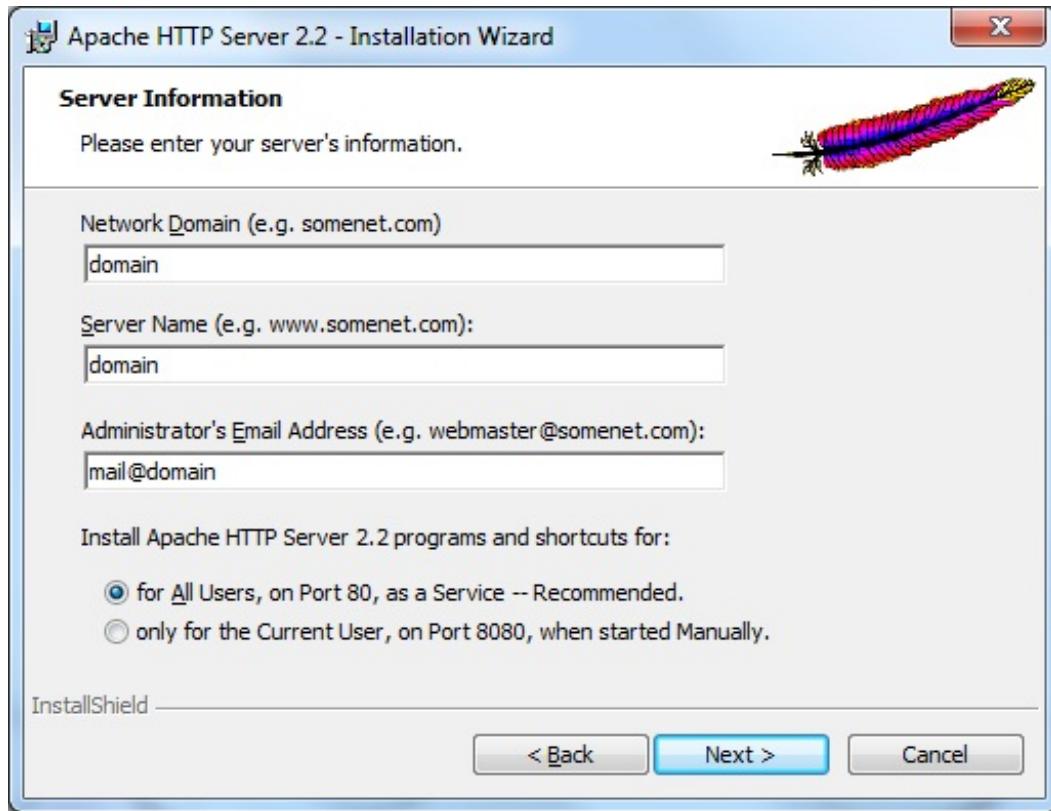


Figure 2.3 — Informations sur le domaine, le nom du serveur et autres

Par la suite, vous aurez le choix entre deux types d'installation : *Typical* et *Custom*. Comme nous ne sommes pas là pour apprendre à configurer un serveur Web aux petits oignons, nous choisirons l'option *Typical* comme indiqué à la figure 2.4.

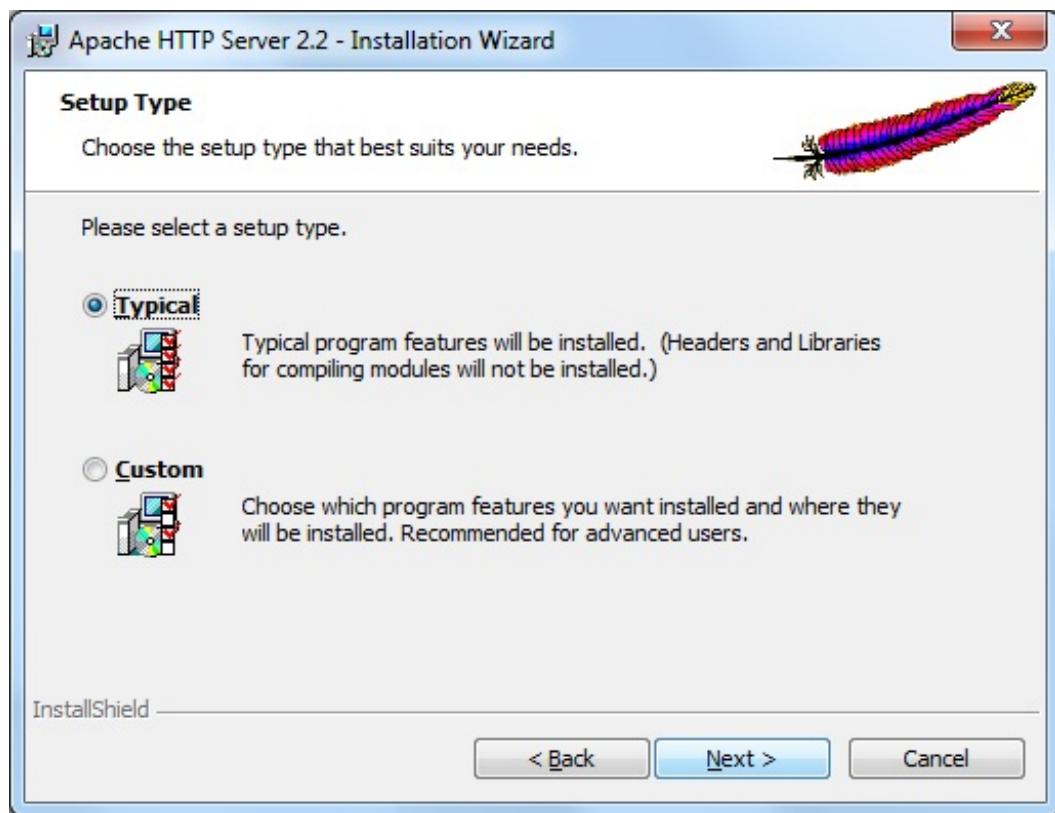


Figure 2.4 — Choix du type d'installation

L'installation à proprement parler va maintenant commencer, cela ne devrait pas prendre très longtemps, quelques minutes tout au plus. Une fois celle-ci terminée, l'assistant d'installation prendra congé, il vous faudra alors regarder dans la barre des tâches de Windows. Vous devriez une petite icône toute petite toute mignonne comme indiqué à la figure 2.5.

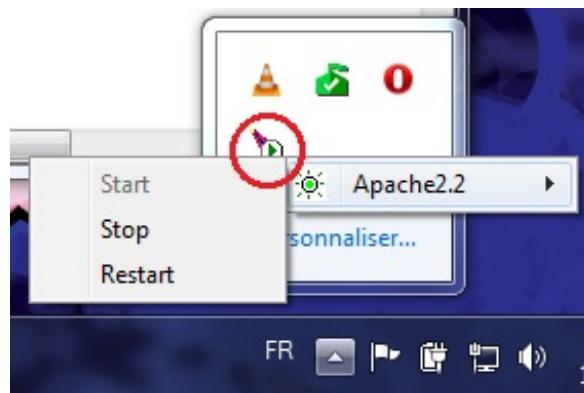


Figure 2.5 — L'icône d'Apache dans la barre des tâches

Ouvrez maintenant votre navigateur favori et rendez vous à l'adresse <http://localhost>, vous devriez obtenir quelque chose de proche de la figure 2.6. Le texte pourrait éventuellement varier, l'important c'est que vous n'ayez pas une erreur disant que l'hôte n'existe pas, qu'il est enjoignable ou je ne sais quoi.

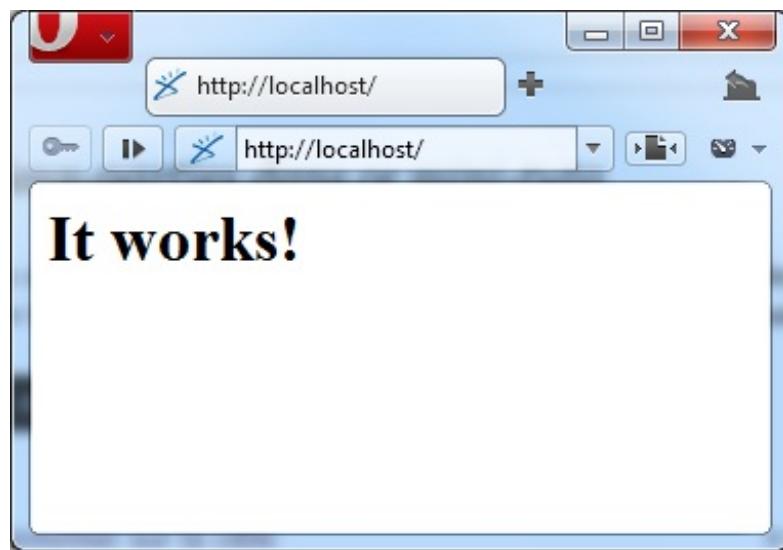


Figure 2.6 — Apache fonctionne !

Qu'est-ce que « localhost » ?

Souvenez-vous qu'Apache est un serveur Web. Pour le contacter, nous devons donc passer par le Web et plus exactement par une adresse Web, une url. De la même façon que l'url <http://siteduzero.com> vous met en contact avec le serveur Web du site du zéro, <http://localhost> nous met en contact avec le serveur Web qui est installé sur votre machine.

Si par malheur cela ne fonctionne pas, pas de panique, nous allons tenter d'y remédier. Pour se faire, il va falloir se rendre dans le gestionnaire de services de Windows. Appuyez simultanément sur les touches Windows et R, cela provoquera l'ouverture de la fenêtre de la figure 2.7. Entrez la valeur « services.msc » et validez, vous découvrirez alors le gestionnaire de services illustrés dans la figure 2.8.

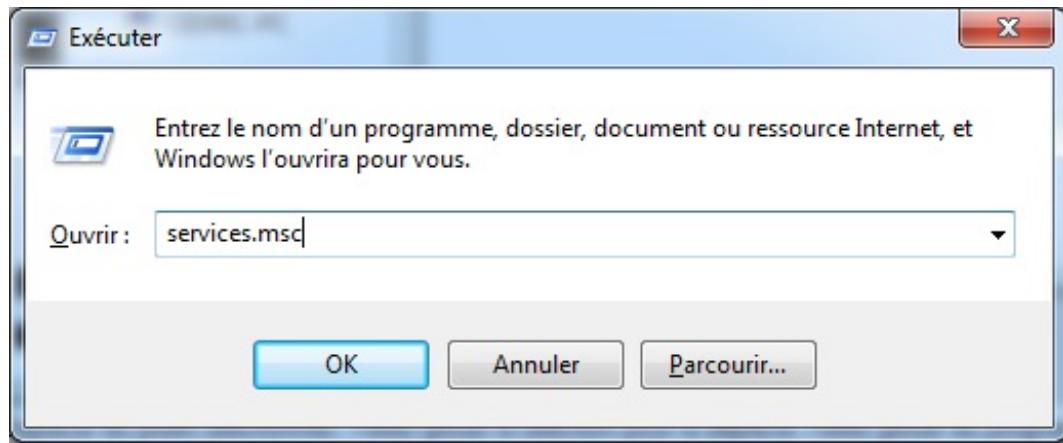


Figure 2.7 — Outil permettant d'exécuter un programme

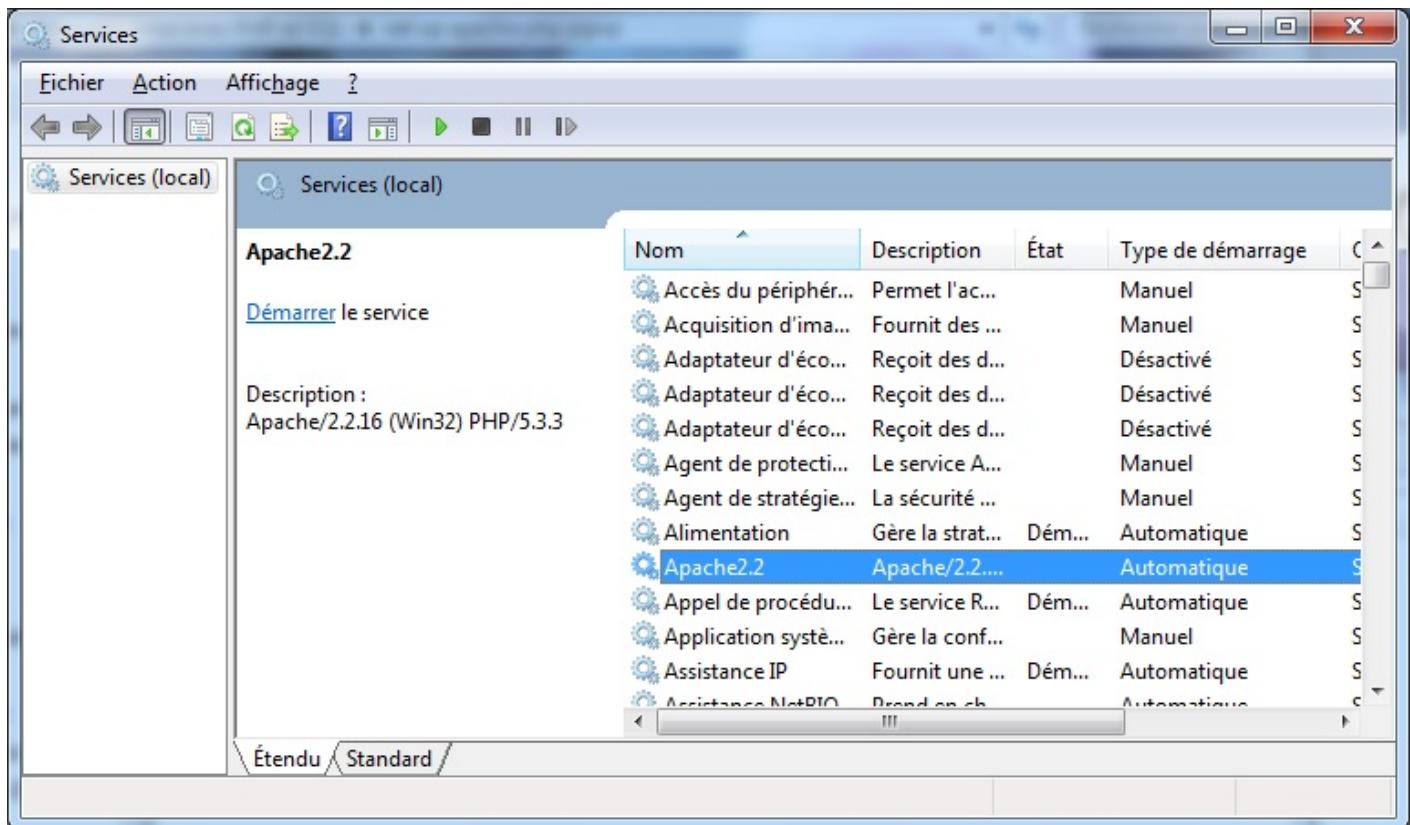


Figure 2.8 — le gestionnaire de services de Windows

Dans la liste de services, vous devriez trouver le service d'Apache HTTPD. S'il ne s'y trouve pas, c'est que l'installation ne s'est pas déroulée correctement ou que vous n'avez pas installé Apache HTTPD en tant que services. Dans les deux cas, tentez de réinstaller le serveur. Si vous le trouvez et qu'il n'est pas dans l'état « Démarré », faites un clic droit dessus et vous pourrez le démarrer.

### Au tour de PHP

PHP n'est pas plus compliqué à installer qu'Apache, commencez donc par vous rendre sur [la page de téléchargement de PHP pour Windows](#). Vous trouverez diverses versions de PHP, prenez la plus récente. Pour ma part, ce sera donc PHP 5.3.3. Parmi les 4 solutions proposées pour votre version de PHP, cherchez après la « VC9 x86 Thread Safe » et téléchargez l'[« installer »](#). Une fois ce téléchargement terminé, exécuter le fichier fraîchement acquis. Tout comme pour Apache, vous serez invité à accepter la licence, faites donc cela. Par la suite vous arrivez à l'écran illustré par la figure 2.9, cochez la case « Apache 2.2.x Module », ça permettra à votre serveur Web d'utiliser l'interpréteur PHP.

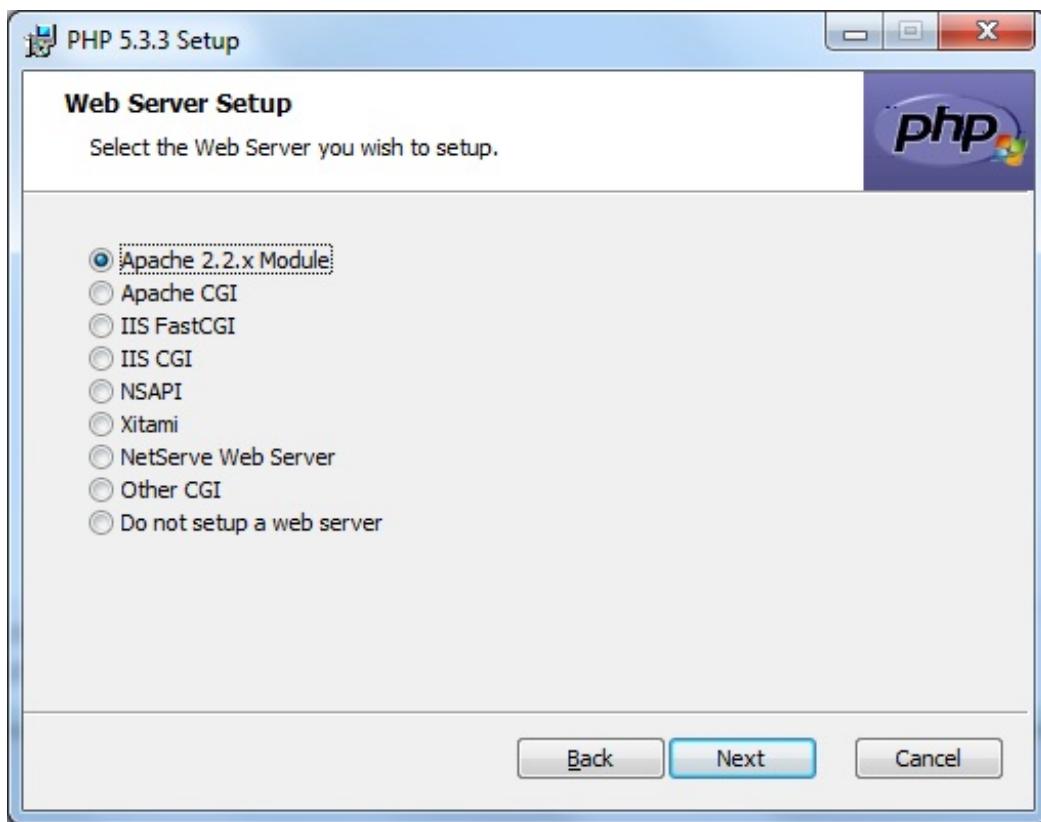


Figure 2.9 — Installons PHP avec Apache

Poursuivez ensuite l'installation, vous serez rapidement à un écran vous invitant à choisir le chemin d'accès à la configuration d'Apache – *Select Apache Configuration Directory*. Cliquez sur le bouton « browse » pour choisir ledit dossier, par défaut il se trouve dans C:\Program Files\Apache Software Foundation\Apache(version)\ comme indiqué à la figure 2.10. Il ne vous reste plus qu'à sélectionner le dossier conf et à valider.

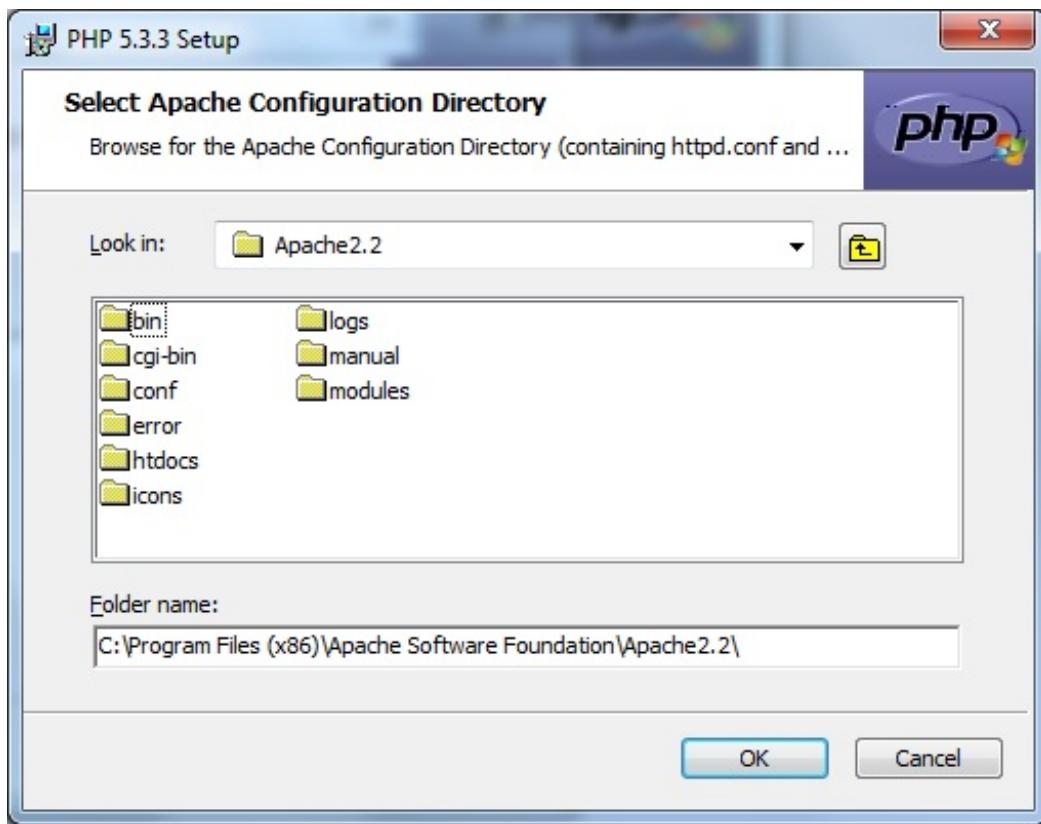


Figure 2.10 — Trouver le dossier de configuration d'Apache

Alors que les écrans continuent à défiler, arrêtez vous à celui intitulé « Choose items to install ». Dans cet écran, vous pouvez voir quels composants PHP installera. Déroulez le menu « extension » et activez Multibyte String et PostgreSQL si ce n'est déjà

fait, comme illustré à la figure 2.11. Dérouler ensuite le menu « PDO » et activez également le PostgreSQL qui s'y trouve.

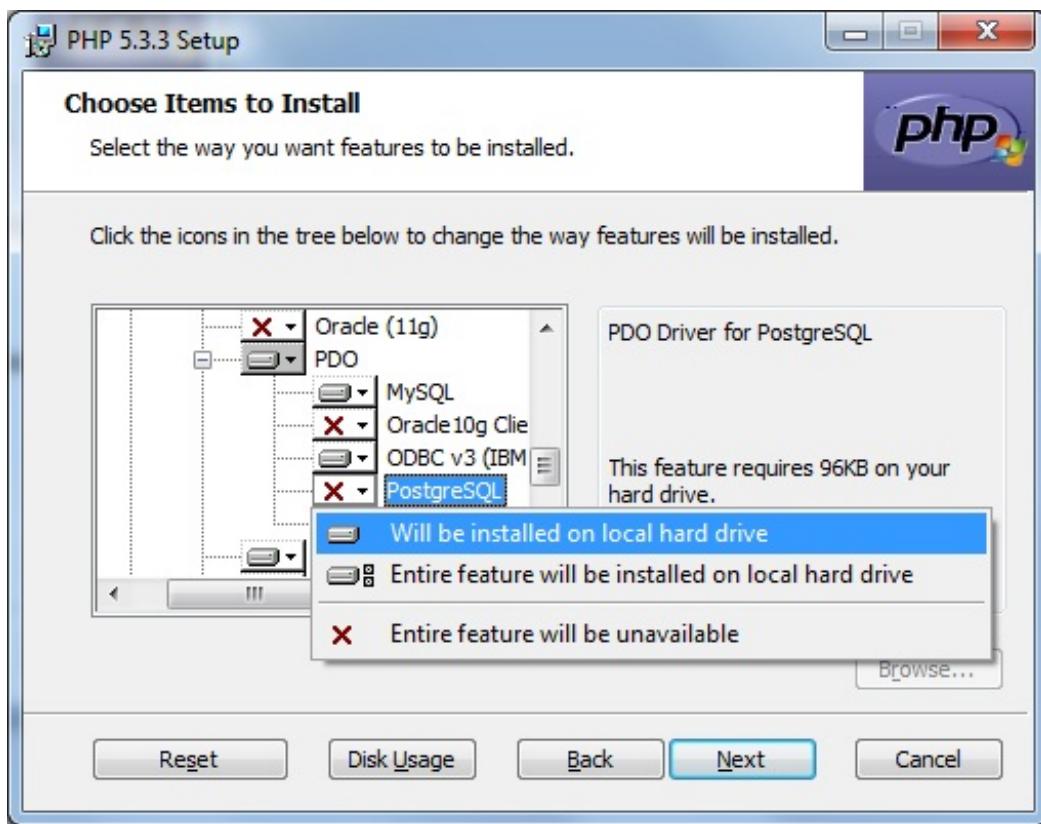


Figure 2.11 — Activons des extensions de PHP

Vous n'avez plus qu'à valider tout ça, l'installation sera alors faite. Félicitation, tout devrait être installé. 😊

Pour vérifier cela, rendez-vous dans le dossier d'installation d'Apache que vous avez choisi précédemment et créer un fichier test.php dans le dossier htdocs. N'oubliez pas d'utiliser votre éditeur de code adoré, placez-y le contenu suivant et sauvegardez :

Code : PHP

```
<?php  
phpinfo();
```

Il est temps de reprendre votre navigateur et de saisir l'adresse suivante : <http://localhost/test.php>. Vous devriez maintenant pouvoir vous émerveiller devant quelque chose de similaire à la figure 2.12.

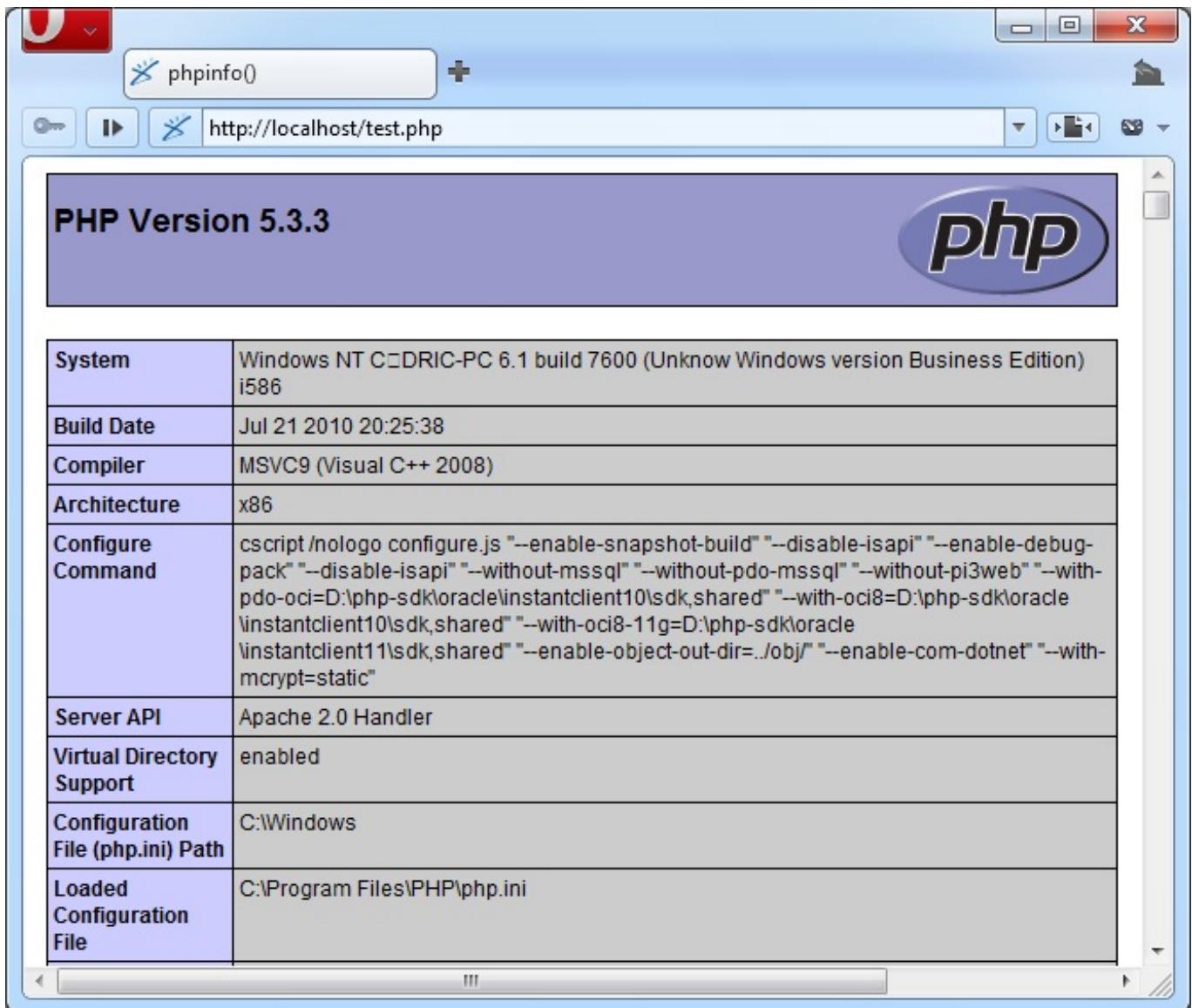


Figure 2.12 — PHP fonctionne

### Museler Apache

Il ne reste plus qu'un détail à régler pour que votre installation soit prête à servir. Actuellement, vous le constaterez si vous redémarrer votre PC ou changer d'utilisateur, Apache démarre automatiquement. Je suppose que comme moi, vous avez une vie, que donc vous ne passez pas tout votre temps à faire du PHP, il est donc inutile que Apache soit tout le temps lancé. Pour éviter cela, il y a deux choses à faire.

Pour commencer, retourner dans le gestionnaire de services de Windows comme expliqué précédemment et trouvez-y la ligne d'Apache. Faites un clic droit dessus et cliquez sur « Propriété », la fenêtre de la figure 2.13 s'ouvrira alors. De cette fenêtre, vous pourrez changer le mode de démarrage d'Apache de « Automatique » à « Manuel ».

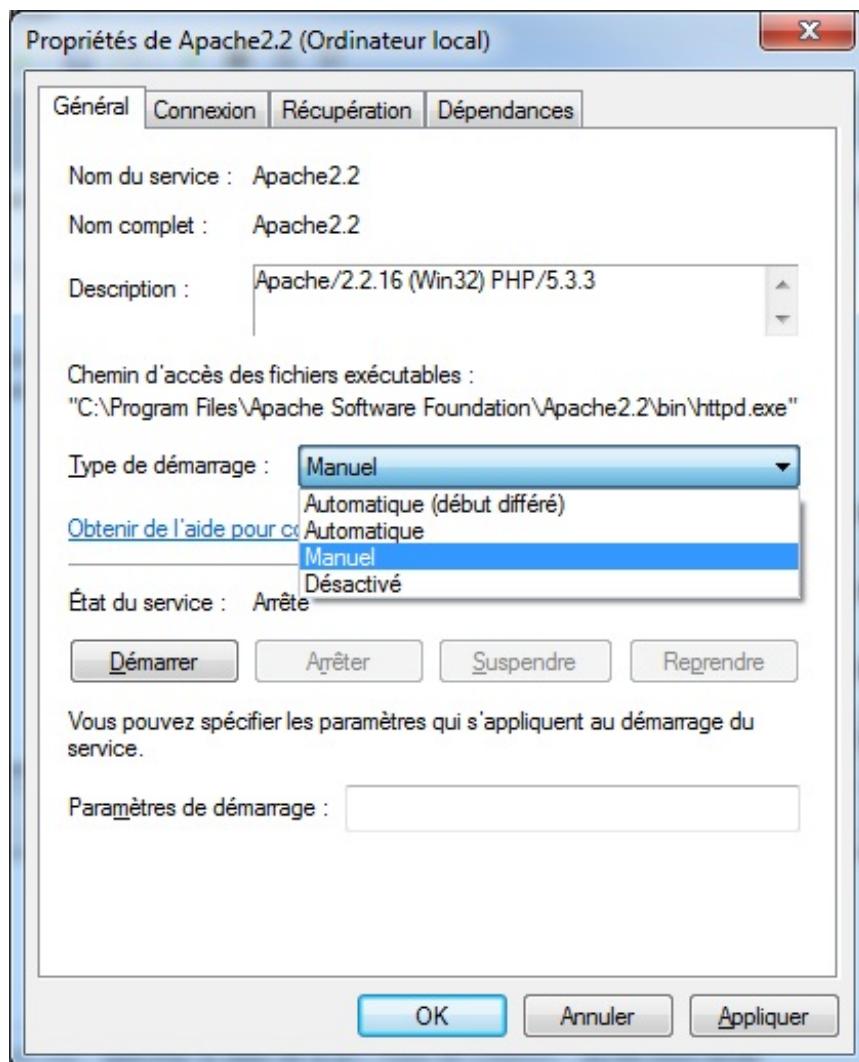


Figure 2.13 — Changer le mode démarrage d'Apache.

Grâce à cela, Apache ne démarrera plus de lui-même, vous devrez le démarrer vous-même. Toutefois, vous remarquerez que l'icône dans la barre des tâches persiste. Ce n'est pas mystérieux, c'est simplement que cette icône ne représente pas le serveur en lui-même mais une application qui permet d'arrêter, de démarrer et de redémarrer le serveur. Cette application nous étant inutile, nous allons également empêcher son exécution automatique.

Pour se faire, appuyez à nouveau simultanément sur les touches Windows et R, mais cette fois saisissez la valeur « msconfig ». Une fois cela validé, une nouvelle fenêtre illustrée par la figure 2.14 s'ouvrira, cliquez sur l'onglet « Démarrage ». Vous trouvez dans la liste juste en dessous une ligne faisant référence à Apache, décochez la, cliquez sur « Appliquer » et finalement « Ok ». Windows vous invitera à redémarrer le système, il n'est pas nécessaire de le faire.

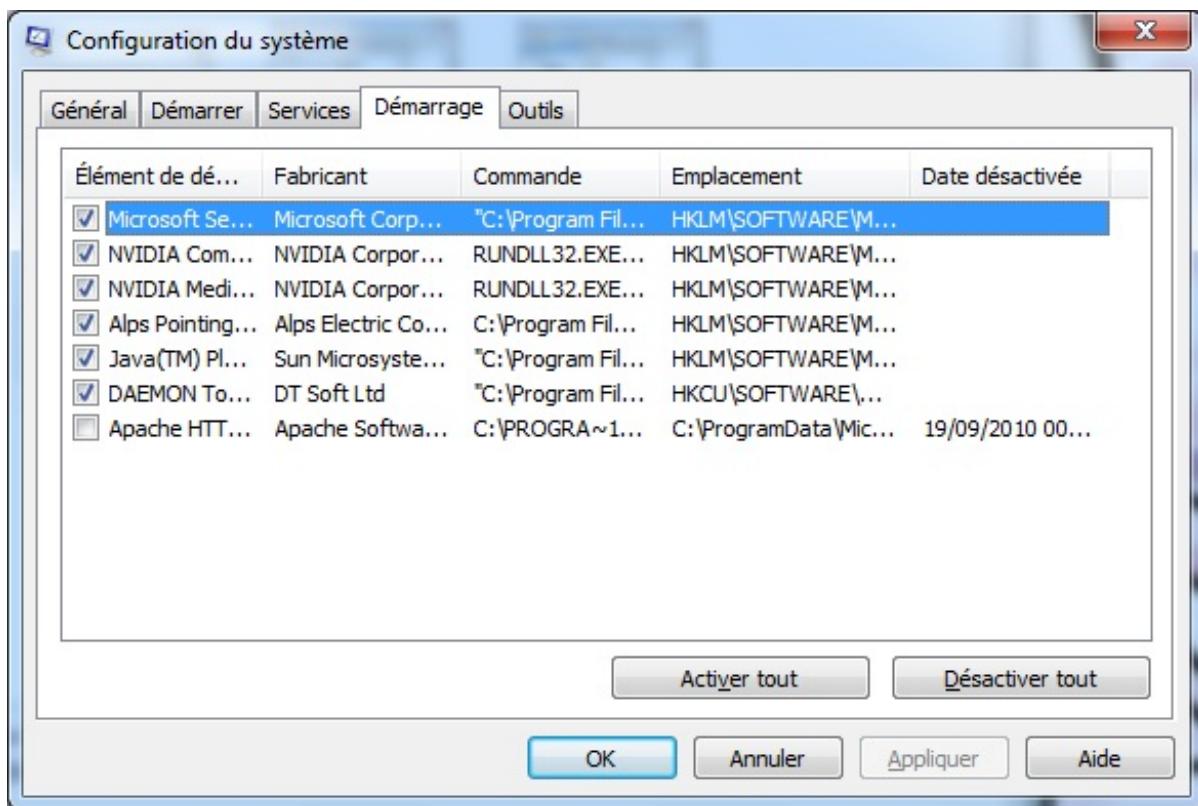


Figure 2.14 — Empêcher le démarrage de l'application Monitor Apache Servers

Apache est à présent définitivement calme, il ne viendra plus vous embêter si vous ne le sollicitez pas. Pour démarrer Apache, deux solutions s'offre à vous. Vous connaissez la première, il suffit de passer par le gestionnaire de services. La seconde fait intervenir l'invite de commande, il se trouve dans la liste des programmes installés. Dans cette invite, il vous suffira d'entrer la commande `net start apache2.2` pour démarrer Apache, comme illustré dans la figure 2.15. Il se peut que vous deviez entrer autre chose que apache2.2, vous trouverez le nom exact du service dans le gestionnaire de services. Pour arrêter le service, rien de bien compliqué, il suffira d'utiliser la commande `net stop` au lieu de `net start`.

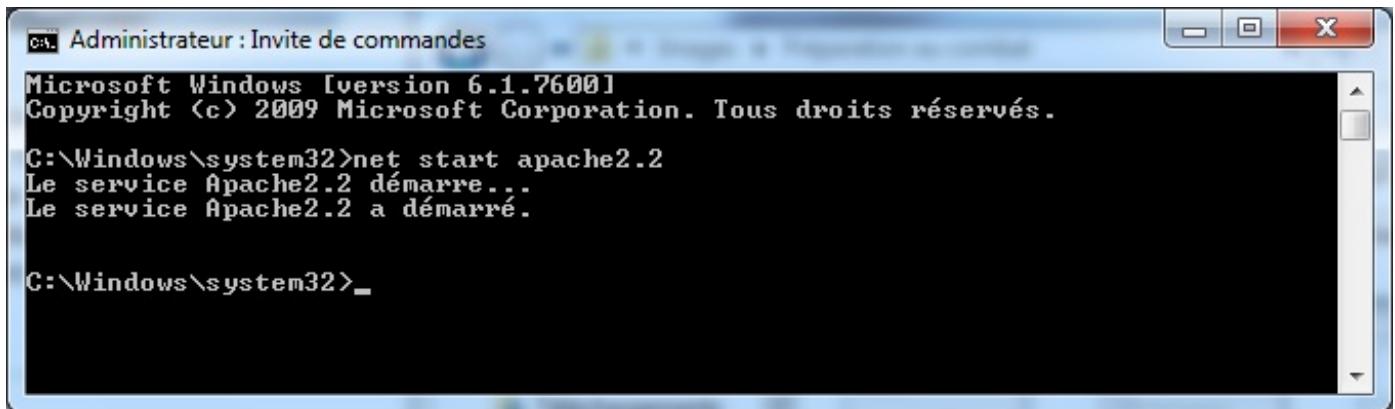


Figure 2.15 — Démarrage d'Apache via l'invite de commande

## Apache et PHP sous GNU/Linux, Mac OS X et les autres

Vous devez avoir les droits d'administration sur votre machine pour procéder à l'installation.



### Installation sous GNU/Linux

Pour l'exemple, j'ai choisi une distribution assez en vogue en ce moment : Ubuntu. La procédure d'installation ne devrait que peu varier en fonction de votre distribution, utilisez votre gestionnaire de paquet favori et tout devrait bien se passer. Si malgré tout vous éprouvez des difficultés, référez-vous à la documentation de votre distribution ou encore à des forums d'entraide. Vous pouvez également me transmettre certaines étrangetés ou certains problèmes relatifs à une distribution en particulier, je les signalerai dans ce cours.

Tout comme sous Windows, la première chose à faire est d'installer le serveur Apache. Rien de bien sorcier, le paquet apache2 est là pour cela :

Code : Bash

```
$ sudo apt-get install apache2
```

Nous allons de suite vérifier que Apache est installé, démarré et fonctionnel :

**Code : Bash**

```
$ service apache2 status
# devrait afficher Apache is running (pid xyz)
# si ce n'est pas le cas, faites un sudo service apache2 start pour
démarrer Apache
$ wget http://localhost
$ more index.html
# devrait afficher <html><body><h1>it works...
```

Qu'est-ce que « localhost » ?

Souvenez-vous qu'Apache est un serveur Web. Pour le contacter, nous devons donc passer par le Web et plus exactement par une adresse Web, une url. De la même façon que l'url <http://siteduzero.com> vous met en contact avec le serveur Web du site du zéro, <http://localhost> nous met en contact avec le serveur Web qui est installé sur votre machine.

Avant d'installer PHP, il va falloir vérifier la version de celui-ci. En effet, la version nécessaire, la 5.3, est relativement récente, il est donc possible que pour certaines distributions, le paquet fourni ne soit pas adapté. Vérifions cela :

**Code : Bash**

```
$ sudo apt-cache show php5
# La partie version devrait donner la valeur 5.3 ou supérieure
```

Si la version de PHP est correcte, parfait, installez le paquet. Si malheureusement ça n'est pas le cas, deux solutions s'ouvrent à vous. La première est de chercher si un gentil contributeur n'a pas créé un dépôt ou un paquet spécifique à PHP 5.3 ou supérieur, auquel cas il vous suffira de changer de dépôt ou de télécharger directement le paquet. Si vous ne trouvez aucun dépôt ou paquet adéquat, la seconde solution sera de compiler PHP.

Commençons par récupérer le code source, vous le trouverez à [cette adresse](#). Prenez la dernière version, le code source est fourni compressé avec Bzip2 ou Gzip, prenez ce que vous préférez, j'ai choisi le Gzip. Reste ensuite à déplacer l'archivre, l'extraire, configurer l'installation et finalement procéder à l'installation.

**Code : Bash**

```
$ sudo cp /home/cedric/Téléchargements/php-5.3.3.tar.gz
/usr/local/src
$ cd /usr/local/src
$ sudo tar -xvzf php-5.3.3.tar.gz
$ cd php-5.3.3.tar.gz
$ ./configure --with-apxs2=/usr/bin/apxs2 --with-gettext --enable-
mbstring --with-pdo-pgsql --with-pgsql
$ sudo make
$ sudo make install
```

Si lors de l'installation vous rencontrez une erreur à cause d'*apxs2* ou de la *libxml2*, il vous faudra sans doute installer deux paquets supplémentaires :

**Code : Bash**

```
# le paquet à installer si vous avez un problème avec apxs2
$ sudo apt-get install apache2-threaded-dev
# le paquet à installer si vous avez un problème avec libxml2
$ sudo apt-get install libxml2-dev
```

Si malgré tout vous avez un problème avec *apxs2*, peut-être que le chemin d'accès est incorrect. Dans ce cas remplacez simplement

/usr/bin/apxs2 par le chemin d'accès vers *apxs2*, vous devriez le trouver aisément.

Une fois PHP installé, reste à tester cela. Mais avant, n'oubliez pas de relancer le serveur Apache via un `$ sudo service apache2 restart`. Pour procéder au test, nous allons créer un fichier PHP nommé *test.php*. Pour ma part, je dois créer ce fichier dans le dossier /var/www. Si ça n'est pas le cas pour vous, chercher le fichier *httpd.conf* qui se trouve dans le dossier *conf* du dossier d'installation d'Apache, vous y trouverez une ligne définissant le « DocumentRoot », c'est le dossier en question. Dans ce fichier, placez le contenu suivant :

**Code : PHP**

```
<?php  
phpinfo();
```

Nous pouvons dès à présent vérifier que cela fonctionne :

**Code : Bash**

```
$ wget http://localhost/test.php  
$ cat test.php | grep 'PHP Version'  
# devrait afficher <a href="http://www.php.net/">...<h1 class="p">PHP  
Version 5.3.2-1ubuntu4.2</h1>
```

Tout comme sous Windows, il peut être intéressant de contrôler quand Apache démarre. Je vous laisse vous référer à la documentation de votre distribution pour empêcher le démarrage du service. Pour démarrer ou arrêter Apache, vous aurez juste à utiliser la commande *service* avec les arguments *start* ou *stop*. L'installation est à présent terminée, félicitations !

### Installation sous Mac OSX et d'autres systèmes

N'ayant pas de Mac et n'ayant jamais vraiment touché à d'autres systèmes que Windows et GNU/Linux, je suis malheureusement incapable de vous aider pour l'installation. Si vous avez des connaissances sur ces systèmes ou des articles traitant de l'installation de ces logiciels sur ces systèmes, merci de m'en faire part, je les intégrerai aussi vite que possible.

## Configurer pour mieux régner

Avant d'utiliser nos outils fraîchement installés, il est nécessaire les configurer. Je vous rassure tout de suite, nous n'allons pas — pour le moment du moins — configurer aux petits oignons les preux PHP et Apache. Mais malgré tout, certaines choses doivent être faites.

## Configuration de PHP

La configuration de PHP est centralisée dans un fichier, le fameux *php.ini*. C'est un fichier assez simple qui fait correspondre des clés à des valeurs. Il y a par exemple une clé pour des identifiants par défaut pour des connexions, et les valeurs associées à ces identifiants.

Ce que nous allons faire de suite, c'est configurer PHP pour qu'il soit utilisable dans de bonnes conditions pour l'apprentissage. La première chose à faire, c'est de localiser le fichier de configuration. cette étape est relativement facile, la réponse se trouve dans les lignes précédentes, dans le fichier de test que nous avons créé. reprenez donc votre butineur favori et rendez vous à l'adresse <http://localhost/test.php>. Dans la page reçue, faites une recherche pour « Loaded Configuration File », vous aurez le chemin d'accès au fichier de configuration juste à côté.

Ouvrez ce fichier et cherchez-y trois lignes commençant respectivement par :

- `error_reporting`,
- `display_errors` et
- `date.timezone`.

Je dis bien des ligne qui **commencent par**, s'il y a un ; devant, ce ne sont pas les bonnes lignes. Sur chacune de ces lignes vous retrouvez le principe de clé et de valeur, la syntaxe étant `<clé> = <valeur>`. Nous allons modifier la valeur de ces trois clés. Mettez ceci comme valeur :

**Code : Ini**

```
error_reporting = E_ALL | E_STRICT  
display_errors = On  
date.timezone = 'Europe/Brussels'
```

Il est à noter que pour la dernière clé, la valeur dépend de votre position géographique. Vous pouvez trouver la valeur la plus adéquate sur [cette page de la documentation PHP](#). Moi par exemple, j'habite en Europe, je clique donc sur le lien correspondant et la ville la plus proche de mon lieu de résidence est Bruxelles, je prends donc la valeur Europe/Brussels, le tout entouré d'apostrophe.

Je vous expliquerai bien évidemment par la suite pourquoi nous avons fait cela.

## Configuration d'Apache

Tout comme PHP, Apache est configuré au moyen d'un fichier, le httpd.conf. Ce fichier se trouve dans le dossier conf du dossier d'installation d'Apache. La syntaxe n'est pas la même que celle du php.ini, mais elle n'est pas plus compliquée. Ouvrez ce *httpd.conf* et cherchez la ligne définissant le *DocumentRoot*. Ce *DocumentRoot* est l'emplacement dans lequel Apache ira chercher les fichiers qui lui sont demandés. Prenons un exemple : si je me rends à l'adresse <http://localhost/test.php>, je contacte le serveur Web à l'adresse <http://localhost> et lui demande le fichier test.php. Ce fichier, où Apache va-t-il aller le chercher ? Dans le *DocumentRoot*. Vous pouvez tester la chose : supprimer ou renommer le fichier test.php que vous avez créé après l'installation d'Apache et vous constaterez que l'url <http://localhost/test.php> vous mène à une erreur.

En fonction de votre système d'exploitation — typiquement Windows —, il se peut que le *DocumentRoot* soit dans un dossier qui nécessite des priviléges d'administrateur, il vous faudrait donc toujours être administrateur dès que vous voudriez créer un nouveau fichier ou en modifier un. Pour éviter cela, nous allons modifier ce *DocumentRoot*. Sur la machine où je me trouve, le *DocumentRoot* est *C:\Program Files\Apache Software Foundation\Apache2.2\htdocs*, je vais changer pour *C:\www* par exemple, ça n'a pas vraiment d'importance du moment que c'est un dossier dans lequel vous pouvez créer et modifier des fichiers.

Reste une dernière modification à effectuer, chercher dans le fichier la ligne <Directory "DocumentRoot">. Pensez évidemment à remplacer *DocumentRoot* par l'ancienne valeur, dans mon cas je dois donc chercher <Directory "C:\Program Files\Apache Software Foundation\Apache2.2\htdocs">. Dans cette ligne, nous allons modifier le chemin d'accès puisque nous avons changé le *DocumentRoot*.

En résumé, au début nous avions par exemple ceci :

### Code : Apache

```
DocumentRoot "C:\Program Files\Apache Software
Foundation\Apache2.2\htdocs"
<Directory "C:\Program Files\Apache Software
Foundation\Apache2.2\htdocs">
```

Et au final nous avons par exemple cela :

### Code : Apache

```
DocumentRoot "C:\www"
<Directory "C:\www">
```



Lorsque vous modifiez le fichier de configuration d'Apache, il est nécessaire que celui-ci le recharge pour que les modifications soient prises en compte. Pensez donc à procéder à ce rechargement si votre système vous le permet ou à défaut à redémarrer complètement Apache.

Bon eh bien voilà, vous avez installé votre propre serveur Web ainsi que PHP, sans oublier un éditeur de code. Vous avez tout ce qu'il faut, vous êtes fin prêt pour attaquer l'apprentissage du PHP.

Un jour peut-être quelqu'un vous dira que vous êtes fou d'avoir installé ces logiciels de cette façon. En effet, il faut savoir que des packs tout prêt existent, packs qui installent PHP, Apache et d'autres logiciels tous seuls comme des grands. Si j'ai choisi de ne pas utiliser ces packs, c'est d'abord parce qu'aucun ne contenait tous les logiciels dont j'avais besoin dans la bonne version et ensuite parce que je trouve qu'il est bon d'avoir une petite idée de comment on installe — même de façon assez basique — ces différents logiciels.

## Premiers pas...

Après ces longues minutes passées à l'installation, au testage et à la configuration des différents logiciels, je vous sens désireux d'entamer le réel apprentissage. C'est pour cela que sans plus attendre, je vous propose de voir ce que sont les balises PHP et les commentaires et quelle est leur utilité.

### Les balises PHP

Prenons un fichier PHP, toute mignon, tout petit, tout doux, celui du chapitre précédent :

#### Code : PHP - Exemple 1

```
<?php  
phpinfo();
```

Je vais le réécrire sous une autre forme, ce sera plus simple pour mon explication :

#### Code : PHP - Exemple 2

```
<?php  
phpinfo();  
?>
```

Comme nous l'avons vu, PHP est un langage interprété, l'interpréteur va donc lire ce fichier et le traiter. Petit problème : un fichier PHP peut contenir du PHP, mais peut également contenir d'autres choses, du texte, du HTML, du CSS, du Javascript et je ne sais quoi d'autre. Ceci est un contenu de fichier PHP parfaitement valide :

#### Code : PHP - Exemple 3

```
Bonjour, robert.  
<?php  
phpinfo();  
?  
Au revoir, Paul le pêcheur.
```

Bien évidemment, l'interpréteur PHP ne comprends pas ces textes, ils ne signifient rien pour lui, puisque ce n'est pas une syntaxe PHP licite.



L'on dit d'un code, d'une syntaxe, qu'elle est *licite* si elle respecte la grammaire et la syntaxe du langage.

Pire encore, si l'interpréteur essayait de faire quelque chose de ce texte, il vous enverrait une jolie erreur, votre code PHP ne servirait donc à rien. C'est pour cela que les balises PHP sont si importantes, elles vont indiquer à l'interpréteur ce qui est du code PHP et qu'il doit donc traiter et ce qui n'en est pas. Si vous voulez signaler le début d'un code PHP, vous devrez donc le lui indiquer via la balise ouvrante – `<?php` – tandis que si vous voulez lui signaler la fin d'un code PHP, vous utiliserez la balise fermante – `?>`.

Comme vous l'avez sans doute remarqué, le code du chapitre précédent ne comporte pas de balise fermante, mais il fonctionne pourtant. Le comportement de l'interpréteur est donc très simple : « quand je rencontre la balise ouvrante, je commence à traiter le code jusqu'à ce que je rencontre la balise fermante, ou que j'ai lu tout le code ». À ce propos, je vous recommande justement **de ne pas mettre de balise fermante, sauf si c'est nécessaire**, comme dans l'exemple 3. Il y a des éléments incompréhensibles pour l'interpréteur avant et après le code PHP, il est donc nécessaire de mettre les deux balises. Cette recommandation vous sera expliquée plus tard.

Au cours de votre apprentissage, de votre parcours *PHPiens*, vous rencontriez ces paires de balises :

- `<?` et `?>`,
- `<?=` et `?>` et
- `<script language="php">` et `</script>`.

Ces paires de balises ont été ou sont encore malheureusement licite. Mais à terme, elles ne le seront plus, vous ne devez donc en aucun cas les utiliser. Ce sont de vieilles pratiques sujettes à disparition qui peuvent déjà être inutilisable en fonction de la configuration de PHP.

Parmi les exemples ci-dessus, vous connaissez maintenant le rôle de chaque élément mis à part celui-ci : `phpinfo();`. Nous verrons ce que c'est par la suite, promis.

Pour le plaisir, je vous propose un mini TP : écrivez moi un contenu PHP licite avec des éléments qui ne sont pas du PHP.

#### Secret ([cliquez pour afficher](#))

##### Code : PHP

```
<?php  
    phpinfo();  
?  
    Je parie que vous vous êtes cassé la tête à chercher un truc  
    complexe...  
<?php  
    phpinfo();  
?  
    alors que c'est vraiment tout bête. :)  
<?php  
    phpinfo();
```

Tout élément qui n'est pas du PHP est exclu des balises ouvrantes et fermantes, c'est donc un contenu PHP licite.

## Les commentaires

Maintenant que vous connaissez les balises qui servent à délimiter le code PHP, nous allons voir un autre élément très important : les commentaires.

Comme tous les langages, le PHP permet de faire des commentaires, c'est-à-dire des éléments entre les balises PHP qui ne seront pas interprétés. Comme ce n'est peut-être pas clair, nous allons reprendre le code d'exemple du chapitre précédent et lui apporter une petite modification :

##### Code : PHP - Exemple 4

```
<?php  
//phpinfo();
```

Mettez cela dans votre fichier `test.php` et rendez vous à l'adresse que vous connaissez maintenant bien, que se passe-t-il devant vos yeux ébahis ? En un mot : rien. Eh oui, j'ai sournoisement glissé un commentaire dans ce code, l'élément `phpinfo()` n'a donc pas été interprété, et comme c'est lui qui était à l'origine de ce charmant tableau, il ne reste que du vide.

Vous l'aurez donc deviné : tout ce qui suit deux slashes (//) est considéré par l'interpréteur PHP comme du commentaire ; l'interpréteur PHP n'interprète pas ce qui est mis en commentaire, il l'ignore.

Quel intérêt, alors ?

Pour l'interpréteur, il n'y en a aucun, il se fiche éperdument de vos commentaires. Mais pour vous, développeurs, les commentaires sont vitaux. Pour le moment, vos scripts sont ridiculement simples et minuscules. Mais par la suite, quand vous commencerez à faire des scripts autrement plus longs et complexes, les commentaires deviendront indispensables pour que vous, ou une personne tierce, puissiez vous y retrouver dans le code.

Imaginez que vous fassiez un très long code, très complexe et faisant intervenir de nombreux éléments. Quelques mois plus tard, pour une raison ou pour une autre, vous devez le modifier. Si vous n'avez pas mis de commentaires, vous allez avoir du mal à retrouver vos marques, et vous perdrez du temps. Vous pourriez même ne rien comprendre du tout.

Mais attention, il ne faut pas non plus tomber dans l'excès inverse. En effet, on voit parfois des scripts dans lesquels chaque ligne est commentée. Éventuellement, il est possible de trouver ça joli, mais ce n'est pas le but des commentaires, ils ne sont pas

faits pour traduire votre script en français !

Ils sont là pour vous aider à vous retrouver dans votre code. Si vous commencez à mettre un commentaire pour dire que vous faites une addition, ça ne sert plus à rien. Vous devez donc faire attention à ne pas en abuser, utilisez-les avec parcimonie, quand vous devez signaler quelque chose de plus général. Si par exemple vous utilisez la méthode de Mr. Patate, il est très utile de mettre un petit commentaire disant que le code qui suit est une implémentation de la méthode de Mr. Patate

Reprenez le code précédent et modifions-le à nouveau :

**Code : PHP - Exemple 5**

```
<?php  
//phpinfo();  
phpinfo();
```

D'après vous, que va-t-il se passer ?

Va-t-on voir deux fois le grand tableau ? Une fois ? Ou ne verra-t-on rien du tout ?

La réponse est : on verra une fois le grand tableau.

Pourquoi ? La réponse est simple. J'ai dit que les // servaient à indiquer un commentaire. Seulement, le commentaire n'est « effectif » que sur la ligne où se trouvent les //.

Le premier phpinfo(); est donc ignoré, car il suit les //, mais comme le second est sur une autre ligne, ce n'est pas un commentaire et il génère un grand tableau.

Vous pouvez vous amuser à mettre le second phpinfo(); sur la même ligne que le premier, les deux seront alors des commentaires et vous verrez une page blanche.

Vous savez donc faire un commentaire sur une ligne, mais imaginons que vous vouliez en faire sur plusieurs. Je reprends le code précédent, et je veux mettre les deux phpinfo(); en commentaire mais en les laissant sur deux lignes différentes.

Le code ressemblera donc à ceci :

**Code : PHP - Exemple 6**

```
<?php  
//phpinfo();  
//phpinfo();
```

Mais vous en conviendrez : si on a beaucoup de lignes à commenter, ça deviendra vite lourd de mettre des // ; si pour une raison ou pour une autre vous vouliez commenter ou décommenter un gros bloc de code, on perdrat beaucoup de temps à taper ou effacer tous les //.

C'est pour cela qu'il existe une seconde méthode pour mettre du texte en commentaire. Mieux qu'un long discours : un bout de code.

**Code : PHP - Exemple 7**

```
<?php  
/*  
phpinfo();  
phpinfo();  
*/
```

Testez ce code. Que se passe-t-il ? À nouveau, vous vous retrouvez face à une page toute blanche.

Vous l'aurez donc compris : tout ce qui se trouve entre /\* et \*/ est considéré comme un commentaire. Attention, toutefois ! Le commentaire commence au premier /\* et se termine au premier \*/.

De nouveau, un bout de code :

**Code : PHP - Exemple 8**

```
<?php  
/*
```

```
/*
phpinfo();
*/
phpinfo();
```

Si vous testez ce code, vous allez voir quelque chose de nouveau et d'inattendu : une erreur de PHP !

Si vous avez bien compris que le commentaire commence au premier /\* et se termine au premier \*/, vous devriez pourvoir m'expliquer pourquoi.

Quand l'interpréteur PHP interprète votre code, il trouve un /\* et se dit : « Un commentaire débute, j'ignore tout jusqu'à ce que je trouve un \*/ » ; il continue de lire votre code et rencontre ce fameux \*/. Juste après cela, il lit phpinfo();, qui est une instruction valide : pas de problème, la lecture continue. Maintenant, il arrive à la dernière ligne, celle qui contient le second \*/. Vous savez que ça indique la fin d'un commentaire sur plusieurs lignes, seulement, l'interpréteur lui, n'a pas trouvé d'ouverture pour ce commentaire multi-lignes. Il se retrouve donc en face d'un texte qu'il doit interpréter, seulement \*/ ne veut rien dire pour lui. L'interpréteur renvoie donc une erreur stipulant qu'il ne comprend pas votre script et le termine.

Si vous utilisez un éditeur de texte avec coloration syntaxique, comme je vous l'ai conseillé dans le chapitre précédent, vous ne devriez jamais faire d'erreur de commentaires. En effet, ces derniers se démarquent du reste du code, par une couleur ou par une police d'écriture différente (comme vous l'avez constaté dans les codes que je vous ai donnés).

Vous voilà incollables sur les commentaires en PHP, alors faites m'en des très utiles et pertinents !

Ce trop cours chapitre se termine malheureusement déjà... mais ne négligez pour autant pas son importance. Comme je vous l'ai dit, et comme je vous le répéterai encore, c'est en ayant de solides connaissances des bases que vous pourrez construire de grandes choses.

De toute façon, ne vous en faites pas, si vous aimez les chapitres plus conséquent, je parie que le prochain saura vous faire vibrer telle la corde d'un arc venant de décocher une flèche !

## Les variables, le cœur du PHP

On va s'attaquer à un chapitre **fondamental**. Vous allez devoir vous accrocher parce que ça va aller très vite. Vous allez être confrontés à l'un des chapitres les plus longs, vous verrez beaucoup de nouvelles choses, alors surtout n'allez pas trop vite. Prenez votre temps, et bien entendu n'hésitez pas à relire plusieurs fois si nécessaire !

### Première approche

Une variable... mais qu'est-ce donc ?

En mathématiques, une variable est un nom, un symbole représentant une quantité inconnue ou quelconque appartenant à un domaine donné.

En informatique, c'est à peu près pareil : une variable associe un nom à une valeur qui peut changer au cours du temps.

On va prendre un exemple. Imaginez que vous ayez un sac rempli de pommes. Le sac est le nom de la variable, et les pommes contenues dans le sac sont la valeur de la variable. Je peux sans problème remplacer les pommes qui sont dans le sac par des poires, par exemple. Le nom de la variable est toujours le même (le sac), mais la valeur de la variable a changé : au lieu de pommes, on a des poires.

Les variables représentent un concept fondamental de tout langage de programmation. C'est pour ça que j'ai appelé cette partie « **Les variables, le cœur du PHP** ». Vous boufferez des variables à la pelle quand vous ferez du PHP (ou tout autre langage de programmation).

Pour l'instant, vous savez que les variables sont caractérisées par leur nom et leur valeur. Seulement, il y a une troisième caractéristique qui est d'une importance capitale : le type de la valeur.

Reprendons l'exemple du sac rempli de pommes. Le sac est le nom de la variable, les pommes sont la valeur de la variable, mais quel est le type de cette valeur ? C'est simple, les pommes sont des fruits. La variable est donc de type « fruit ». Maintenant, si je mets des steaks dans mon sac, le nom de la variable reste le même, la valeur change, mais le type change également ! En effet, un steak, c'est de la viande : la variable est alors de type « viande ».

Si j'avais remplacé les pommes par des poires, la valeur de la variable aurait changé, mais pas le type ! Eh oui, les poires sont également des fruits.

Vous pouvez voir que le type de la variable correspond toujours au type de la valeur : en effet, les deux doivent correspondre.

Le seul but de cette tirade sur les abricots est d'introduire le concept de variable, de nom, de valeur et de type. Si vous ne comprenez pas ce concept, n'allez pas plus loin, relisez et essayez de comprendre à nouveau.

### Les variables à la sauce PHP

*Chouette, on a compris le concept de variable.* Seulement, faut que je vous avoue un truc : chaque langage de programmation a des règles particulières en ce qui concerne les variables. Bien entendu, ici je vais vous parler des variables avec le PHP.

Mieux qu'un long discours, on va créer notre première variable en PHP ! Tapez ce code, enregistrez-le dans un fichier PHP, ouvrez ce fichier et ouvrez grand les yeux !

#### Code : PHP

```
<?php  
$ma_variable;
```

Bravo, vous venez de créer votre première variable. Mais vous vous retrouvez quand même devant une page blanche, c'est tout à fait normal. On a créé une variable, on l'a déclarée, mais on ne fait rien avec.

On va analyser ensemble la ligne de code, nouvelle à vos yeux, **\$ma\_variable;**.

Le premier symbole est un \$, et sert à dire à l'interpréteur PHP que c'est une variable. Quand vous manipulez une variable en PHP, vous devez toujours faire précéder le nom de la variable par un \$.

Après ce premier symbole, vous avez le nom de la variable à proprement parler. Ici, le nom de la variable est ma\_variable.

Attention toutefois ! Vous ne pouvez pas nommer vos variables n'importe comment. Voici la règle de nommage des variables en PHP : vous pouvez utiliser n'importe quel caractère alphabétique (minuscules et majuscules comprises) ou numérique, ou bien un underscore (\_). Toutefois, le nom de la variable ne peut pas commencer par un chiffre.

Voici quelques exemples de noms valides :

- \$variable ;
- \$VARIABLE ;
- \$variable2 ;
- \$variable\_3 ;

- `$_variable`.

Et maintenant, des noms invalides :

- `$1true` ;
- `$réponse` ;
- `$var^`.

Dans la liste des noms de variable valides, vous pouvez trouver `$variable` et `$VARIABLE`. Vous devez savoir qu'en PHP, le nom des variables est **sensible à la casse**, ce qui veut dire que l'interpréteur PHP fait la différence entre des lettres minuscules et majuscules. Et par conséquent, ce code crée deux variables distinctes auxquelles on peut donner des valeurs différentes :

#### Code : PHP

```
<?php  
$variable;  
$VARIABLE;
```

Dans ce code, comme dans tous les autres, vous pouvez remarquer un symbole autre que le \$ qui revient très souvent : le point-virgule (;). Ce symbole est également très important, il sert à dire à l'interpréteur PHP que l'instruction est terminée. **Vous devez mettre un point-virgule à la fin de chaque instruction**, sans quoi l'interpréteur PHP vous dira qu'il y a une erreur dans votre script.

Mais qu'est-ce qu'une instruction ? Une instruction est un bout de texte qui représente un ordre licite pour l'interpréteur. `phpinfo()` par exemple est une instruction.

Vous vous souvenez des premiers codes ? Ils ressemblaient à ceci :

#### Code : PHP

```
<?php  
phpinfo();
```

Qu'est-ce qu'on retrouve ? Le fameux point-virgule qui marque la fin d'une instruction. Ici, l'instruction est `phpinfo()`, ça donne l'ordre à l'interpréteur de créer ce grand tableau que vous connaissez si bien maintenant.

Revenons à nos variables. Jusqu'à présent, on a vu comment créer une variable (on peut aussi appeler ça **déclarer** une variable). Merveilleux. Si je me souviens bien, je vous ai embêtés avec deux autres notions : la valeur et le type d'une variable. Jusqu'ici, ces deux notions ne sont pas intervenues. On a juste déclaré une variable, sans lui donner de valeur, ni de type. Mais ne me regardez pas avec ses yeux tristounets, on va voir ça dans un instant ; d'ailleurs, vous devriez faire une pause, parce qu'on va un peu compliquer les choses.

## Valeurs et opérateur d'affectation

Je vous ai dit précédemment que vous veniez de déclarer une variable : en théorie c'est vrai, en pratique, ça ne l'est pas. En effet, l'interpréteur PHP est un paresseux et vous le constaterez au travers de plusieurs choses. Donc nous déclarons notre variable, mais nous n'y mettons aucune valeur. Elle ne contient rien et est donc inutile. Je ne sais pas pour vous, mais quand j'ai quelque chose d'inutile, je le mets de côté ou je m'en débarrasse, mais je ne le laisse certainement pas trôner au milieu de mon salon. En PHP, c'est pareil : votre variable n'est pas prise en compte, elle n'est pas vraiment déclarée, elle n'existe pas. Pour qu'elle existe, il faut que nous en fassions quelque chose, il faut que nous lui donnions une valeur.

Dans notre histoire de sac, de pommes, de steaks et de poires, donner une valeur signifie qu'on remplit le sac avec quelque chose.

Grossièrement, on peut dire que c'est la même chose en PHP. Malheureusement, je me vois mal aller ouvrir mon ordinateur pour le remplir. C'est pourquoi les développeurs du PHP ont créé un outil très utile : l'opérateur d'affectation (=).

Un petit code... ça faisait longtemps :

#### Code : PHP

```
<?php
```

```
$ma_variable = VALEUR;
```

N'essayez pas de tester ce code, vous aurez droit à une belle erreur. Comme d'habitude, analysons ce code. Dans la ligne qui nous intéresse, nous retrouvons un élément familier, une variable. Directement à sa droite, un nouveau symbole et ce qui représente une valeur. Ce nouveau symbole, cet opérateur d'affectation (=) nous permet de donner, d'affecter une valeur à une variable. Si la variable n'existe pas avant, c'est-à-dire que si nous ne lui avions jamais affecté de valeur auparavant, alors l'interpréteur PHP la déclare avant de faire l'affectation.



L'opérateur d'affectation est bien un égal (=) ; toutefois, écrire \$ma\_variable = valeur; ne veut pas dire que \$ma\_variable est égale à valeur. Ce bout de code veut dire qu'on affecte (ou *on attribue*) la valeur de l'opérande de droite à la variable \$ma\_variable. C'est une question de terminologie, mais c'est important de le savoir !

Après **le nom**, on vient de voir comment donner **une valeur** à une variable. Maintenant, je me dis que vous devez être impatients d'entendre parler **des types** ; accrochez vos ceintures !

## Une histoire de typage

Nom et valeur ; maintenant, on va jouer avec la troisième notion : le type. Mais pourtant, cette histoire de type est encore plus complexe qu'il n'y paraît.

Comme je l'ai dit précédemment, chaque langage a ses propres règles pour les variables. C'est vrai pour le nom des variables, et ça l'est également pour le type des variables. On peut classer les langages de programmation en fonction de la force, du niveau de leur typage. Il y a des langages fortement typés et d'autres, faiblement. Cette notion de langage fortement ou faiblement typé n'est pas régie par une règle stricte. Mais dans le cas du PHP, beaucoup de monde s'accorde pour dire qu'il est faiblement typé.

Quand vous faites votre code PHP, vous n'avez pas à vous soucier du type de vos variables. L'interpréteur PHP gère cela lui-même et modifie le type à la volée si c'est nécessaire.

Comme je suis sûr que je vous ai embrouillés, je vais vous raconter une nouvelle histoire.

Je vais chez le fermier, et je lui demande de me donner 3 kilos de beurre et 24 œufs. Est-ce que je peux additionner les deux ? Est-ce que ça a un sens d'additionner des œufs et du beurre ?

Eh bien non, ça n'a pas le moindre sens.

Par contre, si maintenant j'additionne le prix de ces deux marchandises, est-ce que ça a un sens ? Est-ce possible de faire l'addition de 4 et 6 euros ?

Oui, car c'est la même chose : des euros plus des euros donnent des euros.

On ne peut pas manipuler en une fois des choses différentes. Vous ne pouvez pas additionner un nombre et une phrase, vous pouvez additionner deux fractions. Mais en PHP, vous n'avez pas à vous soucier de ça. Imaginons qu'on a deux variables : l'une contient un nombre et l'autre contient autre chose, une chaîne de caractères, le prénom « Ève » par exemple. Si je voulais faire la somme de ces deux variables dans un langage fortement typé, j'aurais des problèmes. En PHP, aucun problème, vous pouvez le faire. L'interpréteur PHP se chargera lui-même de donner le bon type à vos variables pour que l'addition puisse se faire. Cette opération, changer le type d'une variable, est appelée **transtypage** ou **cast**.

Et vous savez quoi ? C'est pareil pour l'opérateur d'affectation !

Si je tape ça :

### Code : PHP

```
<?php  
$ma_variable = 5;
```

Je déclare une variable qui a ma\_variable pour nom, qui est de type int (pour *integer*, entier en français) et de valeur 5.  
Un autre exemple :

### Code : PHP

```
<?php  
$autre_variable = 'Eve';
```

Cette fois, je déclare une variable qui a autre\_variable pour nom, qui est de type string (chaîne de caractères) et de valeur « Eve » .

Vous ne vous êtes pas occupés du type de vos variables. En fonction de la valeur que vous vouliez affecter à telle ou telle variable, l'interpréteur PHP donnait le bon type à la bonne variable.

Si vous avez bien suivi ce que j'ai dit, vous devriez maintenant vous dire quelque chose comme : « Euh, t'es bête ? Tu nous parles d'un truc totalement inutile, tu essaies de nous embrouiller pour rien ou quoi ? ! »

Je l'admet : le typage n'est *a priori* pas quelque chose d'important en PHP. Et pourtant, ça nous sera extrêmement utile par la suite.

En fait, je vous mens un peu (mais vraiment très peu, c'est plus une impasse sur un détail qu'autre chose). Je vous dis qu'une variable est de type x ou y, mais pour être rigoureusement exact, je devrais dire que la variable hérite du type de la valeur qu'on lui affecte. Retenez bien cela, c'est très important.

## Les types primitifs

Après mon long discours sur le typage, que diriez-vous de découvrir les types de variables en PHP ?

La liste **n'est pas exhaustive**, je ne parle ici que de ce que j'appelle les types primitifs, c'est-à-dire les types de bases. Il en existe cinq.

## Les entiers, alias INT (ou INTEGER)

Vous les avez déjà rencontrés juste avant, mais on va quand même reparler des entiers. Une variable de type entier contient toujours une valeur de nombre entier.

Par exemple, on trouve les entiers 1, 565, 70, etc.

Toutes les variables qui suivent sont des entiers :

### Code : PHP

```
<?php  
$a = 5;  
$b = 234;  
$c = -9;
```

Pour déclarer un entier, il vous suffit donc de mettre un \$, suivi du nom de la variable, d'un égal et enfin du nombre que vous voulez affecter à la variable. Sans oublier à la fin le sempiternel point-virgule pour monter au PHP que l'instruction est finie.

## Les nombres à virgules, alias FLOAT

Les entiers, c'est cool. Mais ce n'est pas suffisant. On peut avoir besoin de nombres à virgule ; le type d'une variable contenant un nombre à virgule est float (vous pouvez aussi rencontrer des double : en pratique, tous deux sont relativement équivalents). Alors, qui peut me créer une variable de type float contenant la valeur 5,25 ?

Facile, vous me dites ? Pas si sûr...

Quand vous écrivez un nombre à virgule, vous séparez la partie entière de la partie décimale avec une virgule. Mais manque de chance, en PHP, et dans bien des langages de programmation, voire peut-être tous, la partie entière est séparée de la partie décimale par un point (.).

Pour obtenir la variable dont je parlais, on utilisera donc ce code :

### Code : PHP

```
<?php  
$d = 5.25;  
// pour ceux qui auraient écrit « $d = 5,25; », ne vous en faites  
pas, on a tous fait cette erreur un jour  
/*  
Vous avez vu comme c'est utile les commentaires ? :p  
Quand je vous disais qu'on allait s'en servir (bon, OK, j'avoue,  
ici ça sert à rien)
```

\* /

## Les vrais ou faux, alias BOOLEEN

Ce type-là est un peu particulier et pas très naturel vu comme ça.

Petite histoire !

Avez-vous déjà joué au jeu « Qui est-ce ? » ? Pour ceux qui ne connaissent pas, ça se joue à deux, chaque joueur choisit la photo d'un personnage et se retrouve en face d'une vingtaine de photos. Les joueurs posent des questions chacun leur tour pour éliminer des photos, et trouver le personnage qu'a choisi l'autre. Pour les éliminer, on pose des questions simples comme « A-t-il les cheveux longs ? » ou « Est-ce une femme ? ». À ce genre de questions, il n'y a que deux réponses possibles : oui ou non.

Eh bien les booléens, c'est pareil, ça ne peut valoir que vrai (true en anglais) ou faux (false en anglais). Pour le moment, ce type de variables doit vous paraître peu intéressant, mais vous verrez par la suite que c'est LE type le plus important.

On crée deux booléens, l'un valant true (vrai) et l'autre false (faux).

**Code : PHP**

```
<?php  
$e = true; // on crée la variable $e et elle vaut true (vrai)  
$f = false; // on crée la variable $f et elle vaut false (faux)
```



Contrairement aux noms des variables, true et false sont insensibles à la casse. On a donc true = TRUE = TRUe = TRuE = ..., et pareil pour false. Mais je vous conseille de ne pas vous amuser à écrire n'importe comment, choisissez une écriture (true ou TRUE, c'est le plus simple) et tenez-vous-y.

## Les chaînes de caractères, alias STRING

On s'amuse bien, mais comment on fera pour afficher quelque chose comme « Bonjour, je m'appelle 'Haku' » ?

Ce genre de valeur, c'est une chaîne de caractères, ou string en anglais.

Ce type de variables est un peu plus complexe que les trois précédents, mais on verra ça en temps voulu.

Pour le moment, tout ce que vous devez savoir, c'est que pour créer une chaîne de caractères, vous devez la placer entre des guillemets ("") ou des apostrophes (''). Pour le moment, on va dire que les deux méthodes sont équivalentes ; ce n'est pas le cas, mais c'est trop tôt pour en parler. Je vais également vous imposer d'utiliser les apostrophes pour délimiter les chaînes de caractères, et ce sans vous expliquer pourquoi. Faites-moi confiance, ça viendra bien assez tôt.

On crée donc une chaîne de caractères qui contient « Bonjour, ça va bien ? » :

**Code : PHP**

```
<?php  
$g = 'Bonjour, ça va bien ?';
```

Pour les gens observateurs, un petit quelque chose devrait vous déranger. En effet, ce code-ci fonctionne sans problème. Maintenant, que se passe-t-il si je veux mettre une apostrophe dans ma chaîne de caractères ?

Testons :

**Code : PHP**

```
<?php  
$h = 'Oh une belle apostrophe : '. Elle est belle !';
```

Zut alors, l'interpréteur PHP nous renvoie une erreur. Ça ne devrait pas vous étonner d'ailleurs, je viens de dire que la chaîne de caractères est délimitée par les apostrophes : par conséquent, quand l'interpréteur de PHP arrive à la première, il comprend qu'une chaîne de caractères commence ; une fois arrivé à la seconde, il comprend que la chaîne se termine et s'attend donc à trouver quelque chose de sensé, comme... un point-virgule. Seulement, il rencontre des caractères qui n'ont aucun sens pour lui : poum, pas content, il renvoie une erreur.

Comment faire pour mettre une apostrophe, alors ? Simple : il suffit de faire précéder l'apostrophe par un *backslash* ()). Reprenons le code :

#### Code : PHP

```
<?php  
$h = 'Oh une belle apostrophe : \'. Elle est belle !';
```

Plus d'erreur, magique ? Non, logique.



C'est la même chose si vous utilisez des guillemets pour délimiter vos chaînes de caractères. En effet, il faudra alors faire précéder les guillemets par un *backslash* ()).

Dernière remarque : il ne faut pas tout mélanger.

Parfois, on voit ceci :

#### Code : PHP

```
<?php  
$ma_variable = '5';
```

La personne qui écrit cela veut déclarer une variable qui contient un entier, mais elle met des apostrophes (ou des guillemets) autour ; la variable est donc de type chaîne de caractères !

Comme le PHP est faiblement typé, ça ne pose pas de problème, mais s'il était fortement typé, ça provoquerait des tonnes d'erreurs. Retenez donc que les apostrophes et les guillemets servent à délimiter une chaîne de caractères, et rien d'autre.

## Un type à part : NULL

null n'est pas vraiment un type de variables, ce n'est donc pas réellement un type primitif. null est une « valeur » qui indique que la variable n'a pas de valeur définie, null représente l'absence de valeur. Mais quel intérêt me direz-vous ? Pour le moment, aucun, mais par la suite nous trouverons quelques utilités à ce null.

Pour obtenir une variable de « valeur » nulle, on lui affectant la « valeur » null.

Mieux qu'un long discours, un code :

#### Code : PHP

```
<?php  
$ma_variable = null;
```



À l'instar de true et false, null est lui aussi insensible à la casse.

## Quel est ton type ?

Ça fait maintenant bien longtemps que je suis là à vous apprendre des notions théoriques qui ne vous ont encore servi à rien pour le moment. Vous ne savez même pas afficher une variable ou une valeur. Franchement, c'est nul, non ?

Je vais faire quelque chose que je n'aime pas, et que j'essaie de faire le moins souvent possible : je vais vous demander d'utiliser

quelque chose sans le comprendre. Vous l'utiliserez tel quel, mais pas pour longtemps, je vous rassure.

Je vais vous montrer comment afficher le type d'une variable ou d'une valeur (souvenez-vous, j'ai insisté sur le fait que vous deviez bien comprendre la différence entre une valeur et une variable), ainsi que le contenu de cette variable ou de cette valeur.

Mais je vais vous montrer un nouveau code qui devrait vous rafraîchir la mémoire pour faire la distinction entre une variable et une valeur :

**Code : PHP**

```
<?php  
  
5;  
'une chaîne';  
true;  
/*  
Ici, je déclare des valeurs  
Comme je n'affecte pas ces valeurs à des variables, elles  
n'existent que quand l'interpréteur PHP les lit  
Une fois que l'interpréteur PHP les a lues, elles n'existent plus  
*/
```

Là, je vois que vous en avez vraiment envie : alors voilà l'instruction qui va enfin vous permettre d'afficher quelque chose.

**Code : PHP**

```
<?php  
  
var_dump('Une belle chaîne');  
$a = 5;  
var_dump($a);  
$b;  
var_dump($b);  
$b = false;  
var_dump($b);  
$c = 3.14;  
var_dump($c);
```

Je vois d'ici vos yeux ébahis et... HORREUR ! Un texte incompréhensible.

Mais on verra ça plus tard. Pour le moment, tout ce que vous devez savoir, c'est que pour afficher le type d'une variable et son contenu, ou bien le type et le contenu d'une variable, il vous suffit de taper : var\_dump(\$variable); ou var\_dump(VALEUR);, VALEUR pouvant être un int (5), un float (3.14), un booléen (true) ou une chaîne de caractères ('Lala').

L'instruction affiche un résultat de ce style :

**Code : Autre**

```
[Type de la variable ou de la valeur]([Contenu de la variable ou de la valeur])
```

Mais il y a une exception. Si vous donnez une chaîne de caractères ou une variable de type chaîne de caractères à var\_dump(), vous aurez ceci :

**Code : Autre**

```
String([Longueur de la chaîne]) "[Contenu de la chaîne]"
```

Si vous regardez attentivement le résultat du code avec tous les var\_dump(), vous verrez une ligne qui ne répond pas aux deux règles que je viens de vous montrer. Vous avez déjà rencontré ce genre de texte : c'est une erreur de PHP.

### Le bonus de fin

Que se passe-t-il si nous utilisons var\_dump() avec une variable qui n'existe pas ? La réponse en images :

Code : PHP

```
<?php  
var_dump ($a);
```

Eh bin zut alors, un beau message d'erreur aussi insultant qu'incompréhensible et un magnifique null. Vous venez de faire les frais de votre première erreur en PHP : jamais vous ne devez utiliser une variable qui n'existe pas. Cela semble logique, non ? Si vous voulez aller sur la lune mais que vous n'avez pas de vaisseau spatial, qu'il n'existe pas, il semble difficile d'atteindre l'objectif. En ce qui concerne le null, je vous ai dit que l'interpréteur PHP pouvait modifier le type à la volée. Dans le cas qui nous occupe, c'est plus ou moins ce qu'il a fait : il a considéré que notre variable qui n'existait pas avait null pour « valeur ». Pfiou ! Si vous êtes toujours vivants : félicitations. Et si vous êtes morts, ben, vous ne devriez pas être capables de lire ça. Si vous n'avez pas eu 20/20 au QCM de ce chapitre, relisez-le. Les variables sont essentielles : si vous ne les maîtrisez pas complètement, vous serez totalement largués dans le prochain chapitre.

## Les opérateurs : partie I

Dans le chapitre précédent, on a parlé de variables. Mais on a également évoqué le terme « opérateur d'affectation ». En PHP, comme dans beaucoup de langages, il existe plusieurs opérateurs.

Ils servent à faire beaucoup de choses : attribuer une valeur, faire une addition, une division, faire des comparaisons, etc.

Dans ce chapitre, je vais vous présenter un nombre réduit d'opérateurs. Mais ce petit nombre va m'amener à introduire une nouvelle notion plus importante encore que les variables : la notion d'expression.

### Retour de l'opérateur d'affectation et introduction des expressions

Dans le chapitre précédent, on a parlé de plusieurs choses : de variables, de types, de valeurs et d'un opérateur d'affectation. Seulement, comme dans la vie tout n'est pas toujours rose, je dois vous avouer quelque chose : vous ne savez pas tout. J'ai volontairement oublié de vous parler de la notion qui est le véritable cœur du PHP : les expressions. Je l'ai fait pour vous simplifier les choses, je ne voulais pas vous embrouiller trop vite ; mais maintenant que vous avez passé le cap des variables, il est temps de bien vous embrouiller. 

Revenons en arrière. Je vous ai dit, et je me cite :

#### Citation : Moi

L'opérateur d'affectation est bien un égal (=) ; toutefois, écrire \$ma\_variable = valeur; ne veut pas dire que \$ma\_variable est égale à valeur. Ce bout de code veut dire qu'on affecte (ou *on attribue*) la valeur de l'opérande de droite à la variable \$ma\_variable. C'est une question de terminologie, mais c'est important de le savoir !

En voulant simplifier, je détourne la réalité. Voici la définition réelle : l'opérateur d'affectation attribue la valeur de l'expression qui se trouve à droite du signe égal à l'opérande qui se trouve à gauche du signe égal.

C'est bien joli, mais qu'est-ce qu'une expression, d'après vous ? N'essayez pas de chercher quelque chose de compliqué, ça tient en quelques mots.

Vu que vous avez l'air de patauger dans la semoule, voilà la définition d'une expression : c'est **toute chose qui a une valeur**. Une expression se caractérise par sa valeur, et par le type de sa valeur (rappelez-vous, les int, float, etc.).

Cette définition entraîne autre chose, que vous comprendrez dans peu de temps : en PHP, tout ou presque est une expression. Il en existe différents « types », mais deux seulement pour les expressions simples : les variables et les valeurs constantes. Vous connaissez un de ces deux « types », les variables ; le second vous échappe encore. Comme son nom ne l'indique pas, une valeur constante est une valeur qui ne varie pas (l'opposé d'une variable, quoi).

Et vous savez quoi ? Je vous ai déjà fait manipuler des valeurs constantes.

Reprenez un bout de code :

#### Code : PHP

```
<?php  
$ma_variable = 5;
```

Avec l'ancienne vision des choses, vous comprenez que ce code déclare une variable, qu'on lui affecte la valeur qui se trouve à droite, c'est-à-dire 5 (un entier).

Avec la vraie définition de l'opérateur d'affectation, voici ce que vous devriez comprendre :

- on déclare une variable ;
- la variable déclarée hérite du type de l'expression qui se trouve à droite du signe égal ;
- l'expression de droite a pour valeur 5 ;
- l'expression de droite est du type entier constant ;
- au final, la valeur de la variable est 5 et son type est entier (int).

Ça paraît compliqué ? Vous n'avez encore rien vu.

Pour l'instant, j'ai uniquement parlé de l'expression de droite. Mais le plus drôle dans tout ça, c'est qu'il y a une deuxième expression !

Mieux qu'un long discours, un bout de code. Dans ce code, toutes les expressions sont entourées de parenthèses :

#### Code : PHP

```
<?php  
($ma_variable = (5));
```

Là, si vous n'êtes pas encore embrouillés, chapeau bas.

Comme vous le constatez, j'ai mis des parenthèses autour de 5, qui est une expression de valeur 5 et de type entier constant.

Mais j'ai également mis des parenthèse autour de '\$ma\_variable = 5;'. Eh oui, c'est une expression.

Reprenez la définition d'une expression. Une expression est quelque chose qui a une valeur.

Mais dites-moi, en affectant une expression à une variable, cette dernière ne prend-elle pas la valeur de l'expression ? Si ! Et donc après une affectation, la variable a une valeur. Or, tout ce qui a une valeur est une expression.

Maintenant, mon petit discours sur les expressions est fini.



Ok... mais à quoi ça va nous servir de savoir ça ?

Je vous l'ai dit : le cœur du PHP, ce sont les expressions. Si vous arrivez à maîtriser ce concept, vous pourrez coder en sachant exactement ce que vous faites, pourquoi vous le faites, comment vous le faites, etc. Et d'ailleurs, désormais, j'utiliserais la notion d'expression très souvent ; alors vous ferez bien de la maîtriser.

## Les opérateurs arithmétiques

Après cette courte introduction sur les expressions, on va pratiquer un petit peu ; ça me servira à vous montrer plus en détail ce que sont les expressions.

À l'école, vous avez dû apprendre quatre opérations de base en mathématiques : l'addition, la soustraction, la multiplication et la division. Le PHP permet de faire ces opérations grâce aux **opérateurs arithmétiques**.

### Opérateur arithmétique d'addition

Comme toujours, code à l'appui, c'est plus simple :

#### Code : PHP

```
<?php  
$ma_variable = 5 + 2;  
var_dump($ma_variable);
```

Vous venez de faire votre première addition en PHP. Maintenant, c'est l'heure de l'explication sur le fonctionnement de l'opérateur arithmétique d'addition : il fait la somme des expressions qui se trouvent à sa gauche et à sa droite. Ici, l'expression de droite est un entier de valeur 2 et l'expression de gauche est un entier de valeur 5. La somme est donc un entier de valeur 7.

Cet opérateur est très simple à comprendre. Toutefois, je vais encore *blabla* dessus.

Qu'est-ce qu'on peut additionner ?

Si je vous demande d'ajouter deux chaînes de caractères, qu'est-ce que vous allez faire ?

Les mettre l'une à la suite de l'autre ?

Ajouter le nombre de caractères des deux ?

Danser le hoola-hop ?

Pour l'interpréteur PHP, c'est pareil, il ne peut pas ajouter des chaînes de caractères. Mais au lieu de faire la liste de ce qu'il ne peut pas ajouter, on va faire la liste de ce qu'il peut ajouter : des entiers (int) et des nombres à virgule (float).

Si vous demandez au PHP d'ajouter autre chose que cela, il va faire la même chose que quand on affecte une valeur à une variable : il va transformer la ou les expressions qu'on veut ajouter et qui ne sont pas des entiers ou des nombres à virgule.

Malheureusement, le transformisme, on verra ça plus tard. Pour le moment, ne tentez pas d'ajouter autre chose que des entiers ou des nombres à virgule.

Dernière chose sur cet opérateur : quel type de valeur retourne-t-il ?

Réfléchissons. On peut lui donner des entiers ou des nombres à virgule, logiquement, il nous renverra la même chose.

Voici un tableau qui donne les différents types de valeur possible :

Types des expressions	Type de la valeur renvoyée
-----------------------	----------------------------

int + int	int
float + int	float
int + float	float
float + float	float

Pour simplifier, on peut dire qu'une addition retourne toujours un float sauf si les deux expressions qu'on additionne sont des int.

## Opérateur arithmétique de soustraction

Si vous avez compris l'opérateur précédent, ça ira très vite : tout est pareil. La seule différence, c'est qu'au lieu de faire une addition, on fait une soustraction.

### Code : PHP

```
<?php  
  
$ma_variable = 5 - 2;  
var_dump($ma_variable);
```

## Opérateur arithmétique de multiplication

À nouveau, très facile si vous avez compris l'opérateur d'addition. On multiplie les expressions entre elles. Tout comme l'addition et la soustraction, la multiplication retourne un nombre à virgule si au moins une des deux expressions qu'on multiplie est un nombre à virgule.

### Code : PHP

```
<?php  
  
$ma_variable = 4.5 * 6;  
var_dump($ma_variable);
```

## Opérateur arithmétique de division

Comme toujours, cet opérateur est très simple à manipuler, il retourne la valeur de la division des deux expressions.

### Code : PHP

```
<?php  
  
$ma_variable = 5 / 2;  
var_dump($ma_variable);
```

Oh ben zut ! Je donne deux entiers, mais le type de ma variable est float et non int.  
Ce n'est pas une erreur, c'est logique !

Après tout, **5/2**, ça vaut 2.5, non ? Et ce 2.5 est un nombre à virgule, non ? Donc la valeur renvoyée est du type float. En réalité, on peut simplifier ça : le type de valeur renvoyée par l'opérateur arithmétique de division est déterminé par le résultat de la division. Si le résultat est un entier (**10/2**, par exemple, donne un résultat entier), l'opérateur renvoie un int. Si le résultat n'est pas un entier (**5/2** par exemple), l'opérateur renvoie un float.

Pour vous en convaincre, faites des var\_dump() de 5 divisé par 5, de 2.5 divisé par 2.5, de 5 divisé par 2.5, etc.

## Calcul complexes et modulo

Résumons vite fait ce chapitre : on a vu une introduction sur les expressions, les quatre opérateurs arithmétiques, leur fonctionnement, les types d'expressions qu'ils acceptent et qu'ils retournent.

Maintenant, on va faire un peu de calcul. Ce n'est pas vraiment complexe, c'est juste pour différencier un calcul comme **4 + 5** d'un calcul comme **5/2 \* 5 + 1**.

En mathématiques, les différentes opérations ont des priorités différentes. En PHP, c'est pareil.

Exemple :

### Code : PHP

```
<?php  
  
$calcul = 4 + 5 * 2;  
var_dump($calcul);
```

Mon but, c'est de faire la somme de quatre et de cinq, puis de multiplier le tout par deux (c'est égal à 18). Seulement, en mathématiques, la multiplication a une priorité plus grande que l'addition, et donc on fait d'abord la multiplication (**5 \* 2 = 10**) et ensuite l'addition (**4 + 10**) ; le résultat est donc 14.

Le résultat n'est donc pas celui que j'attendais. En mathématiques, pour forcer la priorité, on utilise des parenthèses. En PHP, c'est pareil. Mon code devient donc :

### Code : PHP

```
<?php  
  
$calcul = (4 + 5) * 2;  
var_dump($calcul);
```

Cette fois, j'obtiens le résultat attendu : 18. C'est pareil pour tous les opérateurs arithmétiques, chacun à sa propre priorité. Voici l'ordre de priorité de ces quatre opérateurs, du plus au moins prioritaire : \*, /, +, -.

En réalité, ce n'est pas tout à fait vrai.

Exemple : je veux diviser 120 par le résultat de la multiplication de 4 par 3. Avec ces priorités, je taperais ceci :

### Code : PHP

```
<?php  
  
$calcul = 120 / 4 * 3;  
var_dump($calcul);
```

Manque de chance : au lieu d'avoir 10, j'obtiens 90.

Pas de panique, c'est normal : je vous ai menti. D'après ce que j'ai dit, l'opérateur de multiplication a une priorité plus grande que l'opérateur de division. C'est faux : ils ont le même niveau de priorité.

Et il en va de même pour les opérateurs d'addition et de soustraction : ils ont le même niveau de priorité.

Pour obtenir le résultat que je veux, je dois donc utiliser des parenthèses :

### Code : PHP

```
<?php  
  
$calcul = 120 / (4 * 3);  
var_dump($calcul);
```

## Un cinquième opérateur arithmétique : le modulo

En mathématiques, quand vous faisiez une division qui ne tombe pas juste, 5 divisé par 2 par exemple, vous aviez deux choses à la fin : le résultat de la division et le reste. Quand vous divisez 5 par 2, le résultat de la division est 2 (on peut mettre deux fois 2 dans 5) et le reste est 1 (5 moins deux fois 2 égal 1).

Il est parfois pratique de calculer le reste d'une division entière (vous verrez un exemple plus tard), et pour cela, le PHP met un autre opérateur à notre disposition : le modulo (%).

Exemple : je veux connaître le reste de la division entière de 17 par 3.

### Code : PHP

```
<?php  
$calcul = 17 % 3;  
var_dump($calcul);
```

J'obtiens bien le reste de la division entière : 2.

 Il est à noter que le niveau de priorité du modulo est le même que le niveau des opérateurs de multiplication et de division.

De même, le niveau de priorité de l'opérateur d'affectation est plus petit que le niveau de priorité des opérateurs de multiplication, de division et de modulo.

Cette partie n'est pas forcément nouvelle pour vous, les règles de calcul en PHP et en mathématiques sont identiques. Mais il était indispensable de faire cela pour introduire la notion de priorité des opérateurs, c'est souvent la cause de nombreux problèmes et on en reparlera plus tard. Retenez bien que tous les opérateurs arithmétiques ont un niveau de priorité supérieur à l'opérateur d'affectation.

## Les expressions, un peu de pratique !

Vous vous demandez peut-être pourquoi j'ai décidé de parler des opérateurs et des expressions dans le même chapitre. La raison est simple : grâce aux opérateurs, on peut mettre en pratique la théorie qu'on a vue sur les expressions. Vous vous en doutez donc, cette partie sera un peu plus pratique et devrait clarifier la notion d'expression.

On va faire un jeu : je vais vous donner quelques codes, et vous tenterez de compter le nombre d'expressions qu'il y a dans ces codes ; exemple :

### Code : PHP

```
<?php  
($ma_variable = (5));
```

Les expressions présentes sont :

- \$ma\_variable = 5
  - 5

Il y a donc deux expressions.



Rappel de la définition d'une expression : tout ce qui a une valeur.

### Code : PHP

```
<?php  
$var = 'lala';
```

```
$var = 5.6;
```

**Secret (cliquez pour afficher)**

Il y a quatre expressions :

- \$var = 'lala'
  - 'lala'
- \$var = 5.6
  - 5.6

Rien de compliqué, il n'y a qu'à compter.

**Code : PHP**

```
<?php  
$var = 5 * 6;
```

**Secret (cliquez pour afficher)**

Il y a quatre expressions :

- \$var = 5 \* 6
  - 5 \* 6
    - 5
    - 6

La seule chose à ne pas oublier, c'est que l'opérateur arithmétique retourne la valeur de l'expression qui est égale à la multiplication des expressions qui se trouvent à sa gauche et à sa droite.

**Code : PHP**

```
<?php  
$a = $b = 5;
```

**Secret (cliquez pour afficher)**

Il y a trois expressions :

- \$a = \$b = 5
  - \$b = 5
    - 5

Ici, un petit piège pour vous rappeler la définition de l'opérateur d'affectation : il affecte la valeur de l'expression de droite à l'opérande de gauche. Ce qui veut dire que l'expression de droite est interprétée avant d'être affectée. L'expression de droite étant elle-même une affectation, on interprète à nouveau l'expression de droite en premier.

Le PHP fait donc ceci : il déclare la variable \$b et lui affecte la valeur 5 ; ensuite, il déclare \$a et lui affecte la valeur de \$b (au sens strict du terme, ce que je dis n'est pas correct, mais dans les faits, on peut le voir comme ça).

**Code : PHP**

```
<?php  
$var = 5 * (6 / ($a = 4)) % 2;
```

**Secret (cliquez pour afficher)**

Il y a dix expressions :

- \$var = 5 \* (6 / (\$a = 4)) % 2
  - 5 \* (6 / (\$a = 4)) % 2
    - 5 \* (6 / (\$a = 4))
    - 5
    - 6 / (\$a = 4)
    - 6
    - \$a = 4
    - 4
  - 2
  - (6 / (\$a = 4)) % 2

Là, il n'y a pas de méthode miracle : il faut compter. Si vous avez trouvé le bon nombre d'expressions, vous cernez bien ce qu'est une expression, et c'est très bien !

Maintenant que ce petit jeu est fini, vous devriez commencer à bien percevoir ce qu'est une expression. Je l'ai dit mais je le redis : c'est l'élément fondamental du PHP. Tout, ou presque, est une expression en PHP.

## Le vrai visage de var\_dump()

Dans le chapitre précédent, j'ai introduit un élément : var\_dump(). Pour l'utiliser, je vous ai dit qu'il fallait mettre une variable ou une valeur entre les parenthèses.

Maintenant que vous êtes incollables sur les expressions, je peux dire les choses comme elles le sont.

Pour utiliser var\_dump(), vous devez mettre une expression entre les parenthèses.

Ce qui implique qu'on peut mettre beaucoup de choses !

Souvenez-vous de ce que l'on faisait : d'abord, on déclarait notre variable en lui affectant une valeur, ensuite on utilisait var\_dump().

Mais une affectation, d'après ce que j'ai dit, n'est-ce pas une expression ? Eh bien si !

À la place de ceci :

### Code : PHP

```
<?php  
  
$var = 'lala';  
var_dump($var);
```

On peut écrire ceci :

### Code : PHP

```
<?php  
  
var_dump($var = 'lala');
```

Ou encore ceci si on est sûrs de ne pas avoir besoin de la variable par la suite :

### Code : PHP

```
<?php  
  
var_dump('lala');
```



Je vous déconseille d'utiliser le second code. Il est correct au niveau de la syntaxe, mais si vous vous amusez à réduire le nombre de lignes en mettant des affectations un peu partout, vous allez rendre votre code beaucoup plus difficile à lire, et vous ne gagnerez rien, que ce soit en termes de performance ou de taille du fichier qui contient votre code.

Un nouveau chapitre fini : beaucoup de nouvelles choses apprises.

Comme je l'ai déjà dit, n'hésitez surtout pas à relire plusieurs fois et éventuellement demander de l'aide si quelque chose ne vous paraît pas clair. Si tout vous paraît limpide, rendez-vous au prochain chapitre qui continuera de parler des opérateurs.

## Les opérateurs : partie II

Pour l'instant, vous avez vu six opérateurs : l'opérateur d'affectation et les cinq opérateurs arithmétiques. Mais le PHP est bien fourni, et il en reste beaucoup à voir. Il doit en rester plus d'une quarantaine.

Je ne vais pas tous les détailler, je vais même en omettre quelques-uns pour le moment, mais vous les verrez tous, promis !

### Opérateurs arithmétiques d'affectation et opérateurs d'incrémentation

Dans le chapitre précédent, on a vu ce qu'étaient les opérateurs arithmétiques et à quoi ils servaient.

Si vous avez eu la bonne idée de vous amuser à tester un peu tout et n'importe quoi, vous êtes sans doute arrivés à ce genre de code :

#### Code : PHP

```
<?php  
  
$var = 5;  
$var = $var + 5;  
var_dump($var);
```

Rien de compliqué pour vous : on déclare une variable en lui affectant la valeur 5, et puis on réaffecte à cette variable sa valeur plus 5. Seulement, vous devez sans doute vous rendre compte que c'est lourd et plutôt embêtant de devoir écrire plusieurs fois \$var.

C'est pourquoi le PHP met à disposition des opérateurs qui sont un mix des opérateurs arithmétiques et de l'opérateur d'affectation. En termes simples, ces opérateurs permettent d'utiliser la valeur d'une variable dans une expression et d'affecter le résultat de cette expression à cette variable.

L'opérateur arithmétique d'addition et d'affectation est `+=`.

Reprendons le code précédent en utilisant ce nouvel opérateur :

#### Code : PHP

```
<?php  
  
$var = 5;  
$var += 5;  
var_dump($var);
```

Le résultat est strictement le même, mais c'est un peu plus court à écrire. Au passage, rappelez-vous ce qu'est une expression. Si votre mémoire est bonne, vous avez sans doute deviné que `$var += 5` est une expression. On peut donc l'utiliser comme n'importe quelle expression ; exemple :

#### Code : PHP

```
<?php  
  
$var = 5;  
var_dump(6 + $var += 3);
```



Cet opérateur a le même niveau de priorité que l'opérateur d'affectation (`=`).  
Mais n'oubliez jamais que l'on peut forcer la priorité en utilisant des parenthèses.

Évidemment, ce n'est qu'un opérateur parmi d'autres. Tous les opérateurs arithmétiques ont un équivalent en opérateur d'affectation ; exemple :

#### Code : PHP

```
<?php  
$a = 4;  
var_dump($a /= 2);  
$b = 10;  
var_dump($b *= 3);  
$c = 7;  
var_dump($c -= 4);  
$d = 9;  
var_dump($d %= 2);  
$e = 18;  
var_dump($e += 1);
```



Ces quatre opérateurs ont également la même priorité que l'opérateur d'affectation.

*A priori*, rien de bien particulier dans ce code, et pourtant... une notion importante en PHP, comme dans bien des langages, y apparaît : l'incrémentation.

Mais qu'est-ce que l'incrémentation ? Incrémenter une variable, c'est lui affecter sa valeur **plus 1** (bien qu'en réalité, on peut dire qu'incrémenter une valeur, c'est lui ajouter quelque chose). Puisque rien ne vaut un bon exemple :

#### Code : PHP

```
<?php  
$a = 5;  
$a++;  
var_dump($a);
```

L'opérateur d'incrémentation est le double plus (++). 5 plus 1, ça donne 6 : le résultat est bien celui que l'on attend. Mais je vais vous emmêler les pinceaux, regardons ce code :

#### Code : PHP

```
<?php  
$a = 5;  
var_dump($a++);
```

Plutôt curieux, non ? Au lieu d'avoir 6, comme on pouvait le supposer, le résultat est 5. Vous venez de faire les frais de la différence entre la *post-incrémantion* et la *pré-incrémantion*. Des mots compliqués, ça rend toujours mieux 😊.

Pour incrémenter une variable, on a fait comme ça :

#### Code : PHP

```
<?php  
$a = 5;  
$a++;  
var_dump($a);
```

Mais on peut également le faire ainsi :

#### Code : PHP

```
<?php  
$a = 5;
```

```
++$a;
var_dump($a);
```

Ces deux codes produiront le même résultat ; la seule différence, c'est que dans le premier je fais une post-incrémantation, et dans le second, une pré-incrémantation. La différence est très simple à comprendre. La pré-incrémantation incrémente la variable avant que sa valeur ne soit utilisée, alors que la post-incrémantation incrémente la variable après que sa valeur a été utilisée. Vous allez comprendre :

#### Code : PHP

```
<?php
$a = 5;
var_dump($a++);
$b = 5;
var_dump(++$b);
```

Le premier var\_dump() affiche que la variable prend la valeur 5 et le second la valeur 6. Second code :

#### Code : PHP

```
<?php
$a = 5;
var_dump($a++);
var_dump($a);
$b = 5;
var_dump(++$b);
var_dump($b);
?>
```

Le premier et le troisième var\_dump() affichent la même chose que la dernière fois. Mais le second et le quatrième, eux, affichent la même chose. Avec cette histoire de post et pré-incrémantation, vous devriez pouvoir expliquer pourquoi. Mais je vais quand même le faire.

Voici ce que fait l'interpréteur PHP : on déclare \$a et on lui affecte la valeur 5, il affiche les infos sur \$a **avant** d'incrémenter \$a. On affiche à nouveau les infos de \$a. On déclare \$b et on lui affecte la valeur 5, il affiche les infos de \$b **après** avoir incrémenté \$b. On affiche à nouveau les infos de \$b.

Tout ce que vous devez retenir sur la post/pré-incrémantation, c'est que dans le cas de la pré-incrémantation, on incrémente d'abord la variable et ensuite l'instruction est exécutée tandis que dans le cas de la post-incrémantation, l'instruction est d'abord exécutée puis la valeur est incrémentée

Vous vous en doutez peut-être, il existe l'inverse de l'incrémantation : la décrémantation. L'opérateur de décrémantation est le double moins (--), et il fonctionne de la même façon que l'opérateur d'incrémantation (ça vaut aussi pour l'histoire de post/pré-décrémantation).

## Opérateurs de comparaison

Quel que soit le langage de programmation que vous pouvez utiliser, vous avez toujours besoin de pouvoir faire des comparaisons. Par exemple, si vous voulez faire un espace membre, pour s'y connecter, il faudra vérifier que le mot de passe que l'utilisateur donne est le même que celui qu'il a donné à son inscription : on va devoir les comparer.

En PHP, il existe en tout et pour tout neuf opérateurs de comparaison. Ça va du simple test d'égalité au test de différence de type. Les opérateurs arithmétiques retournaient la valeur du calcul et donnaient soit un type int, soit un type float en fonction de la valeur du calcul. Les opérateurs de comparaison renvoient des booléens. Souvenez-vous, j'en ai parlé dans le chapitre sur les variables.

Les booléens peuvent prendre deux valeurs : true (vrai) et false (faux). Les opérateurs de comparaison renvoient true si la comparaison est vraie et false si elle est fausse. Pas très compliqué. Voici un tableau des différents opérateurs de comparaison :

Opérateur	Fonction
=	Retourne true si les valeurs des expressions à gauche et à droite sont égales.

<b>==</b>	Retourne true si les valeurs des expressions à gauche et à droite sont égales et du même type.
<b>!=</b>	Retourne true si les valeurs des expressions à gauche et à droite sont différentes.
<b>!==</b>	Retourne true si les valeurs des expressions à gauche et à droite sont différentes ou sont de types différents.
<b>&lt;&gt;</b>	Voir l'opérateur !=.
<b>&gt;</b>	Retourne true si la valeur de l'expression de gauche est strictement plus grande que la valeur de l'expression de droite.
<b>&gt;=</b>	Retourne true si la valeur de l'expression de gauche est plus grande ou égale à la valeur de l'expression de droite.
<b>&lt;</b>	Retourne true si la valeur de l'expression de gauche est strictement plus petite que la valeur de l'expression de droite.
<b>&lt;=</b>	Retourne true si la valeur de l'expression de gauche est plus petite ou égale à la valeur de l'expression de droite.

Maintenant que la théorie est là, faisons place à la pratique, vérifions que 5 est strictement plus petit que 6 :

#### Code : PHP

```
<?php
$var = 5 < 6;
var_dump($var);
```



5 < 6 est une expression, on aurait donc pu juste écrire var\_dump(5 < 6) au lieu d'utiliser inutilement une variable.

Quelques tests en vrac :

#### Code : PHP

```
<?php
$a = 5;
var_dump($a == 5);
$b = '7';
var_dump($b == 7);
var_dump($b === 7); // 7 est de type int, alors que $b est de type
                   string. L'opérateur == ne vérifiant pas le type, il retourne true,
                   vu que les deux valeurs sont les mêmes. Mais comme l'opérateur
                   vérifie également le type, et comme les valeurs des deux
                   expressions ne sont pas de même type, il retourne false.
$c = 'meuh';
var_dump($c !== 'meuh');
var_dump($c == $d = 'meuh');
// etc.
```



Tous les opérateurs de comparaison ont un niveau de priorité inférieur à n'importe lequel des opérateurs arithmétiques. Parmi les opérateurs de comparaison, il y a deux groupes : >, >=, <, <= et ==, ===, !=, !==, <>. Dans chacun de ces groupes, le niveau de priorité est constant. Le niveau de priorité du premier groupe est supérieur à celui du second. Enfin, les deux groupes ont un niveau de priorité supérieur au niveau de priorité du groupe d'opérateurs suivant : =, +=, -=, \*=, /=, %=.

Pour l'instant, ces opérateurs ne servent à rien, sauf peut-être à vérifier que l'interpréteur PHP sait compter ; mais ils serviront beaucoup dans le prochain chapitre.

## Opérateurs logiques

Après ces quelques opérateurs, faisons une courte pause. Je vous ai fait comprendre qu'il y avait en tout cinquante-deux opérateurs en PHP. On en a vu vingt-deux (cinq arithmétiques, six d'affectation, neuf de comparaison, un d'incrémentation et un de décrémentation). Il en reste donc trente.

Je ne vais pas tous les détailler ici. Certains étant pour le moment totalement superflus, on les verra petit à petit tout en avançant

dans l'apprentissage de la syntaxe du PHP.

Mais avant de vous laisser tranquilles avec les opérateurs, on va encore en voir six : les opérateurs logiques (en réalité, il n'y en a que quatre, mais il y en a qui ont la même fonction avec une priorité différente !).

Dans la partie précédente, on a vu comment faire des comparaisons. Mais maintenant, qu'écririez-vous si je vous demandais de vérifier que 22 est plus grand que 6 et plus petit que 52 ?

Vous arriveriez sans doute, après un peu de réflexion, à ceci :

#### Code : PHP

```
<?php

$a = 22 > 6;
$b = 22 < 52;
$test = $a == $b; // ce qui vaut true
// on peut condenser ce code en :
$test = 22 > 6 == 22 < 52; // > et < ayant un niveau de priorité
plus grand que == (relisez la partie précédente), on n'est pas
obligés de mettre des parenthèses, mais je les mettrai quand même
pour améliorer la lisibilité du code.
```

Il faut cependant l'avouer : ce n'est pas naturel. Vérifier si le résultat des deux tests est le même, il faut y penser. Et si vous avez besoin de faire plus de tests, ça devient vite extrêmement pénible. C'est pour ça que le PHP met les opérateurs logiques à notre disposition. En voici la liste et leur rôle respectif (ils renvoient aussi true ou false, comme les opérateurs de comparaison) :

Opérateur	Fonction
&&	Retourne true si les valeurs des expressions à gauche et à droite sont égales à true.
	Retourne true si la valeur de l'expression à gauche est égale à true ou si la valeur de l'expression à droite est égale à true.
AND	Voir l'opérateur &&.
OR	Voir l'opérateur   .
XOR	Retourne true si la valeur de l'expression à gauche est égale à true ou que la valeur de l'expression à droite est égale à true MAIS si les valeurs des expressions à droite et à gauche ne sont pas toutes les deux égales à true.
!	Retourne true si la valeur de l'expression de droite vaut false.

Les deux opérateurs « doublons » sont faciles à trouver . Comme je l'ai dit, la seule différence est le niveau de priorité. Voici d'ailleurs les six opérateurs dans l'ordre décroissant de leur niveau de priorité : !, &&, ||, AND, XOR, OR. Ils ont tous des niveaux de priorité différents. Mais tous les opérateurs logiques ont un niveau de priorité inférieur au plus faible des opérateurs arithmétiques.

Maintenant que l'on maîtrise ces opérateurs, reprenons l'exercice précédent :

#### Code : PHP

```
<?php

$test = 22 > 6 && 22 < 52;
var_dump($test);
```

Vous êtes fin prêts pour faire des comparaisons rocambolesques.

Un petit exemple : je veux vérifier que 18 est compris entre 10 et 30 (compris), mais est différent de 20. Si c'est le cas, le code doit afficher true.

#### Code : PHP

```
<?php
var_dump(18 >= 10 && 18 <= 30 && 18 != 20);
```

?>

Vu que tous les opérateurs fonctionnent de la même façon, je ne vais pas m'attarder là-dessus. Par contre, je vais dire un petit mot sur l'opérateur logique !.

Si les cinq autres opérateurs logiques utilisent deux expressions (une de chaque côté de l'opérateur), celui-ci, comme les opérateurs d'incrémentation et de décrémentation, n'utilise qu'une expression. Un exemple de son utilisation :

#### Code : PHP

```
<?php  
$test = 5 == 8; // vaut false  
var_dump(!$test); // affiche true  
$test = 4 % 2 === 0; // vaut true  
var_dump(!$test); // affiche false  
?>
```

On peut bien sûr réduire ce code, mais faites bien attention à la priorité des opérateurs. D'ailleurs, comme on en a déjà vu pas mal, voici un cadeau : une liste des opérateurs triés par ordre décroissant de niveau de priorité (les opérateurs d'une même ligne ont la même priorité).

- `++, --`
- `*, /, %`
- `+, -`
- `>, >=, <, <=`
- `==, ===, !=, !==, <>`
- `&&`
- `||`
- `=, +=, -=, /=, *=, %=`
- AND
- XOR
- OR

Et bien entendu, n'oubliez pas qu'on peut **toujours** forcer la priorité en utilisant des parenthèses. D'ailleurs, utilisez-les...

Regardez un peu cette instruction : `$test = 4 % 2 === 0;`

N'est-elle pas plus compréhensible ainsi : `$test = ((4 % 2) === 0);` ?

Il ne faut évidemment pas tomber dans l'excès, c'est à vous de juger de ce qui est nécessaire pour que votre code soit lisible mais pas trop chargé.

Les opérateurs, c'est fini !

Qu'en pensez-vous ? Avec tout ça et les prochains chapitres, vous pourrez bientôt faire de grandes choses avec le PHP.

Si vous êtes toujours avec moi, c'est parti pour le chapitre sur les structures de contrôle, on y verra toute la puissance de l'association des opérateurs de comparaison et des opérateurs logiques.

## Les structures de contrôle

On a appris ce qu'étaient les variables, les expressions, les opérateurs, mais pour l'instant, ce n'est pas encore un truc qui vous en met plein la vue.

Dans ce chapitre, on va tout mettre ensemble pour aborder des structures qui vont vous montrer ce que le PHP a dans le ventre. Couplé avec les quelques chapitres suivants, vous pourrez faire plein de petites choses marrantes.

### Structures conditionnelles

Grâce aux opérateurs de comparaison et aux opérateurs logiques, on peut faire des tests sur des expressions, sur des variables, sur des valeurs. Seulement, ça n'a pas grand intérêt si on ne peut pas utiliser ces résultats. Voilà ? tout frais tout chaud ? un bout de code comme on les aime :

#### Code : PHP

```
<?php

$var = 5;
$test = $var <= 10;
if($test)
{
    var_dump(' $var est plus petite ou égale à 10');
}
else
{
    var_dump(' $var est plus grande que 10');
}
```

Testez donc ce code, et que se passe-t-il ? Un seul var\_dump() affiche quelque chose. Changez la valeur de \$var (donnez-lui la valeur 15, par exemple) et observez : le second var\_dump() finira par être plus bavard. Mine de rien, on vient de voir une des structures de contrôle du PHP : la structure conditionnelle.

Mais comment ça marche ?

Vous voyez déjà deux nouveaux mots-clés, if et else. En français, ils signifient **si** et **sinon**. En sachant ce que ça veut dire, vous devriez comprendre comment fonctionne ce code. On va d'ailleurs l'épurer pour ne conserver que la structure conditionnelle :

#### Code : PHP

```
<?php

if(Expression)
{
    // instructions si Expression est évaluée à true
}
else
{
    // instructions si Expression est évaluée à false
}
```

Voilà donc la structure de base : j'ai utilisé le mot Expression au lieu de la variable \$test, vous savez ce qu'est une expression, on l'a vu. On va donc affiner le fonctionnement de cette structure : si l'expression placée entre les parenthèses du if est évaluée à true, le bloc d'instructions du if est exécuté, sinon c'est le bloc d'instructions du else qui l'est.

Mais qu'est-ce qu'un bloc d'instructions ? Ce qu'on appelle ainsi, c'est ce qui est compris entre les accolades ({}). Dans ces accolades, vous pouvez mettre diverses instructions. Par exemple, si vous faites un script pour afficher une news, vous allez d'abord vérifier que la news existe (avec l'expression adéquate entre les parenthèses qui suivent le if). Si elle existe, vous mettrez diverses instructions qui afficheront la news, mais si elle n'existe pas, c'est le bloc d'instructions du else qui sera exécuté, bloc d'instructions qui affichera une erreur par exemple. Les deux blocs ne seront jamais tous les deux exécutés : c'est soit l'un, soit l'autre, mais pas les deux. Vous pouvez comparer cela à une route qui se sépare en deux : vous prenez soit la rouge à gauche, soit à droite, mais certainement pas et à gauche et à droite en même temps comme c'est illustré à la figure 7.1.

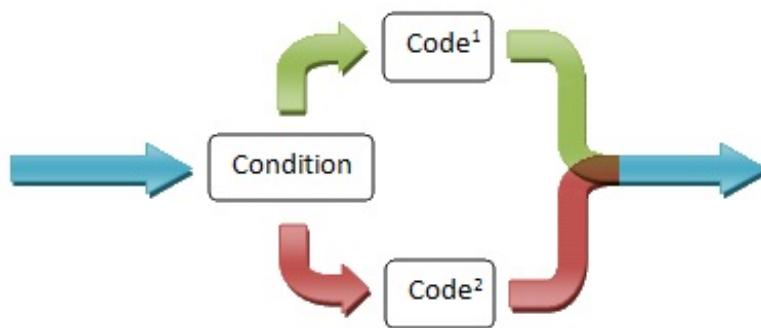


Figure 7.1 — Un seul code sera exécuté, jamais les deux !

Vous n'êtes pas obligés de créer des variables pour rien. Qu'est-ce que je veux dire ? Regardez mon premier code : je déclare deux variables, \$var et \$test. Mais je n'utilise pas \$test ailleurs que dans l'expression que je mets entre les parenthèses du if, ma variable est donc inutile. Car \$var <= 10 est une expression, rappelez-vous !

On peut donc réécrire ce code comme tel :

#### Code : PHP

```
<?php

$var = 5;
if($var <= 10)
{
    var_dump('$var est plus petite ou égale à 10');
}
else
{
    var_dump('$var est plus grande que 10');
}
```

Ce code produit exactement le même résultat, mais il est meilleur car on n'utilise pas de variable inutile, et le script est donc plus performant, plus rapide. Amusez-vous à tester tout et n'importe quoi comme condition, mettez plusieurs instructions dans les différents blocs, bref, essayez un peu tout. Si vous savez ce qu'est une expression, que vous avez compris comment on utilise cette structure conditionnelle, tout ira sans problème.

Puisque c'est simple, encore un mot sur cette structure et sa petite sœur :

#### Code : PHP

```
<?php

$var = 5;
if($var <= 10)
{ // si $var est plus petite ou égale à 10, on le dit
    var_dump('$var est plus petite ou égale à 10');
}
elseif($var === 50) // sinon, si $var est égale à 50, on le dit aussi
{
    var_dump('$var est égale à 50');
}
elseif($var === 42) // parce que 42, c'est grand
{
    var_dump('$var est égale à 42');
}
else // sinon, on dit que $var est plus grande que 10 et différente
de 50
{
    var_dump('$var est plus grande que 10 et différente de 50');
}
```

Le mot-clé rajouté, `elseif`, veut dire **sinon si**. Ça vous permet de faire plus de tests. Prenons un exemple : imaginons que vous demandiez à un visiteur de choisir une couleur parmi trois propositions. S'il choisit la première, vous afficheriez qu'il aime le rouge, sinon s'il choisit la seconde, vous lui afficheriez qu'il aime le bleu, sinon c'est qu'il a choisi la troisième, vous lui afficheriez qu'il aime le rose.

Il y a quatre blocs d'instructions différents dans ce code, mais encore une fois, il n'y en a qu'un et un seul exécuté. Vous devez bien comprendre que dès qu'un des blocs est exécuté, les autres ne le seront pas : on sort de la structure conditionnelle et le script reprend après celle-ci. C'est pourquoi, quand vous faites des conditions multiples, vous devez toujours mettre la condition la plus probable (celle qui a le plus de chances d'être vraie) dans le `if`. Ainsi, l'interpréteur PHP ne perdra pas de temps à évaluer les autres conditions. Nous pouvons à nouveau comparer cela à un choix lors de la séparation d'une route, sauf que cette fois-ci la route se sépare plusieurs fois comme illustré à la figure 7.2

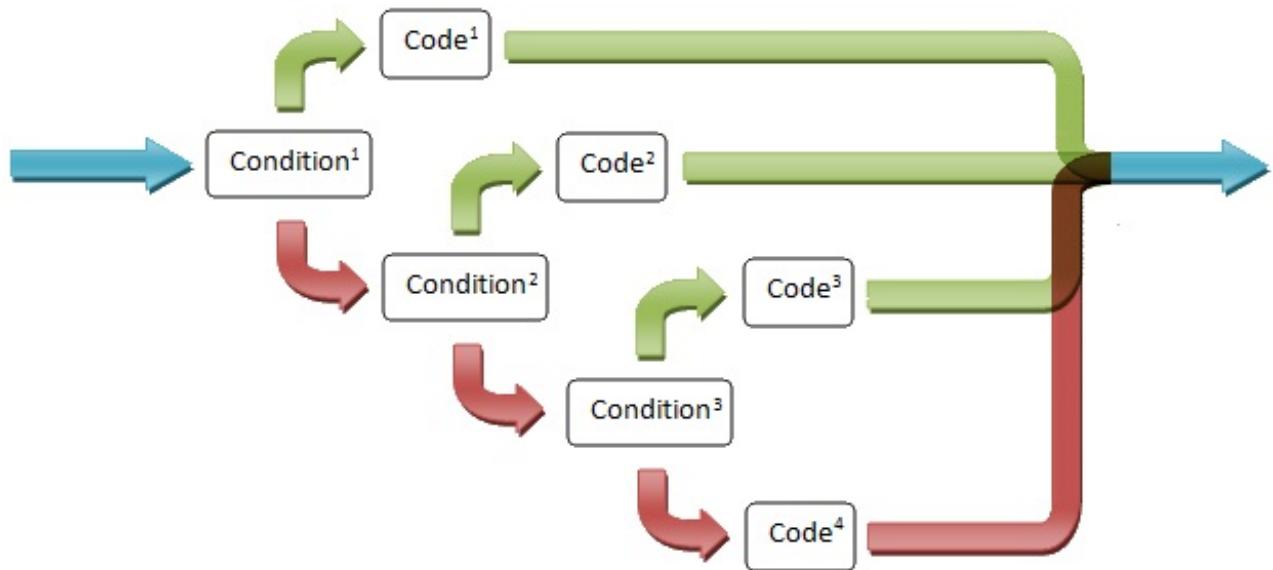


Figure 7.2 — À chaque condition, une nouvelle séparation se fait

Petite note sur le `else` : il est facultatif. Vous n'avez pas besoin de spécifier un `else` si vous n'en avez pas besoin. Imaginons que je veuille afficher quelque chose si `$var` est égale à 70. Dans le cas contraire, je ne veux rien afficher. Au lieu de mettre un bloc d'instructions vide :

#### Code : PHP

```
<?php  
  
$var = 70;  
if($var == 70)  
{  
    var_dump('Oh yeah');  
}  
else  
{  
}
```

On peut oublier le `else` :

#### Code : PHP

```
<?php  
  
$var = 70;  
if($var == 70)  
{  
    var_dump('Oh yeah');  
}
```

Les conditions (comme toutes les autres structures de contrôle, d'ailleurs) ont une place très importante en PHP. Je ne mentirais pas en vous disant que vous allez en bouffer jour après jour, alors tâchez de les maîtriser ! Et pour ça, pas de secret, il faut que vous jouiez avec.

Dernier petit détail qui pourra vous servir : on peut interrompre le bloc de PHP dans les conditions. Je vous ai montré où placer votre code PHP, eh bien sachez que ce code fonctionne :

#### Code : PHP

```
<?php

if(true)
{ // mettez false à la place, et vous verrez
    ?>
    <p>Bonjour !</p>
    <?php
}
else
{
    ?>
    <p>Bonsoir !</p>
    <?php
}
```

Ici, vous voyez que j'ai directement mis le mot-clé true entre les parenthèses du if : on peut le faire, après tout, true a une valeur, c'est donc une expression. Et comme il faut mettre une expression entre les parenthèses, on peut tout à fait le faire.

### Évaluation

Depuis quelques lignes déjà, j'emploie souvent un mot : évaluer. Mais qu'est-ce que ça veut dire ?

Quand vous mettez une expression entre les parenthèses d'un if ou d'un elseif, cette expression est **évaluée comme un booléen** (c'est-à-dire que l'expression est calculée puis convertie, transtypée en sa valeur booléenne). Une expression évaluée en booléen ne peut prendre que deux valeurs bien connues : true ou false. Si l'expression est évaluée à true, le bloc d'instructions est exécuté, sinon il ne l'est pas.

Dans le cas des opérateurs de comparaison et des opérateurs logiques, c'est simple étant donné que ces opérateurs ne renvoient que true ou false. Il n'y a donc aucune ambiguïté possible. Mais imaginons que je sois fou et que fasse ceci :

#### Code : PHP

```
<?php

if('lala')
{
    var_dump('La chaîne lala est évaluée à true');
}
else
{
    var_dump('La chaîne lala est évaluée à false');
```

Je vois d'ici vos grands yeux ébahis : oui, cette chaîne de caractères est évaluée à true. Surprenant, dites-vous ? Je vous réponds que c'est normal. L'interpréteur PHP évalue certaines expressions à true, d'autres à false. La liste des expressions évaluées à true est d'une longueur... infinie, contrairement à la liste de ce qui est évalué à false. C'est pourquoi je vais vous la donner :

- le booléen false ;
- l'entier (int) de valeur 0 ;
- le nombre à virgule (float) 0.0 ;
- la chaîne de caractères vide et la chaîne de caractères « 0 » ;
- null.

La liste n'est pas complète, mais vous ne connaissez pas encore tous les types d'expressions possibles ; ça viendra par la suite.

Je suppose que maintenant, vous savez quoi faire : tester. Amusez-vous à mettre des valeurs différentes de celles-là pour vérifier que c'est bien évalué à true, et testez ces valeurs pour vérifier qu'elles sont bien évaluées à false.

## Autre structure conditionnelle

Les structures précédentes, if, elseif et else, conviennent parfaitement pour traiter des cas où il y a beaucoup de conditions différentes. Mais il y a une autre structure qui peut faire la même chose : switch.

Personnellement, je ne l'utilise presque jamais. Beaucoup prétendent que le switch améliore la lisibilité du code. Pour ma part, je ne vois aucune différence notable mais bon. Vous êtes totalement libres de choisir la structure qui vous plaira, mais moi je préfère le if elseif. Il y a de rares cas où je l'utilise, mais je vous expliquerai ça plus tard, quand on parlera de performance.

Bref, passons aux codes :

### Code : PHP

```
<?php

$nb = 5;
if($nb == 0)
{
    var_dump('$nb vaut 0');
}
elseif($nb == 1)
{
    var_dump('$nb vaut 1');
}
elseif($nb == 2)
{
    var_dump('$nb vaut 2');
}
elseif($nb == 3)
{
    var_dump('$nb vaut 3');
}
elseif($nb == 4)
{
    var_dump('$nb vaut 4');
}
else
{
    var_dump('$nb vaut 5');
}
```

Maintenant, voici exactement la même chose avec un switch :

### Code : PHP

```
<?php

$nb = 5;
switch($nb)
{
    case 0 :
        var_dump('$nb vaut 0');
        break;
    case 1 :
        var_dump('$nb vaut 1');
        break;
    case 2 :
        var_dump('$nb vaut 2');
        break;
    case 3 :
        var_dump('$nb vaut 3');
        break;
    case 4 :
        var_dump('$nb vaut 4');
        break;
    default :
        var_dump('$nb vaut 5');
```

```
}
```

Le résultat est le même, seule la syntaxe diffère. Expliquons quand même cette syntaxe. On la réduit à sa plus simple expression :

#### Code : PHP

```
<?php

switch(Expression)
{
    case Valeur :
        // instruction 1
        // instruction 2
        // ...
        break;
    case AutreValeur :
        // instruction 1
        // instruction 2
        // ...
        break;
    default :
        // instruction 1
        // instruction 2
        // ...
}
```

Entre les parenthèses du switch, vous devez mettre l'expression dont vous voulez tester la valeur. L'interpréteur PHP commence par évaluer cette expression. Ensuite, il va lire le switch ligne par ligne. Lorsque l'interpréteur PHP rencontre un case Valeur :, deux comportements sont possibles :

- soit la valeur de l'expression évaluée est égale à la valeur du case ;
- soit la valeur de l'expression évaluée n'est pas égale à la valeur du case.

Dans le premier cas, l'interpréteur PHP va exécuter les instructions qu'il rencontre jusqu'à ce qu'il trouve un break;. Cette instruction est un peu particulière, je vais en parler brièvement maintenant mais on la reverra plus en détail par la suite. break est une structure de contrôle qui permet de sortir, de briser la structure de contrôle qui l'encadre. Ici, la structure qui encadre le break, c'est le switch ; donc, dès que l'interpréteur PHP rencontre un break;, il sort du switch et les instructions du switch qui n'ont pas été exécutées ne le seront jamais.

Dans le second cas, l'interpréteur PHP va continuer à lire le switch jusqu'à rencontrer un case dont la valeur est égale à celle de l'expression évaluée.

Vous remarquez un dernier mot-clé, default. C'est un case un peu particulier, qui doit être placé en tout dernier. Si tous les case ont échoué (c'est-à-dire que la valeur de l'expression évaluée n'est égale à aucune des valeurs des différents case) et qu'il y un default, alors les instructions de ce dernier bloc seront exécutées. Comme default est toujours le dernier bloc d'un switch, il est inutile de mettre un break puisque le switch se termine juste après.

Le mieux pour bien comprendre, c'est encore et toujours de tester tout et n'importe quoi.

Sachez également qu'il est possible de ne pas mettre d'instruction dans un case ; si ça arrive, l'interpréteur PHP lira les instructions du case suivant :

#### Code : PHP

```
<?php

$i = 1;
switch($i)
{
    case 1 :
    case 2 :
        var_dump('$i est plus petit ou égal à deux');
        break;
    default :
        var_dump('hu ?');
}
```

## Structure de boucle : partie I

À côté des structures conditionnelles, un autre type de structure tient une grande place : les boucles. Concrètement, ça peut servir à beaucoup de choses : afficher un classement, une liste de nombres, etc.

Mieux qu'un long discours :

### Code : PHP

```
<?php  
$lala = 0;  
while(++$lala <= 10)  
{  
    var_dump($lala);  
}
```



Comme vous le voyez, je commence à mélanger tout ce qu'on a vu : expressions, opérateurs en tous genres, variables, structures, etc. Donc s'il y a quoi que ce soit que vous n'avez pas bien compris, allez relire le chapitre qui en parle.

Je suppose que vous avez déjà testé ce code et que vous trouvez le résultat lamentable. Mettez un peu 1000 à la place de 10, et on verra si vous trouvez toujours ça lamentable. 😊

Voici la structure réduite à sa plus simple expression :

### Code : PHP

```
<?php  
while(Expression)  
{  
    // instructions  
}
```

Cette structure est extrêmement simple à comprendre : tant que Expression est évaluée à true, la boucle exécute le bloc d'instructions (les instructions comprises entre les accolades).

Concrètement, que se passe-t-il ?

L'interpréteur PHP arrive sur le while (qui, en français, veut dire « pendant que » ou « tant que ») et évalue l'expression qui est entre parenthèses. Si l'expression est évaluée à true, l'interpréteur PHP exécute le bloc d'instructions puis réévalue l'expression. Si elle est toujours vraie, on recommence et ainsi de suite. Voilà d'ailleurs un risque : une boucle infinie. Imaginons que je mette simplement le booléen true comme expression, ce sera toujours vrai, la boucle ne s'arrêtera jamais (mais l'interpréteur PHP est gentil, il vous indiquera une erreur à force de tourner en rond). Faites donc bien attention aux expressions que vous mettez entre les parenthèses. Vous devez être sûrs qu'elle sera évaluée au moins une fois à false pour que la boucle se termine. Tout ceci est illustré à la figure 8.3.

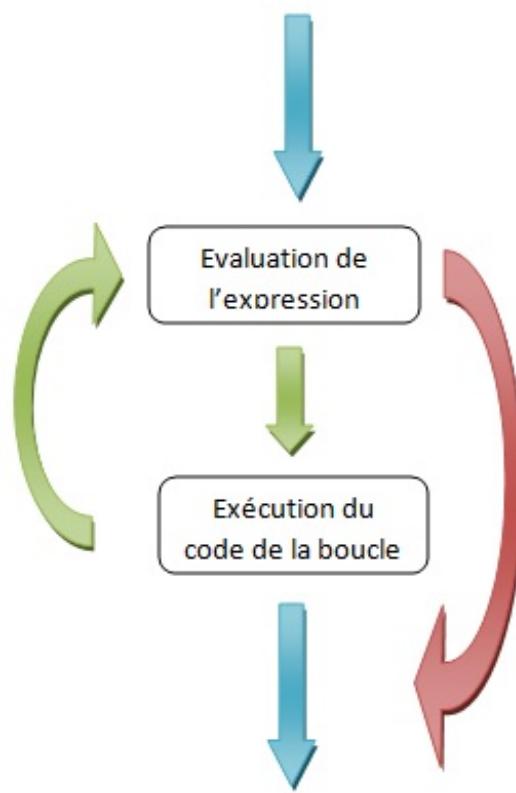


Figure 8.3 — Principe du while

Mais en fait, ceci est faux, l'expression pourrait toujours être évaluée à true ; observons ce code :

**Code : PHP**

```
<?php  
$i = 0;  
while(++$i) // condition toujours vraie, rappelez-vous de ce qui est  
évalué à false  
{  
    if($i === 45) // j'arrête ma boucle après 45 itérations  
    {  
        break;  
    }  
}
```

Vous vous souvenez de break, cette instruction qui sert à sortir de l'instruction de contrôle qui l'encadre ? Eh bien grâce à lui, cette boucle qui paraissait infinie connaît une fin.



break ne peut servir à sortir que des structures de boucle ou d'un switch (pas d'un if, elseif ou else donc).

Dans le code, j'ai glissé un nouveau terme dans mes commentaires : **itération**. Une itération, c'est une exécution de votre boucle. L'évaluation de l'expression et l'exécution du bloc d'instructions, c'est une itération. Ce terme reviendra souvent par la suite, mémorisez-le bien.

## Structure de boucle : partie II

À l'instar des structures conditionnelles, il existe plusieurs structures de boucle, quatre en tout. On en a vu une, on va en découvrir une deuxième tout de suite. La troisième sera vue dans le chapitre suivant et la quatrième à un autre moment .

Bref, voici sans plus attendre un code qui met en avant cette structure :

**Code : PHP**

```
<?php
```

```
$lala = 0;  
do  
{  
    var_dump($lala);  
} while($lala++);
```

Comme d'habitude, la structure réduite à sa plus simple expression :

**Code : PHP**

```
<?php  
  
do  
{  
    // instructions  
} while(Expression);
```

Si vous avez compris comment fonctionne while, vous devriez comprendre celle-ci aussi. Tant que l'expression entre les parenthèses du while est évaluée à true, on refait un tour, la boucle se termine donc dès qu'elle est évaluée à false comme vous pouvez le voir à la figure 8.4.

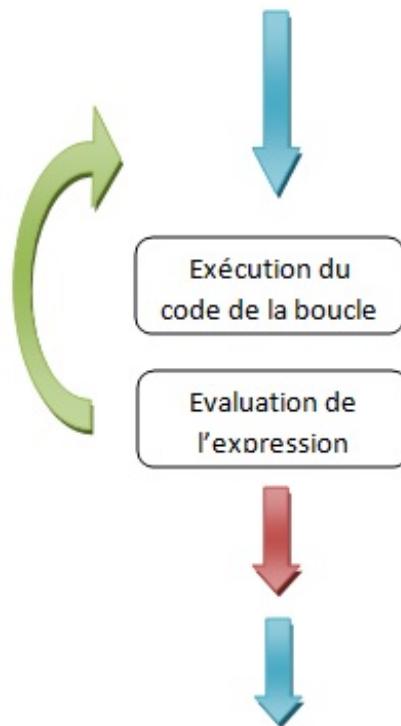


Figure 8.4 — Principe du do while

Mais alors pourquoi créer ces deux structures ? À première vue, elles paraissent très similaires, voire identiques. Mais un détail les différencie : le nombre d'itérations. La première structure de boucle qu'on a vue fera de 0 à n itérations. Cette dernière structure fera de 1 à n itérations.

Testons :

**Code : PHP**

```
<?php  
  
while(false)  
{  
    var_dump('while');  
}
```

```
do
{
    var_dump('do while');
} while(false);
```

Si vous savez lire, vous constatez que « while » ne s'affiche jamais. L'expression étant false, la boucle ne fait pas une seule itération. Par contre, un « do while » s'affiche. Eh oui : même si l'expression du do while est évaluée à false, le bloc d'instructions sera quand même exécuté une fois. D'où la différence entre le nombre d'itérations que j'ai mentionnée ci-dessus.  
Si jamais vous avez une erreur de ce type :

#### Code : Console

```
Parse error: syntax error, unexpected $end in MesRepertoires\page.php on line 21
```

C'est probablement que vous avez oublié de fermer une accolade ; alors vérifiez qu'elles aillent toujours par paires !

Vous aurez bientôt droit à un TP, foncez lire le chapitre sur les fonctions.  
Mais si vous avez du mal à apprêhender quelque chose, prenez votre temps, comme toujours !

## Les fonctions

Je dois vous l'avouer, j'attendais ce chapitre depuis longtemps. En effet, dans ce chapitre, je vais vous expliquer un élément dont on s'est beaucoup servi, mais dont vous ignorez le fonctionnement : var\_dump(). Comme vous devez vous en doutez vu le titre du chapitre, var\_dump() est ce qu'on appelle une **fonction**.

### Première approche

On va commencer, comme à notre habitude, par une petite histoire.

Quand vous êtes de gentils petits garçons (ou de gentilles petites filles, pas de discrimination), il peut arriver que vous demandiez à votre maman de vous éplucher une pomme. Si elle voit votre grand sourire, elle dira oui, ira chercher une pomme, l'épluchera, la coupera et vous la servira sur une petite assiette.

En ne faisant que dire : « Tu peux m'éplucher une pomme ? », vous avez fait exécuter plusieurs actions à votre maman. C'est le principe même d'une fonction.

Allons plus loin et demandons maintenant à maman si elle peut nous éplucher une orange. Elle va faire la même chose, à savoir : dire oui, aller chercher l'orange, l'éplucher, la couper et la servir sur une assiette.

En résumé, si vous dites : « Tu peux m'éplucher une pomme ? », votre mère exécute toutes les actions (prendre, éplucher, couper et servir) sur la pomme.

Si vous dites : « Tu peux m'éplucher une orange ? », elle fera les mêmes actions, mais sur une orange.

Une fonction est caractérisée par trois choses : son nom, ses paramètres et sa valeur de retour.

Si l'on applique ça à l'exemple précédent, le nom est « Tu peux m'éplucher », le paramètre est « une pomme » (ou « une orange ») et la valeur de retour est la pomme ou l'orange découpée servie sur une petite assiette.

Beaucoup d'appareils que vous utilisez tous les jours ont ce genre de fonction, par exemple une friteuse. La seule chose qu'on change, c'est la température à laquelle on veut chauffer la graisse et la friteuse fait le reste en fonction de ça.

Imaginez que vous ayez cinquante variables et que vous deviez calculer le volume de la sphère de rayon égal à la valeur de la variable. Il existe une formule, vous pourriez faire cinquante fois le calcul, un calcul par variable. Mais ça serait très long à écrire. Tandis que si vous faisiez une fonction, vous taperiez une fois la formule, puis il vous suffirait de taper le nom de la fonction avec les variables (qui seront les paramètres) et la fonction vous retournerait le volume de la sphère.

Pour résumé, une fonction est une structure, un objet auquel vous donnez des **entrées** sur lesquels elle **travaille** pour vous donner une **sortie**, comme illustré à la figure 8.1

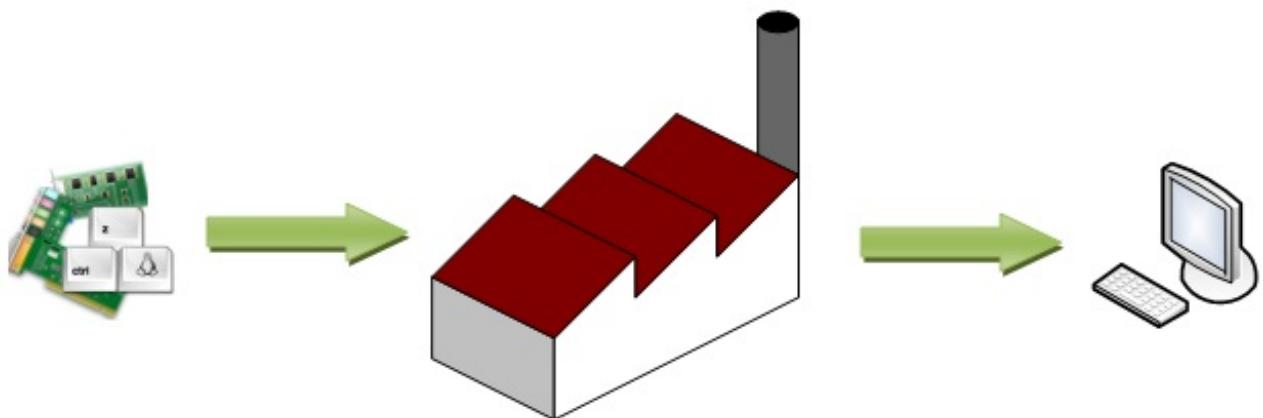


Figure 8.1 — Principe d'une fonction, celle-ci nous monte un ordinateur à partir des composants

### Votre première fonction

Maintenant que vous savez quel est le but des fonctions, on va en créer une, voici le code :

Code : PHP

```
<?php  
  
function volumeSphere($rayon)  
{  
    return (4 * 3.14 * $rayon * $rayon * $rayon) / 3;  
}
```

On va étudier ce code, mais d'abord le simplifier :

#### Code : PHP

```
<?php  
  
function NAME ($PARAM)  
{  
    // bloc d'instructions  
    return Expression;  
}
```

Voilà le code minimal pour créer une fonction (je mens, vous comprendrez après pourquoi).

Tout d'abord, il y a le mot-clé **function**, il indique au PHP qu'on veut déclarer une fonction (à ne pas confondre avec une déclaration de variable). Vient ensuite le nom de la fonction, ici NAME, c'est avec ce nom que vous appellerez la fonction par la suite.

Entre les parenthèses, vous trouvez \$PARAM, c'est une variable, c'est un paramètre (une information dont la fonction a besoin pour faire son job).

On trouve aussi des accolades, comme pour les structures de contrôle ; et un dernier mot-clé subsiste : **return**. Ce mot-clé indique quelle expression la fonction doit retourner.

Si on l'applique à l'exemple de départ, on aurait une fonction du genre :

#### Code : PHP

```
<?php  
  
function tuPeuxMeplucher ($type)  
{  
    return couper(eplucher(prendre($type)) );  
}
```



couper(), eplucher() et prendre() sont également des fonctions ; en effet, on peut appeler des fonctions à l'intérieur même d'une fonction.

Pour appeler cette fonction et donc recevoir un fruit, on aurait un code du style :

#### Code : PHP

```
<?php  
  
$fruit = 'Une pomme';  
var_dump(tuPeuxMeplucher($fruit));
```



Bien évidemment, ne testez pas ces codes, ils ne sont pas fonctionnels.

Revoilà notre var\_dump()... il va nous servir. **C'est une fonction**. Souvenez-vous de ce que j'ai dit sur son utilisation : j'ai dit qu'il fallait mettre une expression entre les parenthèses pour afficher le type et la valeur de l'expression.

Ce qui sous-entend que... ? Eh oui, une fonction peut être une expression. Souvenez-vous de ce qu'est une expression, quelque chose qui possède une valeur. Une fonction retourne une expression, or une expression est une valeur : une fonction retourne donc une valeur. Et comme la fonction retourne une valeur, elle possède forcément une valeur, ce qui implique que c'est une expression (je vous conseille de relire deux ou trois fois cette phrase, soyez sûr de la comprendre).

J'ai dit que je vous mentais : c'est vrai. Le mot-clé **return** qui indique quelle expression doit retourner la fonction n'est pas indispensable. On peut l'enlever, la fonction ne retournera donc rien et ce ne sera donc pas une expression. Si toutefois vous tentez de l'utiliser comme une expression, par exemple dans une addition, l'interpréteur PHP agira de la même façon qu'avec une variable qui n'existe pas et considérera que la fonction retourne null.

En général, on utilise davantage de fonctions qui retournent des valeurs, mais il existe des tas de fonctions qui n'en retournent pas — un exemple que vous connaissez bien est var\_dump(). Essayez donc de faire un var\_dump() d'un var\_dump() pour voir.

**Code : PHP**

```
<?php  
var_dump(var_dump(1));
```

var\_dump(1) affiche « int(1) », c'est son rôle, mais ne retourne rien comme je vous l'ai dit. Et donc le var\_dump() englobant nous affiche un triste et solitaire null.

**Souvenir, souvenir**

Si vous n'avez pas bien compris le concept d'expression et d'opérateur, c'est maintenant que vous allez le sentir. 😊

On a vu qu'une fonction pouvait être une expression, on peut donc assigner cette expression à une variable grâce à l'opérateur d'affectation.

Petit exemple avec la fonction du volume d'une sphère :

**Code : PHP**

```
<?php  
// on déclare la fonction, elle ne fait rien, l'interpréteur PHP va  
juste savoir qu'elle existe  
function volumeSphere($rayon)  
{  
    return (4 * 3.14 * $rayon * $rayon * $rayon) / 3;  
}  
// on choisit un rayon arbitrairement  
$diametre = 5;  
// on calcule le volume (on a besoin du rayon comme paramètre ; or,  
le rayon est égal au demi-diamètre)  
$volume = volumeSphere($diametre/2);  
var_dump($volume);
```

Ce code vous affichera le volume d'une sphère de rayon 2.5.

Il est évidemment inutile d'utiliser une variable pour rien (\$volume) dans ce genre de cas ; il suffit de mettre la fonction directement dans le var\_dump() comme ceci :

**Code : PHP**

```
<?php  
$diametre = 5;  
var_dump(volumeSphere($diametre/2));
```

Les deux codes reviennent strictement au même si vous n'avez plus besoin du volume par la suite, mais le second est meilleur car on ne crée pas de variable inutilement.

Il peut arriver que votre fonction ait besoin de plusieurs paramètres, par exemple une fonction qui calculerait l'aire d'un rectangle (l'aire est égale à la longueur multipliée par la largeur). Si vous avez besoin de plusieurs paramètres, il suffit de les séparer par une virgule tant dans la déclaration de la fonction que dans son utilisation. Exemple :

**Code : PHP**

```
<?php  
function aireRectangle($largeur, $hauteur)  
{  
    // on peut faire ainsi :  
    $aire = $largeur*$hauteur;  
    return $aire;  
    /*
```

```
// Mais aussi ainsi (et c'est même recommandé, car il y a moins
// d'instructions et de variables inutiles) :
return $largeur*$hauteur;
// D'ailleurs, vous devriez y avoir pensé quand j'ai dit que return
retournait la valeur de l'expression
*/
}

var_dump(aireRectangle(5, 4)); // l'aire doit être égale à 20,
testez donc
```

Il y a un troisième cas qui peut se présenter, celui où la fonction possède des paramètres facultatifs, c'est-à-dire qui ne sont pas indispensables. Reprenons l'exemple de la sphère :

#### Code : PHP

```
<?php

function volumeSphere($rayon = 5)
{
    return (4 * 3.14 * $rayon * $rayon * $rayon) / 3;
}

var_dump(volumeSphere());
var_dump(volumeSphere(10));
var_dump(volumeSphere());
```

Ce code va vous afficher les trois volumes, le premier et le troisième étant les mêmes.

Vous le voyez, la différence est claire : dans le premier et le troisième appel de la fonction, on ne donne pas le rayon de la sphère, car j'ai rendu ce paramètre facultatif.

Regardez la déclaration de la fonction : au lieu d'avoir juste \$rayon, j'ai mis **\$rayon = 5**. En faisant ça, je donne une valeur par défaut à cette variable. Si je ne donne pas de valeur pour ce paramètre, l'interpréteur PHP utilisera celui par défaut (ce qui explique pourquoi le premier et le dernier appel de la fonction renvoient la même valeur).

Attention, cependant. Quand votre fonction possède plusieurs paramètres mais que vous ne voulez pas les rendre tous facultatifs, vous devez mettre les paramètres facultatifs le plus à droite possible dans la déclaration de la fonction.

#### Code : PHP

```
<?php

// MAUVAIS
function aireRectangle($largeur = 10, $hauteur)
{
    return $largeur*$hauteur;
}
// correct
function aireRectangle($hauteur, $largeur = 10)
{
    return $largeur*$hauteur;
}
```

Mais pourquoi ?

Réfléchissez. Comment appelleriez-vous la première fonction en utilisant son paramètre facultatif ?

#### Code : PHP

```
<?php

aireRectangle(, 5); // ainsi ? Incorrect
aireRectangle(5); // ainsi ? Incorrect (là, en réalité, vous donnez
le paramètre $largeur mais pas $hauteur)
```

Avec la seconde fonction, plus de problème :

**Code : PHP**

```
<?php  
  
aireRectangle(5); // OK, on renseigne juste la hauteur, et la  
largeur utilisée est celle par défaut  
aireRectangle(5, 90); // on renseigne la hauteur et la largeur
```



Les noms de vos fonctions subissent la même règle que les noms des variables : « *ils commencent par un underscore ou un caractère alphabétique, puis n'importe quel caractère alphanumérique ou un underscore* ».



Les valeurs par défaut que vous donnez à vos fonctions doivent être des constantes, vous ne pouvez pas donner une variable ou le résultat d'une fonction comme valeur par défaut à un argument d'une fonction.

## Les fonctions, c'est trop fort !

Vous avez eu un avant-goût : maintenant, on passe à table.

Plusieurs petites choses sympas sont possibles avec les fonctions comme les fonctions « héritables » et les fonctions récursives. Mais commençons d'abord par les fonctions conditionnelles.

Lorsque l'interpréteur PHP va interpréter votre code, il va commencer par analyser sa syntaxe. S'il y a une erreur, il vous l'indique. Après cette analyse, l'interpréteur PHP va enregistrer temporairement toutes les fonctions que vous aurez créées avant même que votre script ne commence réellement à être exécuté. C'est pour ça que vous pouvez utiliser une fonction dans votre code avant qu'elle ne soit déclarée ; exemple :

**Code : PHP**

```
<?php  
  
lala();  
  
function lala()  
{  
    var_dump('lala');  
    /*  
     * Cette fonction affiche quelque chose à l'écran  
     * Mais ce n'est pas une expression  
     * Comme il n'y a pas de return, il n'y a pas de valeur de retour  
     * Appeler la fonction n'apporte pas de valeur  
     * Et comme ce qui n'a pas de valeur n'est pas une expression...  
    */  
}
```

Maintenant, que se passe-t-il si je mets ceci ?

**Code : PHP**

```
<?php  
  
if(Expression)  
{  
    function lala()  
    {  
        var_dump('lala');  
    }  
}  
  
lala();
```

Mettez donc d'abord true comme expression dans le if, et puis false. Vous constaterez qu'en fonction de ça, vous aurez soit String(4) "lala" qui s'affiche, soit une erreur incompréhensible (pour le moment, du moins).

Quand je dis que l'interpréteur PHP enregistre toutes les fonctions avant même que le script ne commence à être exécuté, ce n'est pas tout à fait vrai. L'interpréteur PHP enregistre toutes les fonctions qui sont en dehors d'un bloc d'instructions (en dehors d'un if, d'un else, d'un for, d'un while, d'une... fonction !). Si j'avais appelé la fonction avant le if, ça aurait toujours retourné une erreur ; en effet, la fonction n'est déclarée qu'après que le if a été interprété. Si vous appelez la fonction avant qu'elle ne soit déclarée, pan : une erreur.

Vous devez donc bien prendre garde quand vous utilisez des fonctions dans des structures de contrôle.

Et voici le second jouet : une fonction dans une fonction. Un petit code :

**Code : PHP**

```
<?php

function a()
{
    function b()
    {
        var_dump(0);
    }
}

b();
```

Un erreur ? Mais comment est-ce possible ?

C'est très simple : la fonction b() est déclarée dans une autre fonction, ce qui implique que tant qu'on n'a pas appelé la fonction dans laquelle elle est déclarée, la fonction n'existe tout simplement pas.

Le code correct serait donc :

**Code : PHP**

```
<?php
function a()
{
    function b()
    {
        var_dump(0);
    }
}

a(); // la fonction a() est exécutée, la fonction b() est donc
déclarée
b();
?>
```

Vous voyez, rien de magique dans le PHP, tout est logique (ou presque, mais bon). Quand vous utilisez cela, faites attention à ne pas utiliser de fonction non déclarée.

Le numéro trois : les fonctions récursives. Ce genre de fonction est plus ou moins souvent utilisées en fonction de ce qu'on fait, ce sont des fonctions qui peuvent s'appeler elles-mêmes au besoin.

Un exemple :

**Code : PHP**

```
<?php

function lala($chiffre)
{
    if($chiffre < 5)
    {
        var_dump('Fin');
    }
}
```

```
    }
    else
    {
        var_dump('Continue');
        lala(--$chiffre);
    }
}

lala(11);
lala(4);
```

Courage, plus que quelques lignes et ce sera fini !

L'avant-dernière chose dont on doit parler, ce sont les fonctions variables.  
Pour l'instant, on appelait toujours nos fonctions de cette façon :

#### Code : PHP

```
<?php

nomFonction($parametre);

?>
```

Mais on peut le faire autrement...

#### Code : PHP

```
<?php

$nom = 'nomFonction';
$nom($parametre);

?>
```

Le résultat sera strictement identique. Quand vous mettez des parenthèses à la droite du nom d'une variable, l'interpréteur PHP va appeler la fonction dont le nom est égal à la valeur de la variable (après avoir converti la valeur en chaîne de caractères au besoin). Ça peut être pratique dans certains cas, on en rencontrera sans doute, et de toute façon vous devez le savoir au cas où vous rencontreriez un script qui utilise cela.

Allez pour le plaisir, dernier point : la **portée des variables**. Je ne vais pas tout vous expliquer, ça serait un peu long. Souvenez-vous de la figure 8.1, j'ai schématisé la fonction par une usine, parce que c'est un assez bon comparatif. Une fonction, toute comme un usine est un peu comme une boîte noire : on sait ce qui y rentre, on sait ce qui en sort, mais on ne sait pas du tout ce qu'il s'y passe, l'usine est un monde à part sans aucun rapport avec le notre. En PHP c'est pareil, vous devez voir vos fonctions comme des petites usines qui font un certain travail et que vous utilisez mais dont vous ignorez le fonctionnement. Prenez par exemple var\_dump() : vous savez ce que vous lui donnez, vous savez ce qu'elle produit, mais vous ne savez pas comment elle le fait. D'ailleurs, qui s'en soucie ? Le propre d'une fonction, son but, c'est d'assurer un rôle, de faire quelque chose de particulier sans que l'utilisateur ne se soucie ni du comment ni du pourquoi. Cela a une conséquence direct sur votre code : n'essayez jamais d'utiliser une variable définie hors d'une fonction dans la fonction, ou inversement (donc, utiliser une variable définie dans une fonction hors de la définition du bloc de code de cette fonction). Sans quoi vous aurez droit à de belles erreurs. La raison tient dans cette idée de monde à part : puisque la fonction — l'usine — est dans un monde à part, elle ne connaît pas les variables qui sont en dehors d'elle-même, ces variables n'existent pas pour elle. Le raisonnement inverse est le même : puisque la fonction est dans un monde à part, les variables en son sein ne sont pas connues de votre script puisqu'elle n'existe tout simplement pas.

Nous reviendrons sur cette notion de portée des variables par la suite, ne vous en faites pas.

Si vous rencontrez une erreur de ce type :

#### Code : Console

```
Fatal error: Call to undefined function meuh() in C:\www\test.php on line 32
```

C'est que vous employez une fonction qui n'existe pas : vérifiez bien le nom que vous avez tapé.

Voilà un chapitre attendu qui doit vous avoir ramolli la tête : n'hésitez surtout pas à le relire, les fonctions tiennent une place très importante dans le PHP (et dans bon nombre de langages de programmation) !

## Les chaînes de caractères, un casse-tête

Lors d'un chapitre précédent, on a vite abordé ce qu'étaient les chaînes de caractères : apostrophes, guillemets, échappement, etc.

Mais il y a encore bien des choses à dire sur ce type bien particulier.

Zoup, jetez-vous sur ce chapitre ô combien intéressant.

### Bref rappel et nouvelle structure

Avant de vous rappeler ce qu'on a déjà vu sur les chaînes de caractères, je vais vous montrer un gentil compagnon : echo. Un exemple :

#### Code : PHP

```
<?php  
echo 'lala';  
$e = 4;  
echo $e;  
echo 4 < 5;
```

Testez ce code et vous verrez s'afficher « lala41 » à l'écran. Vous devriez déjà avoir compris ce que fait echo, mais au cas où : echo est une structure du langage qui affiche la valeur de l'expression qui se situe à sa droite. C'est un peu comme var\_dump(), à la différence près qu'on ne s'occupe pas du type de la variable, echo n'affiche que la valeur de l'expression.

Il y a une seconde différence entre echo et var\_dump() : le premier élément est une structure de langage alors que le second est une fonction. Ce n'est qu'une nuance, mais ça a son importance.

 Vous pouvez utiliser des parenthèses pour délimiter l'expression qu'echo va afficher, c'est d'ailleurs à cause de cela que beaucoup pensent que c'est une fonction, alors que techniquement, c'est une structure de langage.

Puisque cela est fait, passons au rappel. On a vu qu'une chaîne de caractères était délimitée par des apostrophes ('') ou des guillemets ("") ; ainsi, ce code crée deux chaînes de caractères :

#### Code : PHP

```
<?php  
'lala';  
'une belle chaine ...';
```

N'oubliez pas que comme je n'affecte pas ces chaînes de caractères à des variables, elles n'existent que quand l'interpréteur PHP les lit.

On se souviendra également de l'échappement, cette méthode qui permet de mettre des apostrophes ou des guillemets dans les chaînes de caractères. En effet, si l'on veut mettre une apostrophe et qu'on utilise des apostrophes pour délimiter la chaîne de caractères, l'interpréteur PHP ne va pas comprendre ce qu'on veut et nous enverra une erreur. Exemple :

#### Code : PHP

```
<?php  
$lala = 'J'aime le rose';  
echo $lala;  
?>
```

En faisant précéder l'apostrophe d'un *backslash* (\), le problème disparaît :

#### Code : PHP

```
<?php  
echo $lala = 'J\'aime le rose';  
?>
```

Si l'on veut afficher un *backslash* suivi d'une apostrophe, il faudra échapper le *backslash* en le faisant précédé par un autre, et on fera de même pour l'apostrophe :

**Code : PHP**

```
<?php  
echo 'Oh oui du compliqué \\\' que c\'est cool';  
?>
```

Enfin, ce système d'échappement est strictement identique si l'on utilise des guillemets pour délimiter les chaînes de caractères, sauf que cette fois ce sont les guillemets et non les apostrophes que l'on devra échapper.

### Différence entre guillemets et apostrophes

Après ce bref rappel, on va pouvoir cerner la globalité de la problématique des deux syntaxes pour créer des chaînes de caractères.

Mieux qu'un long discours, un exemple :

**Code : PHP**

```
<?php  
  
$pseudo = 'Haku';  
  
echo 'Bonjour $pseudo';  
echo "Bonjour $pseudo";
```

Devant vos yeux ébahis, alors qu'à première vue les deux instructions echo devraient afficher la même chose, une différence apparaît.

La première instruction affiche « Bonjour \$pseudo » alors que la seconde affiche « Bonjour 'Haku ». Voilà la première différence entre les apostrophes et les guillemets : dans une chaîne de caractères délimitée par des guillemets, les variables sont remplacées par leur valeur. C'est plutôt pratique, ça permet de simplifier les codes. Exemple :

**Code : PHP**

```
<?php  
  
$pseudo = 'lala';  
echo 'Bienvenue ' ;  
echo $pseudo;  
echo ', comment allez vous ?';
```

Avec des guillemets, on peut faire ainsi :

**Code : PHP**

```
<?php  
  
$pseudo = 'lala';  
echo "Bienvenue $pseudo, comment allez-vous ?";
```

Le rendu est le même, mais le second code est plus court et un peu plus élégant. Toutefois, certains problèmes peuvent se poser :

**Code : PHP**

```
<?php
```

```
echo "hohé $noel, comment vas-tu ?";
```

Ce n'est vraiment pas de chance, mais on a une belle erreur qui s'affiche. En effet, quand l'interpréteur PHP lit la chaîne de caractères, il voit un \$. Si le caractère qui suit le \$ est un *underscore* (\_) ou un caractère alphabétique (minuscule ou majuscule, peu importe), il remplacera ce nom par la variable correspondante. Comme la variable \$noel n'existe pas, l'interpréteur PHP nous le signale en nous renvoyant une erreur. Souvenez-vous de la convention de nommage des variables pour savoir ce qui sera considéré comme un nom de variable par l'interpréteur PHP (ou allez relire le chapitre sur les variables). Quelques exemples pour bien comprendre :

#### Code : PHP

```
<?php  
  
$lala = 4;  
echo "Tu as $lala enfants"; // OK, la variable existe  
echo "Tu as $lili filles"; // pas bon, la variable n'existe pas  
echo "Tu as $9"; // OK, on se souvient que le nom d'une variable  
commence par un underscore ou un caractère alphabétique
```

Si malgré tout, vous avez besoin d'afficher le symbole \$ devant un *underscore* ou un caractère alphabétique, il suffit de l'échapper en le faisant précédé d'un *backslash* :

#### Code : PHP

```
<?php  
  
echo "lala \$soleil";
```

On peut mieux saisir la différence entre les deux types de chaînes, maintenant : l'une est passive, et l'autre active. Les chaînes délimitées par des apostrophes ne subiront aucun changement, elles resteront telles qu'on les a écrites et peu importe ce que la chaîne contient. Par contre, en fonction des caractères présents dans les chaînes délimitées par des guillemets, il peut se produire des choses comme un remplacement d'un bout de la chaîne par la valeur d'une variable.

Il existe plusieurs symboles qui ont une signification pour les chaînes délimitées par des guillemets, les voici :

Symbole	Signification
\n	Fin de la ligne
\t	Tabulation
\r	Retour à la ligne

Il y en a encore deux autres, mais je ne les présenterai pas car ils ne sont pas très intéressants pour le moment ; vous pouvez cependant trouver la liste exhaustive [ici](#).

Pour échapper ces symboles, il suffit encore une fois de les précédé d'un *backslash*. Voici quelques exemples illustrant les différents symboles (regardez la source de la page et non la page en elle-même : clic droit sur la page > **Afficher la source / le code source**) :

#### Code : PHP

```
<?php  
  
echo 'lala \n lala';  
echo "lala \n lala";  
echo "lala \\n lala";  
  
echo 'lala \t lala';  
echo "lala \t lala";  
echo "lala \\\t lala";
```

```
echo 'lala \r lala';
echo "lala \r lala";
echo "lala \\r lala";

$a = 6;

echo 'lala $a lala';
echo "lala $a lala";
echo "lala \$a lala";

echo 'lala $b lala';
echo "lala $b lala";
echo "lala \$b lala";
```

Et voilà : on en a fini avec les différences entre les deux types de chaînes de caractères, n'hésitez surtout pas à tester un peu tout et n'importe quoi pour vous familiariser avec.

## Nouvel opérateur : concaténation



Puisque vous maîtrisez les deux types de chaînes de caractères, je vais mélanger les deux sans vergogne ; alors si vous croyez ne pas avoir tout compris, relisez, testez, sinon vous serez largués.

Imaginons que nous ayons deux variables et que nous voulions les afficher, comment ferait-on cela ?

### Code : PHP

```
<?php

$a = 'je suis';
$b = 'moi';
echo $a;
echo $b;
```

C'est long et répétitif, vous ne trouvez pas ? Eh bien ne pleurez plus, les développeurs du PHP ont prévu le coup en créant un opérateur génial : l'opérateur de concaténation. Redites ce mot dix fois de suite à haute voix et très rapidement.

Ne cherchez pas la définition de ce mot dans le dictionnaire, vous avez peu de chances de la trouver. Heureusement, on a appris à faire quelque chose qui va nous sauver la vie : chercher par nous-mêmes. Si vous voulez savoir ce qu'est la concaténation, bougez-vous les doigts. 😊

### Secret (cliquez pour afficher)

Le terme concaténation est issu du latin **con** (avec) et **catena** (chaîne) : il désigne l'action de mettre bout à bout deux chaînes. En programmation, on appelle la concaténation de deux chaînes de caractères la chaîne formée de ces deux chaînes mises bout à bout.

Si vous ne savez pas ce qu'est la concaténation, allez sur Google ou sur un autre moteur de recherche, sinon préparez-vous à entrer dans un monde merveilleux.

L'opérateur de concaténation varie en fonction des langages ; en JavaScript, c'est le symbole plus (+). En PHP, c'est le symbole point (.) .

Vous devez vous souvenir comment fonctionnent les opérateurs, non ? Si ce n'est pas le cas, allez relire ce chapitre !

Mettons la théorie en pratique :

### Code : PHP

```
<?php

$a = 'Je suis';
$b = 'moi';
$c = $a . $b;
echo $c;
```

Fantastique, n'est-ce pas ?

Comme tout opérateur qui se respecte, l'opérateur de concaténation concatène deux expressions et en retourne une issue des deux autres. Ainsi, on peut réécrire le code de cette façon :

#### Code : PHP

```
<?php  
$a = 'Je suis';  
$b = 'Moi';  
echo $a . $b;
```

C'est logique, après tout. On sait qu'`echo` affiche la valeur de l'expression qui est à sa droite. On sait également que l'opérateur de concaténation retourne une expression. Enfin, on sait que l'expression retournée par l'opérateur de concaténation est égale aux deux expressions qu'il doit concaténer mises bout à bout.

Ça fonctionne avec tous les types de variables vus précédemment, à une exception près : lequel, à votre avis ?

#### Code : PHP

```
<?php  
echo 'a' . '1' . 'c';  
echo 'a' . 1 . 'c';  
echo 'a' . 5.65 . 'c';  
echo 'a' . true . 'c';
```

Faites TRÈS attention quand vous voulez concaténer des nombres à virgule : en effet, cette instruction : `$a = 'b' . 5 . 65;` ne produit pas le même résultat que celle-ci : `$a = 'b' . 5.65;`.

Prenons un exemple simple : je veux concaténer 5 et 7, j'écris naïvement cette instruction : `$a = 5.7;`.

Faites un `var_dump()`, et ô surprise, ça nous dit que c'est un nombre à virgule. Normal : pour créer un nombre à virgule, on utilise également un point.

C'est pourquoi je vous conseille de toujours mettre un espace entre l'opérateur de concaténation et les expressions ; exemple : `$a = 5 . 7;`.

Là, on n'a plus d'erreur. C'est une simple habitude à prendre, mais elle vous évitera bien des pièges.

Vous avez dû remarquer quelque chose de bizarre pour le dernier `echo` : en effet, ça nous affiche « `a1c` » au lieu de « `atruec` ». C'est tout à fait normal : l'opérateur de concaténation ne peut concaténer que des chaînes de caractères ; si on lui donne d'autres types, l'interpréteur PHP les transtype en chaînes de caractères. Pour les entiers et les nombres à virgule, ça ne pose pas de problème puisque les valeurs ne diffèrent pas. Mais quand on transtype un booléen en chaîne de caractères, ça donne soit 1 si le booléen vaut `true`, soit 0 si le booléen vaut `false`. On verra un peu plus tard quels types sont transtypés, et en quoi, dans différents cas !

On peut bien évidemment concaténer des variables, après tout ce sont des expressions comme les autres :

#### Code : PHP

```
<?php  
$var =  
'a';  
$var = $var . 'a';  
echo $var;
```

Ça fonctionne, mais vous ne trouvez pas ça lourd d'écrire quatre fois `$var` ? Personnellement, ça m'arrache les mains. C'est pour ça qu'à l'instar des opérateurs arithmétiques, il y a un second opérateur de concaténation : l'opérateur d'affectation de concaténation. Exemple :

#### Code : PHP

```
<?php
```

```
$var = 'a';
$var .= 'a';
echo $var;
```

Vous voilà incollables sur la concaténation. C'est souvent un concept qui pose problème : on oublie d'échapper des apostrophes ou des guillemets dans des chaînes de caractères, l'opérateur perd son rôle, on a des erreurs en pagaille, bref, ça va mal. Toutefois, si vous vous souvenez de son rôle (qui est de concaténer deux **expressions**) et que vous savez ce qu'est une expression, alors tout devrait bien se passer.



L'opérateur de concaténation a la même priorité que les opérateurs arithmétiques d'addition et de soustraction.

L'opérateur d'affectation de concaténation à la même priorité que les opérateurs `+=`, `-=`, etc.

À présent, les chaînes de caractères ne devraient plus avoir de secret pour vous, et c'est tant mieux car vous vous en servirez souvent et de plusieurs façons.

## Le type manquant : les arrays

Dans bien des langages de programmation, un autre type de variables très utile existe : les arrays (ou tableaux). Les utilisations sont très diverses, mais aussi très simples. En retenant un ou deux points théoriques, vous pourrez faire de grandes choses ! (Je peux vous l'assurer : quand vous commencerez à évoluer par vous-mêmes, vous boufferez des arrays au petit dej !)

### Faisons connaissance

Si vous avez l'âme bricoleuse, vous avez sans doute une boîte à outils, un genre de boîte qui vous permet de ranger et transporter facilement tous vos outils.

Imaginez-vous un beau matin, le soleil brille, les coqs chantent, le chat ronronne et le chien mâche délicatement votre paire de chaussures préférée. En bon ouvrier que vous êtes, vous allez dans votre atelier pour préparer les outils en prévision de la dure journée de labeur qui se profile.

Manque de chance, vous ne trouvez plus vos outils. En cherchant, vous sortez votre tournevis d'un pot de fleurs, le marteau du dessous de la table, la perceuse pendue au plafond et bien d'autres dans des endroits plus insolites les uns que les autres.

Manque de chance, cette recherche vous a fait perdre beaucoup de temps et vous devez sauter le petit déjeuner, vous êtes alors très grognons.

Un jour, un ami vous amène une invention révolutionnaire : une boîte à outils. Il vous explique son fonctionnement en deux mots : vous rangez vos outils dans cette boîte, vous vous souvenez où est la boîte et ainsi, vous avez tous vos outils à portée de main en un instant.

Heureusement que cet ami est là, car il vient de vous expliquer ce qu'est un array. Schématiquement, on pourrait comparer un array à une boîte à outils. La boîte à outils, c'est l'array. Les compartiments de la boîte à outils, ce sont les **index**, les **clés** de l'array (ce qui permet d'associer un nom simple à une expression) et les outils rangés dans les compartiments, ce sont les données, les expressions.

Comme je suis presque sûr que vous n'avez pas suivi cette explication un peu tirée par les cheveux, on va passer à la pratique, ça devrait vous aider à mieux cerner ce qu'est un array. Un petit code pour la route :

#### Code : PHP

```
<?php  
  
$var = array('lili', 'lala');  
echo $var[0];
```

Ce code vous affiche « lili », mais comment ça marche ?

Pas de panique : comme toujours, une explication va suivre. Vous avez sous les yeux la façon la plus simple de créer un array et de le remplir avec quelques données ; seulement, ce n'est pas pratique du tout à expliquer, je vais donc vous donner un nouveau code qui est strictement identique au précédent au niveau du résultat, mais qui permettra de mieux comprendre ce que je dis :

#### Code : PHP

```
<?php  
  
$var = array(  
    0 => 'lili',  
    1 => 'lala'  
) ;  
echo $var[0];
```

### Qui es-tu vraiment ?

Après cette brève introduction, voici la définition exacte de ce qu'est un array : une association ordonnée. Une association est un type qui **fait correspondre des valeurs à des clés** (ou index).

Dans le code précédent, vous voyez quatre lignes :

- \$var = array(
- 0 => 'lili',
- 1 => 'lala'
- );

La première ligne indique qu'on va créer un array *via* la structure array() et affecter cet array à la variable \$var.  
La deuxième ligne crée une association dans l'array qui associe la chaîne de caractères « lili » à la clé 0.  
La troisième ligne crée une association dans l'array qui associe la chaîne de caractères « lala » à la clé 1.  
La dernière ligne marque la fin de la structure (la parenthèse fermante) et la fin de l'instruction (le point-virgule).

Je vous ai dit que ce code était équivalent au premier, mais pourquoi ?

Quand je tape \$var = array('lili', 'lala');, l'interpréteur PHP crée un array et l'assigne à la variable \$var. On donne les données (les chaînes « lili » et « lala »), mais pas les clés ; or, pour obtenir une association, on doit avoir la valeur ET la clé, l'un ne va pas sans l'autre. C'est pourquoi, lorsque vous tapez cela, l'interpréteur PHP va lui-même attribuer des clés numériques de 0 (les clés commencent à 0 et non à 1, très important !) à n-1 (avec n le nombre de valeurs que vous donnez à la structure array()).

Vous verrez sans doute souvent la distinction entre des arrays associatifs et des arrays à index numériques : cette distinction **n'a aucun sens**. En effet, que ce soit vous ou l'interpréteur PHP qui donnez les clés à utiliser dans l'array, un array est et sera toujours une liste d'associations.

Maintenant qu'on a créé un array, on est contents, mais si on l'utilisait ? Vous avez vu une autre instruction qui affiche une valeur de l'array : echo \$var[0];. Voici la syntaxe de base pour accéder à une valeur d'un array :

#### Code : PHP

```
<?php  
$var [Expression] ;
```

\$var est le nom de la variable à laquelle l'array a été affecté, et Expression est l'expression dont la valeur est la clé de l'association à laquelle vous voulez accéder. Rappelez-vous, on crée un array ainsi : \$var = array('lili', 'lala');. Pour accéder à la chaîne « lili », qui est associée à la clé 0, l'expression entre les crochets doit avoir l'entier 0 comme valeur : \$var[0];.

Comme on doit mettre une expression entre crochets, on peut utiliser beaucoup de choses comme clé. Exemples :

#### Code : PHP

```
<?php  
$var = array(  
    'string' => 'bonjour',  
    4 => 'int',  
    4.3 => 'float',  
    NULL => 'null',  
    '' => 'string vide'  
) ;  
  
echo $var['string'];  
echo $var[4];  
echo $var[4.3];  
echo $var[NULL];  
echo $var[''];  
  
?>
```

Si vous testez ce code, vous verrez des choses étranges.

Vous verrez qu'il affiche deux fois « float » et deux fois « string vide », mais que « int » n'est jamais affiché. Eh bien devinez quoi : c'est normal. Quand vous voulez créer une association avec une clé de type float (nombre à virgule), l'interpréteur PHP transtype ce nombre à virgule en entier (il prend la partie entière).

Ce comportement explique qu'on affiche deux fois « float » et pas « int ». Mais pourquoi ?

C'est simple : quand on veut créer l'association 4.3 => 'float', l'interpréteur PHP transtype 4.3 en 4, l'association qu'on veut créer est donc 4 => 'float'. Seulement, cette clé a déjà été utilisée ! Et de la même façon que pour une variable, l'interpréteur PHP va attribuer la nouvelle valeur à la clé 4.

C'est exactement la même chose pour le null, sauf qu'au lieu d'être transtypé en entier, il est transtypé en chaîne de caractères vide («»).

Un exemple de la puissance des arrays :

**Code : PHP**

```
<?php

$pair = array('pair', 'impair');
echo '9 est un nombre ' . $pair[9 % 2];
// même chose avec une fonction (inutile, mais juste pour
l'exemple)
function pair($nb)
{
    return $nb % 2;
}
echo '10 est un nombre ' . $pair[pair(10)];
```

## Tu commences à me plaire...

Maintenant qu'on a bien vu ce qu'était un array et comment en créer, que penseriez-vous si l'on pouvait ajouter des associations dans un array déjà existant et de modifier les associations qui existent déjà ?

Comme pour la création d'un array, il y a deux façons d'attribuer une clé à une association qu'on veut ajouter à un array qui existe déjà. On peut soit décider nous-mêmes de la clé, soit laisser l'interpréteur PHP faire son choix.

**Code : PHP**

```
<?php

$var = array(); // je crée un array vide
$var['lala'] = 'Bonjour'; // j'ajoute l'association 'lala' =>
'bonjour' à l'array affecté à $var
$var[] = 'Hello';
// j'ajoute une association, mais je laisse l'interpréteur PHP
décider de la clé qui sera utilisée pour cette association
echo $var['lala'] . $var[0];
```

Mais comment savoir quelle clé l'interpréteur PHP va attribuer à une association ?

C'est très simple. L'interpréteur PHP va rechercher la clé numérique la plus grande (s'il n'y a aucune clé numérique, l'interpréteur PHP utilise l'entier 0 comme clé) et y ajouter 1 pour obtenir la nouvelle clé.

Si vous essayez d'ajouter une association à un array qui n'existe pas, l'interpréteur PHP va créer l'array et ajouter l'association ; ainsi, ce code est tout à fait correct :

**Code : PHP**

```
<?php

$stab['truc'] = 'lala';
echo $stab['truc'];
```

La modification des associations est encore plus simple que cela, on a déjà goûté à ça en réalité. Pour modifier la valeur d'une association, il suffit d'ajouter une association qui utilise la même clé, exemple :

**Code : PHP**

```
<?php

$var = array(5 => 'bonjour'); // on crée un array
echo $var[5]; // on affiche la valeur de l'association qui a la clé
5
$var[5] = 'aurevoir';
// on ajoute une association à l'aide d'une clé qui existe déjà, on
modifie donc la valeur de cette association
echo $var[5]; // on affiche la nouvelle valeur de l'association qui
```

a la clé 5

Pour supprimer une association d'un array, il faut utiliser la fonction unset() :

#### Code : PHP

```
<?php  
  
$var = array('lili', 'lala');  
echo $var[0];  
unset($var[0]); // on supprime l'association qui a la clé 0  
echo $var[0]; // affiche une erreur, cette association n'existe  
plus
```



La fonction unset() ne sert pas qu'à supprimer une association d'un array ; en effet, on peut supprimer n'importe quelle variable avec cette fonction, par exemple : unset(\$var); supprimera la variable \$var.

Puisqu'on a eu droit à une belle erreur, en avant-première, une petite explication sur deux erreurs qui peuvent souvent survenir avec les arrays :

#### Code : Console

```
Notice: Undefined offset: 4 in C:\www\test.php on line 37
```

Cette erreur indique que vous essayez d'utiliser une clé qui ne correspond à aucune association. Le 4 correspond à la clé que j'ai voulu utiliser, C:\www\test.php est le chemin du fichier qui a généré l'erreur, et 37 le numéro de la ligne qui l'a engendrée. Quand vous avez une erreur, regardez-la toujours attentivement pour savoir à quelle ligne aller chercher la cause de l'erreur (elle peut être une ligne ou deux au-dessus ou en dessous de la ligne indiquée).

#### Code : Console

```
Warning: Illegal offset type in C:\www\test.php on line 38
```

Cette erreur indique que vous essayez de créer une association avec une clé invalide à la ligne 38 du fichier C:\www\test.php. Ce code-ci générera une erreur similaire :

#### Code : PHP

```
<?php  
  
$var = array(); // un array vide  
$var[array()] = 5; // erreur, on ne peut pas utiliser un array  
comme clé d'une association d'un array
```

Voilà deux erreurs bien courantes. Mais maintenant, on va s'attarder sur **une grave erreur** de bien des codeurs. Il est possible que vous rencontriez parfois ce genre d'instruction :

#### Code : PHP

```
<?php  
  
$tab[cle] = 'truc';
```

Ce bout de code semble anodin, et pourtant, une erreur potentiellement très embêtante s'y est glissée. En lisant ce code, vous avez dû comprendre qu'on veut créer un array \$tab et ajouter l'association cle => 'truc'. Seulement, on a

oublié les apostrophes (ou les guillemets) autour de cle, et ce n'est donc pas une chaîne de caractères. Si l'interpréteur PHP était plus strict, votre script planterait totalement en faisant ça, mais manque de chance, il est très laxiste ; en effet, il va convertir cle en une chaîne de caractères et là, plus de problème (mais vous aurez quand même droit à une erreur de la part de l'interpréteur PHP pour vous signaler que quelque chose n'allait pas).

Je ne peux pas vous expliquer en quoi c'est une erreur pour le moment, car on n'a pas encore vu ce qu'était une constante.

Retenez juste que quand vous voulez créer une association avec une chaîne de caractères comme clé, vous ne devez jamais oublier les apostrophes ou les guillemets.

## Des fonctions et structures bien pratiques

Comme je l'ai dit précédemment, les arrays sont très pratiques, mais c'est uniquement grâce à toutes les fonctions associées que vous utiliserez la pleine puissance de ce type de variables. Je ne vais pas toutes les présenter ici, mais vous pouvez trouver la liste dans [cette partie](#) de la documentation PHP.

Pour bien vous mettre en jambes, je vais vous montrer une nouvelle structure de contrôle (souvenez-vous de if, while et do while) : foreach. Cette structure permet de parcourir un array, et c'est plutôt pratique.

Imaginons que vous voulez faire une liste de prénoms et tout afficher d'un coup. Vous mettez les prénoms dans un array, un foreach et tout s'affiche ! Ajouter, modifier ou supprimer un prénom devient alors très simple et rapide, il suffit de modifier l'array. Exemple :

### Code : PHP

```
<?php  
  
$prenom = array('Georges', 'Julien', 'Pierre', 'Marie', 'Audrey',  
    'Eve');  
foreach($prenom as $line)  
{  
    echo $line . ' ';
```

Plutôt puissant, n'est-ce pas ?

Le fonctionnement de foreach est très simple à comprendre. La seule chose qui peut peut-être vous dérouter, c'est ce \$line. En fait, pour ne pas dépendre de la clé de l'association, l'interpréteur PHP associe, à chaque tour de boucle, la variable \$line à la valeur de l'association qu'il vient de lire (on lit le tableau dans l'ordre des clés, une clé à la fois).

Il se peut que vous ayez besoin de connaître la clé qui correspond aux différentes valeurs ; eh bien soyez rassurés, foreach le permet !

### Code : PHP

```
<?php  
  
$prenom = array('Georges', 'Julien', 'Pierre', 'Marie', 'Audrey',  
    'Eve');  
foreach($prenom as $key => $line)  
{  
    echo 'Clé : ' . $key . ' ; Valeur : ' . $line . '<br />';
```

Exactement comme dans le code précédent, \$line contiendra la valeur de l'association et \$key la clé. Vous pouvez donner n'importe quel nom à ces deux variables ; j'ai choisi key et line parce que les noms parlent d'eux-mêmes.

## Es-tu là, petite valeur ?

Parfois, vous aurez besoin de vérifier si une valeur est présente dans un array. Il existe une fonction pour cela :

**in\_array(\$value, \$array)** .

Cette fonction retourne true si \$value est la valeur d'une des associations de l'array, sinon false. Exemple :

### Code : PHP

```
<?php
```

```
$meuh = array('lili', 'lala');
var_dump(in_array('lili', $meuh));
var_dump(in_array('ohéh', $meuh));
```

## Es-tu là, petite clé ?

On vient de voir comment vérifier si une valeur était présente dans l'array, mais pourquoi ne pas aussi vérifier si une clé existe ? Il y a en réalité deux méthodes : une propre aux arrays et une qui teste l'existence d'une variable en général. La fonction propre aux arrays : **array\_key\_exists(\$key, \$array)** .

Cette fonction retourne true si la clé \$key existe dans l'array \$array, sinon false. Exemple :

Code : PHP

```
<?php

$1 = array(
    5 => 'hu',
    'beuh' => 'meuh',
    7 => 9
);

var_dump(array_key_exists(5, $1));
var_dump(array_key_exists('lala', $1));
var_dump(array_key_exists('beuh', $1));
```

La seconde méthode, la plus générale, est une autre fonction : **isset(\$var)** . Cette fonction retourne true si la variable passée en paramètre existe, false sinon. Exemple :

Code : PHP

```
<?php

var_dump(isset($hu)); // false
$tab = array('lili');
var_dump(isset($tab)); // true
var_dump(isset($tab[0])); // true, la clé existe
var_dump(isset($tab['euh...'])); // false, la clé n'existe pas
```



On peut tester l'existence de plusieurs variables en n'utilisant qu'une seule fois la fonction **isset()**, on les séparera alors par une virgule : **isset(\$a, \$b, \$c)**. La fonction retournera false si au moins une des variables n'existe pas (n'est pas définie).

## Quelle est ta clé ?

On peut très facilement obtenir la valeur associée à une clé, il nous suffit de taper **\$array[cle]**. Mais l'inverse, trouver une clé associée à une valeur, peut aussi être intéressant. La fonction **array\_search(\$value, \$array)** retourne la clé associée à la valeur \$value ou false si la valeur n'existe pas dans l'array.

Code : PHP

```
<?php

$tab = array('lili', 'lala');
var_dump(array_search('lili', $tab));
var_dump(array_search('yo', $tab));
```

## Dis-moi tout...

De temps à autre, pour déboguer un script par exemple, vous aurez besoin d'afficher d'un coup toutes les clés et toutes les valeurs associées d'un array. On pourrait le faire avec un foreach, mais pourquoi s'embêter quand une fonction existe ?

### Code : PHP

```
<?php

$yo = array(
    'meuh' => 'Oh yeah',
    true => true,
    6 => 5.67
);

echo '<pre>';
print_r($yo);
echo '</pre>';
```

Ça rend plutôt bien, n'est-ce pas ? N'oubliez pas de mettre une balise <pre> avant de faire le print\_r(), ça permettra d'avoir un joli texte bien organisé (essayez sans si vous doutez de ça 😊).

## Je veux des arrays !

Il peut arriver que vous ayez besoin de créer des arrays bien particuliers, par exemple un array qui contient les lettres de *a* à *e*, de *a* à *z*, des chiffres de 0 à 9, de 1 à 234, etc.

Au lieu de faire des boucles ou de taper des dizaines de lignes, on va utiliser la fonction **range(\$start, \$end)**. Exemple :

### Code : PHP

```
<?php

$a_e = range('a', 'e');
$a_z = range('a', 'z');
$maj_a_z = range('A', 'Z');
$one_ten = range(0, 9);
$euh = range(56, 564);

echo '<pre>';
print_r($a_e);
print_r($a_z);
print_r($maj_a_z);
print_r($one_ten);
print_r($euh);
echo '</pre>';
```

## Découpage

Imaginez que vous demandiez à l'utilisateur une date au format jj/mm/aaaa, comment feriez-vous pour mettre les jours, les mois et les années dans un array ? Simple avec **explode()**.

### Code : PHP

```
<?php
```

```
$str = '17/08/2007';
echo '<pre>';
print_r(explode('/', $str));
```

Vous l'aurez compris, on coupe la chaîne avec le séparateur (ici '/') et on colle le tout dans un array.

## Collage

Vous savez découper une chaîne pour en faire un array, mais ça peut être utile de faire l'inverse : créer une chaîne de caractères à partir d'un array. Pour cela, il y a **implode()**.

### Code : PHP

```
<?php

$array = array(17, 8, 2007);
echo implode('/', $array);
```

Encore une fois, on utilise un séparateur, mais cette fois-ci il sert de « colle ».

Bon, il existe beaucoup d'autres fonctions, je ne vais pas tout montrer ici. Il vous suffit d'aller voir sur le lien que j'ai donné un peu plus haut pour voir la liste des fonctions (descendez jusqu'à la table des matières pour avoir le nom et le rôle de toutes les fonctions).

## Derniers mots et structure de contrôle

Il existe une autre fonction très utile : **count()**. Exemple :

### Code : PHP

```
<?php

$a = array(0, 9, 8, 'r');
echo count($a);
```

Vous l'aurez compris, cette fonction compte le nombre d'associations qu'il y a dans l'array et vous renvoie ce nombre (la fonction **sizeof()** fait exactement la même chose, seul le nom change).

Ça pourra vous servir si vous voulez obtenir la dernière clé d'un array sans devoir le parcourir entièrement.

Mais pourquoi je dis ça ici ?... Pour introduire la dernière structure de boucle qu'offre le PHP, la quatrième, **for()**. Cette boucle est un peu plus complexe que les trois autres, mais ne craignez rien.

Voici cette structure réduite à sa plus simple expression :

### Code : PHP

```
<?php

for(Expression_une; Expression_deux; Expression_trois)
{
    // bloc d'instructions
}
```

Vous l'avez sans doute compris, la difficulté réside dans la ligne où se trouve le **for()**. Vous voyez qu'entre les parenthèses, il y a trois expressions différentes séparées par des points-virgules.

L'expression une est l'expression d'initialisation, elle n'est exécutée qu'une seule et unique fois, au tout début de la boucle. L'expression deux est l'expression de condition ; tant qu'elle vaudra true, la boucle continuera (comme pour while et do while). Cette expression est exécutée à chaque tour de boucle.

L'expression trois est l'expression itérative, elle est également exécutée à chaque tour de boucle et vous permet de faire certaines actions. Ceci est illustré à la figure 10.1.

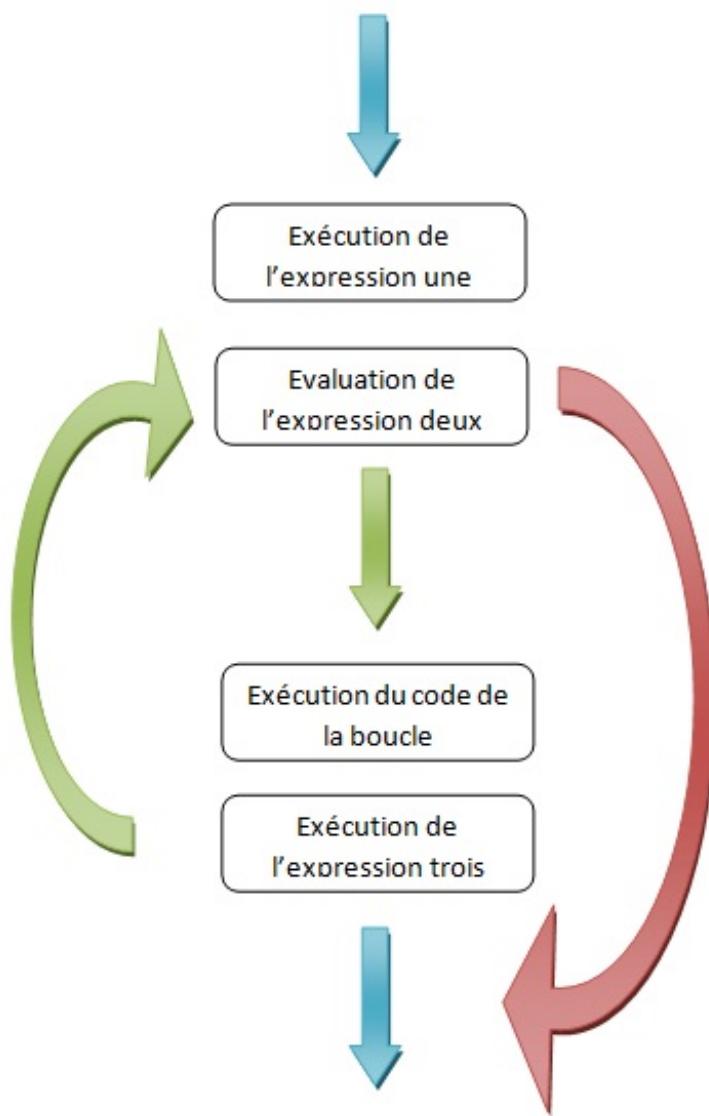


Figure 10.1 — Illustration du *for*

Vous pouvez ne rien mettre dans ces expressions ; ainsi, cette syntaxe est correcte :

**Code : PHP**

```
<?php  
for(; ;)  
{  
    echo 'bonjour';  
}
```

Les expressions qui ne sont pas renseignées sont évaluées à true et donc, vous l'aurez compris, je viens de vous donner le code pour faire une boucle infinie. Il peut arriver qu'on ait besoin de générer ce genre de boucle, mais c'est rare.

Généralement, cette structure de contrôle est utilisée pour faire un nombre donné d'itérations. Par exemple, si je veux afficher les dix premiers chiffres :

**Code : PHP**

```
<?php
```

```
for($i = 0; $i < 10; $i++)
{
    echo $i . ' ';
}
```

On veut afficher les dix premiers nombres pairs ?

#### Code : PHP

```
<?php

for($i = 0; $i <= 20; $i += 2)
{
    echo $i . ' ';
}
```

Il peut arriver qu'on ait besoin de plusieurs instructions dans l'une ou l'autre expression du for ; on peut le faire, mais on les séparera par une virgule et non par un point-virgule.

#### Code : PHP

```
<?php

for($i = 0, $j = 20; $i < $j; $i++)
{
    echo $i . ' ';
}
```



Mais quel rapport avec les arrays ?

J'y viens !

Souvent, on voit des codeurs parcourir des arrays avec for et count() (alors que foreach est fait pour ça !). Il y a encore pire : ils mettent le count() dans l'expression deux du for.



Mais où est le problème ?

Celui qui se pose cette question peut aller relire l'explication sur le for. En effet, les instructions deux et trois du for sont ré-exécutées à chaque tour de boucle. Vous allez donc compter le nombre d'éléments du tableau autant de fois qu'il y a d'éléments dans le tableau. Vous allez utiliser inutilement la fonction count() un grand nombre de fois et vous perdrez en performance.  
**Voici ce qu'il ne faut surtout pas faire :**

#### Code : PHP

```
<?php

$tab = range('a', 'z');
for($i = 0; $i < count($tab); $i++)
{
    echo $tab[$i];
}
```

La fonction count() sera appelée vingt-six fois au lieu d'une... Ça vous amuserait, vous, de faire vingt-cinq fois le même calcul ? Vous perdriez votre temps.

Par contre, ceci est beaucoup mieux :

**Code : PHP**

```
<?php  
  
$tab = range('a', 'z');  
for($i = 0, $count = count($tab); $i < $count; $i++)  
{  
    echo $tab[$i];  
}
```

Comme le `count()` est dans la première expression, il n'est exécuté qu'une et une seule fois.  
Mais le mieux est bien évidemment `foreach` qui est fait pour ça :

**Code : PHP**

```
<?php  
  
foreach(range('a', 'z') as $lettre)  
{  
    echo $lettre;  
}
```

## Les faux jumeaux

Vous croyiez que j'en avais fini avec vous ?  
Quelle désillusion, je dois encore vous parler de l'histoire des faux jumeaux.

Les faux jumeaux, ce sont les chaînes de caractères et les arrays. Mais pourquoi ?  
Parce qu'une chaîne de caractères, c'est un array de caractères ! Exemple :

**Code : PHP**

```
<?php  
  
$str = 'bonjour';  
for($i = 0, $lim = strlen($str); $i < $lim; $i++)  
{  
    echo $str[$i];  
}
```

Vous reconnaisserez les crochets, symptomatiques des arrays. Toutefois, si les chaînes de caractères ressemblent aux arrays, elles n'en sont pas.  
Vous vous souvenez de ce que j'ai dit ? Que pour parcourir un array, le *must* était d'utiliser un `foreach`. Pourquoi ne l'ai-je pas fait, à votre avis ?  
Tout simplement parce que si j'avais fait un `foreach` sur ma chaîne de caractères, j'aurais eu une erreur de ce style :

**Code : Console**

```
Warning: Invalid argument supplied for foreach() in C:\www\test.php on line 42
```

Cette erreur survient quand vous donnez un type qui n'est pas attendu à une fonction ou à une structure (par exemple si une fonction demande un entier et que je lui donne un array, vous aurez une erreur de ce style).

### Les chaînes de caractères ne sont pas des arrays.

La seule similitude entre ces deux types, c'est qu'on peut accéder à chacun des éléments en utilisant la syntaxe des crochets. Les clés dans une chaîne de caractères vont de 0 à n-1 avec n le nombre de caractères, tout comme les arrays.  
Si vous voulez obtenir un véritable array à partir d'une chaîne de caractères, vous pouvez utiliser la fonction `str_split()`.

Il est possible que vous voyiez cette syntaxe :

**Code : PHP**

```
<?php  
$str = 'bonjour';  
echo $str{0}; // affiche 'b'
```

Utiliser des accolades est strictement équivalent à utiliser des crochets. Seulement, la syntaxe avec les accolades sera tout simplement **supprimée** dans les prochaines versions de PHP. Donc si vous ne voulez pas avoir à modifier vos scripts plus tard, prenez la bonne habitude et utilisez les crochets ([]).

Courage courage, bientôt les bases seront finies, bientôt les visiteurs pourront interagir avec vos scripts. Mais avant tout ça, faisons une petite pause avec un chapitre plus relax après ce looong chapitre.

## Bons plans

Ce chapitre, un peu en décalage avec les autres, va vous présenter deux ou trois éléments qui flottent autour du PHP et qui sont très importants ! Ça ne vous aidera pas à devenir les dieux du PHP (sauf pour la partie sur le transtypage), mais ça vous aidera quand même.

### Présenter son code, tout un art

Pour l'instant, les codes que je vous ai montrés étaient tous simples et très courts. Mais quand on commencera à s'amuser un peu plus, les scripts deviendront très vite bien plus longs, bien plus complexes. Et si vous ne savez pas présenter votre code, vous aurez beaucoup de mal à vous relire par la suite.

Pourquoi est-ce important de pouvoir se relire facilement ? Réfléchissez et imaginez que vous fassiez un script génial mais que vous le codiez comme des gorets. Vous l'oubliez quelques mois, et puis vous devez le retravailler pour l'une ou l'autre raison. Si votre code est bien présenté, bien commenté, vous pourrez directement reprendre votre travail, alors que si vous avez joué aux codeurs-gorets, vous perdrez un temps fou à essayer de vous comprendre, et parfois sans succès.

La première chose à faire, c'est d'indenter son code. Mais qu'est-ce que l'indentation ? L'indentation est une pratique qui vise à rendre le code source plus clair en rajoutant des tabulations, ou des espaces, pour bien séparer les blocs d'instructions. Voici deux codes, strictement identiques ; l'un est indenté, l'autre ne l'est pas.

#### Code : PHP

```
<?php

function create_cache_config()
{
    $req = mysql::query('SELECT cfg_fieldname, cfg_fieldvalue FROM '
        . $GLOBALS['cfg']['sql_prefix'] . 'config', IF_ERROR_DIE);
    $tmp = '<?php' . EOL . '$cfg = array(' . EOL;
    while($res = mysql::fetch_assoc($req))
    {
        if(ctype_digit($res['cfg_fieldvalue'])))
            $tmp .= "" . $res['cfg_fieldname'] . "' => " .
                (int)$res['cfg_fieldvalue'] . ',' . EOL;
        else
            $tmp .= "" . $res['cfg_fieldname'] . "' => '" .
                str_replace("'", "\'", $res['cfg_fieldvalue']) . ',' . EOL;
    }
    $tmp[strlen($tmp) - 1] = ' ';
    $tmp .= ');' . EOL;
    if(!($file = fopen('./Cache/config.php', 'w')))
        trigger_error('create_cache_config::error ~ Unable to open file
./Cache/config.php', E_USER_ERROR);
    if(fwrite($file, $tmp) === false)
        trigger_error('create_cache_config::error ~ Unable to write into
file ./Cache/config.php', E_USER_ERROR);
    fclose($file);
}

?>
```

#### Code : PHP

```
<?php

function create_cache_config()
{
    $req = mysql::query('SELECT cfg_fieldname, cfg_fieldvalue FROM '
        . $GLOBALS['cfg']['sql_prefix'] . 'config', IF_ERROR_DIE);
    $tmp = '<?php' . EOL . '$cfg = array(' . EOL;
    while($res = mysql::fetch_assoc($req))
    {
        if(ctype_digit($res['cfg_fieldvalue'])))
            $tmp .= "" . $res['cfg_fieldname'] . "' => " .
                (int)$res['cfg_fieldvalue'] . ',' . EOL;
        else
    }
```

```
        $tmp .= "" . $res['cfg_fieldname'] . " => " .
str_replace("'", "\'", $res['cfg_fieldvalue']) . "','" . EOL;
    }
    $tmp[strlen($tmp, ',')] = ' ';
    $tmp .= ')' . EOL;
    if(!$file = fopen('./Cache/config.php', 'w'))
        trigger_error('create_cache_config::error ~ Unable to open
file ./Cache/config.php', E_USER_ERROR);
    if(fwrite($file, $tmp) === false)
        trigger_error('create_cache_config::error ~ Unable to write
into file ./Cache/config.php', E_USER_ERROR);
    fclose($file);
}

?>
```

Ne trouvez-vous pas le deuxième code beaucoup plus lisible ? En un seul coup d'œil, on sait où commence tel ou tel bloc d'instructions et où il finit. Et encore, j'ai choisi un exemple « simple » ! Quand vous imbriquerez des conditions, des boucles et autres joyeusetés les unes dans les autres, ça va vite devenir un joyeux bordel si vous n'indentez pas correctement.

Il existe plusieurs styles d'indentation, le K&R, le BSD/Allman, le GNU, etc. Aucun n'est réellement meilleur qu'un autre, ça dépend de vos goûts. Personnellement, j'ai opté pour le style BSD/Allman ; petit exemple :

#### Code : PHP

```
<?php

while ($lala)
{
    echo 'machin';
    // blabla
}

?>
```

Le style BSD/Allman fait un retour à la ligne après les parenthèses du while pour mettre l'accolade ouvrante à la ligne qui suit. Vous êtes libres de choisir le style qui vous plaît. L'important, c'est d'en choisir un et de s'y tenir. Si vous mélangez les styles, ça serait encore pire que sans indentation.

Le second point très important, c'est le nom de vos variables. En effet, le nom d'une variable est un élément capital : s'il est trop long, ça serait embêtant à taper (par exemple \$NombreDeMessagesParPage est un nom de variable certes très clair, mais d'une longueur affolante). Il faut donc vous limiter et ne pas exagérer.

Il ne faut cependant pas non plus tomber dans l'excès inverse. Si vous mettez des noms de variable trop courts, vous risquez de ne plus savoir ce qu'est telle ou telle variable (par exemple, \$inf est un nom très court, mais ça ne me dit rien du tout).

Il vous faut donc trouver un juste milieu entre clarté et longueur. L'idéal, c'est d'avoir un style de variables. Prenez les mots que vous utilisez le plus souvent dans vos noms de variables, et trouvez des diminutifs. Par exemple, au lieu de Nombre, vous pouvez mettre Nb, au lieu de Information, mettez Info. Vous pouvez également supprimer tous les mots blancs comme « de », « par », etc.

Si je reprends l'exemple du \$NombreDeMessagesParPage, ça donnerait \$nb\_mess\_page. C'est bien plus court, mais c'est tout aussi lisible pour peu qu'on conserve toujours la même convention de nommage.

Les variables, voilà une drôle d'affaire, d'ailleurs. Vous vous êtes déjà demandés quand il était utile de créer une variable ? Souvent, il arrive que l'on voie ce genre de script :

#### Code : PHP

```
<?php
function cube ($nb)
{
    $calcul = $nb*$nb*$nb;
    return $calcul;
}
```

```
$cote = 5;
$volume = cube($cote);

echo 'Le volume d\'un cube de côté 5cm est ' . $volume;
?>
```

Quelqu'un peut-il me dire en quoi les variables \$calcul, \$cote et \$volume sont utiles ?

Ce genre de variables, j'appelle ça des parasites. Toute variable prend une certaine place dans la mémoire du serveur ; plus il y a de variables, moins il y a de mémoire, plus le serveur est lent (bon, c'est un peu trop simpliste mais l'idée est là).

Quand vous savez que vous n'utiliserez pas une variable, ne la créez pas !

Souvent, les codeurs ont peur des arrays au début, alors on voit souvent des bouts de code comme ceux-ci :

#### Code : PHP

```
<?php
$max_size = $array['max_size'];
$min_size = $array['min_size'];
$max_width = $array['max_width'];
$min_width = $array['min_width'];
?>
```

Vous rigolez peut-être, mais beaucoup de codeurs font ça, et **ça ne sert à rien**. Vous créez des variables totalement inutiles et qui ralentissent le serveur. Dans la partie II de ce tutoriel, on verra bien plus d'exemples pratiques où l'on serait tenté de créer des variables inutiles ; mais pour le moment, retenez juste qu'il faut bien réfléchir avant de créer des variables. Voilà le script précédent sans variable inutile :

#### Code : PHP

```
<?php
function cube ($nb)
{
    return $nb*$nb*$nb;
}

echo 'Le volume d\'un cube de côté 5cm est ' . cube(5);
?>
```

Le rendu est le même, mais j'ai économisé trois variables inutiles !

Encore deux petits détails qui me font grimper au plafond quand je les rencontre sur des scripts : le premier concerne les chaînes de caractères délimitées par des guillemets. Vous savez que dans ces chaînes de caractères, les variables sont remplacées par leur valeur. Alors parfois, on a droit à ce genre de choses :

#### Code : PHP

```
<?php
$a = fonction_quelconque();
echo "$a";
?>
```

Qui dans la salle peut me donner une seule bonne raison de mettre des guillemets autour de \$a ?

Ça ne sert à rien, le rendu sera strictement identique si on les enlève, et en plus ça sera bien plus performant. En effet, analyser la chaîne de caractères et éventuellement remplacer une variable par sa valeur, ça prend un temps fou ! Alors qu'afficher simplement une variable, ça va à une vitesse supraluminique.

Second détail : parlons de concaténation. Qui peut me dire où se trouve la bêtise dans ce script (hormis les variables inutiles, c'est pour l'exemple) ?

#### Code : PHP

```
<?php  
$a = 5;  
$b = 6;  
$c = 11;  
  
echo '' . $a . '+' . $b . '=' . $c . '';  
?>
```

Pour ceux qui sont attentifs, ce sont ces deux chaînes vides ("") qui n'ont pas le moindre sens. À quoi ça sert de concaténer une chaîne vide ?

Prenons un exemple réel : imaginez que vous colliez des bâtonnets bouts à bouts ; à quoi cela servirait-il de coller un bâtonnet de taille nulle ? À part perdre du temps à le coller, ça ne sert à rien du tout.

Eh bien c'est pareil quand on concatène des chaînes vides : le PHP perd du temps à faire la concaténation alors que concaténer une chaîne vide ne modifie en rien l'autre chaîne !

Ça paraît stupide dit comme ça, mais beaucoup de gens font cette stupidité.

## La bible du codeur PHP

Dans ce tutoriel, je vous ai déjà donné un ou deux liens vous renvoyant vers l'une ou l'autre partie du site <http://fr.php.net>. Mais qu'est donc ce site ?

Ce site est la version française de <http://php.net>, le site officiel du PHP Group, c'est-à-dire le groupe de personnes qui a créé et fait évoluer le PHP.

Ce site est une mine d'or. En théorie, pour devenir un bon codeur PHP (ou du moins, pour connaître tous les aspects techniques du langage), il vous suffit de parcourir ce site. Seulement, de prime abord, il paraît austère et peu attrayant. Je vais vous expliquer un peu comment vous en sortir.

Tout d'abord, quand vous arrivez sur ce site, regardez dans la liste de liens en haut. Vous pouvez voir **Download**, **Documentation**, **Faq**, etc. Le lien qui nous intéresse est **Documentation**. Quand vous cliquez dessus, vous arrivez sur une page en anglais. Ne soyez pas déstabilisés, on va retrouver le français dans peu de temps. Vous pouvez voir un tableau sur cette page, il y a les lignes **Formats**, **View Online** et **Download**. Ce qui nous intéresse, c'est de voir la documentation en ligne (sur le site du PHP Group) : on va donc regarder dans la ligne **View Online** (« voir en ligne », en français).

Il y a une liste des langues dans lesquelles la documentation a été traduite. Comme on parle français, on va cliquer sur le lien **French** en gras. Félicitations, vous voilà sur la table des matières de la documentation PHP ! C'est à partir de là que vous pouvez atteindre tous les articles de la documentation.

Vous pouvez voir que tout est classé par thème ; regardez par exemple le point III, *Référence du langage*. Si vous regardez les noms des pages de ce thème, vous devez les reconnaître. Fonctions, variables, expressions, structures de contrôle, on a vu tout ça dans ce tutoriel. Il y en a d'autres que l'on n'a pas vus, comme les références, les classes et les objets, mais ce n'est pas très important pour le moment.

Ce que je veux que vous compreniez, c'est que tout est classé par thème. Et donc, si vous cherchez quelque chose dans la documentation, la première chose à faire est de se demander quel est le thème de ce que vous cherchez. Si je veux des informations sur le if(), je sais que c'est une structure de contrôle, j'irai donc voir dans le point III.16, *les structures de contrôle*.

Si vous cliquez sur ce lien, vous arriverez à une nouvelle table des matières : la table des matières des structures de contrôle. Eh oui, il y a beaucoup de structures, on les a donc toutes insérées dans des pages différentes pour ne pas tout mélanger.

Cliquez donc sur le lien du if et vous voilà face à la description de son rôle, de son fonctionnement, etc. Vous avez aussi des exemples de code : je vous invite à les tester ; bien souvent, ça vous aide à comprendre.

Si vous continuez à descendre dans la page, vous verrez parfois des notes (en rouge, en jaune) qui sont des ajouts, des précisions et que vous devez lire. Ça contient parfois des informations cruciales.

Encore un peu plus bas, vous avez des commentaires sur la page : ce sont des notes de lecteurs, tout comme vous, qui ont voulu apporter leur pierre à l'édifice. Les commentaires sont très souvent en anglais, donc si vous ne parlez pas anglais, ils ne vous aideront pas. Pour ceux qui parlent anglais, je vous invite à lire les notes des lecteurs. On peut y trouver des informations très pratiques.

Mais attention à ne pas accorder une confiance aveugle en ces notes, il peut y avoir de grosses erreurs (déjà vues plusieurs fois) ; vous devez donc toujours critiquer ce que vous lisez.

Maintenant qu'on a vu un peu ce qu'était la documentation (doc' pour les intimes), je vais vous montrer ce dont vous vous servirez le plus souvent : la liste des fonctions.

On a appris à faire des fonctions, mais il en existe déjà des tonnes, plus de mille !

Sachant cela, il est très important, avant de créer une fonction, de vérifier qu'elle n'existe pas déjà. Mais comment le savoir ?

C'est très simple, on va prendre un cas réel. On va imaginer que je veuille créer une fonction qui me calcule la racine carrée d'un nombre.

La première chose à faire, c'est de se demander quel est le thème de ma recherche. Je cherche une fonction, le thème est donc les fonctions. Je vais à la table des matières de la documentation, et je descends jusqu'au VI, *La référence des fonctions*.

Je sais que je cherche une fonction, mais qu'est-ce que je sais d'autre ? Je sais que les racines carrées, ça fait partie des mathématiques. Je vais donc chercher la table des matières des fonctions mathématiques.

Je regarde donc dans la liste et j'arrive sur le point LXXIX, *Mathématiques*. Cool, c'est ce que je veux. Je clique sur le lien et me voilà sur la table des matières des fonctions mathématiques.

Descendez un peu dans la page, et vous verrez la table des matières. Cette table liste l'ensemble des fonctions et donne une petite explication très succincte sur le rôle de chacune d'elle. Je vais donc chercher une section où l'on parle de racine carrée. Je cherche, je cherche encore et je tombe sur la fonction `sqrt()` dont la description est *Racine carrée*. Génial, il y a de grandes chances que ça soit ce que je cherche ! Je clique donc sur le lien et me voilà sur la page concernant la fonction `sqrt()`.

Sur cette nouvelle page, vous avez plusieurs informations. Tout d'abord, en gros et en bleu, vous avez le nom de la fonction (ici, `sqrt()`).

Juste en dessous, vous avez des parenthèses avec un texte comme ceci par exemple : PHP 4, PHP 5. Ça vous indique pour quelles versions du PHP cette fonction existe. Si vous utilisez une version du PHP qui n'est pas indiquée entre ces parenthèses, vous aurez une erreur fatale. Vous pouvez avoir quelque chose de plus complexe comme : PHP 4 >= 4.3.0, PHP 5. Ça veut dire que la fonction est définie pour des versions du PHP 4 à partir du PHP 4.3.0, et pour PHP 5.

Vous avez ensuite une nouvelle ligne contenant le nom de la fonction et une description succincte. Dans notre exemple, on voit `sqrt - Racine carrée`.

Vient ensuite un second grand titre en bleu, *Description*. Là, vous avez des informations plus complètes sur la fonction. Tout d'abord, vous avez le **prototype** de la fonction, dans notre exemple :

#### Code : Console

```
float sqrt ( float arg )
```

Mais comment lire ça ?

Tout d'abord, le premier mot, `float`. C'est le type de la valeur que la fonction vous renvoie, ici un `float`, un nombre à virgule donc. Souvenez-vous des types primitifs et des arrays. Il est possible que vous ayez le mot `void` ; si c'est le cas, ça veut dire que la fonction ne retourne aucune valeur.

Vient ensuite le nom de la fonction, ici `sqrt()`.

Les parenthèses suivent, elles vont vous donner le nombre de paramètres et le type demandé. Dans le cas de `sqrt()`, il n'y a qu'un paramètre, un nombre à virgule, le nombre dont on veut calculer la racine carrée. Pour chaque paramètre, vous aurez un nom et un type. Certains types vous sont encore inconnus, comme `mixed` ou `callback`. Ces deux derniers types sont des **pseudo-types**, il n'existent pas réellement, c'est juste pour simplifier la lecture.

Voici les trois pseudo-types :

- `mixed` : indique qu'on peut donner n'importe quel type pour l'argument ;
- `callback` : indique qu'on doit donner le nom d'une fonction de callback ;
- `nombre` : indique qu'on doit donner un nombre, entier ou non.

Pour les fonctions de callback, on verra ça plus tard, ne vous en faites pas.

Il est possible de voir certains arguments entre crochets (`[]`) ; ne soyez pas étonnés, ça veut simplement dire qu'ils sont facultatifs, vous n'êtes pas obligés de les renseigner (souvenez-vous, je vous ai montré comment créer des fonctions avec des arguments facultatifs).

Une fois le prototype lu et compris, vous tombez sur la description plus complète de la fonction. Veillez à bien la lire, c'est très important.

Dans le cas de la fonction `sqrt()`, il n'y a qu'un paramètre et on devine facilement à quoi il sert. Seulement, ce n'est pas le cas de toutes les fonctions. Prenons par exemple la fonction `mysql_real_escape_string()`.

Regardez après la description, vous avez un nouveau grand titre : *Liste des paramètres*. Cette section va vous dire à quoi sert tel ou tel paramètre. Il est très important de lire cette section, c'est elle qui vous permettra de savoir comment utiliser la fonction.

La section qui suit vous parle de la valeur de retour, elle est également **très importante**. Imaginez le cas suivant : j'affecte à

une variable le retour d'une fonction qui peut renvoyer soit un nombre, soit un booléen. Ensuite, j'utilise cette variable comme paramètre pour une fonction qui demande un nombre et rien d'autre. Que se passerait-il si, pour une quelconque raison, la première fonction renvoyait un booléen ? Vous auriez une erreur et la deuxième fonction planterait ! Les erreurs de ce genre sont très courantes (on verra un très bel exemple dans la partie II de ce tutoriel) ; soyez donc très prudents et vérifiez toujours le type de vos variables quand il n'est pas sûr (si une fonction peut retourner deux types et qu'il n'y en a qu'un qui vous convient, vous devez vérifier ce que renvoie la fonction avec un if, par exemple).

On continue de descendre et nouvelle section : les exemples. Comme je l'ai dit précédemment, ils sont très utiles et permettent de mieux appréhender la fonction ; n'hésitez donc pas à les essayer !

Le reste, c'est du connu, donc j'ai fini ! Vous savez maintenant comment trouver une fonction sans connaître son nom, alors si vous venez sur un forum pour demander si telle ou telle fonction existe, vous n'aurez plus d'excuse et vous serez flagellés publiquement. 😊

Bon, un autre cas peut se présenter : celui où vous connaissez le nom d'une fonction, mais vous ne savez pas ce qu'elle fait, ou comment elle le fait.

Souvent, la réponse à un problème tient en une fonction. La personne qui répond donne donc uniquement le nom de la fonction à la personne qui demande, ça lui suffira pour résoudre le problème. Malheureusement, cette personne n'a pas lu ce chapitre, et elle ne sait pas comment trouver le fonctionnement de la fonction : qu'est-ce qu'elle fait ? Elle poste un nouveau message pour demander comment on utilise la fonction. Plutôt triste, non ?

Dans cinq minutes, vous saurez comment ne plus avoir à poster ce genre de message inutile 😊.

Le site dont on parle depuis tantôt possède un outil très pratique : un moteur de recherche. Il vous suffit de taper le nom de la fonction dans le petit formulaire en haut à droite, et pan, vous tombez sur la description. Mais il y a mieux !

Vous pouvez accéder à la description de n'importe quelle fonction via le lien : <http://fr.php.net/%s>.

%s représente le nom de la fonction : ainsi, pour la fonction strpos() par exemple, on a : [http://fr.php.net\(strpos](http://fr.php.net(strpos).

Mais ça ne s'arrête pas là : tapez math au lieu du nom d'une fonction, boum, vous tombez sur l'index des pages associées aux fonctions mathématiques 😊.

Petit bonus pour ceux qui utilisent Firefox : un raccourci bien pratique. Commencez par aller dans le menu **Marque-pages**, puis dans l'option **Organiser les marque-pages**. Ajoutez ensuite un marque-page via le bouton prévu à cet effet, donnez-lui le nom qui vous plaît, mettez <http://fr.php.net/%s> comme adresse Web et pour le mot-clé, écrivez quelque chose d'équivoque et de pas trop long (comme **doc**, par exemple). Enregistrez le tout et tapez **doc strpos** dans la barre d'adresse de Firefox : miracle, vous tombez sur la page qui décrit la fonction strpos().

Comme vous l'avez sans doute deviné, il vous suffit de changer ce qui suit le mot-clé **doc** pour atterrir sur la page concernée.

Et voilà, vous êtes amis avec la documentation : si vous suivez ces démarches, vous pourrez toujours trouver ce que vous y cherchez !

## Le transtypage, c'est pratique

Depuis quelques chapitres déjà, je vous parle de quelque chose que le PHP fait très souvent : le transtypage. C'est une action qui change le type d'une expression en un autre, et c'est indispensable.

Souvenez-vous de mon exemple avec l'addition : qui peut me dire combien ça fait 'Pomme' + 'Poire' ? 😊

Eh bien avec le transtypage, je peux vous répondre, ça fait 0 !

On peut transtyper des variables de deux façons : soit avec des opérateurs, soit avec des fonctions.

Je vais commencer par parler un peu des fonctions, les opérateurs suivront et la table de conversion fermera la marche.

Il y a trois fonctions qui permettent de transtyper une expression en un type bien précis : strval(), floatval() et intval().

Rien qu'en voyant le nom des fonctions, vous devriez comprendre qui fait quoi.

strval() transtype une expression en une chaîne de caractères, exemple :

### Code : PHP

```
<?php  
  
$var = 5;  
var_dump($var);  
$var = strval($var);  
var_dump($var);  
  
?>
```

Les var\_dump() nous le confirment, on a bien transtypé notre variable !  
Les deux autres fonctions s'utilisent de la même façon :

**Code : PHP**

```
<?php  
  
$var = 'lala';  
var_dump($var);  
$var = intval($var);  
var_dump($var);  
$var = floatval($var);  
var_dump($var);  
  
?>
```

Trois fonctions, beaucoup de types, il y a comme un air de problème, non ?

En effet, les autres types n'ont pas de petite fonction de ce genre (on pourrait les créer, mais c'est inutile). Heureusement pour eux, une quatrième fonction vient sauver les meubles : settype().

Cette fonction prend deux paramètres : la variable qu'on veut transtyper (et non une expression, cette fois) et le type qu'on veut lui donner.

Voici un exemple de son utilisation, très simple :

**Code : PHP**

```
<?php  
$var = '';  
var_dump($var);  
settype($var, 'int');  
var_dump($var);  
settype($var, 'bool');  
var_dump($var);  
settype($var, 'float');  
var_dump($var);  
settype($var, 'array');  
var_dump($var);  
settype($var, 'null');  
var_dump($var);  
settype($var, 'string');  
var_dump($var);  
?>
```



Vous remarquez que je n'affecte pas la valeur de retour de la fonction à \$var : en effet, settype() travaille directement sur la variable et non sur une copie. Mais on verra ça plus tard dans le chapitre sur les références.

Sous vos yeux ébahis, vous voyez le type de \$var changer à tout bout de champ.

Malheureusement, settype(), j'aime pas. Pourquoi ? Parce qu'on modifie directement la variable.

Souvenez-vous de mon exemple de tantôt, avec une fonction qui peut retourner soit un nombre, soit un booléen, et une autre qui demande un nombre et rien d'autre.

Si je veux m'assurer d'avoir un nombre, je peux soit faire une condition, soit transtyper ma variable. Voici comment on ferait avec settype() :

**Code : PHP**

```
<?php  
  
$var = fonction_une();  
settype($var, 'int');  
echo fonction_deux($var);
```

?>

Trois instructions et une variable inutile, plutôt laid, non ? Alors qu'avec les trois fonctions de départ...

#### Code : PHP

```
<?php  
echo fonction_deux(intval(fonction_une()));  
?>
```

Plus court et plus sexy, que demander de mieux ? 😊

Ben, en fait, il y a mieux ! Je n'aime vraiment pas m'user les doigts sur mon clavier et ça m'ennuie de taper intval(\$var). C'est l'heure de faire connaissance avec les opérateurs !

Je vais reprendre un code plus ancien et obtenir le même résultat en économisant mes doigts :

#### Code : PHP

```
<?php  
$var = '';  
var_dump($var);  
var_dump((int)$var);  
var_dump((bool)$var);  
var_dump((float)$var);  
var_dump((array)$var);  
var_dump((string)$var);  
?>
```

J'ai un peu menti : il manque le type NULL. Mais comme on ne transtype jamais une expression en un NULL, ça n'a aucune importance. Voilà donc un code plus court, plus simple.

Reprendons notre code plus sexy et rendons-le encore plus attrant :

#### Code : PHP

```
<?php  
echo fonction_deux((int)fonction_une());  
?>
```

Et voilà, vous savez comment transtyper des expressions et des variables !



La priorité de ces opérateurs est la même, ils sont tous moins prioritaires que les opérateurs ++ et --, mais plus prioritaires que les opérateurs \*, / et %.

Ne nous arrêtons pas en si bon chemin : c'est bien de savoir transtyper, mais c'est mieux de savoir ce que donne le transtypage d'un type en un autre.

## Transtypage en booléen

Pour commencer en douceur, on va transtyper tous les types connus en booléens. Comme vous le savez, le booléen vaut soit false, soit true.

Étant donné qu'il y a un nombre quasiment infini de valeurs qui donnent true, voici ce qui donne false quand on transtype une expression en un booléen :

- le booléen false ;
- l'entier 0 (zéro) ;
- le nombre à virgule 0.0 ;
- la chaîne vide " et les chaînes '0' et "0" ;
- un array vide (sans aucune association) ;
- le type NULL.

Il vous suffit de tester pour vous en assurer. Toute autre valeur que celles-ci vaudra true après transtypage en booléen.

## Transtypage en entier

Si on veut convertir un booléen en entier, l'entier vaudra 0 si le booléen vaut false et 1 si le booléen vaut true.

Si on veut convertir un float en entier, l'entier vaudra l'arrondi inférieur si le float est positif, mais s'il est négatif, l'entier vaudra l'arrondi supérieur (on arrondit vers le nombre le plus proche de 0).

Code : PHP

```
<?php  
var_dump((int)4.65);  
var_dump((int)-4.65);  
?>
```

Si on veut convertir une chaîne de caractères en un entier, plusieurs cas sont possibles :

- si la chaîne commence par un nombre ('0hello' ou '3.9lala' par exemple), l'entier résultant sera égal à la partie entière de ce nombre ;
- si la chaîne ne commence pas par un nombre, l'entier résultant vaudra 0.

Pour les autres types (array, null, etc.), le transtypage en entier n'est pas défini, ça peut donner n'importe quoi. Toutefois, comme il est plutôt gentil, si vous voulez par exemple transtyper un array en entier, le PHP va d'abord transtyper l'array en booléen puis le booléen en entier.

## Transtypage en nombre à virgule

Si vous transtypez un booléen en nombre à virgule, deux cas sont possibles : si le booléen vaut true, le float vaudra 1, sinon 0.

Si vous transtypez un entier en nombre à virgule, le float vaudra l'entier (logique, 1.0 est équivalent à 1).

Si vous voulez transtyper une chaîne de caractères en float, c'est exactement comme pour les entiers, ça dépend par quoi commence la chaîne.

 Ce nombre est un float : -1.3e3. C'est en fait une autre écriture pour (-1.3)\*10^3.

Pour les autres types, c'est comme avec un entier.

## Transtypage en chaîne de caractères

Si vous convertissez le booléen true en chaîne de caractères, vous obtenez la chaîne '1' ; si le booléen vaut false, vous obtenez une chaîne vide.

Si vous convertissez un entier ou un nombre décimal, vous obtenez une chaîne représentant ce nombre (c'est-à-dire une chaîne qui, transtypée dans le type de départ, donne la valeur de départ).

Si vous tentez de convertir un array en chaîne de caractères, vous obtiendrez la chaîne 'Array' (faites donc un echo array(); et vous venez de prouver que le PHP convertit l'expression en chaîne de caractères avant de l'afficher).

NULL est également converti en une chaîne de caractères vide ("").

## Transtypepage en array

C'est le cas le plus simple de tous : si vous transtypez un type en array, vous obtiendrez un array avec cette valeur comme valeur de l'association portant la clé 0.

Code : PHP

```
<?php  
var_dump( (array) 1);  
var_dump( (array) 'lala');  
// etc.  
?>
```

Et voilà, on en a fini avec le transtypepage. 😊

Drôlement costaud, ce PHP, vous ne trouvez pas ? 😊

Ce chapitre était certes long, mais pas très complexe. N'hésitez pas à venir le relire, c'est un bon mémo. 😊

Et en guise de cadeau : pas de QCM !

## Transmettre des variables

Après dix chapitres intensifs sur le PHP, vous commencez tout doucement à avoir un peu d'expérience. Jusqu'à maintenant, vous créiez votre code, vous alliez sur la page et vous regardiez simplement le résultat sans demander plus. Et si on donnait un peu de vie à tout ça en donnant la possibilité au visiteur d'interagir avec votre script sans le modifier ?

Bienvenue dans le monde magique et maléfique (très important, ce mot 😊) des formulaires et de la transmission de variables.

### 35, rue machin

Il existe plusieurs méthodes pour transmettre des variables, mais on va commencer par la plus simple : les adresses (URL).

Pour l'instant, l'URL vous permet juste de dire à votre navigateur quel fichier appeler, mais les URL vous cachent bien des choses, car elles peuvent transmettre des variables !

Créez un fichier PHP, appelez-le **url.php** et tapez ce code :

#### Code : PHP

```
<?php  
  
echo '<pre>';  
print_r($_GET);  
echo '</pre>';  
  
?>
```

Si vous avez bien suivi le chapitre sur les arrays, vous devez savoir que `print_r()` est une fonction qui permet d'afficher joliment un array. Vous pouvez donc en conclure que `$_GET` est un array. Mais vous remarquez aussi que je n'ai pas défini `$_GET`. Vous pourriez donc penser que vous aurez une erreur en testant ce script : qui veut tenter le diable ?

Faites-le tous. 😊

Pour arriver sur cette page, vous taperez dans votre navigateur une URL du style <http://localhost/alias/url.php> ou encore <http://127.0.0.1/alias/url.php>. Quand vous arrivez sur cette page, quelle chance, pas d'erreur et on a même du texte ! On voit ceci :

#### Code : Autre

```
Array  
(  
)
```

Ça ne fera pas de vous les rois du monde, mais ça vous indique quelque chose : `$_GET` est un array, est défini, alors que je ne l'ai pas fait dans mon script. 😊

Ne faites pas ces grands yeux, c'est tout à fait normal. En effet, le PHP crée lui-même des variables, on les appelle des **superglobales**.

C'est l'occasion de faire un petit écart et d'introduire une nouvelle notion : la portée des variables. Vous vous souvenez de ce que j'ai dit à propos des variables et des fonctions ?

Je vous ai dit de ne jamais utiliser de variables déclarées hors d'une fonction dans une fonction ou l'inverse. Si j'ai dit ça, c'est à cause de la portée des variables.

Il existe trois « genres » de variables : les variables superglobales, les variables globales et les variables locales. On va prendre un code tout simple :

#### Code : PHP

```
<?php  
  
function machin()  
{
```

```
$a = $_GET;  
return $a%2;  
}  
  
$b = machin();  
if(true)  
{  
    var_dump($b);  
}  
  
?>
```

Les trois genres de variables sont présents dans ce simple code. Mais quelle variable représente quel genre ? \$\_GET est une variable superglobale, \$b est une variable globale et \$a est une variable locale.

La différence entre ces trois genres est simple. La variable locale n'existe que dans le bloc d'instructions (délimité par des accolades ({})) dans lequel elle a été définie. Si vous tentez d'y accéder en étant hors de ce bloc d'instructions, vous aurez une erreur. Les variables que vous définissez dans une fonction sont locales. Mais celles que vous définissez dans des structures de contrôle ne le sont pas, elles sont globales.

Une variable globale est accessible partout, sauf dans les blocs d'instructions qui créent des variables locales (pour le moment, seul le bloc d'instructions d'une fonction peut faire ça).

Ces deux principes expliquent pourquoi, si je déclare une variable dans une fonction, je ne peux y accéder de l'extérieur de ma fonction, et pourquoi, si je déclare une variable hors d'une fonction, je ne peux l'utiliser dans une fonction.

 Les paramètres d'une fonction sont aussi des variables locales.

Reste le troisième « genre », les superglobales. Comme vous l'avez peut-être deviné, ce sont des variables qui sont accessibles absolument partout, même dans les fonctions. C'est pourquoi mon code précédent fonctionne, d'ailleurs.

 Vous ne pouvez pas créer de variables superglobales, mais vous pouvez modifier celles qui existent déjà.

Voilà, ce court passage sur la portée des variables se termine, c'est assez important de saisir cette notion.

Mais revenons à nos moutons. Je vous ai dit que le PHP créait lui-même des variables, des superglobales ; dans la version actuelle du PHP, ces variables sont au nombre de neuf : \$\_SERVER, \$\_ENV, \$\_COOKIE, \$\_SESSION, \$\_GET, \$\_POST, \$\_FILES, \$\_REQUEST et enfin \$\_GLOBALS.

Vous voyez qu'elles sont très faciles à reconnaître : en effet, elles sont toutes en majuscules et leur nom commence (à une exception près) par un *underscore*.

Faites des print\_r() de ces variables pour voir toutes les informations qu'elles recèlent, une vraie mine d'or !

Pour l'instant, on ne s'intéresse qu'à une d'entre elles : \$\_GET. Reprenez votre script de tout à l'heure, et au lieu d'aller à l'adresse <http://localhost/alias/url.php> ou encore à <http://127.0.0.1/alias/url.php>, allez à l'adresse <http://localhost/alias/url.php?test=1&bla=2>. Et voilà, vous venez de transmettre des variables par une URL. Vous pouvez voir que le tableau \$\_GET contient deux associations.

Mais comment ça fonctionne ? C'est simple : pour transmettre des variables avec des URL, il vous suffit d'ajouter un point d'interrogation, puis de mettre l'association clé / valeur ainsi : cle=valeur. Si vous voulez mettre plusieurs associations, il faut les séparer par un & (dans le code source HTML, il faut mettre & et non & pour que le code HTML soit valide).

Testez un peu tout et n'importe quoi, ?true=4&true=bla&r=5.45, ?ihhan=meuh, etc. pour vous familiariser avec ça. Exemple (le fichier s'appelle url.php et se trouve au chemin http://localhost/url.php, modifiez chez vous au besoin) :

Code : PHP

```
<?php  
  
if(isset($_GET['nom']))  
{  
    echo $_GET['nom'];  
}
```

```
echo '<a href="http://localhost/url.php?nom=Eve">Je m\'appelle  
Eve</a>';  
echo '<a href="http://localhost/url.php?nom=Audrey">Je m\'appelle  
Audrey</a>';  
echo '<a href="http://localhost/url.php?nom=Leslie">Je m\'appelle  
Leslie</a>';  
echo '<a href="http://localhost/url.php?nom=Flore">Je m\'appelle  
Flore</a>';  
  
?>
```

Vous remarquerez que j'ai utilisé la fonction `isset()` qui, comme vous devez le savoir, vérifie si une variable existe. Si j'ai fait ça, c'est pour éviter une erreur. En effet, si je met directement `echo $_GET['nom']` mais qu'au lieu d'aller à l'adresse `http://localhost/url.php?nom=quelque_chose` je vais à l'adresse `http://localhost/url.php`, que se passe-t-il ? L'association qui porte la clé `nom` n'est pas créée étant donné qu'elle n'est pas dans mon URL sous la forme cle=valeur, je vais donc tenter d'utiliser une association qui n'existe pas : le PHP me retournera une erreur d'**`undefined index`**. 😊

**`$_GET` est un array.** Désolé d'écrire si gros, mais je suis obligé. 😊 Beaucoup de personnes viennent et demandent de l'aide sur les fonctions `$_GET`, `$_POST` ou autres. Mais `$_GET` et les autres superglobales ne sont pas des fonctions (il y aurait des parenthèses), ce sont des arrays et rien d'autre !

### Gardons la forme !

Pour cette partie, si vous n'avez pas de bases en HTML, vous ne comprendrez rien. Donc si vous ne savez pas ce qu'est un formulaire, je vous invite à aller voir un autre tutoriel, sans quoi vous n'irez nulle part.

Bon, si vous savez ce qu'est un formulaire (des zones de texte, des cases à cocher, etc.), reprenez votre page `url.php` et mettez ce code :

#### Code : PHP

```
<?php  
  
echo '<pre>';  
print_r($_POST);  
echo '</pre>';  
  
?>
```

Vous savez ce qu'est `$_POST`, **C'est un array**. Oups, mauvais bouton. 😊

Sans plus attendre, jouons avec les formulaires (*form* en anglais) :

#### Code : PHP

```
<?php  
  
echo '<pre>';  
print_r($_POST);  
echo '</pre>';  
  
?>  
<form action=".//url.php" method="post">  
    <p>  
        <input type="text" name="test" value="Oh Yeah" />  
        <br />  
        <textarea name="lala" rows="5" cols="5">Bonjour le  
        monde</textarea>  
        <br />  
        <input type="checkbox" name="ohoh" checked="checked" />  
        J'aime le rose  
        <br />  
        <select name="euh">  
            <option value="1">Meuh</option>  
            <option value="2">La</option>
```

```
        <option value="3" selected="selected">Vache</option>
    </select>
    <br />
    <input type="radio" name="prout" value="1" checked="checked"
/> Oui
    <input type="radio" name="prout" value="0" /> Non
    <br />
    <input type="hidden" name="ninja" value="ramen" />
    <br />
    <input type="submit" value="Valide-moi !" name="valid" />
</p>
</form>
```

Amusez-vous à changer les valeurs, cocher et décocher les cases, etc., et regardez ce que devient `$_POST`.

Vous pouvez constater qu'en fonction de ce que vous mettez dans votre formulaire, les associations de l'array `$_POST` changent. C'est tout à fait normal. En effet, `$_POST` est un array vide à la base. Mais quand vous soumettez un formulaire, votre ordinateur envoie une requête particulière au serveur (un *header*), et en fonction des éléments du formulaire et de leur valeur, le PHP remplit `$_POST` avec les associations qui conviennent.

On va donc expliciter le rôle de chacun d'eux.

## Les zones de texte

Cet élément du formulaire permet de créer une zone de texte d'une ligne.

### Code : HTML

```
<input type="text" name="machin" value="truc" />
```

Quand je mets ce code HTML, mon navigateur me crée une zone de texte et la préremplit avec la chaîne de caractères qui se trouve dans l'attribut **value**, *truc* dans l'exemple.

Quand vous soumettez le formulaire, cet élément va créer une association dans `$_POST`, la clé sera la valeur de l'attribut **name** (*machin* dans l'exemple) et la valeur sera le texte qui est dans la zone de texte. Rien de bien compliqué. Si on fait un echo `$_POST['machin']`; ça nous affichera ce qu'il y a dans la zone de texte par exemple.

## Les zones de texte multi-lignes

Cet élément du formulaire permet de créer une zone de texte de plusieurs lignes.

### Code : HTML

```
<textarea name="lala">Bonjour, je suis une valeur par
défaut</textarea>
```

Le texte qui se trouve entre les deux balises sera mis dans la zone de texte automatiquement (et s'il n'y a rien, il n'y aura pas de texte par défaut). L'attribut **name** servira de clé dans `$_POST` alors que la valeur sera ce qui est contenu dans la zone de texte.



Quand vous faites un retour à la ligne, votre navigateur insère un `\n` (nouvelle ligne, rappelez-vous) dans la chaîne de caractères.

Un echo `$_POST['lala']`; affichera le texte qui se trouve dans la zone de texte.

## Les cases à cocher

Cet élément du formulaire permet de créer une case à cocher.

**Code : HTML**

```
<input type="checkbox" name="oh" checked="checked" />
```

Là, deux cas sont possibles : si la case est cochée, le PHP créera une association dans `$_POST` dont la clé sera la valeur de l'attribut `name` et la valeur sera la chaîne de caractères 'on' ; mais si la case n'est pas cochée, le PHP ne créera pas d'association. Pour savoir si une case est cochée ou non, il faut donc utiliser la fonction `isset()` ou `array_key_exists()`.



Si vous mettez `checked="checked"`, la case sera cochée automatiquement ; enlevez-le pour que la case ne soit pas cochée par défaut.

## Les listes

Cet élément du formulaire permet de créer une liste dans laquelle on choisira une seule et unique valeur.

**Code : HTML**

```
<select name="euh">
  <option value="1">Oh</option>
  <option value="2" selected="selected">Yeah</option>
  <option value="3">!</option>
</select>
```

Quand vous validez un formulaire contenant cette liste, le PHP va créer une association dans `$_POST` dont la clé sera la valeur de l'attribut `name` (pour ne pas changer...) et dont la valeur sera la valeur de l'attribut `value` de l'option que vous aurez choisie. Si je choisis l'option 'Oh', la valeur de l'association sera '1', si je choisis 'Yeah', ça sera '2', et si je choisis '!' ça sera 3.



Rajouter l'attribut `selected="selected"` sélectionnera l'option par défaut.

## Les boutons

Cet élément du formulaire vous permet de créer des boutons à cocher (ça ressemble un peu aux listes).

**Code : HTML**

```
<input type="radio" name="prout" value="1" checked="checked" />
<input type="radio" name="prout" value="2" />
```

Ces éléments de formulaire doivent toujours aller ensemble ; s'il n'y en a qu'un, ça revient à faire une case à cocher. Lorsque qu'on valide ce formulaire, le PHP va créer une association dans `$_POST` qui aura pour clé l'attribut `name` (qui est commun à plusieurs input) et comme valeur la valeur de l'attribut `value` du bouton qu'on aura coché.



À nouveau, rajouter `checked="checked"` permet de cocher un bouton par défaut.

Si vous ne cochez aucun des boutons, aucune association ne sera créée.

## Le champ caché

Cet élément permet de créer un champ caché aux yeux du visiteur.

**Code : HTML**

```
<input type="hidden" name="truc" value="Oui" />
```

Quand vous validerez le formulaire, le PHP créera une association dans `$_POST` qui aura comme clé la valeur de l'attribut `name` et comme valeur la valeur de l'attribut `value`.

## Le bouton de validation

Cet élément vous permet de valider le formulaire en cliquant dessus.

**Code : HTML**

```
<input type="submit" name="valid" value="Valide-moi" />
```

En validant cela, le PHP créera une association qui aura pour clé la valeur de l'attribut `name` et comme valeur la valeur de l'attribut `value`.

En général, on ne met qu'un input de ce genre, mais il peut arriver qu'on ait besoin d'en mettre plusieurs pour laisser au visiteur le choix de faire telle ou telle action.

## Un bouton plus coloré !

Il existe un autre bouton qui a le même rôle que le submit, à la différence près que c'est en cliquant sur une image et non sur un bouton qu'on valide le formulaire.

**Code : HTML**

```
<input type="image" src="lien_image.png" name="valid_image" />
```

Vous remarquez que je n'ai pas mis d'attribut `value`. En effet, quand on utilise cet input et qu'on valide le formulaire, le PHP ne créera pas l'association qui aura pour clé la valeur de l'attribut `name`. À la place, le PHP va créer deux associations : une qui aura comme clé le caractère 'x' et une autre qui aura comme clé le caractère 'y'.

Ces deux associations sont des coordonnées, ça indique à quel endroit l'utilisateur a cliqué sur l'image.

## Aspiration !

Le tout dernier élément que je vais vous montrer nous servira bientôt : il permet d'*uploader* des fichiers (en clair, les envoyer de votre ordinateur vers le serveur). Son utilisation implique une légère modification de la balise `<form></form>`, mais on verra ça en temps voulu quand on fera un script pour permettre aux visiteurs d'*uploader* des images sur votre serveur !

**Code : HTML**

```
<input type="file" name="olala" />
```

N'essayez pas de chercher une association dans `$_POST` en utilisant cet input, vous ne trouverez rien. Cet élément est un peu différent des autres et l'association sera créée dans la superglobale `$_FILES`.

Et voilà : c'était vraiment lourd, cette partie (car très répétitive, il faut l'avouer), mais maintenant vous savez vous servir d'un formulaire en PHP !

## Never Trust User Input

Maintenant que je viens de vous exposer comment laisser vos visiteurs interagir avec vos scripts, je dois vous avouer quelque chose : vous allez vous faire *hacker* votre site...

Désolé, je n'ai pas pu résister, mais soyez tranquilles, c'est faux, votre site ne va pas se faire *hacker*. Toutefois, ce que je vous ai montré est la porte ouverte à toutes sortes de risques de failles dans votre code. La plupart des failles de sécurité liées au PHP tirent leur origine de \$\_GET et \$\_POST (ainsi que de \$\_COOKIE, mais c'est pour plus tard).

Je ne peux pas encore vous expliquer en détail pourquoi ces formulaires sont dangereux, puisque c'est surtout dans la partie II de ce tutoriel qu'il y a vraiment des risques et qu'on peut se faire avoir.

Mais dans tous les cas, soyez rassurés. La sécurité en PHP tient en quelques mots : **Never Trust User Input**.

Littéralement et en français, ça veut dire « Ne jamais croire (faire confiance) aux entrées de l'utilisateur ». Quand vous mettez un formulaire à disposition sur votre site, vous le faites généralement pour faire un sondage, ou pour laisser la possibilité aux gens de s'exprimer. Mais beaucoup de dérives sont possibles, à plusieurs niveaux. Ce petit texte n'a pas pour vocation de vous enseigner toutes les ficelles de la sécurité, car c'est avant tout une histoire de comportement. La seule faille que je puisse vous montrer actuellement, c'est la faille XSS.

Reprenez votre page [url.php](#) et mettez-y ce code, puis validez le formulaire.

#### Code : PHP

```
<?php

if(isset($_POST['texte']))
{
    echo $_POST['texte'];
}

?>
<form action=".//url.php" method="post">
    <p>
        <textarea name="texte" rows="10" cols="50">
            <script type="text/javascript">
                alert('Ohoh, vous avez été piraté par `Haku !!!');
            </script>
        </textarea>
        <input type="submit" name="valid" value="go" />
    </p>
</form>
```

Testez ce code et... Oh mon Dieu ! Je vous ai piraté. Non, rassurez-vous, je n'ai rien fait de méchant, votre PC ne va pas exploser, votre disque dur ne va pas danser la samba, les *aliens* ne vont pas débarquer ; j'ai juste mis un petit code JavaScript qui permet d'afficher une boîte de dialogue. Ce n'est pas plus méchant qu'une plume.

Vous voyez donc que si on laisse l'utilisateur mettre ce qu'il veut, il peut faire des choses qui ne sont pas prévues à la base et qui peuvent être ennuyeuses, voire même dangereuses.

Par exemple, faire une boucle infinie qui affichera tout le temps des boîtes de dialogue comme celle que j'ai affichée, ça serait ennuyeux. Vous devriez tuer le processus de votre navigateur internet pour sortir de cette situation.

Mais en quoi est-ce dangereux ? C'est simple. En PHP, il existe une superglobale, \$\_COOKIE, qui permet de sauvegarder des informations sur l'ordinateur d'un visiteur. Ne criez pas au scandale, les cookies sont de simples fichiers texte, ils ne peuvent rien faire sur votre ordinateur, ce n'est pas *Big Brother*. Ce sont juste des fichiers texte qui permettent au script de sauver des informations propres au visiteur (par exemple, si vous allez sur un site qui propose plusieurs designs, il stockera une information dans un cookie qui permettra de ne pas avoir à vous redemander quel design vous voulez à chaque visite).

Mais on peut mettre des informations beaucoup plus sensibles dans les cookies, comme par exemple des mots de passe pour vous connecter à des espaces membres (les mots de passe ne sont pas là tels quels, ils sont « cachés » grâce à une fonction de hashage).

Et c'est là que la faille XSS arrive. Vous avez vu que je peux injecter du code JavaScript et le faire s'exécuter. Le problème, c'est que le JavaScript **peut accéder aux cookies**. Et donc, si je remplaçais le code qui affiche une boîte de dialogue par un code plus vicieux, qui vous redirigerait vers une page piégée, je pourrais obtenir votre mot de passe et me faire passer pour vous sur tous les sites où vous employez ce mot de passe (c'est d'ailleurs pourquoi il est conseillé de ne pas utiliser tout le temps le même mot de passe, il faut en avoir plusieurs, idéalement un par site ou application).

Maintenant, comment s'en protéger ?

La méthode est simple et a fait ses preuves : utilisez la fonction PHP htmlspecialchars().

Cette fonction transforme cinq caractères en leur équivalent HTML. Bon, et alors ? Eh bien c'est simple.

Si vous tapez ceci dans une page HTML :

#### Code : HTML

```
<script type="text/javascript">alert ('hoho');</script>
```

... vous allez revoir une boîte de dialogue.

Maintenant, on va remplacer les chevrons (< et >) par leur équivalent HTML respectif : &lt; et &gt;. Remettez ce code dans une page HTML et regardez ce qu'il se passe :

#### Code : HTML

```
&lt;script type="text/javascript"&gt;alert ('hoho');&lt;/script&gt;
```

Magie, plus de boîte de dialogue. La raison est simple : votre navigateur sait que s'il rencontre les balises <script></script>, il doit interpréter ce qu'il y a entre les balises comme du JavaScript. Mais comme on a remplacé les chevrons par leur équivalent HTML (c'est-à-dire une suite de caractères qui affiche la même chose mais qui n'a pas la même valeur), votre navigateur n'interprète pas ce qu'il y a entre les balises comme du JavaScript.

Et voilà comment se protéger de cette faille.

Reprendons le code du début en y adjoignant la fonction htmlspecialchars(), et admirons le résultat :

#### Code : PHP

```
<?php  
  
if(isset($_POST['texte']))  
{  
    echo htmlspecialchars($_POST['texte']);  
}  
  
?  
<form action=".url.php" method="post">  
    <p>  
        <textarea name="texte" rows="10" cols="50">  
            <script type="text/javascript">  
                alert('Ohoh, vous avez été piraté par `Haku !!!');  
            </script>  
        </textarea>  
        <input type="submit" name="valid" value="go" />  
    </p>  
</form>
```

Voici la liste des caractères remplacés par htmlspecialchars() :

- & => &amp;
- < => &lt;
- > => &gt;
- " => &quot;
- ' => &#039;



Certains caractères ne sont remplacés que si vous donnez un second argument à la fonction ; pour savoir comment ça fonctionne, allez voir dans la documentation, on a vu comment s'en servir !

## L'enfer des guillemets magiques

Après un petit blabla fort intéressant sur la faille XSS, on va prendre un peu de temps pour discuter d'un autre problème plutôt ennuyeux (je voulais dire emmerdant, mais ce n'est pas très pédagogique !) : les guillemets magiques.

Ça me pose un problème d'en parler ici, puisque la cause de ces guillemets magiques (les magic\_quotes) et ce qui a justifié leur

création vous sont encore inconnus. Je vais donc en parler, mais très brièvement.

Testez ce code :

**Code : PHP**

```
<?php  
  
$text = 'lala des " et encore des "' . " et des ' sans oublier les  
"';  
echo $text;  
echo addslashes($text);  
  
?>
```

Vous voyez que les deux echo (echo qui est une structure, pas une fonction, je le rappelle) n'affichent pas la même chose. On aurait pu s'en douter, vu que l'un affiche directement \$text, alors que l'autre affiche la valeur de retour d'une fonction : addslashes().

Le but de cette fonction est très simple : elle rajoute un *backslash* () devant ces quatre chaînes de caractères : ", ', \ et NULL.

Bon, vous allez me demander quel est l'intérêt de cette fonction ; je pourrais vous répondre, mais ça ne servirait à rien parce qu'on n'a pas encore abordé le sujet. Tout ce que je peux dire, c'est que pendant longtemps, cette fonction a été la défense de bien des codeurs face à une possible faille.

Mais quel rapport avec les formulaires ? Eh bien c'est simple. Dans le fichier de configuration du PHP (un truc obscur que l'on n'a jamais vu), il y a des directives, c'est-à-dire des ordres que le PHP doit suivre.

L'une de ces directives s'appelle **magic\_quotes\_gpc**. Si cette directive est activée, le PHP va appliquer la fonction addslashes() à toutes les valeurs présentes dans les arrays GPC (\$\_GET, \$\_POST et \$\_COOKIE). Et donc, on risque de se retrouver avec une panoplie de backslashes en trop !

Idéalement, il faudrait que cette directive disparaisse : elle n'a été implémentée que pour pallier le manque de mesures de sécurité des codeurs PHP. Seulement, comme cette directive est désactivable, la configuration peut changer. Si vous faites votre script sur un serveur qui a désactivé la directive et qu'ensuite, vous décidez de changer de serveur et d'aller sur un autre qui a activé cette directive, vous aurez des backslashes dans tous les coins et vous devrez modifier votre script pour réparer les dégâts. Heureusement, la version 6 du PHP fera disparaître cette directive. Malheureusement, il y a des tas de serveurs qui resteront avec le PHP version 5 pendant un bon moment, et même avec le PHP version 4.

Je vais vous donner un code qui va vous permettre de ne plus avoir à vous soucier des guillemets magiques et vous l'expliquer.

**Code : PHP**

```
<?php  
  
if(get_magic_quotes_gpc())  
{  
    $_POST = array_map('stripslashes', $_POST);  
    $_GET = array_map('stripslashes', $_GET);  
    $_COOKIE = array_map('stripslashes', $_COOKIE);  
}  
  
?>
```

Le fonctionnement est très simple : d'abord, on vérifie si les magic\_quotes sont activés avec la fonction get\_magic\_quotes\_gpc(). Si c'est le cas, on va utiliser la fonction inverse d'addslashes() : stripslashes().

Pour appliquer une fonction à l'ensemble des valeurs d'un array, on utilise la fonction array\_map(). Le premier argument est le nom de la fonction qu'on veut appliquer (stripslashes() dans notre cas) ; le second argument est l'array sur lequel on va travailler.

Cette fonction est rudimentaire et peut être améliorée, mais pour le moment, ça suffira amplement.

Mais comme je l'ai dit, l'idéal est de les désactiver. Donc si vous avez accès au fichier de configuration du PHP (**php.ini** pour les intimes), suivez cette procédure pour désactiver les magic\_quotes :

- ouvrez le **php.ini** (chez moi, il est dans C:\Wamp\php\php.ini) ;
- recherchez la chaîne 'magic\_quotes\_gpc' ;
- mettez Off comme valeur pour les trois directives des magic\_quotes pour obtenir ceci :

**Code : Autre**

```
; Magic quotes  
;  
; Magic quotes for incoming GET/POST/Cookie data.  
magic_quotes_gpc = Off  
  
; Magic quotes for runtime-generated data, e.g. data from SQL, from exec(), etc.  
magic_quotes_runtime = Off  
  
; Use Sybase-style magic quotes (escape ' with '' instead of \').  
magic_quotes_sybase = Off
```

Reste à redémarrer votre serveur pour que la modification soit effective.

Boum, plus de `magic_quotes` !

On reviendra là-dessus dans la partie II de ce tutoriel de toute façon, pour découvrir pourquoi cette directive existe 😊.

**Je suis perdu !**

Je vais maintenant vous parler très brièvement de quelque chose qui n'a rien à voir avec les formulaires ou la transmission de variables : les constantes.

Si j'en parle ici, c'est que je n'avais pas de place ailleurs. 🍪

On a déjà entendu parler des constantes, mais jusqu'à maintenant, on ne parlait que de valeur constante (comme par exemple la valeur constante 5, ou la valeur constante 'lala'). Mais cette fois, je vais vous parler des vraies constantes, c'est-à-dire des sortes de variables, dont on définit le nom et la valeur une fois, et dont la valeur sera impossible à modifier tout au long du script.

Voici comment on déclare une constante :

**Code : PHP**

```
<?php  
  
define('NOM_CONSTANTE', 'VALEUR_CONSTANTE');  
echo NOM_CONSTANTE;  
  
?>
```

Pour déclarer une constante, on utilise la fonction `define()`. Le premier argument est le nom qu'aura la constante ; le second est la valeur qu'elle prendra (une expression quelconque).

Vous voyez que j'ai fait une instruction bizarre, nouvelle : `echo NOM_CONSTANTE;`

À première vue, vous avez dû vous dire que j'ai voulu écrire une chaîne de caractères mais que j'ai oublié de la délimiter avec des apostrophes ou des guillemets. Il n'en est rien. En tapant cette instruction, je voulais afficher la valeur de la constante.

Pour utiliser la valeur d'une constante, il suffit de taper son nom sans l'encadrer de guillemets ou d'apostrophes. Une constante est une expression comme une autre, au même titre qu'une variable, qu'une fonction qui retourne une valeur, etc. Vous pouvez donc l'utiliser pour beaucoup de choses (concaténation, assignation, calcul...).

Quand vous avez une variable, vous pouvez redéfinir sa valeur, exemple :

**Code : PHP**

```
<?php  
  
$a = 5;  
echo $a;  
$a = 7;  
echo $a;  
  
?>
```

Ce code fonctionne à merveille, mais si on tente de faire ça avec des constantes...

#### Code : PHP

```
<?php  
define('A', 5);  
echo A;  
define('A', 7);  
echo A;  
?>
```

Pan ! Une belle erreur pointe le bout de son nez. Elle est là pour nous dire qu'on tente de redéfinir la valeur d'une constante (enfin, elle dit plutôt que la constante existe déjà).

Mais quel intérêt ?

Je dirais qu'il y a deux cas majeurs où les constantes sont intéressantes.

Le premier, c'est si vous voulez vous assurer qu'une valeur ne change jamais. Une valeur qui resterait constante du début à la fin du script. Par exemple, pour configurer un script (nombre de news affichées par page, si on veut afficher les signatures sur un forum ou non, etc.), c'est très utile, parce qu'on ne risque pas de changer cette valeur.

Le second cas, c'est ce que j'appelle les arguments muets. Quand vous créez une fonction, vous lui donnez souvent des arguments, mais il peut arriver que vous ayez besoin de donner des arguments que je qualifie de muets.

Voici la liste des arguments muets :

- un entier ;
- un nombre à virgule ;
- un booléen.

Pourquoi sont-ils muets ?

Tout simplement parce qu'on ne sait pas quel est leur rôle, à quoi ils servent. Un exemple vaut mieux qu'un discours, et c'est cadeau :

#### Code : PHP

```
<?php  
echo fonction_affiche_news(NB_NEWS_PAGE);  
?>
```

#### Code : PHP

```
<?php  
echo fonction_affiche_news(20);  
?>
```



La constante NB\_NEWS\_PAGE est définie comme telle : define('NB\_NEWS\_PAGE', 20);.

Imaginez que ce n'est pas votre script : vous tombez sur l'appel de cette fonction. Qui peut me dire à quoi sert le nombre 20 ? À afficher une catégorie particulière de news ? À limiter le nombre de news par page ?

On ne peut pas le deviner sans aller voir la déclaration de la fonction, et on perd du temps.

Tandis qu'avec une constante, on a un mot, un nom qui nous permet d'identifier l'utilité de ce paramètre. Quand les scripts sont petits et qu'on les a développés seuls, l'intérêt est limité, je l'avoue. Mais quand on a de gros scripts qu'on crée à plusieurs, ça devient vite le bordel et les constantes permettent d'améliorer un peu la lisibilité du code !

Que de choses à retenir... Si vous ne deviez en retenir qu'une et une seule, c'est le **Never Trust User Input**. Tant que l'utilisateur ne peut pas affecter votre script, il n'y a quasiment aucune chance que votre script pose un quelconque problème de sécurité ; mais dès que l'utilisateur peut interagir avec votre script, si vous ne prenez pas garde, c'est la porte ouverte à toutes sortes de failles de sécurité.

Retenez bien qu'il ne faut jamais, jamais, jamais faire confiance à l'utilisateur. Même si vous faites un script de membres (on en fera un) et qu'il y a des administrateurs (des gens de confiance, en théorie), il ne faut pas leur faire confiance. Quand vous faites un script, toute autre personne que vous est un danger potentiel. J'insiste beaucoup là-dessus. La sécurité, ce n'est pas une histoire de fonction, mais une histoire de comportement. Vous devez prévoir toutes les dérives possibles de votre script et les empêcher.

Mais ne paniquez pas, hein (👻) : même si j'en ai envie, je ne vais pas vous laisser patauger dans cette affaire tout seuls, on verra très bientôt les grandes lignes des principales failles de sécurité et on essayera de s'en protéger !

## Premier TP : une calculatrice

Alors que le douzième chapitre pointe le bout de son nez, il est temps de faire le point.

On a déjà vu ensemble une grande partie des bases du PHP : avec tout ce qu'on a vu, vous pouvez faire de belles choses. Dans la deuxième partie de ce tutoriel, on va tout mettre en pratique. Mais avant, on va faire un TP pour que vous puissiez juger vous-mêmes de l'état de vos connaissances.

Ce TP est également le dernier chapitre de la partie I, alors soyez attentifs. 😊

### Objectifs

Le but de ce TP est que vous réalisiez une calculatrice. Mais ne vous en faites pas, vous allez en faire une très simple.

Premièrement, parce que vous n'avez pas tous des notions en mathématiques suffisantes pour faire quelque chose de plus compliqué.

Deuxièmement et principalement parce que vous allez déjà en baver pour faire quelque chose de simple. 🍪

Je suis sûr que très peu d'entre vous réussiront à faire quelque chose qui fonctionne parfaitement.

Mais ne vous inquiétez pas, ce TP est difficile, c'est volontaire. Je veux que vous compreniez par vous-mêmes que vous n'êtes pas loin. Vous avez vu les bases, mais vous n'avez pas encore beaucoup pratiqué. Vous manquez donc cruellement d'expérience, et comme le dit le dicton : « C'est en forgeant qu'on devient forgeron. »

Ne soyez donc pas étonnés si vous pataugez dans la semoule tout au long de ce TP. 😊

Cette calculatrice sera très simple, elle sera constituée d'un formulaire présentant deux petites zones de texte et une liste déroulante. Les zones de texte serviront au visiteur pour entrer les deux nombres qu'il veut traiter. On demandera à l'utilisateur d'entrer **uniquement des nombres entiers**. On pourrait prendre des nombres à virgule, mais je n'ai pas envie (et de toute façon, ça ne change rien !).

La liste déroulante donnera accès à cinq opérations, les opérations de base :

- addition ;
- soustraction ;
- division ;
- multiplication ;
- modulo.

Une fois que le visiteur aura saisi les nombres et le type d'opération, votre script devra afficher le calcul et la réponse.



Faites attention au cas où le visiteur voudrait diviser (ou faire le modulo) par zéro. En effet, en mathématiques, on ne peut pas diviser un nombre par zéro, ça donne un résultat indéfini. C'est pareil en PHP (et une erreur s'affichera).

Vous devrez donc vérifier les nombres que le visiteur entrera. S'il veut diviser ou faire le modulo par zéro, vous devrez afficher un message : *une erreur est survenue : vous ne pouvez pas diviser par zéro*.

Toutefois, il y a une deuxième chose que vous devez vérifier. Certains navigateurs, via des extensions, permettent de modifier les formulaires (tout le code source HTML, même !). On peut donc modifier les attributs **value** et compagnie. C'est là qu'est le risque. Pour identifier quelle opération il faudra faire, vous mettrez des valeurs particulières dans les attributs **value**. Vous devrez donc vérifier que vous recevez une de ces valeurs, sans quoi vous devrez afficher le message : *une erreur est survenue : opération indéfinie*.

Si vous affichez une de ces deux erreurs, vous ne devrez pas afficher le calcul et la réponse.

Mais dans tous les cas, vous afficherez toujours le formulaire pour qu'on puisse faire d'autres calculs.

### Quelques conseils

Avant tout, je vais vous imposer quelque chose : votre code devra être présenté d'une certaine façon. Je vais vous donner le squelette de votre code :

#### Code : PHP

```
<?php  
/* Déclaration des constantes */  
/* Déclaration des fonctions */  
/* Traitement des données */  
/* Affichage du calcul et du formulaire */
```

?>

Vous viendrez intercaler votre code dans ces commentaires. Si je fais ça, ce n'est pas pour vous embêter, j'ai mes raisons mais vous ne les connaîtrez pas tout de suite. Il est possible que certaines sections de code soient vides, par exemple la déclaration des constantes. Je vous le dis tout net : il n'y a pas besoin de constantes dans ce script. Mais gardez quand même le commentaire pour bien montrer que vous savez séparer votre code et le regrouper par « thème ».

Comme c'est votre premier TP, je vais être gentil et vous donner un aperçu en français de ce que sera votre code :

#### Code : PHP

```
<?php

/* Déclaration des constantes */
/* Déclaration des fonctions */
// Création d'une fonction qui prendra trois paramètres : les deux
// nombres sur lesquels on travaille et une information qui permettra
// de savoir quelle opération appliquer
// La fonction retournera soit une chaîne de caractères, soit le
// résultat du calcul
// Les chaînes de caractères que la fonction peut retourner sont
// les deux erreurs que j'ai mentionnées précédemment
// Vous devrez donc gérer ici les deux erreurs possibles
/* Traitement des données */
// Tout d'abord, vous allez vérifier si toutes les données existent
// et ne sont pas vides
// Pour vérifier si les variables existent, vous utiliserez l'une
// des deux fonctions qu'on a vues dans le chapitre sur les arrays
// Pour vérifier si les variables sont vides, regardez ce que
// retourne un : trim($var) != ''
// Si tout est OK, vous appellerez la fonction qu'on a déclarée
// plus tôt
// En fonction de ce que retourne la fonction, vous devrez vérifier
// si une erreur s'est produite
// Pour ce faire, il suffit de regarder quel est le type de valeur
// que la fonction nous retourne
// Si tout s'est bien passé, ça retourne un INT, sinon une STRING
// Pour vérifier si une expression est de type entier, on utilise
// la fonction is_int(). Pour vérifier si une expression est de type
// chaîne de caractères, on utilise la fonction is_string()
/* Affichage du calcul et du formulaire */
// Si le formulaire a été soumis, vous afficherez :
// soit une erreur si une erreur s'est produite,
// soit le calcul s'il n'y a pas eu d'erreur.
// Ensuite, vous afficherez le formulaire
```

?>

Et voilà, vous avez tout en mains. 😊 En vous appuyant sur ces quelques commentaires et sur tout ce que vous devriez avoir appris jusqu'à présent, vous devriez arriver à faire ce que je vous demande.

Il ne me reste plus qu'à vous souhaiter bonne chance, et bon courage surtout. 😊

#### Correction !

Bon, si vous êtes là, c'est que soit vous avez brillamment réussi, soit vous vous êtes lamentablement plantés et vous cédez à la tentation.

Si vous n'avez pas réussi ce script, mais que vous avez cherché longtemps (plusieurs dizaines de minutes au moins, si pas des heures !), regardez la correction, ça vous aidera.

Par contre, si vous n'avez pas cherché, ne regardez pas ! Ça ne vous apportera rien du tout.

#### Code : PHP

```
<?php
```

```
error_reporting(E_ALL);

/* Déclaration des constantes */
/* Déclaration des fonctions */
function operation($a, $b, $operation)
{
    if($operation == 0)
    {
        return $a + $b;
    }
    elseif($operation == 1)
    {
        return $a - $b;
    }
    elseif($operation == 2)
    {
        return $a * $b;
    }
    elseif($operation && $operation < 5)
    {
        if($b != 0)
        {
            if($operation == 3)
            {
                return $a / $b;
            }
            elseif($operation == 4)
            {
                return $a % $b;
            }
        }
        else
        {
            return 'Une erreur est survenue : vous ne pouvez pas
diviser par zéro';
        }
    }
    else
    {
        return 'Une erreur est survenue : opération indéfinie';
    }
}
/* Traitement des données */
if(isset($_POST['a']) && trim($_POST['a']) != '' &&
isset($_POST['b']) && trim($_POST['b']) != '' &&
isset($_POST['operation']) && trim($_POST['operation']) != '')
{
    $display = true;
    $result = operation((int)$_POST['a'], (int)$_POST['b'],
(int)$_POST['operation']);
    $error = is_string($result);
    if($_POST['operation'] == 0)
    {
        $signe = ' + ';
    }
    elseif($_POST['operation'] == 1)
    {
        $signe = ' - ';
    }
    elseif($_POST['operation'] == 2)
    {
        $signe = ' * ';
    }
    elseif($_POST['operation'] == 3)
    {
        $signe = ' / ';
    }
    else {
        $signe = ' % ';
    }
}
```

```

    }
    else
    {
        $display = false;
        $result = false;
        $error = false;
    }
    /* Affichage des résultats et du formulaire */
    if($display)
    {
        if($error)
        {
            echo $result;
        }
        else
        {
            echo (int)$_POST['a'] . $signe . (int)$_POST['b'] . ' = ' .
        $result;
        }
    }
    ?>
<form action="index.php" method="post">
    <p>
        Opérande n°1 :<input type="text" name="a" />
        <br />
        Opérande n°2 :<input type="text" name="b" />
        <br />
        Opération :
        <select name="operation">
            <option value="0">Addition</optionoption value="1">Soustraction</option>
            <option value="2">Multiplication</option>
            <option value="3">Division</option>
            <option value="4">Modulo</option>
        </select>
        <br />
        <input type="submit" value="Calculer" />
    </p>
</form>

```

Pas si mal, ce code, hein ? 😊

Vous avez dû obtenir un code proche, mais si vous n'avez pas la même chose, ce n'est pas grave. C'est même normal. Le but du jeu, pour le moment, c'est que le script fonctionne. Je n'ai pas mis de commentaires.

Il y a deux raisons à cela.

La première, c'est que le script est ridiculement simple et court.

La seconde, c'est que tous les commentaires sont un petit peu au-dessus, cachés dans les conseils.

Maintenant que vous avez eu un aperçu de « vrai » code, je dois vous avouer un truc : avec ce que vous connaissez, on peut faire bien plus court et efficace !

## Peut mieux faire !

On vient de voir un code qui **fonctionne**. J'insiste bien là-dessus, il fonctionne. Mais il est loin d'être idéal. C'est un code que je qualifierais de simplet, il n'y a pas de réflexion, pas d'utilisation de ce qu'on a appris pour simplifier le script. Et la logique même du script est assez moyenne et naïve.

C'est pour ça que je vais vous montrer un second code : le rendu est **strictement identique**, mais ce second script est bien mieux !

### Code : PHP

```

<?php

error_reporting(E_ALL);

/* Déclaration des constantes */
/* Déclaration des fonctions */
function operation($a, $b, $operation)

```

```
{  
    // On vérifie si l'opération est définie  
    if($operation >= 0 && $operation <= 4)  
    {  
        // On crée un array contenant trois des cinq opérations  
        // On ne peut pas mettre les deux dernières car si $b vaut  
        zéro, on aura une erreur  
        $calcul = array(  
            $a + $b,  
            $a - $b,  
            $a * $b  
        );  
        // Si l'on veut diviser ou faire le modulo et que $b est  
        égale à zéro, on retourne une erreur  
        if($b == 0 && $operation >= 3)  
        {  
            return 'Une erreur est survenue : vous ne pouvez pas  
diviser par zéro';  
        }  
        // Si l'on veut diviser ou faire le modulo et que $b est  
        différente de zéro  
        // On complète le tableau des différents calculs possibles  
        if($b != 0 && $operation >= 3)  
        {  
            $calcul[] = $a / $b;  
            $calcul[] = $a % $b;  
        }  
        // Enfin, on retourne le résultat qui correspond à  
        l'opération demandée !  
        return $calcul[$operation];  
    }  
    else  
    {  
        return 'Une erreur est survenue : opération indéfinie';  
    }  
}  
/* Traitement des données */  
// On vérifie que toutes les valeurs des associations contenues dans  
$_POST sont remplies  
$is_valid = true;  
foreach($_POST as $val)  
{  
    if(trim($val) == '')  
    {  
        $is_valid = false;  
    }  
}  
// On vérifie que le formulaire est soumis et bien rempli  
if(isset($_POST['a']) && isset($_POST['b']) &&  
isset($_POST['operation']) && $is_valid)  
{  
    $display = true;  
    $result = operation((int)$_POST['a'], (int)$_POST['b'],  
(int)$_POST['operation']);  
    $error = is_string($result);  
    // On utilise à nouveau un array pour simplifier le traitement  
    // On récupère le signe qui est associé à $_POST['operation']  
    $signe = array('+' , '-' , '*' , '/' , '%');  
    if(!$error)  
    {  
        $signe = $signe[(int)$_POST['operation']];  
    }  
}  
else  
{  
    $display = false;  
    $result = false;  
    $error = false;  
}  
/* Affichage des résultats et du formulaire */
```

```

if($display)
{
    if($error)
    {
        $result;
    }
    else
    {
        echo (int)$_POST['a'] . $signe . (int)$_POST['b'] . ' = ' .
    $result;
    }
}
?>
<form action="index.php" method="post">
    <p>
        Opérande n°1 :<input type="text" name="a" />
        <br />
        Opérande n°2 :<input type="text" name="b" />
        <br />
        Opération :
        <select name="operation">
            <option value="0">Addition</option>
            <option value="1">Soustraction</option>
            <option value="2">Multiplication</option>
            <option value="3">Division</option>
            <option value="4">Modulo</option>
        </select>
        <br />
        <input type="submit" value="Calculer" />
    </p>
</form>

```

Vous voilà confrontés à deux scripts qui font la même chose. L'un des deux est meilleur, le second. Mais pourquoi ? Tout d'abord, la longueur. En effet, si l'on omet les commentaires, le second script est moins long que le premier.

Mais bon, si ce n'était qu'une histoire de nombre de lignes, je vous le dis tout net : ça ne sert à rien de s'embêter.

Vous constatez que les modifications sont nombreuses dans les sections concernant la déclaration des fonctions et concernant le traitement des données. En effet, l'affichage est quelque chose de relativement statique : il n'y a pas de moyen pour améliorer réellement ça. Au mieux, on peut gagner quelques microsecondes et améliorer la clarté.

C'est toujours le traitement des données qui prend le plus de temps dans un script PHP, c'est donc là-dessus que vous devez porter votre attention pour essayer d'améliorer ça.

La plus grande amélioration dans la fonction, c'est la diminution du nombre de structures conditionnelles. Dans le premier script, il y en a six, alors que dans le second il n'y en a que trois, moitié moins.

Mais en quoi est-ce utile ?

Eh bien, quand vous avez une structure conditionnelle, le PHP évalue l'expression entre les parenthèses, ce qui prend du temps. Moins d'évaluation, moins de temps perdu. C'est particulièrement vrai dans les boucles. En effet, dans les boucles, les conditions sont réévaluées à chaque fois, donc si l'on peut supprimer ne serait-ce qu'une d'entre elles, on pourrait éviter beaucoup d'évaluations inutiles.

Pour éviter de devoir faire beaucoup de conditions, vous voyez que j'ai utilisé un array. En effet, les clés de cet array vont de 0 à 4, tout comme les attributs **value** de mon formulaire. Je peux donc très simplement récupérer le résultat du calcul en utilisant un array et la valeur de l'attribut **value**.

Mais je ne pouvais pas mettre les cinq opérations tout de suite dans mon array. En effet, si je l'avais fait, j'aurais eu une erreur :

#### Code : Console

```
Warning: Division by zero in C:\wamp\www\OwnBB\index.php on line 93
```

Cette erreur m'indique qu'à la ligne 93, j'ai divisé un nombre par zéro. Si vous rencontrez cette erreur, faites des echo et des var\_dump() de vos expressions pour savoir d'où vient le problème (c'est grâce à ces deux structures / fonctions qu'on peut identifier la plupart des bogues !).

Pour la déclaration des fonctions, c'est fini. On va maintenant passer au traitement des données, qui a aussi subi un petit *lifting*.

La première chose qui a été modifiée, c'est la condition pour vérifier si les variables existent et ne sont pas vides. Tout d'abord, vous devez savoir une chose que je ne vous ai pas encore dévoilée : quand un formulaire est soumis (avec un bouton submit ou un bouton image), toutes les associations que peut créer ce formulaire existent. Peu importe que vous ayez rempli ou non le formulaire, tout existe dès que le formulaire est soumis. La seule exception est la case à cocher, mais ça, vous le savez déjà.

On peut donc affirmer que si le formulaire n'a pas été soumis, `$_POST` est un array vide, sinon c'est un array rempli avec des associations. Or, si l'on se souvient bien de nos notions de transtypage, on sait que quand on transtype un array en booléen, deux cas sont possibles :

- si l'array est vide, le booléen vaut false ;
- sinon, le booléen vaut true.

Maintenant, le foreach : vous savez que ça sert à parcourir un array. Ce qui est bien avec cette structure, c'est qu'elle ne génère pas d'erreur si l'array qu'on lui passe est vide.

Avant mon foreach, je crée une variable booléenne qui vaut true.

Je parcours le tableau et j'assigne à cette variable le retour de `trim($val) == "`.

Cette expression peut retourner soit true, soit false. Si la variable est constituée uniquement d'espaces ou si elle est vide, ça retourne true, sinon false.

En clair, si une seule des valeurs contenues dans `$_POST` est vide (la chaîne vide " ou une chaîne faite uniquement d'espaces), `Sis_valid` vaudra false.

Je n'étais pas obligé de vous montrer cela, ça n'a pas grand intérêt ici. Mais quand vous aurez à vérifier si quarante petites zones de texte sont remplies, ça vous sera très utile.

La condition, vous devez la comprendre : on vérifie que `$_POST` n'est pas vide et que tous les champs sont remplis.

La deuxième grosse amélioration, c'est la suppression des cinq conditions ! On la résume en une seule grâce aux arrays pour la deuxième fois. 

Ce second script est plus propre et plus efficace que le premier. Ce que je veux que vous compreniez, c'est qu'il y a toujours moyen de voir un problème sous un autre angle. Le but du jeu, c'est de trouver l'angle qui vous donnera la solution la plus performante. Cette recherche de la solution la plus efficace, on appelle ça l'optimisation.

En PHP, le meilleur moyen d'optimiser son code (qu'il soit le plus rapide possible), c'est de revoir sa logique, d'essayer de supprimer des structures coûteuses en performance (comme des conditions, des boucles, des fonctions lourdes...).

Beaucoup de gens vont vous donner des petits « trucs », comme utiliser ' à la place de " pour délimiter vos chaînes de caractères. Ce genre de « truc » n'est pas faux, c'est vrai. Mais le temps que vous gagnerez est vraiment négligeable comparé à ce que vous pourriez gagner en revoyant votre logique. C'est pour ça que je n'appelle pas ce genre de pratique de l'optimisation, mais du bricolage, du chipotage.

C'est comme si vous deviez aller de Paris à Marseille. Vous avez une 2 CV, vous demandez à votre voisin : « À ton avis, j'irais plus vite si je gonfle un peu mes pneus ? » Ça ne vous fera gagner que quelques minutes.

Alors que si vous revoyiez votre logique et que vous preniez le TGV à la place de cette voiture, vous gagneriez des dizaines d'heures.

Désormais, quand vous regarderez un script, vous devrez toujours vous demander : « Ne peut-on pas l'améliorer ? » Il ne faut pas se contenter de quelque chose qui fonctionne, il faut toujours viser la performance. Et d'ailleurs, il y a sans doute encore moyen d'améliorer ce second script.  Après tout, peut-être n'ai-je pas eu l'angle de vue qui permettrait d'améliorer encore tout ça.

Pfiou, voilà une bonne chose de faite.

Après ce TP, plusieurs cas sont possibles.

Si vous avez réussi du premier coup, félicitations, c'est très bien.

Si vous avez dû chercher un peu, ou même beaucoup, mais que vous y êtes arrivés malgré tout, je vous félicite encore plus car vous vous êtes donné du mal et vous avez persévétré.

À ceux qui auraient totalement raté le TP, ne vous en faites pas. C'est en codant qu'on devient codeur.

Pour être honnête, je n'ai jamais réussi un seul TP du tutoriel qui m'a appris les bases du PHP, et ça ne m'a pas empêché d'aller assez loin. 

Si vous êtes chauds pour la partie II, qui s'annonce très amusante, que faites-vous encore là ?

## Partie 2 : MySQL, PHP et MySQL

Derrière bon nombre de scripts PHP se cache un élément quasiment incontournable : une base de données. C'est ce qu'est MySQL. Vous allez rapidement comprendre en quoi cet outil est pratique ! Mais avant de mêler le PHP et MySQL, on va commencer par apprendre à se servir de MySQL.

### MySQL, qui es-tu ?

Jusqu'à maintenant, vous avez appris les bases du PHP, un langage de scripts orienté vers le Web. Seulement, bien que vous ayez acquis plusieurs notions, plusieurs outils, les scripts que vous pouvez créer pour le moment sont assez limités.

On va remédier à cela en vous donnant la possibilité de créer des scripts que vous connaissez déjà sans doute : des scripts de news, de livre d'or, de compteur de connectés, etc.

Une base de données n'est pas indispensable pour faire tout cela, on peut s'en sortir sans. Mais la solidité d'une base de données facilitera énormément le développement de ces scripts.

#### Parle-moi de toi...

MySQL, MySQL... Je vous ai dit que c'était une base de données. Manque de chance, je vous ai déjà menti. Si MySQL n'était qu'une base de données, je vous le dis tout net : on n'aurait aucun intérêt à l'utiliser.

L'intérêt de MySQL réside dans le fait qu'il soit un **Système de Gestion de Base de Données Relationnelles**, que j'abrégerai dorénavant en SGBDR.

Mais qu'est-ce donc qu'un SGBDR ?

Un SGBDR, c'est une base de données structurée suivant des principes relationnels. Une base de données (que j'abrégerai en BDD) est un ensemble structuré et organisé qui permet le stockage et l'exploitation d'une grande quantité d'informations, de données.

Mais quel est l'intérêt ?

Prenons un exemple concret : un forum. Dans un forum, vous avez des catégories, des sujets et des messages. Les sujets appartiennent aux catégories et les messages appartiennent aux sujets. Il y a donc des relations entre ces trois éléments.

Si on n'utilise pas de SGBDR, on va mettre en place tout un système de gestion avec des fichiers en PHP (on verra comment manipuler les fichiers avec le PHP plus tard). Seulement, ce système va être très long et complexe à mettre en place, ce qui est plutôt ennuyeux. Qui plus est, mettre au point un tel système, fonctionnel et performant, ce n'est pas à la portée du premier venu. Il faut de l'expérience et beaucoup de connaissances pour gérer tous les cas de figure.

Maintenant, si on utilise un SGBDR, que va-t-on faire ? Tout d'abord, on ne créera pas de système de gestion ; en effet, on en a déjà un à disposition : le SGBDR. Tout ce que vous, les créateurs du script, aurez à faire, c'est de donner des ordres au SGBDR (« crée tel sujet dans telle catégorie », « donne-moi la liste des messages du sujet *machin* », « supprime ça », etc.).

Les avantages sont multiples, mais en priorité, je citerais la facilité d'utilisation, les performances et les possibilités de gestion.

Le but du jeu, ça va donc être de vous apprendre à communiquer avec un SGBDR, et plus particulièrement avec MySQL. En effet, pour communiquer avec MySQL, il va falloir apprendre un nouveau langage : le SQL.

Eh oui, pas de chance, vous n'en avez pas encore fini 😊.

Mais pourquoi un autre langage ? La raison est simple : les SGBDR sont organisés d'une certaine façon, on ne peut donc pas faire tout et n'importe quoi n'importe comment, il y a des procédures à respecter pour que le SGBDR fasse ce qu'il faut.

Si on devait prendre un exemple plus concret, imaginez simplement que vous disiez « Peux-tu m'indiquer le chemin ? » à quelqu'un qui ne parle pas français : est-ce qu'il vous comprendra ?

Non, il ne va rien comprendre et il essaiera de vous dire qu'il n'a rien compris.

Avant de vous parler du rapport entre MySQL et le PHP, je dois vous avertir de quelque chose. Dans la partie précédente de ce tutoriel, j'ai été assez théorique et j'allais presque toujours jusqu'au bout du détail.

Cette fois, ça ne sera pas le cas : je ne vais pas vous apprendre à maîtriser les SGBDR, mais je vous montrerai comment vous en servir.

Si je fais ça, c'est tout simplement parce que je ne m'y connais pas assez en SGBDR pour faire un tutoriel qui vous expliquerait tout sur tout.

Mais ne vous inquiétez pas, ce qui va suivre répondra aux attentes de 99.99 % d'entre vous. À ceux qui restent, vous n'aurez plus qu'à trouver un tutoriel plus complet ou à aller lire des documentations. 😊

#### On s'organise ?

La première des notions à acquérir pour utiliser un SGBDR, c'est l'organisation structurelle des bases de données. MySQL est un

logiciel, au même titre que Firefox, Kopete, Windows Media Player. Mais MySQL n'a qu'un seul et unique but : gérer des bases de données.

Le nombre de bases de données que MySQL peut gérer n'a qu'une seule limite : l'espace disponible sur votre disque dur. Autant dire qu'avant de remplir tout ça, vous aurez des tonnes de bases de données. 😊

Si on devait faire une analogie avec le monde réel, imaginez un bâtiment. Dans ce bâtiment, vous avez plusieurs pièces : chacune de ces pièces est une base de données.

Mais ces bases sont également organisées ! Elles sont organisées en **tables**. Chaque table porte un nom qui lui est propre, et elle est liée à la base de données à laquelle elle appartient. Ces tables sont elles-mêmes organisées comme l'est un classeur (ou un tableau) : en **colonnes** et en **lignes**.

Ces quatre notions organisationnelles (base, table, colonne et ligne) constituent la base de tout. C'est le fondement de l'organisation d'une base de données.

Pour reprendre l'analogie de tout à l'heure, si les pièces sont les bases de données, alors on trouve des armoires dans ces pièces. Les armoires représentent les tables. Chacune des armoires comporte différents tiroirs, ce sont les lignes. Et dans ces tiroirs, il y a différents compartiments, ce sont les colonnes. Et dans chaque compartiment, il y a une donnée.

Et bien évidemment, le type de compartiments, leur nombre, leur taille, etc. sont les mêmes pour tous les tiroirs de l'armoire.



Vous entendrez aussi parfois le terme *tupple*. Dans les bases de données, un *tupple* est une ligne dans une table.

Voici un schéma d'une base de données qui exprimera sans doute un peu mieux mes pensées :

Base de données						
	pseudo	password	signature	...		
Ligne 1	Haku	hahaha	Meuh	xxx	xxxx	xxx
Ligne 2	delphiki	ron	J'aime !	yyy	zzz	

	titre	contenu	...		
Ligne 1	Oh yeah	euh ..	rrrr	yyyy	iiii

Le bleu délimite la base de données. Les deux rectangles verts sont deux tables. Celle du dessus est une table qui contiendrait une liste de membres, par exemple. Il y a trois colonnes à cette table : « pseudo », « password » et « signature ». Dans cette table, il y a deux *tuples*, deux lignes de données.

La deuxième table pourrait contenir des news. Il y aurait deux colonnes, « titre » et « contenu », et une seule news (une ligne et une news par ligne).

Le schéma est laid, mais l'important est que vous compreniez l'organisation générale.

## La console à tout âge

Avant de nous lancer dans le SQL, il faudrait que vous lanciez le bon logiciel. ☺

Si vous êtes sous Linux, c'est très simple, vous avez juste à ouvrir un terminal et à taper : **mysql -u root**.

Si vous êtes sous Windows, ça peut être très simple ou un peu moins. Mais on va s'amuser un peu !

Tout d'abord, vous allez devoir trouver un fichier, un exécutable : **mysql.exe**.

Ensuite, vous allez devoir trouver le dossier dans lequel vous avez installé le serveur web (WAMP, EasyPHP, etc.).

Généralement, ça se trouve à la racine du disque dur ou dans le dossier **Program Files**. Par exemple, chez moi, j'accède au dossier de WAMP avec le chemin *C:\Wamp\*.

Une fois que vous avez ce chemin, vous allez ouvrir l'invite de commande de Windows. Pour cela, un petit raccourci : appuyez simultanément sur la touche « Windows » et la touche « R ». Tapez cmd dans l'invite de commande et validez !

Vous devriez voir la console de Windows s'ouvrir sous vos yeux. Maintenant, on va lancer la console de MySQL !

Tout d'abord, tapez ceci :

### Code : Console

```
cd "C:\Wamp\mysql\bin\"  
mysql -u root
```



Pensez à remplacer C:\Wamp\ par le chemin que vous aviez trouvé précédemment !

Si tout fonctionne, vous devriez arriver à ça :

The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe - mysql.exe --user=root'. The window displays the following text:

```
Microsoft Windows [version 6.0.6000]
Copyright <c> 2006 Microsoft Corporation. Tous droits réservés.

C:\Users\Cédric>cd "C:\Wamp\mysql\bin\  
C:\wamp\mysql\bin>mysql.exe --user=root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 5.0.27-community-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> _
```

Si vous n'arrivez pas à quelque chose de similaire, recommencez depuis le début, vous avez dû vous tromper dans le chemin du dossier.

Maintenant, que tout le monde regarde. En théorie, vous êtes les dieux suprêmes qui peuvent tout faire sur les bases de données. ☺

Mais vérifions cela en tapant ceci dans la console :

### Code : SQL

```
SHOW DATABASES;
CREATE DATABASE test;
SHOW DATABASES;
DROP DATABASE test;
SHOW DATABASES;
```

Vous devriez obtenir quelque chose de similaire à ceci :

The screenshot shows a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe - mysql.exe --user=root". The window displays the following MySQL session:

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| dev |
| mysql |
| ngu |
| ownbb |
| phpmyadmin |
| softbb |
+-----+
7 rows in set (0.00 sec)

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| dev |
| mysql |
| ngu |
| ownbb |
| phpmyadmin |
| softbb |
+-----+
7 rows in set (0.00 sec)

mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.01 sec)

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| dev |
| mysql |
| ngu |
| ownbb |
| phpmyadmin |
| softbb |
| test |
+-----+
8 rows in set (0.00 sec)

mysql> DROP DATABASE test;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW DATABASES;
```

Si vous n'obtenez pas quelque chose de proche (seuls les noms des bases pourraient différer), c'est que vous n'êtes pas connectés en tant que root. Si c'est le cas, relisez la partie sur l'ouverture de la console !

Si vous ne deviez retenir que deux choses de ce chapitre, ce serait ceci :

- le PHP et le SQL sont deux langages distincts et totalement indépendants l'un de l'autre ;
- une base de données s'organise en tables, elles-mêmes organisées en lignes et en colonnes.

Après cette courte introduction, on va accélérer le rythme et partir à la découverte du SQL !

## Créons une base de données et une table pas à pas

Avant de pouvoir travailler sur une base de données, on a besoin d'une... base de données ! Mais comme c'est très simple et rapide, on va aussi apprendre à créer une table. Dans l'exemple, ça sera une table pour gérer les messages d'un livre d'or (un TP qu'on fera prochainement !).

Soyez très attentifs et relisez, relisez, relisez encore et encore car la création des tables n'est pas spécialement complexe, mais il faut retenir plusieurs choses. 😊

### La base avant tout

Comme je vous l'ai dit dans le chapitre précédent, MySQL est un gestionnaire de bases de données, nous allons donc naturellement commencer par en créer une. Pour ce faire, ouvrez votre console MySQL comme je l'ai expliqué au chapitre précédent.

Si vous y êtes, entrez donc ces requêtes dans la console :

#### Code : SQL

```
SHOW DATABASES;
CREATE DATABASE test;
SHOW DATABASES;
```

Vous devriez obtenir quelque chose comme ceci :

```
C:\Windows\system32\cmd.exe - mysql -u root
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| dev |
| mysql |
| ngu |
| ownbb |
| phpmyadmin |
| softbb |
+-----+
7 rows in set (0.00 sec)

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| dev |
| mysql |
| ngu |
| ownbb |
| phpmyadmin |
| softbb |
+-----+
7 rows in set (0.00 sec)

mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.03 sec)

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| dev |
| mysql |
| ngu |
| ownbb |
| phpmyadmin |
| softbb |
| test |
+-----+
8 rows in set (0.00 sec)

mysql> -
```

SHOW DATABASES; est une requête qui retourne la liste des bases de données dont MySQL s'occupe. Comme vous le voyez sur l'image, mon premier SHOW DATABASES; me retourne sept bases (information\_schema, dev, mysql, ngu, ownbb, phpmyadmin et softbb).

MySQL me renvoie également un message, **7 rows in set**, qui se traduit en français par **7 lignes dans le jeu**. On appelle jeu (ou plus exactement, jeu de résultats) l'ensemble des lignes qu'une requête retourne.

Je vous ai dit qu'une base de données s'organisait en lignes et en colonnes, eh bien c'est pareil pour les requêtes. Les requêtes qui renvoient un résultat retournent toujours un jeu de résultats qui s'organisent en lignes et en colonnes. Si vous regardez l'image, vous pouvez remarquer que la liste de mes bases est présentée sous forme de tableau. Chaque base de données a sa propre ligne. Et tout en haut de la liste, il y a l'en-tête du tableau dans lequel on peut lire « Database ».

Souvenez-vous du chapitre précédent, quand je vous ai dit que chaque colonne portait un nom, eh bien c'est pareil pour les requêtes. Le jeu de résultats d'une requête est organisé en lignes et en colonnes, et de plus, chaque colonne porte un nom. Et vous verrez plus tard pourquoi c'est très important que les requêtes donnent un nom aux colonnes.

Ensuite, je crée une base qui s'appelle **test** avec la requête CREATE DATABASE test;.

MySQL me renvoie alors un message, **Query OK, 1 Row Affected**, qui se traduit en français par **Requête OK, 1 ligne affectée**. MySQL me dit donc que ma requête a bien été exécutée et que cette requête a affecté une ligne. La requête précédente retournait un jeu de résultats, mais pas celle-ci.

En réalité, il y a deux grands types de requêtes : celles qui retournent un résultat et celles qui affectent et modifient des données existantes. Par modification, c'est aussi la création, la suppression, la modification de données, etc.

Quand vous utilisez une de ces requêtes qui affectent des lignes, MySQL vous renvoie le nombre de lignes que votre requête a affectées. Ici, on crée une base, on affecte donc une et une seule ligne. Il est souvent intéressant de savoir combien de lignes une requête affecte, ça peut servir notamment pour résoudre des erreurs dans votre code. Mais ça permet bien d'autres choses.

La troisième requête ne vous est plus inconnue, elle retourne un jeu de résultats qui contient la liste des bases. Et comme on s'y attendait, on a bien une base qui est venue s'ajouter aux autres, la base `test`. On a donc deux preuves que cette base a bien été créée.

Maintenant, on va provoquer une erreur qui me permettra d'aborder deux points théoriques, dont un qui me tient très à cœur : le préfixage.

Exécutez donc cette requête, que reconnaissiez sans doute :

#### Code : SQL

```
CREATE DATABASE test;
```

Et pan, voilà ce que MySQL nous renvoie :

#### Code : Console

```
ERROR 1007 (HY000) : Can't create database 'test'; database exists
```

En français, ça donnerait :

#### Code : Console

```
ERROR 1007 (HY000) : Ne peut pas créer la base de données 'test'; la base de données
```

L'erreur est assez simple à comprendre, la base de données `test` existe déjà, on tente de créer une base portant le même nom, MySQL nous renvoie une erreur qui dit que la base existe déjà, c'est tout à fait logique.

La première des choses dont je veux vous parler, c'est d'une option. Si vous tentez de créer une base de données qui existe déjà, vous aurez droit à une erreur. Seulement, vous serez parfois amenés à créer une base de données sans savoir si elle existe déjà. Vous avez donc deux possibilités : la première, on joue la sécurité, on vérifie que la base n'existe pas avant de la créer. La seconde, plus expéditive, consiste à créer la table et regarder si on a une erreur.

Seulement, ces deux possibilités sont ennuyeuses et moches : la première nous force à faire deux requêtes et la seconde pourrait générer des erreurs. C'est pour ça que MySQL met à notre disposition une troisième possibilité : celle de créer la base si elle n'existe pas déjà.

Voici la requête :

#### Code : SQL

```
CREATE DATABASE IF NOT EXISTS test;
```

Hop, magique, plus d'erreur !

Mais un problème se pose alors : comment savoir si la base existait déjà ?

C'est simple, regardez un peu votre console... Vous devriez trouver ceci quelque part :

#### Code : Console

```
Query OK, 0 row affected, 1 warning
```

Comme vous le voyez, MySQL nous dit qu'aucune ligne n'a été affectée. Et voilà comment vous pouvez savoir si la base existait déjà : en regardant le nombre de lignes que la requête a affectées.

MySQL nous indique aussi autre chose : **1 warning, un avertissement** en français. N'y prêtez pas attention, ce n'est pas important pour l'instant.

On va maintenant parler du préfixage, qui est une convention que je me suis fixée. Vous n'êtes absolument pas tenus de la suivre, mais c'est une pratique que je trouve extrêmement utile. Un préfixe, c'est une particule placée au début d'un mot pour en modifier le sens.

Imaginons que vous utilisez deux applications en PHP qui nécessitent de créer une base, et imaginons que par hasard, les deux applications donnent le même nom à leur base. Vous allez avoir un problème, vu que vous ne pourrez pas créer deux bases portant le même nom. Et c'est là que le préfixage intervient : en ajoutant une particule devant le nom de la base, le problème disparaît !

Si les deux applications voulaient appeler leur base **lala**, on aurait eu un problème. Mais maintenant, comme chacune des applications donne un préfixe, par exemple **ooo\_** et **uuu\_**, le problème ne se pose plus puisque les bases portent le nom de **ooo\_lala** et **uuu\_lala**.

Bon, en réalité, il est très rare que des applications en PHP créent des bases, donc le préfixage des noms des bases n'a que peu d'intérêt. Mais quand on jouera avec des tables, des colonnes, des jointures, etc., vous verrez combien préfixer les noms de bases, tables et colonnes peut nous faciliter bien des requêtes.

Avant de passer à la création de tables, je vais encore vous montrer une requête et vous dire un mot sur le nom des bases. S'il est utile de savoir créer une base, il est également utile de pouvoir en supprimer une. Voici comment faire :

#### Code : SQL

```
DROP DATABASE test;
```

Cette requête supprimera la base de données qui s'appelle **test**.

Juste pour le fun, exécutez à nouveau cette requête pour obtenir ce joli message d'erreur :

#### Code : Console

```
ERROR 1008 (HY000) : Can't Drop Database 'test'; database doesn't exist
```

Ça nous dit simplement qu'on ne peut pas supprimer (**drop** signifie « supprimer » en français) la base '**test**' car elle n'existe pas.



**Faites très attention** avec le **drop**. C'est très pratique, mais c'est totalement irréversible. Si vous supprimez une base de données, vous supprimez tout ce qu'il y a dedans, tables et données comprises. Soyez donc sûrs de ce que vous faites ! (On apprendra à faire une sauvegarde dans peu de temps.)

En PHP, il y avait une convention à suivre pour nommer ses variables. Eh bien c'est pareil pour le nom des tables. Pour comprendre cette convention, il faut savoir une chose : quand vous créez une base de données, vous créez en réalité un dossier sur votre disque dur (qui porte le nom de votre base de données) qui se trouve dans le répertoire `\mysql\data\` (chez moi, il est dans `c:\Wamp\mysql\data\`). La convention de nommage est donc simple, vous pouvez utiliser n'importe quel caractère autorisé dans un nom de dossier, sauf le *slash* (/), le *backslash* () et le point (.).

Toutefois, c'est encore une convention que je me fixe, mais ne donnez pas de noms trop bizarres à vos tables. Idéalement, limitez-vous à des caractères alphabétiques et à l'*underscore* (\_).

#### On passe à table ?

Maintenant qu'on a des bases, on va créer une table. Rien de plus simple, je vous rassure. 😊

La petite requête de mon cœur :

#### Code : SQL

```
CREATE TABLE test;
```

Mais manque de chance, MySQL vient nous ennuyer avec une nouvelle erreur :

#### Code : Console

```
ERROR 1046 (3D000) : No database selected
```

Ce qui nous donne en français :

#### Code : Console

```
ERROR 1046 (3D000) : Aucune base de données sélectionnée
```

Là, si vous ne trouvez pas cette erreur normale, vous devriez retourner lire le chapitre qui parlait de la structure d'une base de données.

Rappelez-vous du schéma du chapitre précédent : on voyait très bien que le bleu (qui représente la base de données) entoure totalement le vert (qui représente les tables). Si le bleu entoure le vert, c'est parce que le vert appartient au bleu. Le vert n'existe que dans le bleu. Ainsi, une table ne peut pas exister sans base de données, une table appartient à une base de données.

Or, si on regarde la requête précédente, rien n'indique dans quelle base je veux créer cette table. On va donc remédier à cela. Mais avouez que ça serait très ennuyeux qu'à chaque requête on soit obligés de spécifier sur quelle base de données on travaille. C'est pour ça que le SQL nous offre une requête très pratique : cette requête va dire à MySQL sur quelle base on veut travailler, et MySQL travaillera avec cette base jusqu'à ce qu'on lui demande de changer de base, ou jusqu'à ce qu'on relance le serveur MySQL.

Plus de blabla, on va passer à la pratique.

On a vu comment faire la liste des bases de données, on va faire la même chose pour les tables.

Exécutez cette requête :

#### Code : SQL

```
SHOW TABLES;
```

Oups, à nouveau l'erreur qui nous indique qu'on n'a pas sélectionné de base. C'est logique, MySQL ne nous montre les tables que d'une seule base à la fois, mais comme il ne sait pas sur laquelle travailler, il nous renvoie cette erreur.

Remédions à cela :

#### Code : SQL

```
SHOW TABLES FROM mysql;
```

Ahah, parfait, on a une liste de tables ! C'est tout à fait logique, on a indiqué à MySQL avec quelle base on veut travailler. Seulement, taper FROM mysql plusieurs fois ne m'enchantera guère, on va donc lui dire une bonne fois pour toutes qu'on veut travailler sur la base **mysql** (c'est une base que MySQL crée de lui-même, c'est tout à fait normal qu'elle soit là).

Pour cela, exécutez ces requêtes :

#### Code : SQL

```
USE mysql;  
SHOW TABLES;
```

Le résultat est strictement identique à la requête précédente ; normal, c'était le but. 😊

On peut utiliser MySQL sans base, mais en pratique, on utilisera toujours des tables ; vous devrez donc toujours utiliser le **USE** pour dire sur quelle base vous voulez travailler.

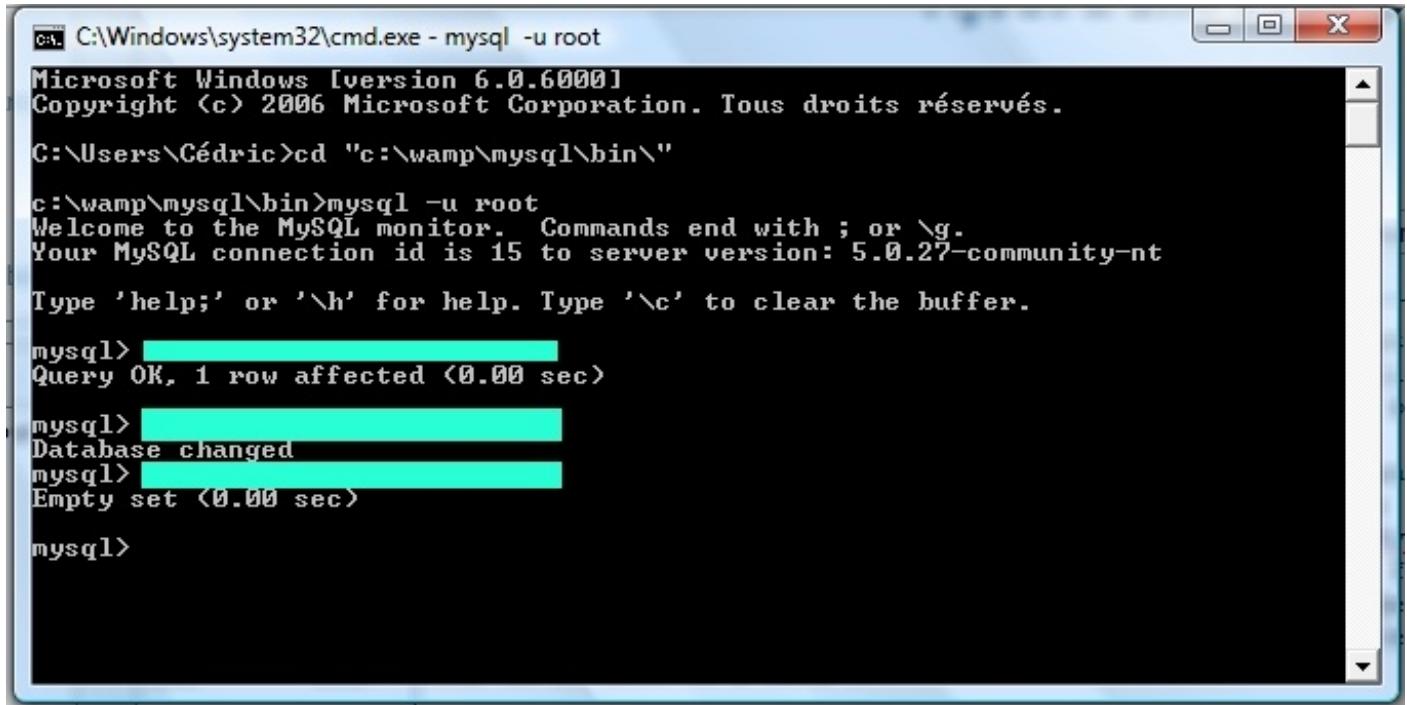
Mais on va se faire une autre base de données : celle-là étant assez importante, ça serait dommage de faire des bêtises dessus. Voici donc un mini TP : vous allez créer une base **dev\_dev** (souvenez-vous du préfixage, c'est pour ça que le dev\_ est là !), dire à MySQL que vous voulez travailler sur cette base et enfin récupérer la liste des tables de cette base toute fraîche.



N'oubliez surtout pas le « ; » à la fin de chacune de vos requêtes. En PHP, il indique la fin de l'instruction, en SQL il

 indique la fin de la requête.

Vous devez arriver à ceci :



```
C:\Windows\system32\cmd.exe - mysql -u root
Microsoft Windows [version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. Tous droits réservés.

C:\Users\Cédric>cd "c:\wamp\mysql\bin\  

c:\wamp\mysql\bin>mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 5.0.27-community-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW TABLES;
Query OK, 1 row affected (0.00 sec)

mysql> Database changed
mysql> Empty set (0.00 sec)

mysql>
```

J'espère que vous avez réussi ou au moins essayé jusqu'au dernier moment ; sinon, regarder la solution ne vous sera pas très profitable.

**Secret** ([cliquez pour afficher](#))

**Code : SQL**

```
CREATE DATABASE dev_dev;  
USE dev_dev;  
SHOW TABLES;
```

Puisque tout est prêt, pourquoi ne pas créer une table ?

On va créer une table pour un livre d'or, je l'appellerai **dev\_gbook**. Si je choisis ce nom, ce n'est pas pour rien. C'est encore pour cette histoire de préfixage. Devant le nom de mes tables, je rajoute toujours le nom de la base sans le préfixe, suivi d'un *underscore*.

Le nom de la base est **dev\_dev**, le suffixe est **dev\_**, je garde donc le **dev** et j'ajoute un *underscore* pour obtenir **dev\_**. Le nom de ma table est **gbook** (abréviation personnelle de *Gold Book*, « Livre d'or » en français), et en rajoutant le **dev\_**, j'obtiens bien **dev\_gbook**.

Dans une même base de données, le préfixe sera donc le même pour toutes les tables.

Essayons donc d'en créer une :

**Code : SQL**

```
CREATE TABLE dev_gbook;
```

Eh oui, on n'est pas encore au bout de nos peines, une nouvelle erreur pointe le bout de son nez :

**Code : Console**

```
ERROR 1113 (42000): A TABLE must have at least 1 COLUMN
```

En français :

#### Code : Console

```
ERROR 1113 (42000) : Une TABLE doit avoir au moins 1 colonne
```

Quand on crée une base, on ne doit pas dire quelles tables iront dans cette base. Mais quand on crée une table, on doit donner sa structure (on verra ce que c'est plus tard ; pour le moment, considérez que la structure, ce sont les colonnes) au moment de la création. Comme on ne donne pas la structure de la table, MySQL nous renvoie une erreur.

Attention à la prochaine partie, ça va faire mal. 😊

### Le plat de résistance !

On va s'attaquer au plus gros morceau de ce chapitre : la création de la structure de la table. C'est à la fois le plus simple et le plus compliqué des sujets de ce chapitre.

C'est simple car si vous arrivez à bien vous représenter la structure d'une table, vous n'aurez aucune difficulté. C'est compliqué parce qu'il y a beaucoup de mots-clés à retenir, ainsi que diverses petites choses.

Le plus simple pour vous guider dans les méandres de la structure d'une table, c'est de partir d'une requête qui crée une table pour toucher un peu à tout.

Mais avant de vous donner cette requête, on va prendre le temps de réfléchir à ce qu'on veut faire. On veut faire une table pour recueillir les messages d'un livre d'or.

La première question à se poser est : « Qu'est-ce qu'on va devoir mettre dans cette table ? »

La première chose, c'est l'évidence même : le message en lui-même. Le message est un texte, une chaîne de caractères de longueur variable (ça peut être très court comme très long).

Deuxième chose très utile, le pseudo de l'auteur du message. Ce pseudo est également une chaîne de caractères, mais on peut considérer que c'est une chaîne contenant assez peu de caractères (j'ai rarement vu des pseudos de plus de 50 ou 100 caractères 😊).

Troisième information qui nous intéresse : la date. Là, c'est nouveau pour vous. On n'a jamais vu comment utiliser des dates, ni en PHP, ni en SQL. En SQL, les dates sont un type de données particulières, ce sont des données de type... date. Eh oui, en SQL aussi, on retrouve cette histoire de typage.

On va encore ajouter deux colonnes à cette table. La première est indispensable, la seconde optionnelle (mais malgré tout très utile).

La colonne optionnelle nous servira à enregistrer l'IP (on a vu ce que c'était au tout début) du posteur du message au moment où il l'a posté. L'intérêt principal, c'est de pouvoir retrouver qui a posté le message (bien que ce soit difficile, voire quasiment impossible) au cas où on en aurait besoin (par exemple s'il a posté un message répréhensible). On a vu que l'IP est sous la forme de quatre nombres séparés par des points. C'est donc une chaîne de caractères. Toutefois, vous devez savoir qu'on peut convertir l'IP en un nombre entier, et on va le faire. La colonne stockera donc des nombres entiers. Vous verrez pourquoi il est plus intéressant de stocker l'IP sous forme d'entier par la suite.

Ces quatre colonnes étaient assez évidentes, il n'y a rien qui ne soit pas naturel. Mais la cinquième est un peu plus étrange à première vue. Quand vous aurez beaucoup de messages dans votre livre d'or, il arrivera peut-être que vous receviez des messages nuisibles (insultes, spams, etc.) qu'il vous faudra supprimer. Seulement, comment va-t-on dire à MySQL quel message supprimer ?

Il faudrait une information qui serait unique, qui serait propre à chaque message.

On pourrait utiliser d'autres choses pour identifier le message, le pseudo par exemple, ou l'IP. Malheureusement, on ne peut pas être certains qu'il n'y aura pas deux messages avec la même IP ou le même pseudo, et si ça arrive, on pourrait supprimer par accident un message qui ne devait pas l'être.

On va donc utiliser un nombre : à chaque message correspondra un nombre, ce sera son **identifiant** (abrégé en **id**). Ce nombre s'incrémentera de 1 à chaque nouveau message posté : ainsi, on n'aura jamais deux messages ayant le même nombre. Et pour supprimer un message, il nous suffira de connaître le nombre qui lui correspond !

Vous l'avez sans doute compris, cette cinquième colonne stockera donc des nombres.

Maintenant que vous savez ce qu'on va mettre dans cette table, voici sans plus attendre la requête tant convoitée :

#### Code : SQL

```
CREATE TABLE dev_gbook (
    gbook_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    gbook_ip INT NOT NULL,
```

```
gbook_msg TEXT NOT NULL,  
gbook_pseudo VARCHAR(100) NOT NULL,  
gbook_date DATETIME NOT NULL  
) ;
```

MySQL devrait vous répondre ceci :

#### Code : Console

```
Query OK, 0 rows affected
```

Vérifions que la table a bien été créée...

#### Code : SQL

```
SHOW TABLES;
```

Si vous voyez un tableau avec comme seule ligne **dev\_gbook**, O.K. : tout est bon !

Comme vous le voyez, comme pour les noms des bases et des tables, j'ai ajouté un préfixe (le nom de la table sans son préfixe) devant le nom de toutes mes colonnes. L'intérêt principal, c'est d'éviter les noms ambigus. Plus tard, on manipulera plusieurs tables en même temps : deux, trois, quatre ou plus encore. Imaginez que vous ayez deux tables ayant une colonne pour un identifiant (id). Si vous nommez les deux colonnes id, comment MySQL saura de quelle colonne vous parlez ?

Avec le nom de la table en préfixe, vous n'aurez jamais ce problème.

Le deuxième avantage, c'est que ça vous évitera des erreurs. Imaginons que je n'aie pas mis le préfixe, le nom de mes colonnes aurait donc été : id, ip, msg, pseudo et date. Mais manque de chance, pour MySQL, DATE a un sens particulier (comme en PHP, je ne peux pas donner n'importe quel nom à mes fonctions), et ça vous retournera une belle erreur de syntaxe (eh oui, les requêtes SQL suivent une syntaxe très précise).

Après ce petit intermède, on va passer au point le plus important : la requête qui crée notre table. Je vais vous expliquer tout ce charabia. ☺

On commence en douceur par la base de la requête :

#### Code : SQL

```
CREATE TABLE dev_gbook () ;
```

Cette requête dit à MySQL de créer une table nommée **dev\_gbook**. La structure de la table sera mise entre parenthèses.

Viennent ensuite cinq lignes ; vous vous en doutez sûrement, chaque ligne dit à MySQL d'ajouter une colonne. On va étudier ces lignes une par une :

#### Code : SQL

```
gbook_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
```

Manque de chance, la première est la plus complexe de toutes. ☹ J'y reviendrai donc plus tard ! On va plutôt commencer par la troisième :

#### Code : SQL

```
gbook_msg TEXT NOT NULL
```

Tout d'abord, vous voyez le nom de la colonne, **gbook\_msg**. Juste après le nom de colonne, on indique le type de la colonne. Ici, on veut un type qui nous permettrait de stocker un texte de longueur variable (qui peut être très long) : le type de colonne le plus

adapté est donc le type **TEXT**.

Viennent ensuite deux mots-clés, **NOT** et **NULL**. Mettre un NOT NULL signifie que vous ne pourrez pas mettre une valeur NULL dans la table (c'est une approximation, j'y reviendrai plus tard). Si vous mettez simplement NULL, vous pourrez mettre des NULL dans votre table (il y a des cas dans lesquels utiliser des NULL est utile ; par exemple, si je voulais donner la possibilité de donner ou non une note, 8/10 par exemple, dans le Livre d'or).



Ne mettre ni NOT NULL, ni NULL n'entraînera pas d'erreur, mais ça revient à mettre NULL.

Passons à la cinquième ligne :

**Code : SQL**

```
gbook_date DATETIME NOT NULL
```

Si vous avez bien compris l'explication précédente, vous devriez être à même de comprendre cette ligne. On indique tout d'abord le nom de la colonne, **gbook\_date**, son type, **DATETIME**, et enfin on dit qu'on ne peut pas mettre de NULL.

Revenons un peu en arrière, à la quatrième ligne :

**Code : SQL**

```
gbook_pseudo VARCHAR(100) NOT NULL
```

Dans cette colonne, des choses bien connues comme le nom, **gbook\_pseudo**, et le NOT NULL. La seule chose qui diffère, c'est le type. Le type VARCHAR est un type de colonnes qui permet de stocker de 0 à 255 caractères, c'est donc pour stocker une chaîne assez courte. Mais qu'est-ce que le 100 entre parenthèses ? C'est tout simplement le nombre maximal de caractères qu'on pourra stocker dans cette colonne. Si vous tentez de stocker une chaîne de 101 caractères, la chaîne sera coupée et seuls les 100 premiers caractères seront sauvés. Si vous connaissez le nombre maximal de caractères (par exemple, si vous imposez des pseudos de 20 caractères maximum), vous avez tout intérêt à mettre ce nombre entre parenthèses. Quand MySQL va lire vos données, s'il sait qu'il y a maximum 20 caractères, il n'aura qu'à lire 20 caractères. Mais si vous n'avez pas indiqué une taille adéquate, que vous avez mis 255 par exemple, MySQL lira 255 caractères pour en récupérer 20 (à nouveau, c'est une approximation, mais l'idée est là).

Encore un peu plus en arrière, on a la seconde ligne :

**Code : SQL**

```
gbook_ip INT NOT NULL
```

Très simple : on crée une colonne qui s'appelle **gbook\_ip**, de type entier (INT) et qui ne peut pas contenir de NULL.

Maintenant que vous êtes un peu plus familiers avec les descriptions des colonnes, on va pouvoir revenir sur la première :

**Code : SQL**

```
gbook_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
```

Je vais décomposer la ligne :

- gbook\_id ;
- INT UNSIGNED ;
- NOT NULL ;
- AUTO\_INCREMENT ;
- PRIMARY KEY.

La première chose, vous connaissez, c'est le nom.

La seconde, c'est comme toujours le type. Mais si vous reconnaissiez le mot-clé INT, qui indique une colonne de type entier,

vous ne savez pas ce qu'est ce UNSIGNED, et à quoi il sert.

Est-ce que vous vous êtes déjà demandés comment votre ordinateur différenciait un entier positif d'un entier négatif ? Eh bien c'est très simple. Quand votre ordinateur stocke un nombre entier négatif, il stocke le nombre entier positif et une petite information (une signature, le signe) en plus qui indique que c'est un nombre négatif. Concrètement, si vous avez besoin de cinq bits pour stocker un nombre positif, vous aurez besoin de six bits pour stocker le nombre négatif correspondant, la signature tient donc sur un bit, appelé bit de signe. Le nombre de bits que MySQL utilise pour stocker un INT et un UNSIGNED INT est le même, mais avec un UNSIGNED INT, vous dites à MySQL que le nombre est toujours positif, le bit de signe est donc inutile, ce qui vous fait gagner un bit pour le stockage du nombre, ce qui vous permet de stocker des nombres plus grands. Vous verrez cela dans le prochain chapitre.

Le NOT NULL, pas besoin d'en rajouter. 😊

Maintenant, un nouveau mot-clé : AUTO\_INCREMENT. Si vous regardez ce mot, vous voyez INCREMENT, c'est assez proche de quelque chose que l'on connaît : l'incrémentation. Rappelez-vous, je vous ai dit que pour identifier chaque message, on utiliserait un nombre qui s'incrémenterait. Seulement, ça serait lourd : il faudrait une requête pour récupérer le plus grand nombre. C'est pour ça que MySQL a ajouté une option qui permet de ne pas avoir à se soucier de ça : l'auto-incrémentation. Quand vous stockerez un message, grâce à l'auto-incrémentation, on n'aura pas à se soucier de l'id à attribuer au message, MySQL s'en chargera pour nous.



L'auto-incrémentation ne fonctionne bien entendu qu'avec des colonnes de type entier.

Les deux derniers mots-clés, PRIMARY et KEY, vous sont aussi inconnus. Pour l'instant, on va les oublier. Sachez juste que quand vous mettez l'option AUTO\_INCREMENT sur une colonne, vous devez obligatoirement ajouter l'option PRIMARY KEY (ce n'est pas tout à fait vrai, mais pour le moment, considérez que c'est comme ça).

Et voilà : vous savez à présent comment créer une table et des colonnes. Il existe plusieurs autres types de colonnes, vous devez toujours utiliser le type qui convient le mieux pour stocker vos données. Si vous devez stocker un nombre, utilisez une colonne de type INT et pas VARCHAR. Les deux fonctionnent, mais la colonne de type INT est bien plus légère et rapide.

Eh bien l'air de rien, vous venez d'apprendre un paquet de choses. 🎉 Si vous avez un seul doute, n'hésitez pas à lire et à relire encore et encore, car le prochain chapitre se basera presque exclusivement sur les types de colonnes et les options. 😊

Petit cadeau qui vous permettra de faire des tests en créant des tables avec différentes colonnes :

#### Code : SQL

```
SHOW COLUMNS FROM dev_gbook;
```

Ça devrait vous aider pour visualiser les différentes colonnes d'une table, leur type, si elles peuvent être nulles, etc.

Malgré et à cause de la quantité d'informations dans ce chapitre, il n'y a pas de QCM. Les points-clés de ce chapitre sont :



- la notion de **jeu de résultats** ;
- la création et la suppression de bases de données ;
- l'obligation de dire à MySQL sur laquelle on travaille ;
- la notion de **type de colonnes** ;
- la notion d'**option de colonne** (NULL / NOT NULL, AUTO\_INCREMENT) ;
- les conventions de nommage des bases, tables et colonnes.

## Gérons nos tables !

Dans le chapitre précédent, on a vu comment créer une table simple. Mais je ne vous ai pas montré tous les types de colonnes, et je n'ai encore rien dit sur la gestion de la table après sa création.

Par gestion, j'entends un peu de tout : ajouter une colonne, en supprimer, en modifier, etc. !

Un chapitre très intéressant qui vous permettra de commencer à vraiment manipuler les tables. 😊

### Une histoire d'entiers

Il est temps de quitter un peu la pratique pour revenir à la bonne vieille théorie. Comme je vous l'ai dit, il faut toujours utiliser le type de colonnes qui correspond le mieux au type de données que vous y stockerez. C'est pour cela que MySQL met à notre disposition un très grand nombre de types de colonnes, près d'une trentaine. 🍪

Tout d'abord, je vais vous donner une requête qui ne fonctionne pas mais qui représente tous les types de colonnes et leurs options :

#### Code : SQL

```
CREATE TABLE nom_table (
    nom_colonne type_colonne [NULL | NOT NULL] [DEFAULT
    valeur_par_defaut] [AUTO_INCREMENT] [[PRIMARY] KEY] [COMMENT
    'commentaire']
);
```

Il y a quelques mots en français, ça devrait donc vous aider. 😊 Tout d'abord **nom\_table**, c'est bien logiquement le nom de la table que vous allez créer. Ensuite vient **nom\_colonne** : c'est le nom que vous donnerez à une colonne. Après, on a **type\_colonne**, ça indique le type de la colonne ; on verra tous les types possibles par la suite.

Maintenant on commence les choses sérieuses : tout d'abord, les crochets. Quand quelque chose est mis entre crochets, il est facultatif, on peut ne pas le mettre. Et vous le voyez, les crochets peuvent s'imbriquer.

Deuxième chose, la barre verticale(), alias le *pipe*. Quand vous avez plusieurs valeurs possibles pour un même élément, on les sépare par une barre verticale.

Après le type de la colonne, on peut trouver soit un NULL, soit un NOT NULL. Je vous ai déjà expliqué en quoi ces mots-clés consistaient, je ne reviens donc pas dessus.

Mais voilà un petit nouveau : DEFAULT. Imaginons que vous fassiez un système de news avec validation des news (c'est-à-dire qu'un administrateur doit autoriser l'affichage de la news pour qu'elle soit visible aux yeux de tous). Pour différencier une news validée d'une news non validée, vous utiliseriez une colonne supplémentaire qui pourrait prendre deux états, *validée* ou pas (on verra plus tard quel type de colonne correspondait à cette donnée). Seulement, vous devez bien vous rendre compte que la plupart du temps, quand on va ajouter une news, elle ne sera pas validée. La colonne qui contiendra cette information aura donc très souvent la même information quand on insérera la news dans la table. Et je ne sais pas pour vous, mais je trouve ça ennuyeux de devoir toujours dire à MySQL que la news n'est pas validée.

C'est pour cela que MySQL nous permet d'utiliser des valeurs par défaut. Si on ne renseigne pas la valeur d'une colonne qui a une valeur par défaut, MySQL stockera la valeur par défaut, sinon il stockera la valeur qu'on lui donne.

Ça doit vous rappeler quelque chose, non ? Oui ! Les fonctions en PHP : souvenez-vous, on pouvait donner une valeur par défaut à un ou plusieurs paramètres ; eh bien c'est pareil avec MySQL.

Pour reprendre l'exemple de la table du Livre d'or, on aurait pu mettre une valeur par défaut sur une colonne : celle qui stocke la date.

 Mais c'est variable la date, non ?

Oui, la date à laquelle le message est posté varie, mais la façon dont on obtient la date ne varie pas (en réalité, on utilise une fonction de MySQL : oui, oui, en SQL aussi il y a des fonctions 😊).

Quand vous pouvez utiliser une valeur par défaut, faites-le. Vos requêtes seront plus courtes et plus rapides à exécuter. **valeur\_par\_defaut** est bien entendu à remplacer par la valeur par défaut.

La prochaine option, également facultative et connue, c'est l'auto-incrémation. Je ne reviens pas non plus là-dessus, mais je vous rappelle quand même qu'on ne peut utiliser l'auto-incrémation que sur des colonnes de type entier !

Voici ensuite quelque chose qu'on a déjà vu, mais dont je n'ai pas parlé en détail, et je ne le ferai pas maintenant. Pour le moment,

retenez juste que vous n'avez pas à utiliser cette option, sauf sur les colonnes qui ont une auto-incrémentation.

Dernier mot-clé : COMMENT. Ce dernier sert à donner un commentaire sur une colonne, sur son rôle. Parfois, les noms des colonnes ne sont pas très explicites (si je nomme ma colonne gbook\_ivba, vous ne savez pas à quoi elle sert !), et c'est assez difficile de savoir à quoi sert l'une ou l'autre. Mais si vous mettez un commentaire, ça sera un jeu d'enfant 😊 (mais bon, le mieux c'est de donner des noms explicites 😊).

Voilà pour le gros de la requête, on va maintenant passer à quelque chose de tout aussi intéressant : les types de colonnes !

Commençons en douceur par les colonnes de types entiers. Eh oui, il y a plusieurs types de colonnes pour stocker des nombres entiers, cinq exactement : TINYINT, SMALLINT, MEDIUMINT, INT et BIGINT.

Si vous parlez un peu anglais, vous pouvez traduire le nom des différents types : très petit entier, petit entier, moyen entier, entier et grand entier.

Si je vous dis que stocker le nombre 200 prend beaucoup moins de place que de stocker le nombre 2 000 000 000, vous êtes d'accord ?

MySQL lui l'est. Si vous ne lui dites pas que vous stockez un petit nombre, MySQL va devoir lire plus d'informations pour s'assurer qu'il lit bien tout le nombre, qu'il n'en oublie pas un morceau. Et donc, si le nombre est petit, MySQL va lire des données pour rien, ce qui est une perte de temps. C'est pourquoi MySQL nous permet de spécifier la grandeur du nombre grâce à ces cinq types. Plus exactement, ces types permettent de stocker un nombre compris dans un intervalle (en gros, le nombre qu'on veut stocker doit être compris entre un nombre plus petit et un nombre plus grand ; on appelle ces deux nombres limites **les bornes**). Voici les intervalles :

Type de colonne	Intervalle
TINYINT	de - 128 à + 127
SMALLINT	de - 32 768 à + 32 767
MEDIUMINT	de - 8 388 808 à + 8 388 807
INT	de - 2 147 483 648 à + 2 147 483 647
BIGINT	de - 9 223 372 036 854 775 808 à + 9 223 372 036 854 775 807

Comme vous pouvez le constater, les intervalles sont plutôt variés et certains sont vraiment énormes. Quand vous stockerez un nombre, vous devrez toujours vous poser la question : « Quelle est la valeur maximale que je pourrais stocker ? ».

Il y a des cas où c'est très simple, par exemple dans le cas d'une colonne servant à stocker un pourcentage de réussite. La limite étant 100 %, la valeur maximale est 100. Le type de colonne le plus adapté est donc TINYINT.

Malheureusement, il n'est pas toujours aisément de trouver cette valeur maximale, par exemple pour l'id des messages de votre livre d'or. Combien y en aura-t-il ? 100 ? 1 000 ? 10 000 ?

Dans ce genre de cas, n'hésitez pas à utiliser le type INT (quand vous aurez plus de deux milliards de messages, vous m'appellerez 😊) qui suffit amplement dans la plupart des cas.

Vous connaissez donc cinq types de colonnes, eh bien je dois vous avouer quelque chose : vous en connaissez en réalité dix, le double !

Pourquoi ? À cause de l'UNSIGNED. Je vous ai parlé de la différence entre le type INT et INT UNSIGNED dans le chapitre précédent : c'est pareil pour ces cinq types, on peut les mettre en UNSIGNED.

Et en réalité, la seule chose qui change entre un entier et un entier UNSIGNED, ce sont les bornes de l'intervalle. La borne inférieure est 0 (le plus petit nombre entier positif), et la borne supérieure est...

Type de colonne	Intervalle
TINYINT UNSIGNED	de 0 à 255
SMALLINT UNSIGNED	de 0 à 65 535
MEDIUMINT UNSIGNED	de 0 à 16 777 615
INT UNSIGNED	de 0 à 4 294 967 295
BIGINT UNSIGNED	de 0 à 18 446 744 073 709 551 615

Si vous avez fait attention aux bornes supérieures, vous aurez peut-être remarqué trois choses.

Premièrement, les bornes supérieures sont égales à *deux fois la borne supérieure des colonnes correspondantes sans le UNSIGNED plus 1*. Par exemple pour le TINYINT, la borne supérieure était 127. Avec la formule, on a comme borne supérieure pour le TINYINT UNSIGNED :  $(2 * 127) + 1 = 254 + 1$ .

La seconde, c'est que la borne supérieure est également égale à une certaine puissance de 2 moins 1 (pour ceux qui ne savent pas encore ce que sont les puissances, 2 puissance 3 est égal à  $2 * 2 * 2$ , 2 puissance 7 est égal à  $2 * 2 * 2 * 2 * 2 * 2 * 2$ ). Pour le TINYINT UNSIGNED, la borne supérieure est égale à  $255 = 256 - 1 = (2^{16}) - 1$ .

La troisième chose, c'est que la borne supérieure est égale à la somme des valeurs absolues des deux bornes du type de colonne correspondant sans le UNSIGNED (pour ceux qui ne savent pas encore ce qu'est la valeur absolue, la valeur absolue de -3 est égale à 3, la valeur absolue de -4 est égale à 4, la valeur absolue de 5 est égale à 5). Pour le TINYINT UNSIGNED, la borne supérieure est égale à  $255 = |-128| + |127| = 128 + 127$ .

Mine de rien, c'est déjà pas mal. 😊

Mais si on avance bien, on n'en a pas encore tout à fait fini avec ces entiers. En plus d'avoir cinq types de colonnes, on peut en plus préciser le nombre de chiffres que le nombre qu'on veut stocker peut comporter. Relisez cette phrase une ou deux fois. 😊

Juste au cas où, je fais un petit écart pour rappeler la différence entre un chiffre et un nombre. Un chiffre est un symbole qui permet de représenter une valeur numérique. Dans le système décimal, il n'y a que dix chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Un nombre est un concept qui représente une unité, une collection d'unités ou une fraction d'unité. 65 est un nombre, par exemple.

Un nombre est écrit avec des chiffres, mais un chiffre n'est pas écrit avec des nombres.

Donc, revenons à nos colonnes. Je vous ai dit qu'on peut indiquer à MySQL le nombre de chiffres qui composent un nombre. Vous comprendrez l'intérêt dans quelques lignes : voici comment indiquer le nombre de chiffres qui composent un nombre.

#### Code : SQL

```
nom_col SMALLINT UNSIGNED
```

On va mettre :

#### Code : SQL

```
nom_col SMALLINT(3) UNSIGNED
```

Si vous tentez de stocker le nombre 9999, c'est ce nombre qui sera stocké. Mais alors, quel intérêt ?

L'intérêt, c'est le ZEROFILL. Imaginons que vous vouliez insérer des nombres avec des zéros devant, 004 par exemple. Si vous utilisez ce que je vous ai appris jusqu'à maintenant, vous n'arriverez jamais à récupérer autre chose que 4 bien que vous tentiez de stocker 004.

Pour arriver à faire cela, MySQL a prévu une autre option, le ZEROFILL. Si vous utilisez le ZEROFILL, MySQL ajoutera des 0 devant le nombre jusqu'à obtenir le nombre de chiffres qui peuvent composer les nombres stockés dans la colonne. Par exemple, je veux stocker le nombre 0004.

Si je fais cette colonne :

#### Code : SQL

```
nom_col SMALLINT
```

je ne récupérerai jamais rien d'autre que 4.

Avec le ZEROFILL, ça ira mieux :

#### Code : SQL

```
nom_col SMALLINT ZEROFILL
```

Aucun souci, j'aurai mon 0004, mais en fait, j'aurai mieux : 000004. Oui, le nombre de chiffres que peut contenir un SMALLINT est six, MySQL rajoute donc cinq zéros.

Pour obtenir mon 0004, je devrai faire ceci :

#### Code : SQL

```
nom_col SMALLINT(4) ZEROFILL
```

Et magie, voilà le 0004 grâce à ce qu'on a vu un peu plus tôt.

Vous pouvez bien sûr utiliser ça avec les UNSIGNED :

#### Code : SQL

```
nom_col SMALLINT(4) UNSIGNED ZEROFILL
```

Là je vous invite sérieusement à faire une pause parce que mine de rien, vous venez de voir plusieurs notions.

### Des types en pagaille

Bon, si vous êtes frais et dispos, on va pouvoir passer à tous les autres types, et il en reste !

Le prochain est encore un type qu'on utilise pour stocker des nombres, mais cette fois, c'est pour stocker des nombres à virgule. Voici comment créer une colonne de ce type :

#### Code : SQL

```
nom_col DECIMAL(5,2) [UNSIGNED] [ZEROFILL]
```

Comme vous le voyez, j'ai mis le UNSIGNED et le ZEROFILL entre crochets, ils sont facultatifs et je n'expliquerai pas leur rôle puisque c'est du déjà vu.

La seule nouveauté, c'est DECIMAL(5,2).

Pour les nombres entiers, on pouvait dire à MySQL combien de chiffres au maximum on voulait stocker dans une colonne ; on peut faire pareil avec les décimaux. Mais en plus de cela, on peut également indiquer à MySQL combien de chiffres après la virgule on veut garder.

Dans mon exemple, je crée une colonne de type DECIMAL qui peut contenir des nombres à cinq chiffres, avec deux chiffres après la virgule.

Ainsi, si je tente de stocker 999999.99999, MySQL stockera 999.99. Si je tente de stocker 55.5555555, MySQL stockera 55.56 (quand MySQL tronque un nombre, il l'arrondit au passage).

À présent, passons à autre chose avec des colonnes qui nous permettront de stocker des dates, des heures, etc.

On a déjà vu un type de colonne pour stocker une date, le type DATETIME. Il permet de stocker des dates avec ce format : AAAA-MM-JJ HH:MM:SS. Ce n'est pas un format français, c'est un format anglais (là-bas on indique l'année, puis le mois, puis le jour, puis les heures, puis les minutes, puis les secondes) mais ne vous en faites pas, on verra comment changer ce format. Il existe d'autres types de colonnes pour les dates. Imaginons que vous n'ayez besoin que de l'année, du mois et du jour. Ça serait assez inutile de stocker l'heure puisqu'elle n'est pas souhaitée. On va donc utiliser un autre type, DATE. Ce type de colonnes permet de stocker une date au format AAAA-JJ-MM.

Il existe encore d'autres types de colonnes, le type TIMESTAMP par exemple. C'est un type assez proche du type DATETIME mais quelques différences subsistent.

Tout comme avec les nombres, les dates que vous pouvez stocker sont limitées par deux bornes. Pour une colonne de type DATETIME, vous pouvez stocker des dates comprises entre 1000-01-01 00:00:00 et 9999-12-31 23:59:59, donc de minuit le 1 janvier de l'an 1000 à 23 heures 59 minutes 59 secondes le 31 décembre 9999, on a de la marge. 🍬

Pour le type DATE, vous pouvez stocker des dates comprises entre 1000-01-01 et 9999-12-31, et pour le type TIMESTAMP, ça va du 1<sup>er</sup> janvier 1970 jusqu'à approximativement l'année 2037.

Passons maintenant au quatrième type de colonne temporelle, le type TIME. Ce type peut poser des problèmes de compréhension si on ne fait pas attention, on va voir pourquoi.

On a vu le type DATE et le type DATETIME, on a vu que le type DATE est une partie du type DATETIME, on peut donc supposer que TIME est la partie manquante.

Et ce n'est pas faux, mais c'est incomplet. En plus de permettre de stocker une heure au format HH:MM:SS, le type TIME convient parfaitement pour stocker une durée, par exemple, 106 heures 23 minutes 43 secondes.

Imaginons que vous fassiez une table destinée à des durées de la traversée de la France à cheval (c'est un exemple 😊), vous aurez peut-être 36 heures 20 minutes, mais vous pouvez aussi avoir 300 heures 56 minutes 06 secondes. Pour stocker cette information, le meilleur type de colonne est TIME.

Mais ce n'est pas tout ! On peut également stocker une durée négative. Je n'ai pas d'exemple d'utilisation sous la main, mais c'est pas grave.

Vous pouvez stocker des durées / heures comprises entre -838:59:59 et 838:59:59, donc de -838 heures 59 minutes 59 secondes à 838 heures 59 minutes 59 secondes.

Finissons-en avec les dates avec le cinquième et dernier type de colonne temporelle, le type YEAR. Ce type de colonne permet de stocker des dates au format YYYY entre 1901 et 2155 (mais ne me demandez pas pourquoi 🤪).

Sans plus attendre, parlons des colonnes servant à stocker des chaînes de caractères. On a déjà vu un de ces types : le VARCHAR. Cette colonne sert à stocker des chaînes avec au maximum 255 caractères, mais on a vu qu'on pouvait diminuer cette limite. Par exemple, si au lieu de 255 caractères :

#### Code : SQL

```
nom_col VARCHAR(255)
```

je n'en veux que 108 :

#### Code : SQL

```
nom_col VARCHAR(108)
```

Il existe un autre type très proche du VARCHAR, le type CHAR. Pour vous, les deux types sont strictement identiques (il faut aussi spécifier le nombre maximal de caractères qu'on pourra y stocker, la limite maximale étant 255), la seule différence réside dans la façon dont MySQL les stockent. Je vous conseille d'utiliser le VARCHAR, il est en général plus économique en terme de place sur le disque dur (le seul cas où il est plus lourd, c'est quand vous stockez des chaînes qui ont le nombre maximal de caractères autorisés : donc, si vous êtes certains que la majorité des données de la colonne aura le nombre maximal de caractères autorisés, utiliser un CHAR est plus économique).

Après les textes courts, on va passer aux textes plus longs avec le type TEXT et ses trois frères. Pourquoi *trois frères* ? C'est tout simplement que le type TEXT, tout comme le type INT, possède plusieurs types identiques, la seule différence étant le nombre maximal de caractères qu'on pourra y stocker. Ces trois frères sont : TINYTEXT, MEDIUMTEXT et LONGTEXT. Pour créer ce genre de colonnes, c'est encore et toujours très simple : vous n'avez même pas à indiquer de limite pour le nombre de caractères comme on devait le faire pour le VARCHAR. Voici quatre colonnes :

#### Code : SQL

```
nom_col_1 TINYTEXT,  
nom_col_2 TEXT,  
nom_col_3 MEDIUMTEXT,  
nom_col_4 LONGTEXT
```

Et voici un tableau reprenant la limite de caractères :

Type de colonne	Nombre maximal de caractères
TINYTEXT	256
TEXT	65 536
MEDIUMTEXT	16 777 216
LONGTEXT	4 294 967 296

Vous l'avez peut-être remarqué, encore une fois, les limites sont des puissances de 2 ( $2^8$ ,  $2^{16}$ ,  $2^{24}$  et  $2^{32}$ ). En fait il y a énormément de mathématiques qui se cachent derrière les bases de données, mais ce n'est pas le sujet. 😊

Le type suivant est un peu particulier, il sert à stocker tout ce que l'on veut sous forme binaire. Vous pourriez par exemple l'utiliser pour stocker une image, mais en pratique, on ne le fera pas. Je ne vous montrerai sans doute aucun exemple dans lequel ce type de colonnes sera nécessaire. Ce type si particulier est BLOB. Et tout comme le type TEXT, il a trois frères : TINYBLOB, MEDIUMBLOB et LONGBLOB.

Là encore, pas besoin de spécifier une taille :

#### Code : SQL

```
nom_col_1 TINYBLOB,  
nom_col_2 BLOB,  
nom_col_3 MEDIUMBLOB,  
nom_col_4 LONGBLOB
```

La limite de caractères de ces colonnes est la même que leur équivalent en TEXT.

Courage, il n'y a plus que deux types ! 😊

Le type qui nous intéresse maintenant va permettre d'insérer des données qui appartiennent à une liste. Reprenons l'exemple de la news validée ou non, les deux seules données possibles sont *validée* et *non validée*. Avec ce type, si on tente d'insérer une autre donnée qu'une de ces deux-là, MySQL stockera une chaîne vide. Ce type si particulier est ENUM.

Pour créer une colonne de ce type, voici comment il faut faire :

#### Code : SQL

```
nom_col ENUM('valeur_1', 'valeur_2')
```

Vous pouvez mettre  $65\ 535$  ( $= (2^{16})-1$ , quand je vous disais que les mathématiques sont partout) éléments possibles dans un ENUM. Et le -1 dans  $(2^{16})-1$  s'explique tout simplement par le fait que si vous tentez de stocker une valeur non-autorisée, MySQL stockera une chaîne vide (en gros, si vous spécifiez trois valeurs possibles, ça revient à en spécifier quatre car MySQL peut stocker une chaîne vide). En conséquence de ça, il est bien sûr totalement inutile de permettre de stocker une chaîne vide, vu que MySQL le fait déjà quand il faut.

Le dernier type, SET, est un peu plus complexe que les autres, je n'en parlerai donc pas. Et de toute façon, il est assez rare qu'on s'en serve.

Petit cadeau, une liste de tous les types de colonnes avec leurs options !

- TINYINT[(nombre\_de\_chiffres)] [UNSIGNED] [ZEROFILL]
- SMALLINT[(nombre\_de\_chiffres)] [UNSIGNED] [ZEROFILL]
- MEDIUMINT[(nombre\_de\_chiffres)] [UNSIGNED] [ZEROFILL]
- INT[(nombre\_de\_chiffres)] [UNSIGNED] [ZEROFILL]
- INTEGER[(nombre\_de\_chiffres)] [UNSIGNED] [ZEROFILL]
- BIGINT[(nombre\_de\_chiffres)] [UNSIGNED] [ZEROFILL]
- DECIMAL(nombre\_de\_chiffres, nombre\_de\_chiffres\_après\_la\_virgule) [UNSIGNED] [ZEROFILL]
- DATE
- TIME
- TIMESTAMP
- DATETIME
- CHAR(nombre\_de\_caractères)
- VARCHAR(nombre\_de\_caractères)
- TINYBLOB
- BLOB
- MEDIUMBLOB
- LONGBLOB
- TINYTEXT
- TEXT
- MEDIUMTEXT
- LONGTEXT
- ENUM('valeur\_1', 'valeur\_2', 'valeur\_3', ...)

Avec tout ça, celui qui ose choisir un mauvais type de colonne, je le mords. 😊



Il y a quelques autres types qui sont des alias (des noms différents pour un même type de colonnes), et une ou deux options que je n'ai pas mentionnées, mais ça viendra plus tard.

## Et la gestion ?

C'est bien joli tout ça, mais maintenant, on va mettre en pratique toute cette théorie pour gérer un peu nos tables. Pour le moment, vous savez créer une table, mais une fois créée, c'est fini, vous êtes cuits.

La première chose qu'on va faire, c'est apprendre à supprimer une table.



**Faites très attention :** tout comme quand vous supprimez une base, si vous supprimez une table, vous supprimez aussi toutes les données que la table contient.

Supprimer une table relève du jeu d'enfant quand on sait supprimer une base de données ; on avait :

### Code : SQL

```
DROP DATABASE nom_base;
```

Et maintenant on a :

### Code : SQL

```
DROP TABLE nom_table;
```

Testons donc ceci. On va d'abord créer une table, puis la supprimer. Et on affichera la liste des tables à chaque fois pour voir ce qui se passe.

### Code : SQL

```
USE dev_dev;
SHOW TABLES;
CREATE TABLE test (test TEXT);
SHOW TABLES;
DROP TABLE test;
SHOW TABLES;
```

En regardant attentivement ce que nous renvoie le SHOW TABLES;, on voit bien que la table est créée puis supprimée.

Après la suppression d'une table, on va voir comment modifier une table. Je vais vous apprendre à faire quatre choses : supprimer une colonne, ajouter une colonne, changer une colonne et enfin changer le type d'une colonne.

Bon, fini le blabla, on va commencer à jouer. On va travailler sur la table **dev\_gbook** que je vous ai fait créer dans le chapitre précédent. En réalité, j'y ai volontairement glissé une petite erreur. Je vous avais dit qu'on transformera l'IP en entier grâce à une fonction PHP, seulement, cette fonction peut retourner un nombre négatif. Or, la colonne qui stockera l'IP, **gbook\_ip**, est de type INT UNSIGNED. On ne peut donc pas y stocker de nombre négatif. On va donc faire joujou avec cette colonne pour illustrer la gestion d'une table.

On va commencer par quelque chose d'assez simple, on va supprimer cette colonne.  
Voici une requête générique pour supprimer une colonne quelconque :

### Code : SQL

```
ALTER TABLE nom_table DROP COLUMN nom_col;
```

Si je l'applique à l'exemple qui nous intéresse, ça donne :

**Code : SQL**

```
ALTER TABLE dev_gbook DROP COLUMN gbook_ip;
```

Testez cette requête et faites un :

**Code : SQL**

```
SHOW COLUMNS FROM dev_gbook;
```

Vous voyez que la colonne **gbook\_ip** a bien été supprimée. 😊



**Faites très attention** : si vous supprimez une colonne, vous supprimez toutes les données contenues dans la colonne.

Maintenant qu'on a supprimé cette colonne, on a un gros problème : on ne peut plus stocker l'IP de la personne qui a posté le message. On va donc recréer cette colonne.

Créer une colonne est assez simple en soi pour autant qu'on sache manipuler les types de colonnes (vous avez intérêt à le savoir après tout ce blabla là-dessus 😊) et leurs options.

Voici la requête générique qui ajoute une colonne :

**Code : SQL**

```
ALTER TABLE nom_table ADD COLUMN nom_col type_col [NULL | NOT NULL]
[DEFAULT valeur_par_defaut] [AUTO_INCREMENT] [[PRIMARY] KEY]
[COMMENT 'commentaire'] [FIRST | AFTER nom_col_prec];
```

Ça fait peur, mais on a déjà vu une grande partie de ce monstre ; d'ailleurs, je vais enlever les éléments qu'on a déjà vus pour simplifier tout ça. Je vais remplacer tout ceci par **option\_col** :

**Code : SQL**

```
[NULL | NOT NULL] [DEFAULT valeur_par_defaut] [AUTO_INCREMENT]
[[PRIMARY] KEY] [COMMENT 'commentaire'];
```

Ce qui me donne :

**Code : SQL**

```
ALTER TABLE nom_table ADD COLUMN nom_col type_col option_col [FIRST
| AFTER nom_col_prec];
```

Mais on va présenter ça autrement pour que ça soit plus agréable :

**Code : SQL**

```
ALTER TABLE nom_table
ADD COLUMN nom_col type_col option_col
[FIRST | AFTER nom_col_prec];
```

Je vous ai dit qu'en PHP, il était de mise de ne mettre qu'une seule instruction par ligne. En SQL, il arrive qu'on ait des requêtes tout simplement monstrueuses avec des tonnes de trucs. Dans ce genre de cas, tout taper sur une ligne est assez rédhibitoire, et ça rend la requête difficile à lire.

C'est pour ça que j'ai pris l'habitude de mettre des retours à la ligne et des indentations même dans mes requêtes. Je ne sais pas pour vous, mais je préfère de loin ma dernière requête à la précédente, elle est bien plus aérée et agréable à lire. Et bien entendu, ces trois lignes ne forment qu'une et une seule requête, étant donné qu'il n'y a qu'un point-virgule (qui sert, je vous le rappelle, à marquer la fin d'une requête). Au fil du temps, vous commencerez à sentir quand il faut mettre des retours à la ligne ou des indentations sans même regarder votre requête.

Bref, on en était à la création d'une colonne. L'ALTER TABLE nom\_table ne vous est pas inconnu, ça indique qu'on modifie (*alter*) une table (*table*).

Les mots-clés ADD COLUMN (**ajouter colonne** en français) indiquent qu'on veut ajouter une colonne. **nom\_col** est le nom de la future colonne, **type\_col** est son type et **option\_col** représente les options (NULL, NOT NULL, etc.).

La nouveauté, c'est ce qu'il y a entre les crochets de la troisième ligne. Imaginons que vous soyez chargés d'organiser l'implantation des produits dans un rayon d'un super-marché : comment allez-vous savoir où placer les salades par rapport aux fruits, les fruits par rapport aux chocolats, etc. ?

On va devoir vous le dire, vous donner un ordre à respecter. Pour les colonnes d'une table, c'est pareil. On les crée dans un ordre particulier (l'ordre dans lequel on les définit quand on crée la table), et cet ordre ne change pas. Si on veut ajouter une colonne, il faut dire à MySQL où ranger cette colonne.

C'est à ça que sert ce qu'il y a entre ces crochets. Mais comme vous le voyez, c'est entre crochets, ça veut donc dire que c'est facultatif. En effet, reprenons l'exemple des rayons, on pourrait dire à la personne qui range : si tu ne sais pas où ça va, mets-le à la fin du rayon.

Eh bien c'est pareil avec MySQL : si vous ne lui dites pas où ranger cette nouvelle colonne, il la mettra à la suite des autres, à la fin, en dernière position.

Maintenant, regardons les deux possibilités qui se cachent dans ses crochets : on a deux mots-clés. Le premier est FIRST : si vous mettez ce mot-clé, MySQL ira mettre cette colonne en première position, ce qui est assez logique si on sait qu'en français, FIRST veut dire *premier*.

Maintenant on a le mot-clé AFTER, qui veut dire *après* en français. Pour positionner notre colonne par rapport aux autres, on va utiliser les colonnes déjà existantes. On va par exemple lui dire de rajouter la colonne après la colonne x ou y. **nom\_col\_prec** est le nom de la colonne qui sera utilisée comme repère.

Reprenons l'exemple de la table **dev\_gbook**, on va recréer la colonne **gbook\_ip** à la même place qu'elle occupait avant qu'on la supprime, c'est-à-dire après la colonne **gbook\_id** et avant la colonne **gbook\_msg**.

On devra donc utiliser le mot-clé AFTER, et la colonne **gbook\_id** sera le repère puisque c'est après cette colonne qu'on veut placer la nouvelle.

Le type de la nouvelle colonne sera INT(10). Et zoup, mini-TP, à vous de créer cette requête. Voici ce que vous devez obtenir :

#### Secret (cliquez pour afficher)

##### Code : SQL

```
ALTER TABLE dev_gbook
ADD COLUMN gbook_ip INT(10)
AFTER gbook_id;
```

Avec un SHOW COLUMNS FROM dev\_gbook;, vous pourrez vérifier si la colonne a bien été créée et si elle est au bon endroit.

Maintenant que notre colonne n'est plus en UNSIGNED, tout va bien. Mais manque de chance, je suis vraiment maladroit et j'ai totalement oublié de dire à MySQL que la colonne ne peut pas contenir de NULL.

Mais c'est très bien en fait, je vais vous montrer comment changer le type d'une colonne. 😊

Voici la requête générique :

##### Code : SQL

```
ALTER TABLE nom_table
MODIFY COLUMN nom_col type_col option_col
[FIRST | AFTER nom_col_prec]
```

Cette requête ressemble énormément à la requête qui ajoute une colonne, et c'est assez normal. La seule différence est que ADD

COLUMN est remplacé par MODIFY COLUMN (*modifier colonne* en français).

Maintenant, à vous de jouer, vous allez modifier la colonne **gbook\_ip** (pour le moment de type INT(10) et qui a l'option NULL) pour lui donner le type INT(10) et l'option NOT NULL. C'est relativement simple si vous avez compris ce que j'ai dit, mais c'est toujours intéressant d'essayer par soi-même.

#### Secret (cliquez pour afficher)

##### Code : SQL

```
ALTER TABLE dev_gbook
MODIFY COLUMN gbook_ip INT(10) NOT NULL
AFTER gbook_id;
```

Affichez à nouveau la liste des colonnes, et on voit bien que la colonne **gbook\_ip** ne peut plus stocker de NULL.

Courage, plus qu'une seule sorte de requête et c'est fini. 😊

C'est à peu près la même chose que la requête précédente, sauf que là on peut également changer le nom de la colonne. Voici la requête générique :

##### Code : SQL

```
ALTER TABLE nom_table
CHANGE COLUMN nom_col nom_col_nouveau type_col option_col
[FIRST | AFTER nom_col_prec]
```

Il n'y a que deux différences avec la requête précédente, la première étant le CHANGE COLUMN à la place de MODIFY COLUMN.

La seconde, c'est l'apparition de **nom\_col\_nouveau** entre le nom de colonne qu'on traite, le nouveau type et les nouvelles options. **nom\_col\_nouveau** est le nom qu'aura la nouvelle colonne (si le nouveau nom est le même que l'ancien, ça revient à utiliser MODIFY COLUMN). Le CHANGE COLUMN permet de renommer simplement une colonne.

On va prendre un exemple inutile : on va transformer la colonne **gbook\_ip** en une nouvelle colonne qui aura le même nom, le même type et les mêmes options. À vous de jouer !

#### Secret (cliquez pour afficher)

##### Code : SQL

```
ALTER TABLE dev_gbook
CHANGE COLUMN gbook_ip gbook_ip INT(10) NOT NULL
AFTER gbook_id;
```

Mais tant qu'on en est à parler de renommage, je vais prendre deux ou trois lignes pour vous montrer comment on peut renommer une table (ça peut servir). La requête générique :

##### Code : SQL

```
ALTER TABLE nom_table
RENAME TO nom_nouvelle_table
```

Et un exemple pour créer une table **dev\_test**, la renommer en **dev\_trash** et enfin la supprimer :

##### Code : SQL

```
SHOW TABLES;
CREATE TABLE dev_test (test TEXT);
```

```
SHOW TABLES;
ALTER TABLE dev_test
RENAME TO dev_trash;
SHOW TABLES;
DROP TABLE dev_trash;
SHOW TABLES;
```

Et voilà, on en a vraiment fini avec la gestion des tables (enfin, pour le moment du moins...) ! 😊

Alors, qu'en dites-vous ? 🎉

Ça commence à devenir intéressant ; dès que je vous aurai appris à insérer des données, les mettre à jour, les supprimer et les sélectionner, on parlera d'un couple qui a du chien : le PHP et MySQL.

Et une fois que vous serez capables d'utiliser ces deux tourtereaux ensemble, on passera aux choses sérieuses avec MySQL. 😊

À nouveau, pas de QCM pour les mêmes raisons que le chapitre précédent. De ce chapitre, vous devez surtout retenir les différents types de colonnes possibles. Bien évidemment, il est loin d'être indispensable de savoir que la limite d'un BIGINT UNSIGNED est 18 446 744 073 709 551 615 ou  $2^{64}-1$ . 😊 Retenez l'essentiel de chacun des types, et ce sera déjà très bien !

Pour la gestion des tables, nous n'avons vu que quatre ou cinq actions possibles, mais cette fois vous devez tout retenir : syntaxe et utilité.

Il est très important de retenir ceci car si vous êtes incapables d'administrer et de gérer vos bases de données, vous n'irez pas bien loin.



## SELECT, ou comment récupérer des données

Après ce (long) chapitre sur la gestion des tables, on va enfin commencer à jouer avec ce que vous utiliserez le plus souvent : les requêtes SELECT.

Elles vous permettent de sélectionner des données venant d'une table ou plusieurs tables, mais pas seulement ! Ce chapitre parlera aussi des chaînes de caractères en SQL, des types, etc.

### MySQL et les types

Dans le chapitre précédent, on a vu que chaque type de données devait être stocké dans le type de colonnes adéquat, mais en réalité, les types de colonnes, pour nous, utilisateurs, n'ont pas grand intérêt.

L'intérêt de ces différents types réside dans les optimisations de stockage que MySQL peut faire.

En SQL, on peut considérer qu'il y a trois grands types de données pour nous utilisateurs : les nombres, les chaînes de caractères et le NULL.

On a déjà parlé de typage avec le PHP, et en SQL, c'est très similaire, mais quelques différences subsistent.

Je vais vous montrer ces différents types en action avec quelques requêtes très simples :

#### Code : SQL

```
SELECT 1;
```

Cette requête vous retourne un tableau avec une colonne (qui a pour nom **1**) et une ligne (qui a pour valeur **1**).

#### Code : Console

```
+---+
| 1 |
+---+
| 1 |
+---+
```

Maintenant, une autre requête qui va sélectionner un nombre à virgule :

#### Code : SQL

```
SELECT 4.2;
```

On obtient un tableau similaire, une ligne et une colonne, de valeur et de nom 4.2.

#### Code : Console

```
+-----+
| 4.2 |
+-----+
| 4.2 |
+-----+
```

Ces deux requêtes font la même chose, elles sélectionnent toutes deux une valeur constante (souvenez-vous, on avait la même chose en PHP).

Maintenant, on va sélectionner une valeur constante de type chaîne de caractères :

#### Code : SQL

```
SELECT 'MySQL est grand';
```

Ce qui nous donne le tableau suivant :

**Code : Console**

```
+-----+
| MySQL est grand |
+-----+
| MySQL est grand |
+-----+
```

On peut déjà tirer une conclusion : quand on sélectionne une valeur constante, MySQL nous retourne une colonne supplémentaire qui a pour nom la valeur de la valeur constante qu'on sélectionne.

Comme vous le voyez, j'ai utilisé des apostrophes ('') pour délimiter ma chaîne de caractères. En effet, en SQL, on délimite les chaînes de caractères avec des apostrophes et rien d'autre !

MySQL ne respecte pas toujours les normes, et c'est le cas pour les chaînes de caractères. MySQL permet l'utilisation de guillemets pour délimiter les chaînes de caractères, mais la norme SQL exige des apostrophes. C'est pourquoi vous devrez toujours utiliser les apostrophes à partir de maintenant.

En PHP, on a vu que pour insérer une apostrophe dans une chaîne de caractères délimitée par des apostrophes, il fallait échapper l'apostrophe en la précédant d'un *backslash*.

En SQL, la règle de l'échappement est toujours de mise, mais... pour échapper une apostrophe, au lieu de la précédé d'un *backslash*, on la précède d'une apostrophe.

Donc, au lieu de faire :

**Code : SQL**

```
SELECT 'MySQL c\'est trop fort !';
```

Vous devrez faire :

**Code : SQL**

```
SELECT 'MySQL c''est trop fort !';
```

Mais... ceux qui auront eu la bonne idée de tester les deux requêtes auront pu constater que les deux fonctionnent ! Eh oui, encore une fois, MySQL déroge à la norme et permet d'utiliser des *backslashes* pour échapper les apostrophes. Mais dorénavant, vous devrez vous plier à la norme et utiliser l'apostrophe.

Pour sélectionner un NULL, rien de plus simple, il suffit de faire ceci :

**Code : SQL**

```
SELECT NULL;
```

Et voilà, maintenant vous connaissez tous les types que MySQL peut vous retourner. Plutôt étrange, non ? Et pourtant, c'est véridique. Maintenant vous pouvez vous demander comment on fait pour indiquer à MySQL qu'on veut récupérer une date par exemple, et c'est en fait très simple :

**Code : SQL**

```
SELECT '2007-09-14';
```

Eh oui, une date en SQL, c'est une chaîne de caractères. C'est pour cela que je vous disais que les types de colonnes n'avaient aucun intérêt pour nous. Que vous utilisez une colonne de type VARCHAR ou DATETIME, à priori rien ne change. Mais pour MySQL, la différence est grande.

Une question devrait vous titiller les méninges depuis quelque temps ; on a vu un élément très important en PHP : les booléens. Ils permettent d'utiliser des conditions, des boucles, etc., mais qu'en est-il en SQL ? Eh bien, le mieux c'est encore d'essayer 😊.

**Code : SQL**

```
SELECT TRUE;
```

Et là, chose curieuse, on obtient ceci :

**Code : Console**

```
+-----+
| TRUE |
+-----+
|   1   |
+-----+
```

Comme vous le voyez, le nom de la colonne est TRUE comme on pouvait s'y attendre, mais la valeur est... 1. Eh oui, en SQL, il n'y a pas de valeur booléenne au sens propre, les booléens sont « remplacés » par des entiers !

Souvenez-vous de ce que j'ai dit sur le transtypage des entiers en booléens en PHP : tout entier est transtypé en TRUE sauf l'entier 0, qui est le seul à être transtypé en FALSE.

Et souvenez-vous également de ce que je vous ai dit sur le transtypage des booléens en entiers, TRUE donne l'entier 1, et FALSE donne l'entier 0.

C'est également le cas en SQL, et on le verra plus tard quand on abordera les opérateurs logiques.

Désormais, vous savez sélectionner une valeur constante avec un SELECT, mais que diriez-vous d'en sélectionner plusieurs à la fois ?

Pour sélectionner plusieurs valeurs constantes, il suffit de les séparer par une virgule :

**Code : SQL**

```
SELECT 1, 2, 3;
```

Mais imaginons que nous voulions faire quelque chose comme cela :

**Code : SQL**

```
SELECT 1, 1, 1;
```

Avouez que c'est plutôt difficile de s'y retrouver, étant donné que toutes les colonnes portent le même nom. Ce qui serait bien, c'est de pouvoir renommer ces colonnes. Essayons cela :

**Code : SQL**

```
SELECT 1 AS nb_pomme, 1 AS nb_poire, 1 AS nb_peche;
```

Ah, c'est quand même mieux, non ?

Dans cette requête, j'ai renommé explicitement ces colonnes, mais sachez qu'on peut également le faire d'une manière implicite :

**Code : SQL**

```
SELECT 1 nb_pomme, 2 nb_poire, 3 nb_peche;
```

Le résultat est strictement identique, mais la seconde requête est plus... risquée. Ne brandissez pas les boucliers, vous ne risquez pas de faire planter votre PC 😊.

Seulement, cette seconde requête est moins claire, un peu plus difficile à lire. C'est pour ça qu'à partir de maintenant, vous devrez toujours renommer vos colonnes explicitement avec le mot-clé AS.



Le nom de colonne que vous donnez avec un AS se plie bien évidemment aux mêmes règles de nommage qu'une colonne normale dans une table.

Et c'est après ce petit passage sur le renommage que je vais vous parler d'une erreur extrêmement courante. Imaginons que par inadvertance, j'oublie de délimiter ma chaîne de caractères avec des apostrophes :

#### Code : SQL

```
SELECT lala;
```

Voyons ce que MySQL me renvoie :

#### Code : Console

```
ERROR 1054 (42S22) : Unknown column 'lala' in 'field list'
```

Ce qui nous donne en français :

#### Code : Console

```
ERROR 1054 (42S22) : Colonne 'lala' inconnue dans 'liste des colonnes'
```

Et c'est là que vous devez être attentifs. En SQL, tout ce qui est délimité par des apostrophes est une chaîne de caractères. Mais toute chaîne de caractères (ou même tout caractère alphabétique) non-délimitée par des apostrophes est une colonne.

Quand j'envoie ma requête précédente à MySQL, il comprend que je lui demande toutes les valeurs contenues dans la colonne **lala**. Seulement, cette colonne n'existe pas, il me renvoie donc une erreur.

L'autre chose qui vous est inconnue, c'est le **field list**. En réalité, on verra un peu plus tard que quand on sélectionne des colonnes, ces dernières viennent toujours d'une ou de plusieurs tables.

Dans la requête, on spécifie bien entendu quelles tables on veut utiliser, et c'est là que le **field list** intervient.

Avant d'aller récupérer des valeurs, MySQL va aller chercher la liste des colonnes des tables qu'on utilise, et si jamais on cherche à utiliser une colonne qui n'existe pas, MySQL nous renvoie cette erreur.

Maintenant, pourquoi est-ce que je dis que cette erreur est extrêmement courante ? Eh bien, quand vous allez utiliser MySQL et le PHP, vous verrez que les requêtes sont des chaînes de caractères. Seulement, dans ces chaînes de caractères, il peut y avoir des chaînes de caractères propres à la requête. Par exemple :

#### Code : PHP

```
<?php  
$str = 'une requête SQL avec 'une chaîne'';  
?>
```

Si vous testez ce code, vous aurez bien sûr une erreur car j'ai totalement oublié d'échapper les apostrophes. Bien souvent, vous utiliserez également des variables, et très souvent la concaténation. Et c'est là que les erreurs surviennent : les codeurs ne réfléchissent pas, et ils oublient qu'en SQL, les chaînes de caractères doivent également être délimitées par des apostrophes. Ce qui fait qu'il faut d'avoir :

#### Code : SQL

```
SELECT 'lala';
```

ils obtiennent :

**Code : SQL**

```
SELECT lala;
```

Ce qui provoque l'erreur ci-dessus. Mais on reverra ça très bientôt quand on parlera de l'utilisation de MySQL avec le PHP.

Dernier petit mot avant la fin de cette partie : les opérateurs. On a vu que le PHP mettait à notre disposition une véritable armée d'opérateurs, c'est également le cas avec MySQL.

Ainsi, on retrouve des opérateurs bien connus comme les opérateurs arithmétiques : addition, soustraction, division, multiplication et modulo.

Mais MySQL met également un opérateur supplémentaire à notre disposition : la division entière.

Voici quelques exemples, d'abord l'addition :

**Code : SQL**

```
SELECT 1+1;
```

La soustraction :

**Code : SQL**

```
SELECT 1-1;
```

La multiplication :

**Code : SQL**

```
SELECT 4*2;
```

La division :

**Code : SQL**

```
SELECT 15/3;
```

L'exemple de la division tombe volontairement juste : en effet,  $15/3 = 5$ . En PHP,  $15/3$  nous retournerait un type entier, mais comme vous pouvez le constater, MySQL nous retourne un nombre à virgule ! On verra plus tard comment se débarrasser des virgules. Le modulo :

**Code : SQL**

```
SELECT 13%5, 13 MOD 5;
```

Comme vous le voyez, les deux valeurs sont identiques, et pour cause, il existe deux opérateurs pour le modulo : **MOD** et **%**. Vous devez savoir que le modulo retourne le reste de la division entière. Par exemple,  $17\%5 = 2$  car  $17 = 3*5+2$ . Mais il existe aussi un opérateur complémentaire, qui au lieu de retourner le reste de la division entière retourne la division entière arrondie à l'entier inférieur ; cet opérateur est **DIV** :

**Code : SQL**

```
SELECT 17 DIV 3;
```

Il est évident que l'opérateur de division (/) et l'opérateur de division entière ne retourneront pas souvent la même chose :

**Code : SQL**

```
SELECT 17/3, 17 DIV 3;
```

Vous devez donc bien réfléchir à ce que vous voulez obtenir et utiliser le bon opérateur.



Tout comme en PHP, il y a une priorité des opérateurs, et on peut utiliser des parenthèses pour forcer la priorité.

Et c'est maintenant qu'on va pouvoir utiliser la pleine puissance du renommage de colonnes 😊.

Allez, regardez ça :

**Code : SQL**

```
SELECT 1+1 AS addition, 1-1 AS soustraction, 4*2 AS multiplication,
15/3 AS division, 17%3 AS modulo_a, 17 MOD 3 AS modulo_b, 17 DIV 3
AS divi_entiere;
```

On peut bien sûr faire des calculs plus complexes en utilisant tous les opérateurs, des parenthèses, etc.

## Les données venant d'une table

Après ce petit moment de repos, on va pouvoir passer aux choses sérieuses : on va enfin travailler sur des données venant d'une table. Le seul petit souci, c'est que pour l'instant notre table est... vide.

Pas de souci, voici de quoi arranger la situation : exécutez simplement cette requête, je vous expliquerai rapidement ce qu'elle fait et comment elle le fait 😊 :

**Code : SQL**

```
INSERT INTO dev_gbook(gbook_id, gbook_ip, gbook_msg, gbook_pseudo,
gbook_date)
VALUES
(1, 1, 'Message n° 0', 'Albert', NOW() - INTERVAL 2 DAY),
(2, 173, 'Message n° 1', 'Arnold', NOW() - INTERVAL 3 DAY),
(3, 917, 'Message n° 2', 'L''andouille', NOW() - INTERVAL 9 DAY),
(4, 1303, 'Message n° 3', 'delphiki', NOW() - INTERVAL 0 DAY),
(5, 777, 'Message n° 4', 'Coca', NOW() - INTERVAL 8 DAY),
(6, 2101, 'Message n° 5', 'Corail', NOW() - INTERVAL 6 DAY),
(7, 2485, 'Message n° 6', 'Winnie', NOW() - INTERVAL 3 DAY),
(8, 99, 'Message n° 7', 'Tigrou', NOW() - INTERVAL 6 DAY),
(9, 577, 'Message n° 8', 'Harry', NOW() - INTERVAL 0 DAY),
(10, 73, 'Message n° 9', 'Zorro', NOW() - INTERVAL 3 DAY);
```

Si vous obtenez le message **10 rows in set** en réponse à cette requête, vous êtes dans le vrai et prêts pour la suite. Vous ne devriez normalement pas tomber sur un autre message si vous avez suivi le tutoriel depuis le début sans modifier trop de choses.

Zou, c'est parti. Notre but est de sélectionner toutes les colonnes de toutes les lignes contenues dans la table. Il y a peu, je vous ai dit que si on oubliait d'entourer une chaîne de caractères dans une requête, MySQL considérerait ça comme une colonne ; vous devez donc déjà avoir une petite idée de la requête qui nous intéresse, que voici d'ailleurs :

**Code : SQL**

```
SELECT gbook_id, gbook_ip, gbook_msg, gbook_pseudo, gbook_date
```

```
FROM dev_gbook;
```

Cette dernière vous retourne un joli tableau de cinq colonnes et de dix lignes ; la table n'ayant que cinq colonnes et dix lignes, on a bien toutes les colonnes de toutes les lignes !

Comme vous le voyez, j'ai écrit la requête sur deux lignes : c'est totalement volontaire. La première ligne ne vous est pas étrangère tant elle ressemble à ce qu'on a vu précédemment.

Il y a tout d'abord le mot-clé **SELECT** qui indique à MySQL qu'on veut récupérer des valeurs. Ce mot-clé est suivi des valeurs qu'on veut récupérer séparées par des virgules, c'est du déjà vu sauf que dans ce cas-ci, on récupère non pas des valeurs constantes mais les valeurs contenues dans les colonnes.

Sur la seconde ligne, un nouveau mot-clé fait son apparition dans ce contexte : **FROM**.

Dans une requête de type **SELECT**, le **FROM** sert à indiquer où MySQL doit aller chercher les données. Ici c'est dans une table, mais nous verrons plus tard qu'on peut aussi mettre autre chose dans le **FROM**.

Bref, après le **FROM**, on trouve **dev\_gbook** qui est le nom d'une table. Ainsi, cette requête peut se traduire en français par : *sélectionner les valeurs des colonnes [...] dans la table dev\_gbook.*

C'est assez simple, mais on va quand même jouer un peu avec ça car c'est la base de tout. Vous allez m'écrire une requête qui ne va me renvoyer que la liste des pseudos et la date des messages, c'est assez simple à faire.

#### Secret (cliquez pour afficher)

##### Code : SQL

```
SELECT gbook_pseudo, gbook_date  
FROM dev_gbook;
```

J'espère pour vous que vous avez réussi, sinon je vous conseille de bien relire tout ce chapitre.

Il est à noter que l'ordre dans lequel vous nommez les colonnes n'a pas d'importance (du moins pour l'instant 😊), ainsi on aurait très bien pu écrire :

##### Code : SQL

```
SELECT gbook_date, gbook_pseudo  
FROM dev_gbook;
```

Les données retournées par la requête sont identiques à celles retournées par la précédente.

Maintenant, et si on voulait sélectionner une donnée qui ne vient pas d'une colonne, comment feriez-vous ? Eh bien c'est simple, mais on va revenir sur une notion qui nous avait bien occupés quand on parlait du PHP : les expressions (eh oui, je vous avais dit que ça vous poursuivrait jusqu'au bout du monde !).

En réalité, voici comment on pourrait écrire *simplement* une requête de type **SELECT** :

##### Code : SQL

```
SELECT Expr[, Expr[, ...]]  
FROM t_table;
```

Quand j'écrirai des requêtes, j'utiliserais quelques raccourcis pour me faciliter l'écriture :

- **t\_table** désigne le nom d'une table quelconque ;
- **t\_col** le nom d'une colonne quelconque ;
- **Expr** une expression quelconque ;
- ce qui est entre crochets (**[]**) est facultatif.

On voit donc qu'en réalité, MySQL se fiche de la différence entre une colonne, une valeur constante et je ne sais quoi d'autre. La seule chose que MySQL nous renvoie, c'est la valeur des expressions qu'on demande.

C'est d'ailleurs pour cela que si on fait :

**Code : SQL**

```
SELECT 1+1;
```

MySQL ne nous retourne qu'une colonne, car comme on l'a déjà vu, `1+1` est une expression. MySQL l'évalue donc (ici, ça se résume à faire une addition) et nous retourne la valeur. Eh bien quand on joue avec des colonnes, c'est pareil. Si on fait :

**Code : SQL**

```
SELECT t_col
FROM t_table;
```

MySQL nous retourne la valeur de l'expression `t_col`, MySQL évalue donc cette expression et comme c'est un nom de colonne, il nous renvoie la valeur de la colonne.

Et donc, il est très facile de sélectionner en même temps des valeurs constantes et des colonnes, par exemple :

**Code : SQL**

```
SELECT 8, gbook_date
FROM dev_gbook;
```

Comme vous le voyez, MySQL nous retourne deux colonnes et dix lignes. Comme le **8** est une valeur constante, on le retrouve à chacune des lignes.

Il est bien évidemment possible de faire des opérations sur les colonnes étant donné que ce sont des expressions, on peut ainsi additionner des colonnes entre elles, additionner une colonne et une constante, etc.

Par exemple :

**Code : SQL**

```
SELECT gbook_id + 100
FROM dev_gbook;
```

Mais regardez un peu le nom de la colonne que MySQL nous retourne : ce n'est pas très élégant. Heureusement, on a vu quelque chose de très pratique dans ce genre de cas : le renommage !

On écrira donc par exemple :

**Code : SQL**

```
SELECT (gbook_id + 100) AS t_col
FROM dev_gbook;
```

Et zou, un joli nom de colonne facile à lire et à écrire 😊. Mais on peut encore faire mieux :

**Code : SQL**

```
SELECT (gbook_id + 100) AS gbook_id
FROM dev_gbook;
```

Comme vous le voyez, on peut renommer une expression avec le nom d'une colonne qui existe déjà ; seulement, cela peut éventuellement poser problème par la suite, je vous le déconseille donc pour le moment 😊.

Vous verrez sans doute des requêtes de la forme suivante :

**Code : SQL**

```
SELECT *
FROM t_table;
```

L'étoile (\*) est un raccourci pour dire **toutes les colonnes**, ça évite de devoir toutes les nommer. Seulement, je vous le dis tout net : ne l'utilisez jamais !



Il y a deux raisons à cela : la première, c'est qu'il est rare, très rare d'avoir besoin de toutes les colonnes d'une table ; pour être honnête, ça ne m'est jamais arrivé dans un code que j'utilise pour autre chose que des exemples. Maintenant, vous vous demandez peut-être pourquoi c'est mal de sélectionner des colonnes que l'on utilise pas : eh bien, c'est assez simple. Imaginez que vous êtes MySQL. Je vous demande toutes les colonnes, vous allez vous embêter à aller chercher dans vos tiroirs, puis vous allez m'envoyer tout ça, par la poste par exemple. Pour vous ça fait un travail de recherche en plus et pour la poste, ça fait quelques kilos en plus à m'apporter 😊.

Tout ça juste pour vous dire que si vous sélectionnez des données inutiles, vous perdez du temps et donc votre performance chute.

La seconde raison est la lisibilité. En effet, si vous avez bien nommé vos colonnes, avec un nom explicite, le simple fait de lire votre requête dira au lecteur quelles informations vous sélectionnez, tandis que si vous utilisez l'étoile, le lecteur sait que vous sélectionnez tout, mais qu'est-ce que le « tout » ? À moins d'avoir la structure de la table sous les yeux, le lecteur ne saura pas du tout quelles informations s'y trouvent.

## Tri, ordre et limitation

Je vous l'accorde, le titre de cette partie n'est pas très parlant mais vous allez vite comprendre pourquoi il est excellent 😊.

Cette partie va aborder trois clauses, trois outils qui sont une partie de la puissance des SGBDR. Avec ces trois outils, on va pouvoir filtrer des résultats, les ordonner et limiter leur nombre.

Jusqu'à présent, on a juste sélectionné des données, mais on n'avait aucun contrôle. Le nombre de lignes que MySQL nous rentrait était forcément le nombre de lignes contenues dans la table : on ne pouvait pas ordonner ces lignes, elles arrivaient dans un ordre précis sans qu'on sache pourquoi et enfin, on ne pouvait pas les filtrer, on ne pouvait pas dire à MySQL « donne-moi les lignes telles que... ».

Parmi ces trois clauses, il y en a une assez simple, celle qui permet d'ordonner les lignes ; une autre un peu plus complexe, celle qui permet de filtrer des résultats ; et la dernière, celle qui permet de limiter le nombre de résultats, est un peu particulière. En effet, elle existe et nous simplifie considérablement bien la tâche, mais elle peut également poser de **gros** problèmes de performance, c'est pourquoi il faut bien la comprendre pour l'utiliser avec parcimonie.

Les deux outils un peu plus complexes, le filtrage et la limitation, seront juste abordés pour le moment, on en reparlera beaucoup (surtout du filtrage en réalité) par la suite. Mais l'outil d'ordonnancement n'aura normalement plus de secret pour vous à la fin du chapitre ; on va d'ailleurs commencer par parler de lui !

## Mettons de l'ordre

Avant de vous montrer comment ordonner vos résultats selon votre bon vouloir, on va revenir quelques instants sur la notion d'ordre dans les SGBDR.

La première chose à savoir, c'est que pour un SGBDR, **la notion d'ordre n'existe tout simplement pas** ! Ce qui veut dire que quand vous ajoutez des données dans une table, votre SGBDR ne les ordonne pas, il les met les unes à la suite des autres comme elles viennent, le SGBDR se moque totalement de l'ordre.

Le seul moment où le SGBDR possède une vague notion d'ordre, c'est quand vous lui demandez de vous donner des lignes. À ce moment-là, on peut dire au SGBDR par rapport à quoi et dans quel ordre on veut que les lignes soient renvoyées.

Zoup, rentrons dans le vif du sujet. Pour sélectionner toutes les valeurs de la colonne **gbook\_id**, on utilisera la requête suivante :

**Code : SQL**

```
SELECT gbook_id
FROM dev_gbook;
```

Vous obtenez alors un tableau avec une colonne et dix lignes. Les lignes contiennent, en allant du haut vers le bas : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Vous constatez qu'il y a malgré tout un ordre, les nombres sont retournés de façon croissante (du plus petit au plus grand). Maintenant, je voudrais avoir l'inverse, c'est-à-dire que je veux avoir les mêmes nombres, mais retournés dans l'ordre décroissant (du plus grand au plus petit). Pour cela, exécutez donc cette requête et admirez le résultat :

**Code : SQL**

```
SELECT gbook_id
FROM dev_gbook
ORDER BY gbook_id DESC;
```

MySQL nous renvoie bien ce qu'on voulait ! Maintenant, analysons un peu cette requête. Les deux premières lignes ne devraient pas vous poser problème, on vient de voir quel était leur rôle. La seule nouveauté, c'est ce **ORDER BY**gbook\_id **DESC**.

Le **ORDER BY** est un mot-clé qui indique à MySQL qu'on va lui dire comment ordonner les résultats, ce qui suit le **ORDER BY** dira à MySQL comment les ordonner.

Le DESC est un autre mot-clé qui dit dans quel ordre MySQL doit retourner les résultats. Il n'y a que deux ordres possibles : croissant (ascendant, ASC) et décroissant (descendant, DESC).

Ceci vous indique quelque chose : on ne pourra ordonner que par des valeurs qui sont liées par un ordre de grandeur ! On peut donc sans problème ordonner par quelque chose de numérique, puisque si on a deux nombres, on peut dire que l'un est plus grand et l'autre plus petit. Maintenant si on veut ordonner par autre chose, une chaîne de caractères par exemple, c'est plus compliqué car il faut savoir comment MySQL va dire que telle chaîne est « plus grande » qu'une autre.

Mais s'il y a quelque chose de très important à dire, c'est qu'il est toujours préférable d'ordonner par des valeurs numériques. L'ordonnancement par des valeurs numériques est très rapide et léger, ordonner par un autre type de valeur est souvent moins rapide et synonyme de perte de performances : essayez donc d'ordonner le plus souvent par des valeurs numériques.

Reste ce qu'on trouve entre **ORDER BY** et DESC. Ce qui se trouve là est toujours une expression, ce qui veut dire que MySQL ordonne en comparant des... expressions !

Dans l'exemple qui nous occupe, l'expression en question est une colonne, *gbook\_id*, mais on aurait très bien pu faire une opération quelconque sur cette colonne.

Essayez donc cette requête pour qu'on rigole 😊 :

**Code : SQL**

```
SELECT gbook_id
FROM dev_gbook
ORDER BY gbook_ip/gbook_id;
```

*A priori*, il n'y a pas beaucoup d'ordre là-dedans, et pourtant...

**Code : SQL**

```
SELECT gbook_id, (gbook_ip/gbook_id) AS op
FROM dev_gbook
ORDER BY gbook_ip/gbook_id DESC;
```

En regardant la colonne **op** retournée par MySQL, vous constatez qu'il a parfaitement ordonné les résultats en suivant notre demande.

Maintenant on va jouer avec l'opposé de DESC : ASC. Exécutez ces deux requêtes et comparez le jeu de résultats de chacune d'elles.

**Code : SQL**

```
-- Première requête
SELECT gbook_id
FROM dev_gbook
ORDER BY gbook_id ASC;
-- Seconde requête
SELECT gbook_id
```

```
FROM dev_gbook  
ORDER BY gbook_id;
```



En SQL, le -- sert à indiquer un commentaire d'une ligne, c'est l'équivalent de // en PHP.

Si vous ne voyez pas de différence entre les jeux de résultats des deux requêtes, arrêtez de vous faire mal aux yeux, vous n'en trouverez pas 😊. En effet, les mots-clés ASC et DESC sont **facultatifs**, et si on n'indique pas explicitement qu'on veut un ordre croissant ou décroissant, MySQL ordonnera les résultats dans l'ordre croissant (ASC). C'est pour cela que les deux requêtes produisent le même jeu de résultats.

Maintenant on va s'amuser encore un peu 😊. Exécutez cette requête :

Code : SQL

```
SELECT gbook_ip, gbook_date  
FROM dev_gbook  
ORDER BY gbook_date;
```

Cette requête retourne la liste des dates des messages triés par la date et ordonnées de façon décroissante. Comme je l'ai dit, pour ordonner, MySQL compare les valeurs et dit : « truc est plus grand / plus petit que machin », et comme on peut comparer deux dates assez simplement, on peut très facilement ordonner avec une date sans perdre beaucoup en performance. MySQL ordonne presque aussi rapidement des dates que des entiers, n'ayez donc pas peur d'utiliser des colonnes (ou, pour être plus exact, des expressions, vu que MySQL ordonne en comparant des expressions) de type DATETIME, DATE, TIME, etc.

Seulement, en regardant le résultat de cette requête, vous constaterez qu'on a des dates identiques. Mais pour des dates identiques, vous constatez qu'il n'y a pas d'ordre particulier pour la colonne **gbook\_ip**. Imaginons que vous souhaitiez récupérer ces mêmes lignes, toujours ordonnées par la date de façon croissante, mais que vous vouliez que pour une même date, les IP soient ordonnées de façon croissante...

Eh bien, c'est possible ! On ferait comme ceci :

Code : SQL

```
SELECT gbook_ip, gbook_date  
FROM dev_gbook  
ORDER BY gbook_date ASC, gbook_ip ASC;
```

On obtient bien ce qu'on voulait. Maintenant, comment ça fonctionne ? Eh bien c'estridicullement simple. MySQL va lire les lignes, il va les ordonner par la date. Ensuite, il va de nouveau lire toutes les lignes, et s'il y a des lignes qui ont la même date, il va ré-ordonner ces lignes en fonction de l'IP (bon, en réalité ce n'est peut-être pas exactement comme ça qu'il procède, mais on peut le voir comme ça pour que ça soit simple à comprendre). Et le plus fort, c'est qu'on peut étendre ce raisonnement à un nombre quelconque d'ordres. On peut sans problème demander à MySQL d'ordonner en fonction de trois, de quatre, de cinquante expressions et même plus. Et bien évidemment, chaque expression peut être ordonnée de façon croissante ou décroissante. Dans ma requête, j'ai demandé deux fois l'ordre croissant, mais j'aurais très bien pu demander un ordre décroissant pour l'une des deux expressions, ou même pour les deux.

La seule chose qu'il ne faut pas oublier, c'est comment MySQL va procéder pour ordonner les lignes. Il commence par ordonner en fonction de l'expression la plus à gauche (la plus proche du **ORDER BY**) et s'il y a des lignes qui ont la même valeur, il ordonnera par la colonne juste à droite, et ainsi de suite jusqu'à ce qu'il n'y ait plus deux lignes avec la même valeur, ou qu'aucune expression ne subsiste comme critère de tri (ou d'ordre).

Avant d'en finir avec l'ordonnancement, encore deux choses !

La première, c'est qu'il est fort probable qu'un jour vous ayez à demander à MySQL le résultat d'une opération (une division, par exemple) et d'ordonner par le résultat de la division.

Avec ce que j'ai dit jusqu'ici, vous écririez par exemple :

Code : SQL

```
SELECT (gbook_ip/gbook_id) AS op
```

```
FROM dev_gbook  
ORDER BY gbook_ip/gbook_id;
```

Seulement, il y a une certaine redondance (répétition) dans cette requête : l'opération est faite deux fois, ce qui est plutôt inutile et coûteux en performance. C'est pourquoi on fera plutôt ceci :

#### Code : SQL

```
SELECT (gbook_ip/gbook_id) AS op  
FROM dev_gbook  
ORDER BY op;
```

Le résultat est strictement le même, on peut ordonner *via* un surnom qu'on a donné à une expression. C'est très pratique, ça allège l'écriture de la requête et ça évite à MySQL de recalculer pour rien 😊.

Mais ceci m'amène à la deuxième chose dont je veux encore vous parler, et qui peut parfois poser problème. Je vous ai dit il y a peu qu'il était parfois dangereux de renommer une expression avec le nom d'une colonne qui existe, on va voir pourquoi 😊.

Imaginons que vous vouliez à nouveau récupérer les valeurs de la division de gbook\_ip par gbook\_id et que vous renommiez cette valeur avec le nom **gbook\_id**. On a alors :

#### Code : SQL

```
SELECT (gbook_ip/gbook_id) AS gbook_id  
FROM dev_gbook;
```

Pas de souci, on a bien ce qu'on veut. Mais soyons fous, et imaginons que vous vouliez récupérer ces mêmes résultats, mais ordonnés de façon décroissante en fonction de la colonne gbook\_id. On ferait :

#### Code : SQL

```
SELECT (gbook_ip/gbook_id) AS gbook_id  
FROM dev_gbook  
ORDER BY gbook_id DESC;
```

Mais maintenant, on va aussi demander à MySQL de nous envoyer les valeurs de la colonne gbook\_id pour vérifier que l'ordre est celui qu'on attend :

#### Code : SQL

```
SELECT gbook_id, (gbook_ip/gbook_id) AS gbook_id  
FROM dev_gbook  
ORDER BY gbook_id DESC;
```

Et là... manque de chance, on voit que ce n'est pas du tout l'ordre qu'on attendait 😊.

Si on regarde attentivement, on constate aisément qu'en fait MySQL a ordonné en fonction du résultat de l'opération, et non en fonction de la valeur de la colonne gbook\_id. Et voilà le risque auquel on s'expose si on renomme une expression avec le nom d'une colonne existante : on n'est pas sûrs de savoir en fonction de quoi MySQL va ordonner les résultats.

Mais il faut quand même pondérer ce risque. En effet, imaginons qu'au lieu d'avoir gbook\_ip/gbook\_id, on ait gbook\_id\*100. Testez et vous verrez que l'ordre est bien celui qu'on voulait. Et pourtant, c'est bien en fonction de gbook\_id\*100 que MySQL a ordonné les résultats. Seulement, il se trouve que si A est plus petit que B, alors A\*100 est également plus petit que B\*100. Il y a donc des opérations qui changent l'ordre, mais il y en a qui le conservent. Si on faisait gbook\_ip/100, on conserverait également le bon ordre. Il faut donc vérifier au cas par cas, ce qui est plutôt rébarbatif, et c'est pour cela qu'il faut être prudents quand on renomme une expression avec un nom de colonne existant. Si ça conserve l'ordre, pas de souci, sinon, gros soucis, et ceci termine ce discours sur ORDER BY ^^.

La requête SELECT générique qu'on a pour le moment est donc :

**Code : SQL**

```
SELECT Expr[, Expr[, ...]]
FROM t_table
[ORDER BY Expr [ASC|DESC] [, Expr [ASC|DESC] [, ...]]];
```

## Filtrons tout ça

Après ce long passage sur l'ordonnancement, on va parler du filtrage des données. Ça vous permettra par exemple de récupérer tous les messages d'un auteur en particulier, ou qui ont été postés entre telle et telle date, et bien d'autres choses.

Le but du jeu est de récupérer tous les messages qui ont été rédigés par l'auteur **L'andouille** (il n'y en a qu'un en réalité). Voici la requête :

**Code : SQL**

```
SELECT gbook_pseudo, gbook_msg
FROM dev_gbook
WHERE gbook_pseudo = 'L''andouille';
```

Si vous regardez le jeu de résultats, toutes les lignes retournées ont **L'andouille** comme valeur dans la colonne **gbook\_pseudo** : c'est ce qu'on voulait.

La nouveauté dans cette requête, c'est la troisième ligne. Le mot-clé **WHERE** indique à MySQL qu'il doit prendre les lignes qui satisfont à certaines conditions. Ces conditions sont ce qui se trouve à droite du **WHERE**.

De manière générique, voici comment j'écrirais cette requête :

**Code : SQL**

```
SELECT Expr[, Expr[, ...]]
FROM t_table
WHERE Expr;
```

Et en lisant ça, vous vous rendez compte que ce qui se trouve à droite du **WHERE** n'est rien d'autre qu'une expression ! 😊

En allant chercher les lignes, MySQL évaluera l'expression du **WHERE** et là, deux cas sont possibles : soit l'expression est évaluée à vrai (1) et MySQL prend la ligne, soit elle est évaluée à faux (0) et MySQL ignore la ligne.

Cette expression peut être simple comme dans la requête précédente, mais elle peut être très complexe. Pour varier les expressions, MySQL nous propose toute une panoplie d'outils : des opérateurs, des fonctions, etc.

Les fonctions, on verra ça un peu plus tard, mais pour que vous puissiez tout de même utiliser le **WHERE**, voici quelques opérateurs qui ne vous sont pas étrangers ; on en verra des bien plus mignons par la suite.

Opérateur	Description
A $\neq$ B	Retourne 1 (vrai) si la valeur de l'expression de gauche (A) est différente de la valeur de l'expression de droite (B).
A = B	Retourne 1 (vrai) si la valeur de l'expression de gauche (A) est égale à la valeur de l'expression de droite (B).
A < B	Retourne 1 si la valeur de l'expression de gauche est strictement plus petite que la valeur de l'expression de droite.
A $\leq$ B	Retourne 1 si la valeur de l'expression de gauche est plus petite ou égale à la valeur de l'expression de droite.
A > B	Retourne 1 si la valeur de l'expression de gauche est strictement plus grande que la valeur de l'expression de droite.
A $\geq$ B	Retourne 1 si la valeur de l'expression de gauche est plus grande ou égale à la valeur de l'expression de droite.
A AND B	Retourne 1 si les deux expressions sont évaluées à 1.

**A OR B**

Retourne 1 si une des deux expressions est évaluée à 1.

Et voilà, vous avez droit à deux opérateurs logiques et six opérateurs de comparaison. La plupart de ces opérateurs ne vous sont pas étrangers, les seuls qui peuvent vous embêter sont les deux premiers :  $\diamond$  et  $=$ .

On a vu qu'en PHP, l'opérateur qui testait la différence était  $!=$ ; en SQL, c'est  $\diamond$ . Toutefois, si vous utilisez un SGBDR assez permissif, comme MySQL, le  $!=$  existe et est équivalent, mais il ne respecte pas la norme, donc utilisez  $\diamond$  et rien d'autre !

On a également vu qu'en PHP, pour tester l'égalité entre deux expressions, on utilisait l'opérateur  $==$ ; en SQL, on utilise juste  $=$ . Il est assez commun de s'embrouiller entre les différents opérateurs du PHP et de MySQL, donc soyez attentifs et faites attention à ce que vous utilisez.

Dans la colonne **Description** du tableau que je viens de vous donner, vous voyez que j'emploie toujours l'expression « retourne 1 ». Toujours en PHP, on a vu que tous les opérateurs logiques et de comparaison retournaient un booléen (false ou true), sauf qu'en SQL, les booléens existent bel et bien, mais sont toujours remplacés par leur équivalent en entier (1 pour true et 0 pour false), comme je l'ai déjà expliqué.

Avec ces petits joujoux, on peut s'amuser avec le WHERE 😊. Par exemple, allons chercher toutes les lignes dont l'id est plus petit que 5 :

**Code : SQL**

```
SELECT gbook_id
FROM dev_gbook
WHERE gbook_id < 5;
```

Maintenant, si on veut tous les id compris entre 2 et 7 (2 et 7 inclus), on fera :

**Code : SQL**

```
SELECT gbook_id
FROM dev_gbook
WHERE gbook_id >= 2 AND gbook_id <= 7;
```

On peut bien sûr combiner tous ces opérateurs comme on le souhaite !

Et c'est là-dessus que se termine cette première approche du WHERE ; la requête générique devient donc :

**Code : SQL**

```
SELECT Expr[, Expr[, ...]]
FROM t_table
[WHERE Expr];
```

## Vertical Limit !

Parlons maintenant du dernier objet de ce chapitre : la limitation du nombre de lignes. Tout d'abord, qu'est-ce que ça veut dire, « limiter le nombre de lignes » ?

Considérons une requête toute simple :

**Code : SQL**

```
SELECT gbook_id
FROM dev_gbook;
```

Celle-ci nous retourne la totalité des valeurs de la colonne gbook\_id, elle nous retourne un **jeu de résultats**. Ce jeu de résultats se caractérise par deux choses : le nombre de colonnes et le nombre de lignes. Le nombre de colonnes est facile à connaître en regardant la requête SQL, il est donné par le nombre d'expressions mises entre le **SELECT** et le **FROM**.

Le nombre de lignes est par contre impossible à connaître en ayant juste la requête de sélection. En effet, si on a aucune condition (avec la clause **WHERE**), MySQL nous retournera toutes les lignes contenues dans la table ; s'il y a une condition, le nombre de lignes dépendra de la condition.

Vous voyez donc que le nombre de lignes est très variable (les requêtes sans **WHERE** sont relativement rares) et que ça peut donc poser problème si on veut récupérer un nombre précis de lignes. Mais on va arranger ça avec la troisième clause : **LIMIT**.

La requête précédente me retournait dix lignes, maintenant je ne veux que quatre lignes :

#### Code : SQL

```
SELECT gbook_id  
FROM dev_gbook  
LIMIT 4;
```

Le résultat est celui que j'attendais : je n'ai bien que quatre lignes. Seulement, ce sont les quatre premières. Comment ferais-je pour avoir les quatre dernières ?

Simple ! Il faut que les dernières deviennent les premières, et inversement 😊. Et pour cela, on a vu un outil très pratique :

**ORDER BY**. Eh oui, on peut utiliser simultanément **ORDER BY**, **WHERE** et **LIMIT** ; et avec un **ORDER BY** approprié, on peut aisément faire en sorte que les premières lignes deviennent les dernières, et inversement.

Pour cela, il faut commencer par trouver comment sont ordonnées les lignes. Ici, on n'a rien dit à MySQL à ce sujet puisqu'on n'a pas mis de **ORDER BY**. Mais en regardant le résultat de la requête précédente sans le **LIMIT**, on constate que les lignes sont triées avec la colonne **gbook\_id** de façon croissante, ce qui équivaut à un **ORDER BY gbook\_id ASC**.

Or, on a vu comment passer à l'ordre décroissant, il suffit de remplacer le **ASC** par **DESC** ; ainsi, on a :

#### Code : SQL

```
SELECT gbook_id  
FROM dev_gbook  
ORDER BY gbook_id DESC  
LIMIT 4;
```

Bingo, on a bien les quatre lignes que l'on attendait.

Malheureusement, on est quand même assez limités : on peut bien récupérer les X premières lignes (ou les dernières en jouant avec **ORDER BY**), mais que se passe-t-il si on a envie de récupérer X lignes à partir d'une ligne donnée ? 🤔

D'abord, regardez ceci :

The screenshot shows a Windows command-line interface window titled 'c:\wamp\mysql\bin\mysql.exe'. It displays the MySQL monitor welcome message and a help prompt. The user has selected the 'dev\_dev' database and run a 'SELECT gbook\_id FROM dev\_gbook;' query. The results are shown in a table with 10 rows, indexed from 0 to 9. The 'gbook\_id' column values are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 respectively.

gbook_id
0
1
2
3
4
5
6
7
8
9
10

10 rows in set (0.03 sec)

mysql> \_

Ce jeu de résultats ne devrait pas vous être inconnu, c'est celui qui est retourné par la requête :

#### Code : SQL

```
SELECT gbook_id  
FROM dev_gbook;
```

Et un peu à droite des différentes lignes, vous pouvez voir des chiffres en rouge qui vont de 0 à 9. Ces chiffres représentent le numéro de la ligne, son index ou encore son **OFFSET**. Il faut savoir que quand MySQL vous renvoie des lignes, il les a ordonnées d'une certaine façon, mais cet ordonnancement crée également une notion de position. La ligne 0 est la première du jeu de résultats, la ligne 1 la deuxième, etc., et c'est sur cet OFFSET qu'on va se baser pour récupérer des lignes un peu comme on le veut.



L'OFFSET des lignes varie de 0 à n-1, n étant le nombre de lignes du jeu de résultats ; rappelez-vous, c'est comme avec un array.

Voici une requête qui va aller me chercher les lignes qui ont un OFFSET compris entre 4 et 8 :

#### Code : SQL

```
SELECT gbook_id  
FROM dev_gbook  
LIMIT 4 OFFSET 4;
```

La nouveauté se situe aux alentours du **LIMIT**. Voici la syntaxe générique de cette requête :

#### Code : SQL

```
SELECT gbook_id  
FROM dev_gbook  
LIMIT nombreLignes OFFSET offsetDepart;
```

**nombreLignes** et **offsetDepart** sont deux expressions de type entiers, dont les valeurs sont supérieures ou égales à zéro. **nombreLignes** indique à MySQL combien de lignes on souhaite récupérer et **offsetDepart** indique à partir de quel OFFSET on va chercher les lignes. Attention, MySQL ne comprendra jamais autre chose que « va chercher X lignes à partir de l'offset Y », vous ne lui direz jamais : « va chercher les lignes dont l'offset est compris entre X et Y ». Au début, c'est une erreur courante.

Par exemple :

- LIMIT 2 OFFSET 10 signifie qu'on demande deux lignes à partir de la ligne qui a l'offset 10 (à partir de la onzième, donc) ;
- LIMIT 5 OFFSET 5 signifie qu'on demande cinq lignes à partir de celle qui a l'offset 5 (à partir de la sixième, donc) ;
- LIMIT 0 OFFSET 99 signifie qu'on ne demande aucune ligne (zéro) à partir de celle qui a l'offset 99 (à partir de la centième, donc).

Il est très probable que vous rencontriez également cette notation : LIMIT X, Y. Vous devez savoir que cette notation n'est pas valide, c'est encore un ajout de MySQL : vous devrez donc toujours utiliser la version LIMIT X OFFSET Y. Seulement, il y a une petite astuce 😊.

LIMIT X, Y est équivalent non pas à LIMIT X OFFSET Y, mais à LIMIT Y OFFSET X. Et c'est d'ailleurs pour ça que cette notation (avec la virgule) est un peu illogique, traduisons en français :

- LIMIT X OFFSET Y signifie qu'on demande X lignes à partir de celle qui a l'offset Y ;
- LIMIT Y, X est équivalent, mais cette fois on le traduit par : on demande, à partir de la ligne qui a l'offset Y, X lignes.

Je ne sais pas pour vous, mais je trouve que la deuxième traduction est plus qu'imbuvable... Utilisez donc toujours la syntaxe LIMIT X OFFSET Y.



Si vous ne spécifiez pas à partir de quel offset vous demandez les lignes (en ne mettant pas le OFFSET Y), MySQL ira chercher les lignes à partir de celle qui a l'offset 0 ; ainsi, LIMIT X OFFSET 0 et LIMIT X sont équivalents et valides 😊.

La requête générique avec le LIMIT est donc :

**Code : SQL**

```
SELECT Expr[, Expr[, ...]]  
FROM t_table  
[LIMIT X [OFFSET Y]];
```

Et voici ce qui termine cette discussion sur la clause **LIMIT**. Je vous ai dit qu'elle pouvait poser des problèmes de performances, mais on verra ça par la suite. Pour le moment, utilisez-la tant que vous voulez 😊.

Mais avant de vous laisser en paix, voici la requête générique qui couple tout ce qu'on a dit :

**Code : SQL**

```
SELECT Expr[, Expr[, ...]]  
FROM t_table  
[WHERE Expr]  
[ORDER BY Expr [ASC|DESC] [, Expr [ASC|DESC] [, ...]]]  
[LIMIT X [OFFSET Y]];
```

Plutôt élégant, n'est-ce pas ? Cette requête vous donne aussi l'ordre dans lequel les trois clauses doivent être spécifiées. En effet, si vous mettez un **ORDER BY** avant un **WHERE** ou un **WHERE** après un **LIMIT**, vous ne récolterez pas grand-chose d'autre qu'une erreur de la part de MySQL (parse error).

Et voici un nouveau chapitre qui se termine. Il était assez long, je vous conseille donc de le relire un peu plus tard après avoir fait une petite pause 😊.

Étant donné sa longueur et la quantité de choses à retenir, il n'y a à nouveau pas de QCM.

Mais rassurez-vous, le prochain chapitre sera plus court et utilisera l'une ou l'autre des notions vues ici, vous pourrez donc éprouver vos connaissances.

Et dans peu de temps, vous aurez droit à quelques TP suivis de l'utilisation conjointe de MySQL et du PHP !

## Supprimer, modifier et insérer des données

Dans ce chapitre, nous allons voir trois types de requêtes : les suppressions, les modifications et les insertions. Une fois qu'on aura vu tout ça, vous pourrez enfin réellement utiliser MySQL. Bien évidemment, il restera encore beaucoup de choses à voir : jointures, sous-requêtes, index, etc. Mais c'est pour plus tard 😊.

### DELETE s'exhibe

Pour aborder ce chapitre en douceur, on va commencer par le type de requêtes le plus « simple » : DELETE.

Comme son nom l'indique (DELETE se traduit par *supprimer*), ce type de requêtes va vous permettre de supprimer des données. Mais attention toutefois : quand je dis données, je pense **ligne** (ou *tupple*). En effet, ce type de requêtes ne permet de supprimer que des lignes. Et c'est assez logique quand on y pense : que voudrait dire « supprimer une valeur d'une colonne » ? Mettre NULL ? Mettre la valeur par défaut ?

Ça n'a aucun sens : si on a une ligne, toutes les colonnes existent, on ne peut pas supprimer une valeur d'une colonne.

Sans plus attendre, voici la syntaxe minimale d'un DELETE :

#### Code : SQL

```
DELETE FROM t_table  
[WHERE Expr];
```

Ça n'est pas bien sorcier, mais explicitons quand même tout cela. La première ligne se compose de deux mots-clés, DELETE et FROM, qui indiquent qu'on veut effectuer une suppression de données, suivis du nom de la table dans laquelle on veut procéder.

La seconde ligne ne devrait pas vous être inconnue, c'est une clause WHERE comme on l'a vu dans le chapitre précédent. Cette clause est facultative, mais en pratique vous l'utiliserez toujours 🍷.

Reprenons l'exemple de la table **dev\_gbook** et imaginons que pour une quelconque raison, un message injurieux se soit glissé dans le tas. Il serait bienvenu de pouvoir éliminer ce message, en le supprimant par exemple.

Seulement, pour supprimer ce message **et seulement** ce message, il faudra bien dire à MySQL « supprime cette ligne, mais pas les autres », et c'est là que la clause WHERE va être utile. En effet, MySQL ne supprimera que les lignes qui satisfont cette clause.

Pour identifier un message dans la table **dev\_gbook**, on a une colonne très utile : **gbook\_id**, qui est un nombre entier positif unique et invariable pour un message donné. On va donc se baser là-dessus pour supprimer le message injurieux (dans l'exemple, l'id du message en question est choisi arbitrairement, 78 par exemple). Si vous avez bien suivi ce que j'ai dit, vous devriez être en mesure d'écrire la requête qui supprimera le message ayant l'id 78 😊.

#### Secret (cliquez pour afficher)

#### Code : SQL

```
DELETE FROM dev_gbook  
WHERE gbook_id = 78;
```

Bien évidemment, vous n'êtes absolument pas tenus d'utiliser l'id pour supprimer des messages ; vous pouvez sans problème supprimer les messages antérieurs à une date donnée (dans ce cas vous utiliserez l'opérateur <, >, etc. pour comparer une date avec la colonne **gbook\_date**), les messages qui ont été rédigés par une IP donnée (là, vous utiliserez l'opérateur = et la colonne **gbook\_ip**) ou je ne sais quoi d'autre, comme par exemple supprimer les messages où **(gbook\_ip % gbook\_id) / 4 > 2** si ça vous amuse 🍸.

Comme je l'ai dit, la clause WHERE est facultative, mais en pratique on l'utilise toujours. Pourquoi ?

Réfléchissons ! Si on ne met pas WHERE, on va supprimer toutes les lignes. Or, il est assez rare que l'on décide de supprimer toutes les données d'une table. Mais il y a une autre raison. Si l'on veut vider sa table, il y a une méthode plus rapide qu'un DELETE : TRUNCATE.

Si vous exécutez cette requête :

#### Code : SQL

```
TRUNCATE TABLE t_table;
```

vous viderez entièrement la table `t_table` ; concrètement, vous supprimerez toutes les lignes. Comme je l'ai dit, cette requête est un peu plus rapide qu'un simple `DELETE FROM t_table`. En effet, `TRUNCATE` est prévu pour supprimer toutes les lignes, alors que `DELETE` permet bien plus de choses, les développeurs de MySQL ont donc optimisé le fonctionnement de `TRUNCATE` et au final, cette méthode est plus rapide.

Une autre différence (abus de langage, ce n'est pas tout à fait vrai, mais le détail n'est pas très important) est l'état de **l'auto\_increment**. Imaginons que vous ayez une table avec une colonne possédant l'option `auto_increment` et que la plus grande valeur de cette colonne est 78. Si vous supprimez toutes les lignes de la table avec `DELETE` et qu'après vous insérez une nouvelle ligne, vous verrez que la valeur de la colonne en `auto_increment` sera 79.

Maintenant imaginons qu'au lieu du `DELETE`, on ait fait un `TRUNCATE TABLE` : si on insère une nouvelle ligne, la valeur de la colonne en `auto_increment` sera... 1.

Eh oui, si `DELETE` conserve la valeur de `l'auto_increment`, ce n'est pas le cas de `TRUNCATE`.

`DELETE` et `l'auto_increment`... deux mots qui vont mal ensemble 😊. En effet, il arrive souvent que les gens aient envie de combler les « trous ».

Imaginez que nous ayons une table contenant trois lignes avec une colonne en `auto_increment` ayant les valeurs 1, 2 et 3. Si je supprime la ligne dont la valeur de la colonne en `auto_increment` est 2, il me restera les lignes 1 et 3. Et si par la suite j'insère une nouvelle ligne dans ma table, la valeur de la colonne en `auto_increment` sera 4.

Or, cette idée semble déplaire à beaucoup de monde 😬. Bien souvent, il arrive que des gens demandent comment combler ces trous.

Ce n'est pas impossible, loin de là, on peut combler ces trous si on le veut. Sauf que... ça ne sert à rien. Il est totalement inutile de combler les trous et c'est surtout illogique. L'unicité est une des raisons de cet illogisme. Reprenons l'exemple du livre d'or : si je supprime un message et que je comble le trou, deux messages différents auront eu le même identifiant.

Maintenant, imaginons que je donne un lien qui permette d'afficher un seul message en se basant sur l'identifiant unique, n'aurais-je pas un problème ?

Eh bien si : imaginons qu'à l'instant  $T_0$  j'écrive ce lien sur un forum quelconque pour utiliser le message du livre d'or comme argument, je supprime le message à un instant  $T_1$  et à l'instant  $T_2$ , un visiteur suit le lien qui est sur le forum ( $T_0 < T_1 < T_2$ ). Le visiteur croit alors tomber sur un argument intéressant et il tombe sur un « heyyyy trop fort ton site loool xptdr » : avouez que ça serait plutôt ennuyeux 😞.

Après ce petit passage sur `l'auto_increment`, on va en terminer avec cette première approche de `DELETE` en parlant de deux clauses facultatives qui vont de pair : `ORDER BY` et `LIMIT`.

Ces deux clauses sont très pratiques quand on ne veut pas supprimer toutes les lignes qui répondent à la clause `WHERE`. Imaginons par exemple que nous voulions supprimer des messages du livre d'or plus anciens qu'une date donnée, mais qu'on veuille conserver un certain nombre de dates données. On pourrait d'abord sélectionner les id des messages et ensuite se baser là-dessus pour les supprimer, mais on peut simplifier ça. Voici la requête générique :

#### Code : SQL

```
DELETE FROM t_table
WHERE Expr1
ORDER BY Expr2 [ASC|DESC]
LIMIT nb_ligne;
```

Cette requête supprimera les lignes qui répondent à la clause `WHERE`, mais pas n'importe comment : MySQL va les supprimer dans l'ordre défini par l'`ORDER BY` et en supprimera au maximum `nb_ligne`.

Il ne faut pas confondre le `LIMIT` d'un `SELECT` et d'un `DELETE`. Le premier permet de dire « x lignes maximum à partir de y », alors que le second permet de dire « x lignes maximum ».

Ainsi, pour supprimer deux lignes antérieures au 4 octobre 2007, j'écrirai cette requête :

#### Code : SQL

```
DELETE FROM dev_gbook
WHERE gbook_date < '2007-10-24 22:26:02'
ORDER BY gbook_date
LIMIT 2;
```

Exécutez cette requête : MySQL vous répond qu'il a supprimé deux lignes (**Ok, 2 rows affected**) alors que quatre lignes répondaient au `WHERE`. Ces clauses sont aussi très pratiques quand on veut supprimer le message le plus ancien, le plus

récent...

DELETE est un outil très puissant que vous devrez apprendre à utiliser, alors n'hésitez pas à relire ce chapitre ! Il n'est pas très long et assez simple, mais ce n'est pas une raison pour le négliger 😊.

Et le plus fort, c'est que nous n'avons pas encore vu toute la puissance de DELETE...

## UPDATE en folie

Après la suppression, passons à la mise à jour ! Si vous avez bien compris DELETE et ses différentes clauses, le type de requêtes que je vais présenter vous paraîtra très simple. Pour faire des mises à jour, le mot magique est UPDATE (qui se traduit par *mettre à jour*).

Sans plus attendre, voici la requête tant espérée :

### Code : SQL

```
UPDATE t_table
SET t_col1 = Expr[, t_col2 = Expr2[, ...]]
[WHERE Expr3]
[ORDER BY Expr4 [ASC|DESC]]
[LIMIT nb_lignes];
```

Passé le cap du choc de la première rencontre, on se rend compte que cette requête ressemble beaucoup à DELETE, et c'est également le cas en pratique.

Tout comme DELETE, UPDATE ne met à jour que les lignes qui satisfont la clause WHERE. Les clauses ORDER BY et LIMIT ont le même rôle et le même fonctionnement que DELETE.

La première ligne, UPDATE t\_table, sert simplement à indiquer à MySQL qu'on veut faire des mises à jour dans la table t\_table.

Reste la ligne la plus intéressante, la seconde. DELETE permettait de supprimer uniquement des lignes, mais ce n'est pas le cas d'UPDATE. En effet, on peut mettre à jour une colonne, deux colonnes, trois colonnes, toutes les colonnes si on le désire 😊.

Le mot-clé SET indique qu'on va lister les colonnes qu'on veut mettre à jour. Après ce mot-clé, vous mettrez le nom des colonnes à mettre à jour suivi d'un égal et de leur nouvelle valeur. Chacune des colonnes sera séparée par une virgule.

L'avantage d'UPDATE, c'est qu'on peut utiliser la valeur actuelle de la colonne !

Imaginons qu'on veuille mettre à jour la table t\_table. Je souhaite ajouter une unité à la colonne t\_col pour toutes les lignes. Ceux qui ignorent qu'on peut utiliser la valeur actuelle des colonnes feront un SELECT et puis mettront à jour en se basant sur le résultat de la requête précédente, mais c'est lourd, et pour cause : on peut faire la même chose avec un simple UPDATE :

### Code : SQL

```
UPDATE t_table
SET t_col1 = t_col+1;
```

Bien évidemment, on peut faire des calculs plus compliqués : je pourrais par exemple attribuer la valeur  $((t\_col \% 2)/t\_col \% 9$  si ça me chante.

Et c'est déjà fini pour UPDATE. Il faut dire que DELETE nous a bien pré-mâché le travail 😊.

## INSERT sous les feux de la rampe

Après la suppression et l'édition, faisons place à l'insertion ! Vous avez déjà eu droit à un avant-goût au chapitre précédent : souvenez-vous de cette requête bizarre que je ne vous avais pas expliquée.

Sans plus attendre, voici une requête d'insertion, INSERT :

### Code : SQL

```
INSERT INTO t_table(col_list)
VALUES (values_list)[, (values_list2)[, ...]];
```

La première ligne se compose de trois parties distinctes :

- INSERT INTO ;

- **t\_table** ;
- **(col\_list)** ;

La première partie est un simple mot-clé qui indique à MySQL qu'on veut faire une insertion, tout comme DELETE indique qu'on veut faire une suppression ; **t\_table** est le nom de la table dans laquelle on veut insérer des données.

La troisième partie est la plus intéressante de cette ligne. En effet, entre parenthèses, vous allez indiquer la liste des colonnes pour lesquelles on va donner explicitement des valeurs.

Je m'explique : quand on a parlé des types, on a vu qu'on pouvait donner des valeurs par défaut aux colonnes quand on crée une table. Eh bien c'est là que les valeurs par défaut interviennent. Si on dit à MySQL qu'on va donner explicitement une valeur en listant la colonne dans **col\_list**, MySQL insérera cette valeur, tandis que si on ne nomme pas une colonne dans **col\_list**, MySQL insérera la valeur par défaut que vous avez spécifiée lors de la création de la table (ou une valeur que MySQL déterminera lui-même, si vous n'avez pas spécifié de valeur par défaut).

Il est donc très important de donner les bonnes valeurs par défaut au moment de la création de la table : comme cela on n'aura pas à donner explicitement toutes les valeurs, ce qui est un gain de temps, de clavier et de lecture 😊.

Les colonnes que vous nommerez dans **col\_list** devront encore et toujours être séparées par des virgules.

Attaquons maintenant la seconde ligne. Le premier mot-clé, **VALUE**, indique à MySQL qu'on veut insérer des valeurs constantes (on verra plus tard qu'on peut faire des choses amusantes comme insérer des données venant d'une autre table !).

Après cela, on trouve la liste des valeurs que vous donnerez à chaque colonne.

Je m'explique : dans **col\_list**, vous avez indiqué le nom des colonnes pour lesquelles vous allez donner une valeur, MySQL s'attend donc à trouver le bon nombre de valeurs dans **values\_list**.

C'est d'ailleurs la cause d'une erreur extrêmement fréquente que voici :

#### Code : Console

```
ERROR 1136 (21S01) : Column count doesn't match value count at row 1
```

En français, cela nous donne à peu près : *Le nombre de colonnes ne correspond pas au nombre de valeurs à la ligne 1*. Si vous rencontrez cette erreur, c'est que vous donnez un nombre de valeurs différent du nombre de colonnes dans lesquelles vous voulez insérer des données.

Une **value\_list** insérera une et une seule ligne ; ainsi, pour insérer plusieurs lignes, vous devrez donner plusieurs **values\_list**, chacune étant délimitée par des parenthèses, séparées les unes des autres par une virgule.

Pour donner un exemple concret, imaginons que je souhaite insérer deux messages dans le livre d'or ; j'aurais quelque chose du genre :

#### Code : SQL

```
INSERT INTO dev_gbook(gbook_ip, gbook_msg, gbook_pseudo, gbook_date)
VALUES
(1, 'lala', 'hi''han', '2007-01-01 00:00:00'),
(2, 'lili', 'ha''hin', '2006-01-01 00:00:00');
```

Et comme vous le constaterez si vous faites un petit SELECT, deux lignes sont venues s'ajouter. Vous voyez également que je n'ai pas spécifié de valeur pour la colonne **gbook\_id**. En effet, cette colonne a l'option **auto\_increment**, MySQL lui donne donc la bonne valeur par défaut.

Il peut arriver que vous rencontriez ce genre de requête :

#### Code : SQL

```
INSERT INTO dev_gbook
VALUES
('1', 1, 'lala', 'hi''han', '2007-01-01 00:00:00'),
('1', 2, 'lili', 'ha''hin', '2006-01-01 00:00:00');
```

Cette requête est correcte et produit exactement le même résultat que la précédente, à la différence près que cette requête-ci est moche 🤢.

Vous pouvez voir que le **col\_list** a disparu. En effet, ce dernier est facultatif, et si on ne le spécifie pas, MySQL considérera qu'on veut donner explicitement des valeurs pour chaque colonne. Le seul petit problème, c'est qu'alors on se retrouve avec des

choses moches comme ces deux " qu'on peut voir dans chacun de mes values\_list. L'idée de mettre ça vient d'un constat assez simple : si on tente d'insérer un type de données qui ne correspond pas au type de la colonne, MySQL donne la valeur par défaut et comme dans ce cas, c'est la valeur de l'**auto\_increment**, tout fonctionne.

Mais tout comme le SELECT \*, il est plus que fortement conseillé de toujours spécifier les colonnes dans lesquelles on veut insérer des données.

La première raison, c'est que les valeurs par défaut existent et qu'il faut les rentabiliser.

La seconde raison, c'est bien évidemment la lisibilité : si on n'indique pas pour quelle colonne on donne des valeurs, il faut avoir la structure de la table pour savoir ce qu'on insère et dans quelle colonne, alors qu'avec la requête précédente, en voyant le nom des colonnes, on peut aisément deviner à quoi sert chaque donnée.

Et puis, soyons fous ! Imaginez que vous vouliez insérer une ligne avec toutes les valeurs par défaut ; si je ne nomme pas explicitement les colonnes, je devrai faire :

#### Code : SQL

```
INSERT INTO dev_gbook  
VALUES ('', '', '', '', ''');
```

Alors que si je nomme les colonnes, je n'ai qu'un :

#### Code : SQL

```
INSERT INTO dev_gbook()  
VALUES();
```

C'est qui qui gagne en nombre de caractères frappés ? 😊

Vous devez également savoir, comme je vous l'ai déjà dit, qu'on peut insérer des données venant d'une autre table. Pour cela, on utilisera la syntaxe INSERT INTO ... SELECT :

#### Code : SQL

```
INSERT INTO t_tabl(col_list)  
SELECT col_list  
FROM t_table2  
[WHERE Expr]  
[ORDER BY Expr [ASC|DESC]]  
[LIMIT X [OFFSET Y]];
```

Il est bien évident que tout ce que j'ai dit précédemment reste valable, et que le nombre de colonnes que vous nommez doit être égal au nombre d'expressions que vous récupérez avec le SELECT.

Et voici ce qui termine cette partie 😊.

Dans ce chapitre, nous avons plusieurs types de requêtes (DELETE, UPDATE et INSERT), ainsi que plusieurs clauses et particularités propres à chacune.

À présent, vous êtes capables de vous débrouiller avec MySQL et vous pouvez espérer utiliser une base de données avec le PHP : c'est d'ailleurs le but du prochain chapitre ! 😊

Après ce chapitre qui sera relativement court, vous aurez droit à un nouveau TP qui mettra à l'épreuve votre maîtrise des formulaires en PHP ainsi que vos connaissances tant en PHP qu'en SQL.

## PHP et MySQL : des alliés de poids

Après ces quelques notions de SQL, on va enfin rentabiliser ce temps passé en utilisant conjointement PHP et MySQL. Dans ce chapitre, vous allez voir comment discuter avec MySQL, comment exécuter des requêtes, comment exploiter les résultats de ces requêtes et bien d'autres choses encore ! 😊

### Retour en arrière

Avant de commencer à jongler avec PHP et MySQL, un petit rappel s'impose. Je vous ai dit précédemment que MySQL était un logiciel qui servait à gérer des bases de données. Je vous ai également présenté une console qui nous a permis de donner des ordres à MySQL en lui donnant des requêtes légales (avec une syntaxe correcte).

Mais il est important de ne pas tout confondre. Quand on utilise la console, **on n'utilise pas MySQL**. En effet, la console n'est qu'un outil qui nous permet de communiquer avec MySQL.

Pour comprendre ça, il suffit de réfléchir un peu. On ne peut pas dialoguer n'importe comment avec MySQL, il faut suivre un certain protocole. Les protocoles représentent une chose importante : ils permettent de décrire comment les différents applications communiquent entre elles, de simplifier le développement d'applications.

En effet, si vous avez envie de créer votre propre logiciel pour communiquer avec MySQL, vous n'aurez pas besoin de voir comment MySQL fonctionne de A à Z, vous aurez simplement besoin d'aller lire la documentation qui décrit le protocole que MySQL utilise ; ensuite, vous pourrez créer votre application. La console MySQL n'est donc qu'une application qui nous permet de dialoguer avec MySQL grâce à l'intégration du bon protocole.

Maintenant, souvenez-vous, pour lancer la console MySQL, nous avons utilisé la commande :

#### Code : Console

```
mysql -u root
```

Si vous n'avez jamais touché à une console ou à un terminal, vous ne savez pas trop ce que ça veut dire, je vais donc vous l'expliquer grossièrement.

Le premier mot, **mysql**, indique qu'on veut lancer l'application MySQL.

Le second mot, **-u**, indique à l'application qu'on lui donne un paramètre (comme pour une fonction) ; le paramètre désigné par **-u** est le nom d'utilisateur avec lequel on veut se connecter à MySQL.

Le troisième mot, **root**, est simplement le nom d'utilisateur, la valeur donnée au paramètre **-u**.

Tapez cette commande revient donc à lancer MySQL et à demander de se connecter avec le nom d'utilisateur **root**. En effet, tout comme sur un système d'exploitation, Windows, Ubuntu, etc., plusieurs utilisateurs peuvent utiliser MySQL. La raison principale, c'est que l'administrateur du PC peut ne pas vouloir donner tous les droits aux utilisateurs de MySQL, il créera donc des comptes avec des droits limités.

Bien évidemment, en plus du nom d'utilisateur, il y a également un mot de passe, sinon n'importe qui pourrait se connecter avec le compte de l'administrateur 😊. C'est là que vous devez vous demander « Mais à quel moment a-t-on donné un mot de passe ? ». La réponse est : jamais.

Le compte **root** est le compte de l'administrateur, et par défaut, il n'y a pas de mot de passe ; c'est pour cela que nous n'avons pas eu besoin d'en donner. Bien évidemment, sur un serveur public, tous les comptes sont protégés par des mots de passe.

Pour vous connecter à MySQL, vous aurez donc besoin de deux choses : le nom d'utilisateur et le mot de passe associé. Avec la console, la connexion se fait toute seule grâce au paramètre **-u**. En PHP, nous n'utiliserons pas la console, mais on verra une autre façon de se connecter à MySQL.

Toutefois, en plus du nom d'utilisateur et du mot de passe, on va avoir besoin de l'adresse du serveur MySQL. En effet, votre serveur MySQL... il se trouve bien quelque part, sur un serveur. Quand on lance la console, on n'a pas besoin de préciser où se trouve MySQL, et c'est assez logique puisque la console est dans le même dossier que MySQL 😊, elle « sait » donc où aller trouver MySQL.

Cependant PHP, lui, ne le saura pas : MySQL peut se trouver sur le même serveur ou sur un serveur distant. L'adresse du serveur dépendra de l'hébergeur chez qui vous mettrez vos scripts, par exemple si vous êtes chez Free, vous utiliserez l'adresse **sql.free.fr** ; si vous êtes en local, vous utiliserez l'adresse **localhost**, si MySQL est sur un des PC d'un réseau local, vous utiliserez son adresse IP sur le réseau local.

Bref, vous savez maintenant de quoi on a besoin pour se connecter à MySQL. Sachez que vos identifiants de connexion (adresse du serveur, nom d'utilisateur et mot de passe) vous sont fournis par votre hébergeur, c'est donc à eux que vous devrez les demander si vous les perdez, si vous ne les recevez pas, etc.

### Dis bonjour à MySQL !

Après cette brève explication, passons à la pratique. Au travers d'une extension, MySQLi, PHP met à notre disposition un

nombre de fonctions assez conséquent pour travailler avec MySQL. Cette liste de fonction est disponible dans la [documentation PHP](#).



Dans la documentation de MySQLi, vous entendrez souvent parler d'objet et de procédural. Pour le moment, reportez-vous toujours au style procédural.

Pour l'instant, on va s'intéresser à la fonction qui est peut-être la plus utile de toutes : celle qui nous permet de se connecter à MySQL. En effet, tout comme la console MySQL, PHP peut utiliser le protocole de MySQL, on peut donc directement communiquer avec MySQL en passant par PHP, ce qui nous évite de devoir passer par un intermédiaire supplémentaire comme la console.

Voici le prototype allégé de la fonction qui permet de se connecter à un serveur MySQL :

#### Code : Console

```
MySQLi mysqli_connect(string $server, string $username[, string $password])
```

Le paramètre **\$server** est l'adresse du serveur auquel on veut se connecter, **\$username** est le nom d'utilisateur avec lequel on veut se connecter et **\$password** le mot de passe à utiliser.

Comme vous le voyez, le paramètre **\$password** est facultatif, on a vu qu'il pouvait ne pas y avoir de mot de passe associé à un compte d'utilisateur. Il faut également que vous sachiez que ce n'est pas le prototype complet ; si vous voulez le voir tel qu'il est en réalité, allez lire la documentation. Je ne présente pas la version complète car elle ne présente pas d'intérêt pour le moment.

Arrêtons-nous un peu sur le type de ce que retourne cette fonction : un objet MySQLi. Ce type est quelque chose que nous n'avons jamais rencontré jusqu'à présent. Sa vraie nature n'a pour l'heure pas d'importance, tout ce que vous devez savoir, c'est que ce type représente une connexion à un serveur MySQL.

Lors de tout échange de données, vous devez connecter les différents intervenants : ceux-ci peuvent ensuite utiliser la connexion établie pour dialoguer entre eux. Ici, le principe est le même, on veut que PHP puisse dialoguer avec le serveur MySQL, nous avons donc besoin d'une connexion. Lorsque nous voudrons envoyer des requêtes au serveur MySQL, il faudra dire à PHP quelle connexion il doit utiliser pour communiquer la requête au serveur MySQL.

Passons au code, connectons-nous à MySQL ! Sortez Notepad++ ou bien votre éditeur de code préféré, mettez-y ce code et testez donc :

#### Code : PHP

```
<?php  
  
mysqli_connect('localhost', 'root');  
// j'aurais aussi pu écrire : mysqli_connect('localhost', 'root', '');  
  
?>
```

Comme vous le constatez, ce code n'affiche rien, et pour cause : nous n'avons pas demandé à PHP d'afficher quoi que ce soit, on lui a juste demandé de se connecter à MySQL. La valeur de retour de cette fonction est un nouveau type de variables un peu particulier : une ressource.

On verra plus en détail par la suite ce que sont les ressources et à quoi elles servent.

Maintenant, ça serait bien de pouvoir exécuter des requêtes. Pour cela, on va faire appel à la fonction adéquate : **mysqli\_query()**. Voici son prototype :

#### Code : Console

```
[MySQLi_Result | bool] mysqli_query(MySQLi $link, string $query)
```

À nouveau, le prototype n'est pas complet, j'ai volontairement omis un paramètre optionnel. Cette fonction est assez simple à comprendre, elle attend deux paramètres : la connexion avec le serveur MySQL et une chaîne de caractères (la requête que vous voulez que MySQL exécute).

La valeur de retour de cette fonction est par contre un peu plus subtile. Dans les chapitres précédents, on a vu plusieurs types de requêtes ; il y avait les requêtes de type SELECT, DELETE, UPDATE, SHOW, etc.

`mysqli_query()` peut retourner trois « valeurs » : un objet `MySQLi_Result`, un booléen (true) ou un booléen (false). La valeur de retour dépend de deux choses : le type de la requête et sa syntaxe.

Si la requête retourne un jeu de résultats (par exemple des requêtes de type SELECT, SHOW, etc.), `mysqli_query()` retournera un objet `MySQLi_Result` qui représente le jeu de résultats si la requête est correcte syntaxiquement, le booléen false sinon.

Si la requête ne retourne pas de jeu de résultats (DELETE, UPDATE, DROP, etc.), `mysqli_query()` retournera le booléen true si la requête est correcte syntaxiquement, le booléen false sinon.

Il est très important de connaître les différentes valeurs de retour possibles en fonction du type de la requête, ça va nous permettre de gérer facilement les erreurs éventuelles. En effet, imaginons que votre requête soit incorrecte :

#### Code : PHP

```
<?php

// on se connecte et stocke la connexion obtenue dans une variable
$link = mysqli_connect('localhost', 'root');
// on exécute une requête sur la connexion précédemment obtenue
mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;');

?>
```

Vous ne pouvez pas savoir si la requête a échoué ou non (mais je peux vous assurer que la requête de mon exemple est incorrecte, sauriez-vous me dire pourquoi ? ), mais si vous connaissez les valeurs de retour possibles, vous pouvez vous en assurer.



Un objet `MySQLi_Result` est évalué à true.

Faisons ce petit test :

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
if(mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;'))
{
    echo 'Requête ok';
}
else
{
    echo 'Erreur de requête';
}

?>
```

Maintenant, il serait intéressant de connaître l'erreur (et comme ça, vous pourrez vérifier si vous aviez trouvé la bonne erreur ). Pour cela, il existe une fonction fournie par PHP : `mysqli_error()`. Cette fonction retourne la dernière erreur générée par une requête SQL ou une chaîne vide, s'il n'y a pas eu d'erreur.

Notre code deviens donc :

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
if(mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;'))
{
    echo 'Requête ok';
}
```

```

else
{
    // nous devons spécifier sur quelle connexion nous voulons voir
    // s'il y a une erreur
    echo 'Erreur de requête : ' . mysqli_error($link);
}

?>

```

Eh oui, l'erreur gagnante était que je n'avais pas sélectionné de base 😊.

Seulement, vous avouerez que si on doit taper tout ça à chaque fois qu'on utilisera **mysqli\_query()**, ça deviendra vite lassant. Et c'est là qu'intervient quelque chose qu'on a vu il y a quelque temps déjà : l'opérateur logique **||** (ou **OR**). En effet, dans tous les cas, si la requête génère une erreur, **mysqli\_query()** retourne false, sinon elle retourne un objet (soit un booléen, soit un objet **MySQLi\_Result**) qui est évalué à true dans tous les cas. Si la fonction retourne false, on va arrêter le script et afficher l'erreur générée ; pour arrêter le script, on utilisera la fonction **exit()** (souvent, vous verrez des gens utiliser **die()** au lieu de **exit()**) ; seulement, **die()** n'est qu'un alias de **exit()** qui pourrait disparaître dans les prochaines versions du PHP, utilisez donc toujours **exit()**.

On écrira donc :

#### Code : PHP

```

<?php

$link = mysqli_connect('localhost', 'root');
mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;') or
exit(mysqli_error($link));

?>

```

Pour comprendre pourquoi le **exit()** n'est exécuté que si la requête génère une erreur, il suffit de savoir que PHP est paresseux 🍪.

En effet, vous devez savoir que l'opérateur **||** (ou **OR**) retourne true si l'un des deux opérandes vaut true. On sait également que PHP lit les expressions de gauche à droite, on peut donc affirmer que si on a : **true || \$variable**, PHP retournera true. Et c'est pourquoi si l'opérande de gauche est évalué à true, l'expression de droite ne sera pas exécutée. PHP est un langage paresseux : s'il connaît avec certitude le résultat d'une expression sans avoir besoin de l'évaluer toute entière, il ne le fera pas et n'évaluera que ce qui est nécessaire.

Je vous conseille de **toujours** mettre un **exit(mysqli\_error())** : vous allez voir par la suite que si votre requête vient à générer une erreur, ça produira des erreurs en cascade et que si vous n'avez pas mis ce petit truc, vous pourriez vous perdre à chercher l'erreur là où elle n'est pas.

Maintenant, on va se connecter à une base pour pouvoir travailler dessus, le code sera donc :

#### Code : PHP

```

<?php

$link = mysqli_connect('localhost', 'root');
mysqli_query($link, 'USE dev_dev;');

$result = mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;') or
exit(mysqli_error($link));

?>

```



Sachez qu'il existe une fonction pour sélectionner une base de données : **mysqli\_select\_db(MySQLi \$link, string \$basename)**. Faire **mysqli\_query(\$link, 'USE db;')** est équivalent à **mysqli\_select\_db(\$link, 'db');**.

Vous pouvez bien évidemment changer de base par la suite en sélectionnant à nouveau une autre base.

Enfin, sachez que la fonction **mysqli\_query()** ne permet d'exécuter qu'une et une seule requête à la fois ! Si vous tentez de faire ceci :

#### Code : PHP

```
<?php  
    mysqli_query($link, 'SELECT t_col FROM t_table; DELETE FROM t_table  
    WHERE Expr;') or exit(mysqli_error($link));  
?>
```

MySQL générera une erreur de syntaxe car si séparément ces deux requêtes sont légales, l'ensemble des deux n'est pas une requête légale. De plus, étant donné qu'on ne peut exécuter qu'une et une seule requête, sachez que le point-virgule (;) qui termine normalement la requête est facultatif, vous pouvez ne pas le mettre. Ainsi, ces requêtes produiront dans ce contexte strictement le même résultat :

#### Code : PHP

```
<?php  
    mysqli_query($link, 'SELECT t_col FROM t_table;') or  
    exit(mysqli_error($link));  
    mysqli_query($link, 'SELECT t_col FROM t_table') or  
    exit(mysqli_error($link));  
?>
```

Maintenant que vous savez vous connecter à MySQL, sélectionner une base et traquer les erreurs générées par vos requêtes, on va passer à l'exploitation des objets MySQLi\_Result. 😊

### Exploiter les ressources

Comme vous l'avez vu, **mysqli\_connect()** et **mysqli\_query()** retournent toutes les deux des types un peu spéciaux : des objets MySQLi et MySQLi\_Result.

Comme je l'ai déjà dit, ce sont des types un peu particuliers, nous les verrons en détail par la suite. Sachez juste que ces types ne sont pour l'heure pas exploitables directement.

Voici un exemple :

#### Code : PHP

```
<?php  
    echo mysqli_connect('localhost', 'root');  
?>
```

Ce code vous affiche une charmante erreur fatale. Si vous tombez nez à nez avec un message comme celui-ci, ne cherchez pas plus loin, c'est que vous avez tenté d'afficher un de ces objets, ce qui est évidemment impossible.

Pour exploiter ces objets, on va d'abord se demander ce qu'ils représentent. On a vu que les requêtes qui retournent des jeux de résultats nous retournaient toujours des tableaux dans la console ; eh bien c'est exactement pareil en PHP : on va utiliser ces objets en demandant à des fonctions fournies par PHP de nous donner des lignes du tableaux.

Pour cela, PHP nous fournit des fonctions : les fonctions **mysqli\_fetch\_\*()**. Ce nom de fonction barbare est une façon personnelle d'écrire : **mysqli\_fetch\_array()**, **mysqli\_fetch\_assoc()** et **mysqli\_fetch\_row()**.

Ces trois fonctions permettent toutes d'obtenir une ligne du jeu de résultats représenté par un objet MySQLi\_Result, et leurs prototypes sont très similaires :

#### Code : Console

```
[array|bool] mysqli_fetch_array(MySQLi_Result $result)
```

```
[array|bool] mysqli_fetch_assoc(MySQLi_Result $result)
[array|bool] mysqli_fetch_row(MySQLi_Result $result)
```

Ces trois fonctions doivent vous paraître étranges... Trois fonctions, trois mêmes types de retour, trois même prototypes, quel est l'intérêt ?



Pour l'instant, on ne se préoccupera pas du booléen que peuvent retourner ces fonctions.

Eh bien l'intérêt réside dans l'array que nous retourne chacune des fonctions. On a vu que les colonnes étaient accessibles soit par leur nom, soit par leur index. Et c'est là que chaque fonction tire son épingle du jeu.

La fonction **mysqli\_fetch\_assoc()** retourne un array dont les clés sont les noms des colonnes et dont les valeurs sont les valeurs des colonnes.

La fonction **mysqli\_fetch\_row()** retourne un array dont les clés sont les index des colonnes et dont les valeurs sont les valeurs des colonnes.

Et la fonction **mysqli\_fetch\_array()** retourne... les deux, bien que ce comportement soit modifiable. Elle retourne par défaut un array où il y a toutes les associations **nom de colonne => valeur de la colonne**, ainsi que toutes les associations **index de la colonne => valeur de la colonne**. Toutes les données sont donc doublées, l'array est donc deux fois plus lourd !

C'est pourquoi je vous déconseille d'utiliser **mysqli\_fetch\_array()**. Qui plus est, vous verrez qu'il est rare, voire extrêmement rare d'utiliser les index des colonnes. En effet, il suffit de changer l'ordre de sélection des colonnes dans la requête et il faut tout changer... Et je ne vous parle même pas de la lisibilité, entre **\$res['gbook\_date']** et **\$res[4]**, la première version me semble quand même infiniment plus lisible, c'est pourquoi je ne peux que vous recommander l'utilisation de **mysqli\_fetch\_assoc()**.

Passons à la pratique !

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
mysqli_query($link, 'USE dev_dev;') or exit(mysqli_error($link));

$result = mysqli_query($link, 'SELECT gbook_id, gbook_msg,
gbook_date FROM dev_gbook;');

$tab = mysqli_fetch_assoc($result);
echo $tab['gbook_id'] . '<br />' . $tab['gbook_date'];

?>
```

Vous avez peut-être remarqué un détail... C'est qu'à aucun moment je n'ai indiqué quelle ligne je voulais utiliser. En effet, avec les **mysqli\_fetch\_\*** (), on ne peut pas utiliser n'importe quelle ligne quand ça nous chante.

Vous devez savoir que dans un objet **MySQLi\_Result**, il y a quelque chose qui revient souvent en programmation : un pointeur. Ce pointeur pointe (💡) sur une et une seule ligne du jeu de résultats représenté par l'objet **MySQLi\_Result**.

Quand vous appelez une des fonctions **mysqli\_fetch\_\*** (), la fonction déplace le pointeur sur la ligne suivante ; ainsi, lorsqu'on appellera à nouveau la fonction, elle nous retournera non pas la même ligne, mais bien la suivante.

On se dit donc que si on veut afficher tous les résultats, une petite boucle pourrait nous être très utile : eh c'est bien le cas. Seulement, quelle sera la condition pour sortir de la boucle ?

On pourrait compter le nombre de lignes dans la ressource (la fonction **mysqli\_num\_rows(\$result)** retourne le nombre de lignes que contient le jeu de résultats représenté par l'objet **MySQLi\_Result \$result**), mais ce serait négliger un détail important : le second type de retour des **mysqli\_fetch\_\*** () .

Eh oui, ces fonctions sont bien conçues et nous permettent de savoir si le pointeur trouve une ligne.

Je vous ai dit qu'à chaque appel des fonctions **mysqli\_fetch\_\*** (), le pointeur se déplace sur la ligne suivante. Mais que se passe-t-il quand le pointeur se trouve sur la dernière ligne ? La fonction déplace le pointeur sur une ligne qui n'existe pas puisqu'il n'y en a plus, et au prochain appel de la fonction, cette dernière tentera d'accéder à une ligne qui n'existe pas. Dans un pareil cas, la fonction retournera le booléen **false**.

Or, il n'y a que deux cas où le pointeur ne pointe sur aucune ligne :

- si on a déjà parcouru toutes les lignes ;

- ou s'il n'y a aucune ligne.

Bien souvent, on voit des gens qui vérifient si leur requête a retourné au moins une ligne avec la fonction `mysqli_num_rows()` ; or, on peut savoir s'il y a au moins une ligne en se basant uniquement sur la valeur de retour des fonctions `mysqli_fetch_*`.

Revenons donc à nos moutons. On ne sait toujours pas comment afficher toutes les lignes d'une ressource. Mais on possède plus d'éléments pour résoudre le problème. On sait maintenant que les fonctions `mysqli_fetch_*` nous retourneront `false` s'il n'y a plus de ligne, et on sait également qu'un array non-vide (ce qui est toujours le cas avec le retour de ces fonctions car sinon, on ne sélectionnerait rien du tout 😊) est évalué à `true`.

Ces deux choses doivent vous faire penser à une boucle en particulier : `while()`.

Voici devant vos yeux ébahis comment on affiche simplement toutes les lignes d'une ressource :

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
mysqli_select_db($link, 'dev_dev');

$result = mysqli_query($link, 'SELECT gbook_msg, gbook_date FROM
dev_gbook;');

while($line = mysqli_fetch_assoc($result))
{
    echo '<fieldset><legend>' . $line['gbook_date']
        . '</legend>' . $line['gbook_msg'] . '</fieldset>';
}

?>
```

Alors, qu'en pensez-vous ? 😊

J'explique brièvement la condition du `while()` : à chaque itération, on va affecter la valeur de retour de `mysqli_fetch_assoc()` à la variable `$line`. À chaque itération, la valeur de `$line`, c'est-à-dire la valeur de retour de `mysqli_fetch_assoc()`, sera évaluée à `true` s'il y a encore au moins une ligne dans la ressource et le bloc d'instructions de la boucle sera alors exécuté.

Si au contraire il n'y a plus de ligne, `mysqli_fetch_assoc()` retournera `false`, `$line` vaudra donc `false` et le bloc d'instructions de la boucle ne sera pas exécuté : on sort de la boucle.

Maintenant, on va provoquer une erreur :

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
mysqli_select_db($link, 'dev_dev');

$result = mysqli_query($link, 'SELECT gbook_msg, gbook_dat FROM
dev_gbook;');
print_r(mysqli_fetch_assoc($result));

?>
```

Et ce code vous retournera quelque chose de ce style :

#### Code : Console

```
Warning: mysqli_fetch_assoc() expects parameter 1 to be mysqli_result, boolean given
```

Cette erreur est monstrueusement courante, vous ne pouvez pas imaginer combien de sujets ont été créés à cause d'elle...

Mais je vous rassure : si vous suivez mon conseil de toujours mettre un **or exit(mysqli\_error())** après vos **mysqli\_query()**, vous n'aurez jamais, *au grand jamais*, cette erreur. En effet, posons-nous la question de l'origine de cette erreur.

D'après ce que PHP vous dit, l'erreur vient de **mysqli\_fetch\_assoc()**, OK. Regardons la ligne correspondante : on ne voit pas grand chose. Le seul élément qui pourrait être un début de réponse, c'est la variable **\$result**. Remontons donc jusqu'à cette variable, on voit qu'elle peut avoir deux types de valeurs : un objet **MySQLi\_Result** ou un booléen. On regarde le prototype de **mysqli\_fetch\_assoc()** et on voit qu'elle attend un objet **MySQLi\_Result**.

On croit lui en envoyer un, **mysqli\_query()** retourne pourtant un objet **MySQLi\_Result** ! Cependant, PHP nous dit que le type de la variable qu'on donne à **mysqli\_fetch\_assoc()** n'est pas correct, que ce n'est pas un objet

**MySQLi\_Result (expects parameter 1 to be mysqli\_result)** veut dire *le paramètre 1 attendu est un MySQLi\_Result*.

Qu'en déduit-on ? 😊

Que **mysqli\_query()** a retourné `false`. Et quand **mysqli\_query()** retourne-t-elle `false` ? Quand la requête a générée une erreur.

Et voilà la cause de l'erreur : c'est la requête qui plante. Souvenez-vous, je vous avais dit que si vous ne mettiez pas le **or exit(mysqli\_error())**, vous vous exposiez à des erreurs en cascade 🍑.

Si vous avez pensé à mettre ce petit truc, vous ne verrez donc jamais cette erreur car si erreur il y a, le **exit()** sera exécuté et l'erreur affichée, n'oubliez donc pas de le mettre et pensez à regarder vos requêtes si vous avez une erreur avec un **mysqli\_fetch\_\*()** 😊.

Avant d'en finir avec ceci, un dernier petit truc dont on a déjà parlé : je vais vous montrer comment vérifier s'il y a au moins une ligne dans un objet **MySQLi\_Result**. En effet, avec ce que je vous ai dit pour l'instant, vous seriez peut-être amenés à tenter ceci :

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
mysqli_query($link, 'USE dev_dev;') or exit(mysqli_error($link));

$req = mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;');

if($res = mysqli_fetch_assoc($req))
{
    while($res = mysqli_fetch_assoc($req))
    {
        echo $res['gbook_id'];
    }
}

?>
```

Si vous avez essayé, vous aurez constaté que ça ne fonctionne pas tout à fait. En effet, il manque la première ligne, la première ligne de la ressource n'est pas affichée, c'est tout à fait logique.

Traduisons ce code en français :

#### Code : Autre

On crée un array `$res` qui contient la première ligne du jeu de résultats  
On déplace le pointeur du jeu de résultats sur la seconde ligne  
Si on obtient un array :  
    On crée un array `$res` qui contient la n-ième ligne du jeu de résultats  
    On déplace le pointeur du jeu de résultats sur la ligne suivante  
    Tant qu'on obtient un array :

On affiche la valeur de la colonne

Comme vous le constatez, à aucun moment on ne demandera d'afficher la première ligne. En effet, l'expression du **while()** est évaluée avant que le bloc d'instructions ne soit exécuté.

Mais on a vu une autre boucle qui pourrait peut-être nous sauver la mise... **do {} while();**. Eh oui, à l'inverse de **while()**, cette boucle exécute le bloc d'instructions avant d'évaluer la condition !

Essayons donc...

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
mysqli_query($link, 'USE dev_dev;') or exit(mysqli_error($link));

$req = mysqli_query($link, 'SELECT gbook_id FROM dev_gbook;');

if($res = mysqli_fetch_assoc($req))
{
    do
    {
        echo $res['gbook_id'];
    }while($res = mysqli_fetch_assoc($req));
}

?>
```

Et miraculeusement, ça fonctionne 😊.

Vérifions ce qu'il se passe s'il n'y a aucune ligne (pour ne sélectionner aucune ligne, j'utilise une condition qui n'est jamais vraie 😞):

#### Code : PHP

```
<?php

$link = mysqli_connect('localhost', 'root');
mysqli_query($link, 'USE dev_dev;') or exit(mysqli_error($link));

$req = mysqli_query($link, 'SELECT gbook_id FROM dev_gbook WHERE
0;');

if($res = mysqli_fetch_assoc($req))
{
    do
    {
        echo $res['gbook_id'];
    }while($res = mysqli_fetch_assoc($req));
}
else
{
    echo 'Aucune ligne ne correspond à vos critères';
}

?>
```

Et voici qui termine cette partie sur l'exploitation des ressources MySQL !

Ce n'est pas parce qu'on fait du PHP qu'on doit faire de vilaines requêtes 😊.

Vous devez indenter vos requêtes, c'est d'une importance capitale. Pour ce chapitre, les requêtes sont ridiculement minuscules et tiennent donc sans mal sur une ligne (peut-être même sur moins de 100 caractères), mais quand on





commencera à faire des requêtes plus exotiques, vous devrez indenter pour arriver à lire quelque chose sans perdre cinq minutes à vous demander où sont le WHERE et le ORDER BY.

Vous êtes maintenant fin prêts pour utiliser PHP et MySQL conjointement : j'aurai sans doute encore quelques détails à rajouter par la suite, mais le plus gros est là 😊.

Comme promis, le prochain chapitre est un TP qui mettra à l'épreuve tout ce que vous avez vu jusqu'à maintenant : création de table, utilisation des formulaires, requêtes d'insertion, requêtes de suppression, requêtes de sélection et affichage d'un jeu de résultats de MySQL 😊.



Il se peut que pour communiquer avec un serveur MySQL, vous entendiez parler des extensions mysql (mysql\_connect(), mysql\_query(), etc.) et de PDO. Sachez que l'extension mysql est l'ancêtre de l'extension MySQLi ? qui signifie *MySQL Improved*, improved voulant dire *amélioré* ? et que PDO est une extension que nous verrons par la suite.

## Second TP : un visionneur de bases de données et de leurs tables

Ce TP va vous permettre de mettre en pratique une partie de tout ce qu'on a vu dans les chapitres précédents, mais c'est surtout votre capacité à faire travailler MySQL et PHP ensemble qui sera mise à l'épreuve.

Ce petit script, qui vous permettra de voir rapidement vos bases de données et vos tables, n'est ni long ni complexe, mais il sert à placer des bases supplémentaires pour que plus tard, on puisse vraiment s'amuser. 

### Objectifs et cycle de développement

Pour ce TP, nous allons faire quelque chose qui peut se révéler très utile : un visionneur de bases de données et de leurs tables. Comme son nom l'indique, cela nous permettra de voir rapidement quelles bases de données existent et quelles tables s'y trouvent.

Pour le moment, on va faire très simple, on se contentera d'afficher le nom des bases et des tables. On pourrait faire bien plus complet, mais le but de ce TP, en plus de vous faire pratiquer un peu l'utilisation conjointe du PHP et de MySQL, est de vous initier à un cycle de développement efficace.

Vous devez savoir que lorsqu'on développe un script ou un logiciel, il est très souhaitable de ne pas faire n'importe quoi n'importe quand. Il y a plusieurs étapes et suivre un ordre qui rendra le travail plus aisés est une bonne chose.

Dans tout développement, la première chose à faire, c'est de définir ce que l'on veut faire, le **but du logiciel**, sa finalité. Cette première étape a déjà été faite : nous voulons faire un script qui nous permettra de voir nos bases de données et nos tables. Normalement, l'étape suivante serait d'écrire formellement les **spécifications** de notre script, c'est-à-dire l'ensemble des fonctionnalités. Toutefois, vu la simplicité du script, c'est relativement inutile. Mais nous allons tout de même le faire !

Vous devez savoir que les spécifications sont d'une importance **capitale**. En effet, toute la suite du développement se basera là-dessus ! Mais pourquoi ?

Eh bien parce qu'on ne construit pas un script n'importe comment. Ce sont les spécifications qui vont définir ce qu'on utilisera (utilisera-t-on une base de données ou non ? Si oui, laquelle ? Devrons-nous faire attention à des détails particuliers ? Si oui, lesquels ? Etc.) et comment on l'utilisera. Pour prendre un exemple qui vous parlera peut-être davantage : si vous voulez faire un forum, vous allez d'abord rédiger les spécifications et à partir de cela, vous créerez toute votre base de données, sa structure. Et la conséquence directe est que si vous modifiez les spécifications, vous devrez tout modifier puisque tout se base dessus. Il est donc très important d'apporter le plus grand soin à la rédaction des spécifications. Voici donc les spécifications de notre script :

- quoi qu'il arrive, on affichera la liste des bases de données ;
- chaque nom de base de données sera accompagné d'un lien qui pointera vers la page courante en passant le nom de la base de données en paramètre `_GET` ;
- si l'utilisateur demande d'afficher les tables d'une base de données en particulier, on affichera lesdites tables ;
- pour demander d'afficher les tables d'une base de données en particulier, l'utilisateur passera le nom de la base de données voulue en `_GET` en utilisant les liens prévus à cet effet.

Je vous l'avais bien dit : les spécifications du script qui nous occupe sont ridiculement simples, mais dans de gros projets, il n'est pas rare que ce document soit parmi les plus gros. 

Après cette étape vient le temps de la **conception**. C'est donc maintenant qu'on va se demander « Comment ? ». Pour reprendre l'exemple du forum, c'est maintenant qu'on penserait la structure de toutes nos tables, les liens qu'il y aura entre elles, etc. Juste après cela, on penserait à l'organisation de nos scripts (oui, on ne fait pas un seul script, il y en aurait plusieurs !), à leurs interactions, leurs similitudes pour éventuellement en fusionner, etc.

Pour le script qui nous occupe, cette étape est encore extrêmement simple. En effet, nous n'aurons pas besoin d'utiliser une base de données : on lira des bases de données mais nous ne stockerons rien, nous n'avons donc pas besoin d'une base de données. De plus, comme l'application se réduit à deux fonctionnalités, elle ne sera composée que d'un seul script, ce qui simplifie encore la tâche. La conception de notre script se résume donc à dire qu'il n'y aura qu'un seul fichier qui se chargera de tout. 

Passons maintenant à une étape qui va faire l'objet d'une étude plus sérieuse : la création de l'**algorithme**. Toutefois, vu l'importance que je vais accorder à l'algorithme du script, nous reviendrons là-dessus un tout petit peu plus tard. Pour l'instant, sachez juste qu'un algorithme est un moyen de résoudre un problème (ici, le problème étant d'afficher la liste des bases de données et éventuellement la liste des tables).

La cinquième étape de développement est la résolution du problème. L'étape précédente nous donne la méthode pour résoudre le problème, mais cette méthode est générique, elle n'est pas spécifique au langage qui nous intéresse. C'est pour ça qu'il faut traduire cet algorithme dans le langage qui nous intéresse, on appelle cela l'**implémentation**. Cette étape doit se baser exclusivement sur les précédentes ! On n'improvise pas l'implémentation, on se contente de suivre l'algorithme.

Et nous voici à la fin du cycle de développement du logiciel. 

Mais ce n'est pas fini pour autant. En effet, notre logiciel est écrit, tout beau tout neuf, mais... ne faudrait-il pas le tester ?

Eh bien si, il va falloir le tester encore, encore et encore pour s'assurer qu'il fonctionne comme les spécifications le décrivent et

qu'il n'y a pas de bug ici ou là.

Cette étape se fait en deux parties.

D'abord, on teste le logiciel, on teste les fonctionnalités. Si on rencontre des bugs, on en prend note ; si on voit que le logiciel ne respecte pas les spécifications, on en prend note.

La seconde partie se base sur la première et a pour but de corriger ce qui a été relevé dans la partie précédente. S'il y a des bugs, il va falloir trouver leur origine et les corriger. Si le logiciel ne respecte pas les spécifications (normalement, c'est assez rare), il faudra le modifier plus en profondeur.

Une fois cette étape finie, on recommence. 😊

Eh oui, il est très rare qu'on trouve toutes les erreurs du premier coup, c'est pour ça qu'il est préférable de recommencer une ou deux fois avec des personnes différentes. Pourquoi donc des personnes différentes ? Tout simplement parce que tout le monde ne teste pas les logiciels de la même façon, que parfois on passe à côté de certaines choses.

Une fois que le logiciel est jugé stable, c'est-à-dire qu'on ne rencontre plus de bug et qu'il respecte les spécifications, on arrive à l'avant-dernière étape : la production. Cette étape consiste simplement à donner le script à ceux qui l'ont commandé pour qu'ils puissent l'utiliser.

La dernière étape est tout simplement le suivi. C'est bien beau d'avoir donné un logiciel, mais si le client rencontre des difficultés, s'il trouve des bugs, il faut bien l'aider et éventuellement le modifier un peu.

Et voilà que se termine ce petit tour d'horizon de la création et de l'utilisation d'un logiciel du point de vue du codeur. Veillez à suivre ce genre de cycle, ça aide énormément.

## L'étape d'algorithme

Comme promis, revenons en long, en large et en travers sur une étape du cycle de développement : l'algorithme. Je vous ai dit précédemment que l'algorithme était, grossièrement, un moyen de résoudre un problème.

En réalité, cette « définition » est vraiment boiteuse. On devrait plutôt dire qu'un algorithme décrit la méthode employée pour résoudre un problème. L'algorithme en lui-même ne résoudra pas le problème, il ne servira qu'à aider les personnes chargées de l'implémentation. Sans algorithme, les codeurs devraient se demander comment ils vont faire telle ou telle chose, dans quel ordre, etc. L'algorithme répond à toutes ces questions, il permettra aux codeurs de n'avoir qu'à le lire et à le traduire.

Pour qu'un algorithme soit facile à comprendre, on utilise un langage très simple appelé **pseudo-code**. Ce pseudo-code est extrêmement proche du français (ou de l'anglais pour les anglophones) mais comporte quelques symboles, structures, etc. pour s'approcher un peu des langages de programmation. Voici par exemple un algorithme très simple qui décrit comment calculer la moyenne des nombres présents dans un tableau :

### Code : Autre

```
$somme <- 0
$nombreElement <- 0
POUR CHAQUE élément DE tableau de nombre
    $somme <- $somme + élément
    $nombreElement <- $nombreElement + 1
SI $nombreElement > 0
    $moyenne <- $somme / $nombreElement
SINON
    AFFICHER "moyenne indéfinie"
```

Bon, je vais vous expliquer quelques bases de ce pseudo-code de ma création (il n'y en a pas d'universel, on utilise en général celui qui convient le plus à l'utilisation qu'on en fera).

Tout d'abord, dans tout algorithme, il y a des variables. Dans mon pseudo-code, les variables sont nommées comme en PHP, c'est-à-dire qu'elles commencent par le symbole \$ et qu'elles respectent les [conventions de nommage de PHP](#).

S'il y a des variables, il faut bien leur donner des valeurs : pour cela, on a vu l'opérateur d'affectation. Dans mon pseudo-code, l'opérateur d'affectation est le symbole <- . L'avantage de ce symbole, c'est qu'il indique le sens de l'affectation : la flèche va vers la gauche, on comprend donc aisément que c'est la valeur à droite de l'opérateur qui est affectée à la variable à gauche de l'opérateur.

Pour ce qui est des opérateurs arithmétiques (addition, soustraction, etc.), on utilise les mêmes tant en pseudo-code qu'en PHP, c'est-à-dire les bons vieux +, -, /, etc.

Enfin, on peut voir qu'il y a des mots en majuscules : ce n'est pas pour insister dessus mais c'est pour indiquer une structure.

Parmi les plus connues, il y a le **SI**, **SINON**, **SI**, **SINON**, **TANT QUE**, **POUR**, etc.

Sachant tout cela, il devient assez facile de comprendre ce pseudo-code et de l'implémenter en PHP :

### Code : PHP

```
<?php
```

```
$somme = 0;
$nombreElement = 0;
foreach($tableauNombre as $element)
{
    $somme += $element;
    $nombreElement++;
}
if($nombreElement > 0)
{
    $moyenne = $somme/$nombreElement;
}
else
{
    echo 'Moyenne indéfinie';
}

?>
```

C'est très ressemblant, on pourrait donc se demander s'il n'est pas plus intéressant de l'écrire directement en PHP. Mais non, ce n'est pas plus intéressant, pour deux raisons.

Premièrement, s'il est vrai que cet algorithme est extrêmement simple, d'autres le sont beaucoup moins. Or, il est beaucoup plus simple de raisonner en bon vieux français que dans un langage de programmation, la syntaxe du français étant plus « simple » puisque c'est celle qu'on utilise tous les jours.

Deuxièmement, un avantage énorme du pseudo-code, c'est qu'il est portable. En effet, le pseudo-code peut être lu par tous, nul besoin de savoir coder en PHP pour écrire et lire un algorithme. Ainsi, si vous écrivez votre algorithme en PHP, qu'est-ce qui vous dit que celui qui voudra utiliser votre algorithme comprend le PHP ?

De plus, ça permet aussi à ceux qui ne connaissent pas les syntaxes des langages de programmation de participer à la création de logiciels. Après tout, pourquoi ne pas envisager une équipe dans laquelle une personne sera chargée d'une seule étape, celle de l'algorithme ? S'il excelle dans ce domaine, ça serait stupide de s'en passer !

## Quelques outils

Avant de vous lancer dans ce TP, je vais vous donner trois outils qui vous seront fort utiles.

Le premier, c'est une fonction. En effet, quand un utilisateur vous demandera d'afficher la liste des tables d'une base de données, il serait bon de vérifier que le nom de la base de données est formaté comme il faut. Ce format dépend de vous, il est normalement décrit dans les spécifications du script (eh oui, vous voyez, ça arrive à tout le monde d'oublier un bout des spécifications, mais heureusement ici ce n'est pas trop grave 😊).

Je décide donc que pour qu'un nom de table soit jugé valide, il ne pourra contenir que des caractères alphabétiques minuscules ou majuscules, ainsi que des *underscores* (\_).

Pour vérifier qu'une chaîne de caractères ne contient que des caractères alphabétiques, c'est très simple, une fonction toute prête est là : **ctype\_alpha(string \$str)**. Cette fonction retourne true si la chaîne \$str ne contient que des caractères alphabétiques, sinon elle retourne false.

 Il existe plusieurs fonctions ctype, elles sont très utiles pour vérifier qu'une chaîne de caractères ne contient que certains caractères. Par exemple, la fonction **ctype\_digit()** permet de vérifier que la chaîne de caractères ne contient que des chiffres. La liste des fonctions ctype se trouve bien entendu dans la documentation officielle.

Toutefois, vu que le format autorise aussi l'*underscore*, cette fonction ne nous aidera pas...

Vous me croyez quand je dis ça ? 😊

Imaginons que vous ayez la chaîne de caractères **lala\_lili\_**, êtes-vous d'accord qu'elle respecte le format mais que **ctype\_alpha()** renverra false ?

Maintenant, enlevons tous les *underscores*, on obtient la chaîne **lalalili**. Et que renverra **ctype\_alpha()** ? Assurément **true**. On peut donc dire que si on enlève tous les *underscores* et que **ctype\_alpha()** renvoie **true**, alors la chaîne de caractères de départ respecte le format.

Supprimer un caractère, ce n'est pas directement possible. Mais êtes-vous d'accord si je dis que supprimer un caractère revient à le remplacer par une chaîne vide ? Eh bien c'est ce qu'on va faire.

En PHP, une fonction nous permet de remplacer un ou plusieurs caractères par d'autres, c'est la fonction **str\_replace()**, dont voici le prototype (volontairement amputé d'un paramètre) :

### Code : Console

```
string str_replace(mixed $search, mixed $replace, mixed $subject)
```

Le paramètre `$search` peut être soit une chaîne de caractères, soit un tableau. Si on donne une chaîne de caractères, on remplacera cette chaîne de caractères. Si on donne un tableau, on remplacera toutes les chaînes de caractères contenues dans le tableau.

Le paramètre `$replace` indique par quoi `$search` sera remplacé. Ce paramètre peut également être une chaîne de caractères, ou un tableau.

Le paramètre `$subject` est simplement la chaîne de caractères (on peut aussi lui donner d'autres choses comme des nombres, mais ils seront convertis en chaîne de caractères) dans laquelle on va effectuer le remplacement.

Voici des exemples avec cette fonction pour comprendre l'utilisation des arrays en paramètre :

#### Code : PHP

```
<?php

$str = 'lala lili';
echo str_replace('a', 'i', $str); // on remplace a par i, ça
affichera donc 'lili lili'
echo '<br />';
echo str_replace(array('i', 'l'), 'z', $str); // on remplace i ET l
par z, ça affichera donc 'zaza zzzz'
echo '<br />';
echo str_replace('il', 'z', $str); // ATTENTION ! Cette fois c'est
la chaîne de caractères <gras>il</gras> qui est remplacée par z, ce
n'est pas du tout la même chose que la dernière fois, ça affichera
'lala lzi'
echo '<br />';
echo str_replace(array('a', 'i'), array('', 'l'), $str); // on
remplace a par une chaîne vide et i par l, ça affichera 'll llll'

?>
```

En conclusion, voici la fonction qui permettra de vérifier qu'un nom de base de données respecte le format :

#### Code : PHP

```
<?php

function isValid($str)
{
    return ctype_alpha(str_replace('_', '', $str));
}

?>
```

Le second outil est une requête SQL. En effet, quand vous allez vouloir afficher la liste des tables d'une base de données, ne serait-il pas bienvenu de pouvoir vérifier que la base de données existe ?

Il y a moyen de faire autrement, mais pour rester dans quelque chose que vous connaissez, voici une requête qui retournera le nom de la base de données dont vous voulez tester l'existence :

#### Code : SQL

```
SELECT schema_name
FROM information_schema.schemata
WHERE schema_name = 'nomBaseDeDonneeATester';
```

Si la base de données existe, cette requête retournera une ligne, sinon elle n'en retournera aucune.

Le troisième outil devrait vous être également très utile, et pour cause : sans lui, vous sauteriez sans parachute. 😊

Lors du précédent TP, je vous avais donné un petit texte qui expliquait ce que le script ferait ; cette fois, pour appuyer sur l'importance d'un bon cycle de développement, je vais vous donner l'algorithme du script. 😊

Une grande partie de ce TP sera donc d'arriver à comprendre et à traduire cet algorithme en PHP :

#### Code : Autre

```

/* Déclaration des constantes */
Déclarer la constante MYSQL_HOST
Déclarer la constante MYSQL_USER
Déclarer la constante MYSQL_PASS
/* Déclaration des fonctions */
Déclarer la fonction isValid() donnée dans l'énoncé
/* Traitement des données */
Connexion à MySQL en utilisant les constantes précédemment définies

$dbResult <- ressource contenant la liste des bases de données
$dbList <- array vide
TANT QUE on n'a pas parcouru toute la ressource
    $dbList[] <- $nomBaseDeDonnee

$showTbl <- booléen indiquant si on veut afficher les tables d'une base de donné
SI $showTbl
    SI le nom de la base de données est valide
        $req <- ressource retournée par la requête donnée dans l'énoncé
        SI il y a une ligne dans $req
            $tblResult <- ressource contenant la liste des tables de la base de
            $tblList <- array vide
            TANT QUE on n'a pas parcouru toute la ressource
                $tblList[] <- $nomTable
        SINON
            $errMsg <- 'Cette base de données n'existe pas'
    SINON
        $errMsg <- 'Le nom de la base de données n'est pas valide'
/* Affichage des données */
SI il y a des bases de données à afficher
    AFFICHER 'Liste des bases de données'
    POUR CHAQUE base de données
        AFFICHER nom de la base de données et un lien (<a href=". /index.php?
db=nomBaseDeDonnees"></a>)
SINON
    AFFICHER 'Aucune base de données n'existe'

SI on demande d'afficher des tables
    AFFICHER 'Vous avez demandé à voir la liste des tables de la base de données'
    SI il y a un message d'erreur
        AFFICHER $errMsg
    SINON SI il n'y a pas de table à afficher
        AFFICHER 'Il n'y a aucune table dans cette base de données'
    SINON
        POUR CHAQUE table
            AFFICHER nom de la table

```

Pour faciliter votre compréhension, voici un ou deux indices : **TANT QUE** est un **while()**, **POUR CHAQUE** est un **foreach()**, et **\$var[] <-** indique qu'on affecte la valeur de l'opérande de droite dans une nouvelle association du tableau **\$var** (on a vu l'opérateur **[]** dans la première partie).

En bonus, un petit cadeau : le nom des colonnes retournées par les requêtes qui donneront la liste des bases et des tables sont facilement reconnaissables en exécutant la requête dans la console MySQL ; toutefois, si vous n'y arrivez pas, vous n'aurez qu'à utiliser **mysql\_fetch\_row()** au lieu de **mysql\_fetch\_assoc()**, et le nom des bases et des tables sera accessible à l'index 0.

Et voilà, il ne me reste plus qu'à vous souhaiter bonne chance ; ne baissez pas les bras même si ça paraît difficile. 😊

#### Correction

J'espère pour vous que vous lisez ces lignes après avoir brillamment réussi ou sué sang et eau pendant des heures pour trouver la solution, sinon vous souffrirez le martyre pendant des siècles. 🍸

Faire un copier-coller de cette correction ne vous servira à rien, la comprendre si vous n'avez pas réussi ou la comparer à votre code si vous avez réussi vous sera par contre très profitable.

**Code : PHP**

```
<?php

error_reporting(E_ALL);

/* Déclaration des constantes */

define('MYSQL_HOST', 'localhost');
define('MYSQL_USER', 'root');
define('MYSQL_PASS', '');

/* Déclaration des fonctions */

function isValid($str)
{
    return ctype_alpha(str_replace('_', '', $str));
}

/* Traitement des données */

$link = mysqli_connect(MYSQL_HOST, MYSQL_USER, MYSQL_PASS);

// On récupère la liste des bases de données
$dbResult = mysqli_query($link, 'SHOW DATABASES;') or
exit(mysqli_error($link));
$dbList = array();
while($db = mysqli_fetch_row($dbResult))
{
    $dbList[] = $db[0];
}

// Si on a demandé à afficher la liste des tables d'une base de
données
// On va vérifier si la base de données existe
// Si c'est le cas, on récupérera la liste des tables de cette base
de données
// Sinon, on renverra une erreur
$showTbl = isset($_GET['db']) && trim($_GET['db']);
if($showTbl)
{
    if(isValid($_GET['db']))
    {
        $req = "SELECT schema_name
FROM information_schema.schemata
WHERE schema_name = '" . $_GET['db'] . "'";
        $req = mysqli_query($link, $req) or
exit(mysqli_error($link));
        if(mysqli_fetch_assoc($req))
        {
            $tblResult = mysqli_query($link, 'SHOW TABLES FROM ' .
$_GET['db'] . ';') or exit(mysqli_error($link));
            $tblList = array();
            while($tbl = mysqli_fetch_row($tblResult))
            {
                $tblList[] = $tbl[0];
            }
        }
        else
        {
            $errMsg = 'Cette base de données n\'existe pas';
        }
    }
    else
    {
        $errMsg = 'Le nom de la base de données n\'est pas valide';
    }
}
```

```

/* Affichage des données */

// On commence par afficher la liste des bases de données
if($dbList)
{
    echo 'Liste des bases de données :<br /><ul>';
    foreach($dbList as $db)
    {
        echo '<li><a href="index.php?db=' . $db . '">' . $db . 
'</a></li>';
    }
    echo '</ul><br /><br />';
}
else
{
    echo 'Aucune base de données n\'existe';
}

// Ensuite, on affiche la liste des tables si on l'a demandée
if($showTbl)
{
    echo 'Vous avez demandé à voir la liste des tables de la base de
données <strong>' . htmlspecialchars($_GET['db']) . '</strong> :<br
/>';
    if(!empty($errMsg))
    {
        echo '<p style="color:red;">Une erreur est survenue : ' .
$errMsg . '</p>';
    }
    elseif(!$tblList)
    {
        echo '<p style="color:red;">Il n\'y a aucune table dans
cette base de données.</p>';
    }
    else
    {
        echo '<ul>';
        foreach($tblList as $tbl)
        {
            echo '<li>' . $tbl . '</li>';
        }
        echo '</ul>';
    }
}
?>

```

Alors, pour commencer, vous vous êtes peut-être demandés pourquoi je vous ai dit d'utiliser des constantes pour vous connecter à votre base de données. La raison est simple : c'est plus pratique. Pour le moment, on n'utilise qu'un seul fichier, mais par la suite, nous en utiliserons plusieurs. Or, s'il est rapide de modifier une fois les paramètres de connexion à MySQL, il est fastidieux de le faire autant de fois qu'il y a de fichiers. Nous verrons plus tard comment il est possible d'utiliser des constantes dans plusieurs fichiers différents. Ainsi, il suffira de modifier un seul fichier pour que toutes les connexions à MySQL soit correctes, un gain de temps énorme !

Pour le reste, comme vous le constatez, j'ai respecté mon algorithme à la lettre.

La seule chose qui peut poser problème, c'est le transtypage. Regardez la ligne n° 34 : vous devez savoir que la fonction **trim()** retire tous les espaces blancs au début et à la fin de la chaîne, ça évite d'avoir un nom de base de données vide. Pour être plus explicite, j'aurais dû écrire `trim($_GET['db']) != "`, mais comme je vous l'ai dit, une chaîne vide est évaluée à false, ce que j'ai écrit est donc tout à fait correct mais un peu moins clair (c'est volontaire 😊).

Regardez également les lignes n° 66 et n° 88 : j'utilise encore le transtypage implicite. Explicitement, j'aurais dû écrire `$showTbl != array()`, mais comme un array vide est évalué à false, c'est inutile, bien que plus clair.

Et voici déjà la fin de ce TP : vous avez réalisé un petit script pas très compliqué mais rudement utile pour voir rapidement et simplement quelles sont les bases de données et les tables présentes.

Bien évidemment, c'est rudimentaire, très rudimentaire, même, on peut faire beaucoup plus de choses ; en fait, on peut complètement administrer ses bases de données avec une interface en PHP. Rendez-vous donc à l'adresse <http://localhost/phpmyadmin/> ou <http://127.0.0.1/phpmyadmin/>. 😊

Eh oui, depuis le début on aurait pu apprendre à se servir des bases de données avec ce gros script installé par défaut avec WAMP, EasyPHP et compagnie. Mais j'ai préféré vous torturer à tout faire à la main. 😊

Mais surtout, ça m'a permis de vous apprendre à vous servir du SQL et pas simplement à cliquer bêtement sur des boutons. Direction le prochain chapitre. 😊

## Fonctions, conditions et quelques opérateurs SQL

Jusqu'à présent, nous nous sommes contentés de requêtes assez simples ; or, vous devez savoir qu'un SGBDR offre beaucoup plus de possibilités que ce que vous avez vu jusqu'à présent.

Mais avant de s'embarquer dans des requêtes de fous, il faut commencer par les bases, bases incomplètes pour le moment. Le but de ce chapitre est donc de vous initier à l'utilisation des fonctions en SQL, de vous montrer toute une panoplie d'opérateurs qui vous permettront d'exploiter votre SGBDR et de vous apprendre à utiliser à bon escient les conditions en SQL.

### Les fonctions, de bons souvenirs

Tout comme PHP, SQL met à votre disposition un grand nombre de fonctions de toutes sortes (des fonctions relatives aux mathématiques, aux chaînes de caractères, aux dates, aux statistiques, etc.).

Il vous faut également savoir que SQL permet de créer ses propres fonctions. Cependant, je ne vous montrerai pas comment créer des fonctions et utiliser ces dernières. Pour expliquer ce choix, je ne citerai que deux raisons : je ne maîtrise pas complètement ce sujet et ce n'est pas indispensable pour le moment.

Bref, revenons à nos fonctions. Vous avez de la chance, si vous savez utiliser des fonctions en PHP, vous savez utiliser des fonctions en SQL 😊.

Les mêmes règles s'appliquent tant en PHP qu'en SQL :

- une fonction possède des paramètres et retourne une valeur ;
- pour donner plusieurs paramètres à une fonction, il suffit de les séparer par une virgule ;
- tous les paramètres que vous donnez à une fonction sont des expressions ;
- les fonctions sont des expressions, vous pouvez donc les utiliser dans des opérations et les renommer avec AS.

Et voilà, j'en ai fini avec la théorie sur les fonctions en SQL ; elle est pas belle, la vie ? 😊

Je ne vais pas passer mon temps à énoncer toutes les fonctions qui existent, il y en a bien trop. Nous allons plutôt faire quelque chose de constructif et apprendre à nous servir de la documentation SQL, et plus exactement la documentation de MySQL (si vous utilisez un autre SGBDR, vous devrez bien évidemment utiliser la documentation adéquate).

L'index de la documentation se trouve à [cette adresse](#) (si vous comprenez l'anglais, je vous invite à utiliser la [documentation anglaise](#) qui est mieux fournie que la version française). Tout comme la documentation du PHP, la documentation de MySQL se découpe en plusieurs chapitres, sous-chapitres et compagnie.

Dans le petit encadré tout à droite sur la page de l'index, vous pouvez voir les chapitres principaux de cette documentation : ils vont de l'installation de MySQL à l'administration de celui-ci. Regardez donc le chapitre 11, ça devrait vous rappeler quelque chose 😊.

Mais celui qui nous intéresse est le douzième : les fonctions à utiliser dans les clauses **SELECT** et **WHERE**. Cliquez donc sur ce lien et vous voici sur le chapitre dédié aux fonctions que MySQL vous fournit de base. Vous pouvez constater que les opérateurs sont également repris dans ce chapitre.

Choisissons un chapitre au hasard, le 12.3 par exemple : les fonctions de chaînes de caractères. Cette nouvelle page qui s'ouvre sous vos yeux reprend l'intégralité des fonctions liées aux chaînes de caractères. Je pense qu'il n'y a pas grand-chose à en dire, la page parle d'elle-même. Les fonctions sont listées de façon alphabétique, vous avez la description de chaque fonction suivie d'un ou plusieurs exemples d'utilisation de ladite fonction.

Allons voir du côté de la fonction **SUBSTRING()**. Tout d'abord, on trouve les prototypes de la fonction :

#### Code : Console

```
SUBSTRING(str, pos), SUBSTRING(str FROM pos), SUBSTRING(str, pos, len), SUBSTRING(str
```

En effet, une fonction peut avoir plusieurs prototypes, vous en avez d'ailleurs un bel exemple avec cette fonction qui peut s'appeler de quatre façons différentes. Les prototypes fournis par la documentation MySQL sont malheureusement moins complets que ceux de la documentation PHP. En effet, ni le type de retour de la fonction ni le type des arguments n'est donné. Pour connaître le type d'argument ou de retour de la fonction, vous devrez lire la description de la fonction ou vous baser sur le nom des paramètres. Ainsi, **str** indique une chaîne de caractères ; de même, **pos** et **len** indiquent des nombres entiers (une position et une longueur sont des nombres entiers).

Vous pouvez également connaître le type de retour et le type des arguments en regardant les exemples (ce que je vous invite toujours à faire). Regardons de plus près le troisième exemple de **SUBSTRING()** :

#### Code : Console

```
mysql> SELECT SUBSTRING('Quadratically', 5, 6);
```

```
-> 'ratica'
```

Les exemples de code sont toujours divisés en deux parties distinctes. La première, la ligne commençant par **mysql>**, montre la requête qu'on utilise. La seconde (la ligne commençant par **->**) montre quant à elle le retour de la requête.  
Ainsi, en voyant que la requête d'exemple me retourne '**ratica**', je comprends que la fonction **SUBSTRING()** retourne bel et bien une chaîne de caractères.

Voici qui termine cette brève présentation de la documentation de MySQL. Je vous conseille vivement de parcourir la liste des fonctions à propos des chaînes de caractères et des mathématiques.

## Un cas particulier : **NULL**

Vous rappelez-vous du temps lointain où vous étiez des débutants complets en SQL ? 😊

Dans le chapitre sur la création d'une base de données et d'une table, je vous ai parlé d'une valeur en vous disant qu'on y reviendrait par la suite. Cette valeur, c'est ce fameux **NULL**, mystique pour certains, magique pour d'autres.

On va faire un petit test pour vous convaincre que cette valeur est assez spéciale. Imaginez que vous alliez à une exposition, elle est composée de plusieurs salons dans lesquels, à un moment donné, il y a un nombre donné de visiteurs. Seulement, comment ferez-vous pour différencier un salon dans lequel il n'y a pas de visiteur et un salon dans lequel **il n'y a jamais eu de visiteur** ? Vous pourriez utiliser une colonne supplémentaire qui prendrait la valeur 0 s'il n'y a eu aucun visiteur, et la valeur 1 s'il y en a déjà eu.

Cependant, c'est négliger une possibilité des SGBDR : **NULL**. Si je crée une table à deux colonnes, une pour le nom du salon de type **VARCHAR(50)** et une autre pour le nombre de visiteurs de type **TINYINT**, me croyez-vous si je vous dit que ça suffit pour savoir s'il y a déjà eu des visiteurs ?

Non ? Vous devriez. 🤔

En effet, si ma colonne qui contient le nombre de visiteurs peut être nulle, alors j'aurai deux états possibles pour cette colonne : nulle ou non-nulle. De plus, une valeur non-nulle pourra être un nombre entier quelconque ! Et donc au final, cette seule colonne me permet de savoir s'il y a déjà eu des visiteurs et le nombre de visiteurs qu'il y a actuellement.

Essayez donc de créer cette table par vous-mêmes :

**Secret** (cliquez pour afficher)

**Code : SQL**

```
CREATE TABLE dev_salon (
    sal_name VARCHAR(50) NOT NULL,
    sal_count TINYINT UNSIGNED DEFAULT NULL
);
```

Et voici quelques données de test :

**Code : SQL**

```
INSERT INTO dev_salon (sal_name, sal_count)
VALUES ('voiture', 54), ('bateau', DEFAULT), ('calèche', 30),
('vélo', 0), ('trottinette', NULL), ('roller', 0);
```

Petite nouveauté : l'utilisation du mot-clé **DEFAULT**. Si vous regardez la définition de la table, vous voyez que j'ai indiqué que **NULL** serait la valeur par défaut. Le mot-clé **DEFAULT** indique à MySQL qu'on veut insérer la valeur par défaut ; ainsi, écrire **('bateau', DEFAULT)** est équivalent à écrire **('bateau', NULL)** puisque **NULL** est la valeur par défaut.

Maintenant, imaginons que vous vouliez avoir la liste des salons n'ayant jamais été visités, vous seriez tentés de faire ceci :

**Code : SQL**

```
SELECT sal_name
FROM dev_salon
WHERE sal_count = NULL;
```

Cependant, MySQL s'obstine à vous renvoyer un jeu de résultats vide (Empty Set). 😬

Je vous rassure, votre PC n'a aucun problème, votre version de MySQL n'est pas pourrie et MySQL est très bien installé. Si NULL est une valeur si particulière, c'est parce qu'elle représente l'absence de valeur. Un NULL signifie que la colonne n'a aucune valeur.

Et comment feriez-vous pour comparer une colonne où il n'y a aucune valeur ? On ne peut comparer que des valeurs ! Par ailleurs, MySQL est très gentil, la plupart des SGBDR vous auraient tout simplement renvoyé une erreur de syntaxe. Mais heureusement, la norme SQL a prévu un opérateur, **IS NULL**, et son opposé, **IS NOT NULL**.

Ainsi, pour sélectionner les salons où il n'y a jamais eu de visiteur, on écrira :

#### Code : SQL

```
SELECT sal_name
FROM dev_salon
WHERE sal_count IS NULL;
```

Tandis que pour avoir ceux où il y a déjà eu des visiteurs, on écrira :

#### Code : SQL

```
SELECT sal_name
FROM dev_salon
WHERE sal_count IS NOT NULL;
-- Vous pouvez aussi utiliser WHERE sal_count >= 0
-- Ça fonctionnera, mais je vous le déconseille
-- En effet, pour commencer c'est moins clair
-- Mais surtout, IS NOT NULL est prévu pour, c'est donc ça qu'il
faut utiliser
```

Puisqu'on a vu comment se servir des fonctions il y a peu de temps, je vais vous en présenter trois :

- COALESCE ;
- IFNULL ;
- NULLIF.

Je ne sais pas par laquelle commencer tant elles sont spéciales. 😊

Faisons cela dans l'ordre alphabétique ; notre première victime est donc COALESCE. Alors, accrochez-vous à votre siège, fixez attentivement votre écran et lisez cette phrase : cette fonction retourne le premier paramètre non-nul qu'on lui donne et on peut donner autant de paramètres qu'on veut.

Le plus simple pour comprendre, c'est de faire des exemples :

#### Code : SQL

```
SELECT COALESCE(NULL, 0); -- retourne 0
SELECT COALESCE(NULL, NULL, 'a', NULL, 7); -- retourne 'a'
SELECT COALESCE(4.3, NULL, 't'); -- retourne 4.3
```

Personnellement, j'utilise assez rarement cette fonction, et pour cause, je n'ai jamais qu'une colonne dont je veux tester la nullité, mais il est toujours utile de savoir s'en servir.

La fonction suivante, IFNULL, est plus souvent utilisée. Vous pouvez la voir comme une version allégée de COALESCE. En effet, cette fonction prend deux paramètres et a le comportement suivant :

- si le premier paramètre est nul, elle retourne le second paramètre ;
- si le premier paramètre n'est pas nul, elle retourne le premier paramètre.

Vous voyez donc que c'est exactement le comportement de COALESCE (on retourne la première valeur non-nulle). On va tout de suite se servir de cette fonction. Reprenons l'exemple des salons : je veux une requête qui va me retourner deux

colonnes ; la première contiendra le nom du salon, et la seconde contiendra la chaîne 'Jamais visité' s'il n'y a jamais eu de visiteur, sinon elle contiendra le nombre de visiteurs dans le salon.

Alors, vous vous en sortez ? 😊 Ne trichez pas. 🍫

#### Secret (cliquez pour afficher)

##### Code : SQL

```
SELECT sal_name, IFNULL(sal_count, 'Jamais visité')
FROM dev_salon;
```

Je n'ai personnellement jamais utilisé la troisième fonction, NULLIF, mais tant qu'on y est, parlons-en. Cette fonction ? un peu spéciale ? prend deux paramètres et a le comportement suivant :

- si les deux paramètres sont égaux, alors elle retourne NULL ;
- si les paramètres sont différents, elle retourne le premier paramètre.

Petit exemple :

##### Code : SQL

```
SELECT NULLIF(4, 5); -- retourne 4
SELECT NULLIF('a', 'a'); -- retourne NULL
```

## Les conditions

Si vous avez bien lu le texte précédent, vous avez dû remarquer que les fonctions IFNULL et NULLIF ont un comportement qui rappelle les conditions, et pour cause : on classe ces deux fonctions dans la famille des conditionnelles. 😊

En SQL, les conditions ne sont pas pareilles qu'en PHP. En PHP, on exécutait ou non des blocs d'instructions en fonction de nos conditions ; or, en SQL, il n'y a pas de bloc d'instructions (du moins, pas dans ce cours 🍫). Les conditions que nous allons voir sont donc plus restreintes, leur seul but est de tester et de retourner des valeurs en fonction du résultat du test. Vous allez très vite comprendre.

La norme SQL prévoit deux méthodes conditionnelles, une structure et une fonction : CASE et NULLIF(). Mais dans le chapitre précédent, nous avons utilisé une autre fonction conditionnelle : IFNULL(). En effet, comme bien souvent, MySQL se dégage du lot en changeant la norme ou en y ajoutant des éléments ; c'est pour ça que MySQL nous offre deux fonctions conditionnelles supplémentaires : IF() et IFNULL(). Il est bien évidemment déconseillé de les utiliser, étant donné que si vous les utilisez et que vous changez de SGBDR, vous risquez de devoir modifier vos requêtes, d'autant plus que ces deux ajouts sont très faciles à obtenir avec ce que la norme nous propose.

Avant de passer au gros morceau, CASE, nous allons expédier la fonction IF(). Cette fonction prend trois paramètres ; si le premier est évalué à vrai, on retourne le second paramètre, sinon on retourne le troisième paramètre.  
Reprendons l'exemple de nos salons. Cette fois, je veux une requête qui me retourne deux colonnes : la première contiendra le nom du salon, et la seconde contiendra la chaîne 'Déjà visité' s'il y a déjà eu des visiteurs, sinon elle contiendra la chaîne 'Jamais visité'.

Indice : pensez à IS NULL et IS NOT NULL. 😊

#### Secret (cliquez pour afficher)

##### Code : SQL

```
SELECT sal_name, IF(sal_count IS NOT NULL, 'Déjà visité', 'Jamais
visité')
FROM dev_salon;
```

Après cette mise en bouche, nous allons nous attaquer à la vraie structure conditionnelle, le « pur-sang » comme certains l'appellent : CASE.

Voici sa structure :

**Code : SQL**

```
CASE [Expr1]
  WHEN Expr2 THEN Expr3
  [ WHEN Expr4 THEN Expr5
  [ WHEN...]]
  ELSE Expr6
END
```

Bon, une explication s'impose, je crois. 😊

Pour commencer, la première ligne : **CASE Expr1**. Cette ligne déclare le CASE et indique quelle expression sera utilisée dans les tests. Souvenez-vous du switch() de PHP, là aussi on donnait une valeur qui allait être testée à chaque fois.

La seconde ligne fait peur, mais elle est tout aussi simple. D'abord, vous avez le mot-clé **WHEN** ; ce mot-clé est suivi du test qu'on veut faire subir à **Expr1**. Vient ensuite le mot-clé **THEN** ; il est suivi d'une expression, et c'est la valeur de cette expression que retournera le **CASE** si l'expression du WHEN est évaluée à vrai.

Il peut y avoir autant de **WHEN... THEN...** que l'on veut.

La ligne suivante ne devrait pas être difficile à comprendre : **ELSE Expr6**. Le **ELSE** équivaut au default du switch() de PHP, c'est-à-dire que si aucune expression d'un **WHEN** n'est évaluée à vrai, c'est l'expression qui suit le **ELSE** qui sera renvoyée.

La dernière ligne, **END**, indique juste la fin du **CASE**, il ne faut pas l'oublier sinon vous aurez une erreur de syntaxe.

Cette fois, je crois que vous avez vraiment besoin d'un exemple. Voici une requête qui va me renvoyer deux colonnes : la première contiendra le nom du salon et la seconde pourra contenir trois valeurs :

- la chaîne 'Jamais visité' s'il n'y a jamais eu de visiteur ;
- la chaîne 'Aucun visiteur actuellement' s'il n'y a pas de visiteur ;
- la chaîne 'Il y a X visiteur(s) actuellement' s'il y a au moins un visiteur.

Pour arriver à écrire la troisième chaîne, je vais utiliser une fonction propre à MySQL pour concaténer : **CONCAT()**. Cette fonction prend autant d'arguments que l'on veut et retourne la concaténation de tous les arguments, ou **NULL** si un des arguments est nul. Exemple :

**Code : SQL**

```
SELECT CONCAT('Ahah', ' ', il_pleut , ' !'); -- retourne 'Ahah, il
pleut !'
SELECT CONCAT('Ahah', ' ', il_pleut , , NULL, ' !'); -- retourne NULL
```

À ce propos, la fonction **CONCAT()** est propre à MySQL, la norme SQL prévoit une fonction (**CONCATENATE()**) et un opérateur de concaténation (**||**), mais MySQL n'en fait encore une fois qu'à sa tête.

Trêve de bavardage, passons à la requête qui nous intéresse :

**Code : SQL**

```
SELECT sal_name,
CASE
  WHEN sal_count IS NULL THEN 'Jamais visité'
  WHEN sal_count = 0 THEN 'Aucun visiteur actuellement'
  ELSE CONCAT('Il y a ', sal_count, ' visiteur(s)
actuellement')
END AS sal_etat
FROM dev_salon;
```



Notez que j'ai renommé mon **CASE** avec **AS** pour éviter que l'affichage dans la console ne soit déformé ; testez sans le **AS** **sal\_etat** si vous voulez.

Alors, qu'en pensez-vous ? Plutôt puissant, non ? 😊

À partir de cette seule structure, vous pouvez réécrire les trois autres fonctions très facilement ; d'ailleurs, essayez, c'est un très bon exercice (réponses ci-dessous).

**Secret (cliquez pour afficher)****Code : SQL**

```
-- NULLIF(Expr1, Expr2)
SELECT CASE
    WHEN Expr1 = Expr2 THEN NULL
    ELSE Expr1
END;

-- IFNULL(Expr1, Expr2)
SELECT CASE
    WHEN Expr1 IS NULL THEN Expr2
    ELSE Expr1
END;

-- IF(Expr1, Expr2, Expr3)
SELECT CASE
    WHEN Expr1 THEN Expr2
    ELSE Expr3
END;
```

## Un peu de repos avec les opérateurs

Pour finir ce chapitre en douceur, je vais vous présenter quatre opérateurs que vous serez souvent amenés à utiliser. Nous en avons déjà vu plusieurs, notamment les opérateurs arithmétiques de base (+, -, /, etc.) et plusieurs opérateurs logiques (AND, OR, etc.).

Nous allons commencer joyeusement par l'opérateur IN(). Cet opérateur permet de comparer la valeur d'une expression à une liste de valeurs.

Imaginez que vous ayez une liste de prénoms : Charles, Popeye, Marie-Henriette, Ryo, Zeus et Marcus. Si vous voulez vérifier qu'une colonne contient l'un de ces prénoms, vous seriez amenés à écrire ceci :

**Code : SQL**

```
SELECT t_col
FROM t_table
WHERE t_col = 'Charles' OR t_col = 'Popeye' OR t_col = 'Marie-
Henriette'
OR t_col = 'Ryo' OR t_col = 'Zeus' OR t_col = 'Marcus';
```

Vous avouerez que c'est... moche, et long à écrire. Mais heureusement, l'opérateur IN() permet d'alléger l'écriture :

**Code : SQL**

```
SELECT t_col
FROM t_table
WHERE t_col IN('Charles', 'Popeye', 'Marie-Henriette', 'Ryo',
'Zeus', 'Marcus');
```

Les deux requêtes donneront exactement le même résultat, mais la seconde est bien plus élégante, elle est bien plus courte ; le plus important, c'est qu'elle est plus performante. En effet, dans la première requête, dans le pire des cas, on fera onze opérations : six comparaisons avec l'opérateur = et cinq opérations logiques avec l'opérateur OR.

Tandis que dans la seconde requête, on fera au pire six comparaisons, ce qui est bien plus léger.

Il existe aussi l'opérateur opposé à IN() : NOT IN(). Ainsi, si on veut les lignes où le prénom n'est pas un des six précédemment cités, on écrira :

**Code : SQL**

```
SELECT t_col
FROM t_table
```

```
WHERE t_col NOT IN('Charles', 'Popeye', 'Marie-Henriette', 'Ryo',
'Zeus', 'Marcus');
```



Il est évident que Expr NOT IN() est équivalent à NOT(Expr IN()), mais la première version a le mérite d'être plus légère au niveau de l'écriture.

Maintenant, imaginez que vous soyez professeur, vous avez donné une interrogation à vos élèves et vous sauvegardez leur résultat dans une base de données. Comment ferez-vous pour sélectionner les élèves dont les notes sont comprises entre 3 et 7 ?

#### Code : SQL

```
CREATE TABLE note_eleve (
    eleve_name VARCHAR(50),
    eleve_note TINYINT NOT NULL
);
INSERT INTO note_eleve(eleve_name, eleve_note)
VALUES ('albert', 5), ('arthur', 5), ('coco', 2), ('charlie', 9),
('marie', 0), ('marc', 4), ('yan', 7);
```

La solution la plus évidente est l'utilisation des opérateurs de comparaison,  $\geq$  et  $\leq$ , et de l'opérateur logique AND, ce qui donnerait :

#### Code : SQL

```
SELECT eleve_name
FROM note_eleve
WHERE eleve_note  $\geq$  3 AND eleve_note  $\leq$  7;
```

Cependant, encore une fois, on fait trois opérations (deux comparaisons et un opérateur logique). C'est là que vous devez penser à ce qu'on vient de voir : IN(). En effet, avec cet opérateur, on réduira le nombre d'opérations :

#### Code : SQL

```
SELECT eleve_name
FROM note_eleve
WHERE eleve_note IN(3, 4, 5, 6, 7);
```

Mais, manque de chance, on peut encore faire mieux. Regardez bien la liste des valeurs possibles dans le IN() : elles sont continues, toutes les valeurs possibles comprises entre 3 et 7 (inclus) sont présentes. Une suite de valeurs comme celle-ci est appelée **recouvrante**, c'est-à-dire que toutes les valeurs de l'intervalle (dans l'exemple, l'intervalle est 3 à 7 inclus) sont présentes. Quand vous avez des valeurs recouvrantes, il existe un opérateur bien plus performant : BETWEEN. Cet opérateur permet de vérifier qu'une expression est comprise dans l'intervalle décrit par deux expressions.

La meilleure façon d'écrire cette requête est donc :

#### Code : SQL

```
SELECT eleve_name
FROM note_eleve
WHERE eleve_note BETWEEN 3 AND 7;
```



Tout comme IN(), BETWEEN possède un opérateur opposé : NOT BETWEEN. De plus, il est évident que NOT(Expr BETWEEN Expr AND Expr) est équivalent à Expr NOT BETWEEN Expr AND Expr.

Ça fait bien longtemps que vous n'aviez plus eu de QCM, en voilà donc un ! 😊

Pour le prochain chapitre, on va s'attaquer à quelque chose de très pratique : la gestion des dates en SQL.



## La gestion des dates

Dans bon nombre d'applications, la gestion des dates est très importante, voire vitale. Par exemple, si vous devez faire un agenda, un outil de rappel pour ne pas oublier une réunion, les dates seront vitales.

Les SGBDR proposent une gestion des dates très poussée avec un grand nombre de fonctionnalités : ainsi, il sera possible de gérer très facilement des décalages horaires, de formater les dates suivant les préférences des utilisateurs, de sélectionner des dates comprises dans des intervalles, et bien d'autres choses encore. 😊

Quand nous avons vu les différents types de colonnes, nous avons vu plusieurs types servant à stocker des dates : DATE, TIME, DATETIME et TIMESTAMP. Nous avons également vu sous quel format les dates sont stockées, mais tout ça ne nous sert à rien si on ne sait pas les utiliser.

Ce chapitre est là pour vous donner les outils nécessaires à l'utilisation des dates en SQL.



Certaines fonctions et structures sont propres à MySQL : si vous utilisez un autre SGBDR, vous devrez aller dans sa documentation pour trouver les équivalents.

De plus, assurez-vous d'utiliser une version récente de MySQL (5 ou supérieure), sinon certaines fonctions ne seront pas disponibles.

### Qui peut me donner la date ?

Pour commencer joyeusement, nous allons voir trois fonctions qui permettent d'obtenir des informations sur la date actuelle. Ces trois fonctions sont :

- CURDATE()
- CURTIME()
- NOW()

La première fonction, CURDATE(), ne prend aucun paramètre et retourne la date actuelle au format américain : par exemple, pour le 16 janvier 2008, la fonction retourne la chaîne de caractères **2008-01-16**.

La fonction suivante, CURTIME(), est le complément de CURDATE(), elle ne prend aucun paramètre et retourne l'heure actuelle. Par exemple, pour 16 heures 49 minutes et 21 secondes, la fonction retourne la chaîne de caractères **16:49:21**.

La dernière fonction est simplement la concaténation des deux autres, elle retourne la date et l'heure actuelles. Par exemple, pour le 16 janvier 2008 à 16 heures 49 minutes et 21 secondes, la fonction retourne la chaîne de caractères **2008-01-16 16:49:21**.

Testons cela en *live* :

#### Code : SQL

```
SELECT CURDATE() , CURTIME() , NOW();
```

Et profitons-en pour faire un petit exercice. Je souhaite qu'une requête me retourne NULL si la concaténation de CURDATE(), d'un espace et de CURTIME() est bien égale à la chaîne de caractères retournée par NOW(). Si ce n'est pas égal, je veux que la requête me retourne la chaîne concaténée.

Quelqu'un a-t-il la solution ? 😊

#### Secret (cliquez pour afficher)

#### Code : SQL

```
SELECT NULLIF(CONCAT(CURDATE() , ' ' , CURTIME()) , NOW());
```

### Formatage et opérateurs simples

Connaître la date, c'est bien ; pouvoir l'afficher comme l'utilisateur le souhaite, c'est mieux. 😊

Pour formater les dates, MySQL nous propose une fonction : DATE\_FORMAT(). Cette fonction prend deux paramètres :

- le premier est la date à formater ;
- le second est le format selon lequel la date sera formatée.

Pour afficher une date formatée, on fera donc ceci :

**Code : SQL**

```
SELECT DATE_FORMAT(date, format);
```

Le paramètre **date** peut être le retour de NOW(), de CURDATE, de CURTIME() ou de toute autre chaîne de caractères respectant le format de stockage des dates en SQL.

Le paramètre **format** est une chaîne de caractères qui contiendra certains symboles. Vous l'aurez compris, le but du jeu est de vous familiariser avec ces symboles.

Ces symboles sont au nombre de 32, ils ont tous une signification particulière. Tout mis à part ces symboles sera recopié littéralement ; ainsi, si j'utilise la chaîne de caractères **format** comme format de date, c'est la chaîne de caractères **format** qui sera retournée par **DATE\_FORMAT()**. En effet, cette fonction ne retourne que ce qu'on lui demande ; si le format ne demande pas qu'on affiche une information temporelle, aucune de ces informations ne sera affichée. Exemple :

**Code : SQL**

```
SELECT DATE_FORMAT(NOW(), 'format');
-- retourne 'format'
```

Ces 32 symboles respectent deux règles :

- ils sont constitués de deux caractères ;
- le premier caractère est toujours le caractère % .

Regardez cet exemple :

**Code : SQL**

```
SELECT DATE_FORMAT(CURDATE(), 'Nous sommes en %Y');
-- retourne 'Nous sommes en 2008'
```

La nouveauté, c'est % Y. Ce symbole sert à afficher l'année de la date sur quatre chiffres. Essayez de remplacer % Y par % y, vous verrez que c'est également un symbole particulier. Il sert à afficher l'année de la date sur deux chiffres.

Maintenant, imaginons que je veuille afficher la chaîne % Y : comment ferais-je ?

Si je l'écris dans le format, elle sera remplacée par l'année de la date. Heureusement, il existe une solution : le symbole % % . Ce symbole permet d'afficher le caractère % . Exemple :

**Code : SQL**

```
SELECT DATE_FORMAT(CURDATE(), '%%Y -> %Y');
-- retourne '%Y -> 2008'
```

Si vous avez compris cet exemple, vous avez tout compris sur le formatage des dates. 😊

Voici un tableau reprenant les 32 symboles et leur effet :

Symbol	Remplacé par...
%%	le caractère %'
%a	l'abréviation du nom du jour en anglais (Mon, Thu, etc.)
%b	l'abréviation du nom du mois en anglais (Jan, Feb, etc.)
%c	le numéro du mois (1, 2, 3, ..., 12)
%d	le numéro du jour (00, 01, 02, ..., 31)

%D	le numéro du jour au format anglais (1st, 2nd, 3rd, etc.)
%e	le numéro du jour (0, 1, 2, ..., 31)
%f	les microsecondes de la date
%h	l'heure au format anglais (00, 01, ..., 12)
%H	l'heure (00, 01, ..., 23)
%I	l'heure au format anglais (00, 01, ..., 12)
%i	les minutes de la date (00, 01, ..., 59)
%j	le numéro du jour de l'année (001, 002, ..., 366)
%k	l'heure (0, 1, 2, ..., 23)
%l	l'heure au format anglais (0, 1, 2, ..., 12)
%m	le numéro du mois (01, 02, ..., 12)
%M	le nom du mois en anglais (January, February, etc.)
%p	la chaîne 'AM' avant midi et la chaîne 'PM' après midi
%r	l'heure au format anglais (hh:mm:ss AM/PM)
%s	les secondes de la date (00, 01, ..., 59)
%S	les secondes de la date (00, 01, ..., 59)
%T	l'heure au format 24 heures (hh:mm:ss)
%u	le numéro de la semaine (00, 01, ..., 53), où lundi est le premier jour de la semaine
%U	le numéro de la semaine (00, 01, ..., 53), où dimanche est le premier jour de la semaine
%v	le numéro de la semaine (00, 01, ..., 53), où lundi est le premier jour de la semaine, utilisé avec le symbole %v
%V	le numéro de la semaine (00, 01, ..., 53), où dimanche est le premier jour de la semaine, utilisé avec le symbole %X
%w	le numéro du jour de la semaine (dimanche -> 0, lundi -> 1, etc.)
%W	le nom du jour en anglais (Monday, Tuesday, etc.)
%x	l'année sur quatre chiffres, où lundi est le premier jour de la semaine, utilisée avec le symbole %v
%X	l'année sur quatre chiffres, où dimanche est le premier jour de la semaine, utilisée avec le symbole %V
%y	l'année sur deux chiffres
%Y	l'année sur quatre chiffres

Fiou, c'est fini. ☺

Bien évidemment, il ne faut pas tous les connaître par cœur, vous pouvez toujours venir consulter cette liste ou celle de la [documentation officielle](#).

Sachez qu'il existe une fonction qui a le comportement opposé de DATE\_FORMAT(). Cette fonction retourne une date suivant le format de MySQL créée à partir d'une chaîne de caractères et d'un format précis. Voici un exemple :

#### Code : SQL

```
SELECT STR_TO_DATE('21/12/2007 20h20', '%d/%m/%Y %H%i');
-- retourne '2007-12-21 20:20:00'
```

Cette fonction peut être pratique si vous demandez aux utilisateurs d'entrer des dates ; en effet, ça permettra aux visiteurs d'utiliser le format de leur pays.

Passons maintenant à quelque chose de simple : la comparaison des dates.

Vous pouvez comparer des dates avec les opérateurs de comparaison que nous avons vus : <, <=, >, >=, <> et =. Exemple :

#### Code : SQL

```
SELECT '2008-01-15' < CURDATE();
-- retourne 1, la date actuelle est plus grande (récente) que la
date du 15 janvier 2008
```

Mais ce ne sont pas les seuls opérateurs que vous pouvez utiliser ! Pour ne citer qu'eux, IN() et BETWEEN sont utilisables avec les dates.

Et d'ailleurs, vous verrez que BETWEEN est extrêmement pratique quand il est utilisé avec des dates ! Imaginez que vous fassiez un système d'archivage et qu'un jour quelqu'un vous demande de ressortir tous les dossiers entre le 1<sup>er</sup> janvier 2008 et le 3 février de la même année. Avec BETWEEN, c'est d'une effarante simplicité :

#### Code : SQL

```
SELECT [...]
FROM t_table
WHERE col_date BETWEEN '2008-01-01' AND '2008-02-03';
```

Petit exercice : je souhaite qu'une requête m'affiche la date actuelle suivant ce format : **jour/mois/année%heureHminutesMsecondesS** (exemple : 16/01/2008%18H50M32S).

#### Secret (cliquez pour afficher)

#### Code : SQL

```
SELECT DATE_FORMAT(NOW(), '%e/%m/%Y%%iM%ss');
```

## Des fonctions, en veux-tu en voilà !

En plus des quatre fonctions précédentes, MySQL met à votre disposition un large panel de fonctions en tout genre pour manipuler les dates. La plupart de ces fonctions servent à obtenir des informations sur une date, comme par exemple la fonction YEAR() qui retourne l'année de la date passée en paramètre. Cette partie va vous présenter les fonctions les plus couramment utilisées.

## Des fonctions d'extraction

### DATE()

Cette fonction sert à extraire la date (jour, mois, année) de la date passée en paramètre.

Si NOW() retourne **2008-01-16 18:59:59**, DATE(NOW()) retournera **2008-01-16**.

Il est évident que DATE(CURDATE()) est équivalent à CURDATE() ; de plus, si la date passée en paramètre ne contient aucune information sur les jours, mois ou années, DATE() retournera NULL. Exemple :

#### Code : SQL

```
SELECT DATE(NOW());
```

### TIME()

Cette fonction sert à extraire la partie TIME (heures, minutes, secondes) de la date passée en paramètre. S'il n'y a pas d'information relative à la partie TIME, la fonction retournera NULL. Exemple :

**Code : SQL**

```
SELECT TIME(NOW());
```

**DAY()**

Cette fonction sert à extraire le numéro du jour du mois (1, 2, ..., 31). Tout comme DATE(), si l'il n'y a pas d'information relative au jour, DAY() retournera NULL. Exemple :

**Code : SQL**

```
SELECT DAY(CURDATE());
```

**WEEK()**

Cette fonction sert à extraire le numéro de la semaine de l'année (1, 2, ..., 53). Tout comme DATE() et DAY(), si l'il n'y a pas d'information relative à la semaine (obtenue à partir du mois et du jour), la fonction retournera NULL. Exemple :

**Code : SQL**

```
SELECT WEEK(CURDATE());
```

**MONTH()**

Cette fonction sert à extraire le numéro du mois de l'année (1, 2, ..., 12). Tout comme DATE() et DAY(), si l'il n'y a pas d'information relative au mois, MONTH() retournera NULL. Exemple :

**Code : SQL**

```
SELECT MONTH(CURDATE());
```

**YEAR()**

Cette fonction sert à extraire l'année (exemple : 2008). Tout comme les fonctions précédentes, si l'il n'y a pas d'information relative à l'année, YEAR() retournera NULL. Exemple :

**Code : SQL**

```
SELECT YEAR(CURDATE());
```

**DAYNAME(), MONTHNAME()**

Ces fonctions servent respectivement à obtenir le nom du jour et le nom du mois, en anglais bien évidemment. Comme toujours, s'il n'y a pas d'information relative au jour, au mois, les fonctions retourneront NULL. Exemple :

Code : SQL

```
SELECT DAYNAME(CURDATE()), MONTHNAME(CURDATE());
```

### HOUR(), MINUTE(), SECOND(), MICROSECOND()

Ces fonctions servent respectivement à extraire les heures, les minutes, les secondes et les microsecondes d'une date. Encore une fois, s'il n'y a pas d'information relative aux heures, aux minutes, aux secondes ou aux microsecondes, ces fonctions retourneront NULL. Exemple :

Code : SQL

```
SELECT HOUR(CURTIME()), MINUTE(CURTIME()), SECOND(CURTIME());
```

## Des fonctions de calcul

### Calculer le nombre de jours entre deux dates

Pour savoir combien de jours séparent deux dates, il existe la fonction DATEDIFF(date, date). Les deux paramètres peuvent être de type DATE ou de type DATETIME, mais seule la partie DATE sera utilisée dans le calcul. Exemple :

Code : SQL

```
SELECT DATEDIFF('2008-01-16', '2007-12-07'), DATEDIFF('2007-12-07',
'2008-01-16');
-- retourne une ligne avec deux colonnes dont les valeurs sont 40
et -40
```

### Ajouter une durée (heures:minutes:secondes) à une date

Si un jour vous avez besoin d'ajouter une durée à une date, vous utiliserez sans doute la fonction ADDTIME(date, time). Le premier paramètre peut être de type DATE, TIME ou DATETIME, mais le second paramètre doit être de type TIME. Exemple :

Code : SQL

```
SELECT ADDTIME(NOW(), '01:00:00'); -- ajoute une heure à l'heure
actuelle
```



Pour soustraire une durée, il suffit de mettre une durée négative : ADDTIME(NOW(), '-01:00:00) soustraira une heure à l'heure actuelle.

### Ajouter un certain nombre de jours à une date

Pour ajouter un certain nombre de jours à une date, il y a plusieurs méthodes ; celle que je présente ici est une forme de la fonction ADDDATE(date, day). Cette fonction retournera la date **date** à laquelle on aura ajouté **day** jours. Exemple :

Code : SQL

```
SELECT ADDDATE(NOW(), 100); -- ajoute 100 jours à la date actuelle
; vous pouvez mettre un nombre de jours négatif pour faire une
soustraction
```

### Connaitre la date du dernier jour d'un mois

Comme vous le savez, les mois n'ont pas tous le même nombre de jours. Certains mois durent 31 jours, d'autres 30, un 28 et tous les quatre ans, il y a même un mois de 29 jours. Pour connaître facilement la date du dernier jour d'un mois, il existe la fonction LAST\_DAY(date). Exemple :

#### Code : SQL

```
SELECT LAST_DAY('2004-02-17'); -- année bissextile, la fonction
retourne bien le 29 février 2004
```

## La magie des intervalles

Voici quelque chose que j'apprécie énormément. Vous devez savoir que toutes les fonctions d'addition et de soustraction peuvent être réécrites à partir des intervalles. Petit exemple avant de passer à la théorie : je vais remplacer la fonction ADDDATE() par un intervalle.

#### Code : SQL

```
SELECT ADDDATE(NOW(), 31), NOW() + INTERVAL 31 DAY;
```

L'avantage des intervalles, c'est que vous pouvez ajouter ou soustraire ce que vous voulez, des années, des mois, des jours, des minutes, des secondes, des microsecondes, des semaines, des trimestres... rien que ça. 😊

Pour déclarer un intervalle, il faut respecter une structure précise que voici :

- d'abord, le mot-clé INTERVAL ;
- ensuite, l'intervalle suivant un format précis : ça peut être soit un nombre, soit une chaîne de caractères ;
- enfin, un mot-clé spécifiant le format de l'intervalle.

Ces trois éléments sont bien présents dans l'exemple précédent, regardez bien. 😊

On peut séparer les intervalles en deux groupes, les intervalles simples et composés. J'appelle intervalle simple un intervalle numérique. Par exemple, INTERVAL 31 DAY est un intervalle simple. Voici la liste de ces intervalles :

Mot-clé	Unité temporelle ajoutée
YEAR	Année
MONTH	Mois
WEEK	Semaine
DAY	Jour
HOUR	Heure
MINUTE	Minute
SECOND	Seconde
MICROSECOND	Microseconde

QUARTER	Trimestre
---------	-----------

Quelques exemples :

#### Code : SQL

```
SELECT NOW() + INTERVAL 2 QUARTER; -- la date dans 2 trimestres
SELECT NOW() + INTERVAL 12 YEAR; -- la date dans 12 ans
SELECT NOW() + INTERVAL 23 MINUTE; -- la date dans 23 minutes
SELECT CURDATE() + INTERVAL 2 WEEK; -- la date dans 2 semaines
```

Les intervalles composés sont, comme leur nom l'indique, composés de plusieurs éléments. Par exemple, si vous voulez la date dans 23 minutes et 18 secondes, vous utiliserez un intervalle composé : MINUTE\_SECOND.

Voici la liste des intervalles composés :

Mot-clé	Format	Unités temporelles ajoutées
YEAR_MONTH	YEAR-MONTH	Année et mois
DAY_HOUR	DAY HOUR	Jour et heure
DAY_MINUTE	DAY HOUR:MINUTE	Jour et minute
DAY_SECOND	DAY HOUR:MINUTE:SECOND	Jour et seconde
DAY_MICROSECOND	DAY.MICROSECOND	Jour et microseconde
HOUR_MINUTE	HOUR:MINUTE	Heure et minute
HOUR_SECOND	HOUR:SECOND	Heure et seconde
HOUR_MICROSECOND	HOUR.MICROSECOND	Heure et microseconde
MINUTE_SECOND	MINUTE:SECOND	Minute et seconde
MINUTE_MICROSECOND	MINUTE.MICROSECOND	Minute et microseconde
SECOND_MICROSECOND	SECOND.MICROSECOND	Seconde et microseconde

Quelques exemples :

#### Code : SQL

```
SELECT NOW() + INTERVAL '5 12' DAY_HOUR; -- la date dans 5 jours et
12 heures
SELECT NOW() + INTERVAL '5-6' YEAR_MONTH; -- la date dans 5 ans et 6
mois
SELECT NOW() - INTERVAL '2 10:05:32' DAY_SECOND; -- la date il y a
2 jours, 10 heures, 5 minutes et 32 secondes
-- Oui, si on peut ajouter des intervalles, on peut également en
soustraire ;o)
SELECT NOW() - INTERVAL '30:30' MINUTE_SECOND; -- la date il y a 30
minutes et 30 secondes
```

Après ce chapitre, vous êtes incollables sur la gestion des dates. Si pour le moment certains fonctions vous paraissent inutiles, attendez un peu qu'on voie la clause GROUP BY, et vous verrez à quel point la gestion des dates est puissante et pratique pour faire des statistiques sur des périodes de durée variable, par exemple. 😊

Comme il n'y a rien de vraiment subtil dans ce chapitre, nul besoin de QCM, c'est juste de la théorie pure.

## Bataillons, à vos rangs... fixe !

Imaginez que vous soyez un officier de l'armée : les soldats sont dissipés, les esprits s'échauffent, vous devez intervenir ! Après avoir aspiré une bonne quantité d'air, vous criez haut et fort : « Bataillons, à vos rangs... fixes ! » Là-dessus, les soldats se calment et se regroupent en plusieurs bataillons bien carrés, bien discernables.

Maintenant, imaginons que nous soyons chargés du recensement des soldats : nous aurions par exemple une table avec quatre colonnes : soldat\_id, soldat\_bataillon, soldat\_nom et soldat\_grade.

Pourriez-vous m'écrire une requête qui me donnerait le nombre de soldats qu'il y a dans chaque bataillon ?

Ne cherchez pas, la réponse est assurément **non**, mais je vous promets qu'à la fin de ce chapitre, vous saurez écrire cette requête.



### Les intérêts et les limitations du regroupement

Comme vous l'avez peut-être compris en lisant l'introduction de ce chapitre, nous allons parler de **regroupement**. Jusqu'à présent, quand nous faisions des requêtes de type SELECT, nous travaillions ligne par ligne, c'est-à-dire que nous appliquions un certain traitement à chaque ligne de la table (un filtrage avec la clause WHERE, un calcul à partir des valeurs des colonnes, etc.).

Mais le SQL nous propose autre chose : travailler groupe par groupe.

Vous savez sûrement ce que sont des groupes, mais voici une définition qui exprime le concept de regroupement en SQL : un groupe est un ensemble d'éléments distincts possédant une ou plusieurs caractéristiques communes. Ces groupes sont aussi parfois appelés **agrégats**.

Mais quel est l'intérêt de travailler groupe par groupe ?

La réponse tient en un mot : statistiques.

Les statistiques sont très pratiques : prenez par exemple les publicités sur un site web. Imaginez que vous ayez 1000 visiteurs, afficheriez-vous les même pubs si 900 personnes ont moins de 18 ans et si 900 personnes ont plus de 65 ans ?

Il est évident que non. Si le lectorat est majoritairement jeune, vous mettriez des publicités sur des jeux vidéo, des écoles, ou n'importe quoi d'autre qui pourrait intéresser des jeunes. Tandis que si le lectorat est majoritairement vieux, les publicités porteraient sur des sujets susceptibles d'intéresser les personnes âgées.

Prenons un autre exemple : un forum. Imaginez-vous étant l'administrateur d'un forum : des membres vous proposent d'ouvrir un autre forum, comment savoir si ce forum est utilisé ?

C'est très simple : vous allez regarder le nombre de messages sur un intervalle de temps donné et le comparer avec les autres forums. Si c'est égal, le forum fonctionne bien. Si c'est supérieur, le forum vous apporte beaucoup de visites et est donc très bénéfique. Si c'est inférieur, le forum n'apporte rien et se révèle un poids mort pour vous.

Pour réaliser cela, vous aurez besoin de faire des statistiques. Et pour faire des statistiques, vous avez besoin de groupes (ou agrégats).

Prenons un cas concret. Voici une table servant à stocker des commentaires relatifs à une news :

#### Code : SQL

```
CREATE TABLE dev_comment (
    com_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, -- id
    du commentaire
    com_newsId INT UNSIGNED NOT NULL, -- id de la news à laquelle
    le commentaire se rapporte
    com_author VARCHAR(50) NOT NULL, -- pseudo de l'auteur du
    commentaire
    com_content TEXT NOT NULL -- contenu du commentaire
);
```

Et voici le contenu de cette table en image :

com_id : 1 com_newsId : 1 com_author : `Haku com_content : Lala	com_id : 6 com_newsId : 3 com_author : `Haku com_content : Olé !
com_id : 2 com_newsId : 1 com_author : Paul com_content : Ah ouais	com_id : 7 com_newsId : 1 com_author : `Haku com_content : Hmmmm
com_id : 3 com_newsId : 2 com_author : Paul com_content : Bonne news	com_id : 8 com_newsId : 2 com_author : `Haku com_content : Tout à fait
com_id : 4 com_newsId : 1 com_author : Modérateur com_content : C'est fini oui ?	com_id : 9 com_newsId : 1 com_author : Paul com_content : Bye !
com_id : 5 com_newsId : 1 com_author : `Haku com_content : Non =O	

Comme on peut le voir, je suis nul en graphisme. 😊  
Mais heureusement, ce n'est pas ce qui nous intéresse ici.

Chaque petite case bleue est une ligne de notre table ; nous avons donc neuf lignes et par conséquent, neuf commentaires. On peut voir également que les commentaires appartiennent à des news différentes : une partie appartient à la news ayant l'id 1, une autre appartient à la news ayant l'id 2, et le reste appartient à la news ayant l'id 3.

Imaginons que nous voulions le nombre de commentaires par news, cette information est une statistique. En conséquence de quoi, nous allons devoir créer des groupes puisque nous avons dit qu'on ne pouvait établir de statistiques que sur des groupes. Mais nous avons aussi dit que les groupes étaient créés à partir d'une ou plusieurs caractéristiques communes. Dans le cas qui nous occupe, puisque nous voulons des statistiques pour chaque news, nous allons regrouper nos commentaires grâce à la colonne **com\_newsId** qui, je le rappelle, contient l'id de la news à laquelle le commentaire se rapporte.

Voici une image montrant ces groupes, toujours en bleu :

com_id : 1 com_newsId : 1 com_author : `Haku com_content : Lala	com_id : 9 com_newsId : 1 com_author : Paul com_content : Bye !
com_id : 2 com_newsId : 1 com_author : Paul com_content : Ah ouais	com_id : 3 com_newsId : 2 com_author : Paul com_content : Bonne news
com_id : 7 com_newsId : 1 com_author : `Haku com_content : Hmmmm	com_id : 8 com_newsId : 2 com_author : `Haku com_content : Tout à fait
com_id : 4 com_newsId : 1 com_author : Modérateur com_content : C'est fini oui ?	com_id : 6 com_newsId : 3 com_author : `Haku com_content : Olé !
com_id : 5 com_newsId : 1 com_author : `Haku com_content : Non =O	

Nous distinguons très nettement les trois groupes créés, et c'est ici que tout se joue, alors accrochez-vous à vos chaussettes.



Vous savez déjà que ces groupes vont nous permettre d'obtenir des statistiques, mais ce que vous ignorez, c'est qu'un groupe possède les mêmes colonnes que les lignes qui le constituent.

Prenez une bonne inspiration et relisez ceci : **un groupe possède les mêmes colonnes que les lignes qui le constituent.**

On a vu que les groupes sont formés à partir de lignes. On sait également que les lignes possèdent plusieurs colonnes. Dans l'exemple, les lignes sont constituées de quatre colonnes : **com\_id, com\_newsId, com\_author et com\_content**. Ce qui implique que notre groupe est également constitué de ces quatre colonnes ; elles existent donc dans ce groupe et sont donc manipulables ! Et c'est bien pour ça que nous allons pouvoir faire des statistiques. En effet, pour établir des statistiques, il y a une famille de fonctions : les **fonctions d'agrégation**. Ces fonctions ne travaillent que sur des groupes.

Mais attention toutefois : quand je dis que ces colonnes sont manipulables, ce n'est pas tout à fait exact. Ces colonnes ne sont pas manipulables comme bon nous semble. Il n'y a en effet que deux façons d'utiliser les colonnes d'un groupe :

- avec des fonctions d'agrégation ;
- et quand la valeur de la colonne est la même pour toutes les lignes constituant le groupe.

Nous verrons les fonctions d'agrégation juste après ; pour le moment, intéressons-nous à la seconde façon. Qu'est-ce que cela veut dire ?

Sur le premier schéma, il y a neuf cases bleues, ces cases bleues sont les lignes de ma table. Donc si je fais un SELECT sans clause WHERE, MySQL me retournera un tableau de neuf lignes et quatre colonnes, comme d'habitude.

Maintenant, quelle sera la taille du tableau retourné par MySQL avec le regroupement et sans clause WHERE ? Quatre colonnes et neuf lignes ?

Hé bien **non**, MySQL nous retournera un tableau de quatre colonnes et... trois lignes ! Si votre regroupement crée huit groupes, MySQL vous retournera un tableau de huit lignes. Et cela amène la conclusion suivante : un groupe peut être considéré comme

une ligne.

Or, si vous avez bonne mémoire, d'une ligne on ne peut obtenir qu'une et une seule valeur pour une colonne donnée. C'est exactement pareil pour les groupes.

Regardons à nouveau le deuxième schéma et plus précisément la colonne **com\_author**. Dans le groupe, il y a deux valeurs différentes pour cette colonne. Vous ne pourrez jamais en avoir plus d'une. Si vous tentez de sélectionner la valeur de la colonne **com\_author** d'un groupe, vous n'aurez qu'une des valeurs de cette colonne. Ainsi donc, pour cette colonne du groupe relatif à la news ayant l'id 2, MySQL me retournerait soit la valeur **Paul**, soit la valeur **'Haku**, mais jamais les deux. Et s'il y avait 50 000 valeurs différentes dans un groupe, MySQL ne me retournerait malgré tout qu'une et une seule valeur.

Généralement, la valeur renournée par MySQL est la première dans l'ordre lexicographique.

Et c'est pourquoi j'ai dit que si la valeur d'une colonne est la même pour toutes les lignes constituant le groupe, alors on peut sélectionner cette colonne. Hé bien oui : reprenons l'exemple de ce groupe relatif à la news ayant l'id 2. Si je sélectionne la valeur de la colonne **com\_newsId**, MySQL me retournera soit 2, soit... 2. Effectivement, comme toutes les valeurs sont les mêmes, peu importe celles que MySQL nous retourne puisqu'elles sont toutes identiques !

Ainsi, vous pourrez toujours sélectionner sans crainte votre critère de regroupement, mais n'essayez surtout pas de sélectionner autre chose car vous n'avez aucun moyen de savoir à quoi cette information correspond.

## La clause magique

Nous allons nous servir de la table **dev\_comment** pour montrer comment regrouper des lignes. Je ne vous donne pas la requête d'insertion, vous avez les données sur les schémas, vous pouvez insérer vous-mêmes les données. Considérez cela comme un mini-TP. 

Pour vérifier que vous avez bien inséré les données, exécutez donc cette requête et utilisez vos yeux :

### Code : SQL

```
SELECT com_id, com_newsId, com_author, com_content  
FROM dev_comment;
```

Passons au plat de résistance : la nouvelle clause. Cette clause s'appelle GROUP BY, et voici sa syntaxe :

### Code : SQL

```
SELECT Expr[, Expr[, ...]]  
FROM t_table  
GROUP BY Expr[, Expr[, ...]];
```

Comme vous le voyez, le critère de regroupement est une expression, on peut donc regrouper en fonction d'une colonne, de la valeur de retour d'une fonction, ou plus généralement d'une expression.

Vous constatez aussi qu'on peut utiliser plusieurs expressions. Le comportement de GROUP BY avec plusieurs expressions ressemble à celui d'ORDER BY. Souvenez-vous, si nous avions un ORDER BY A, B, on ordonnerait les lignes en fonction de A et s'il y avait plusieurs lignes ayant la même valeur pour A, on ordonnerait celles-ci en fonction de B.

Avec GROUP BY, si on a un GROUP BY A, B, on va d'abord créer des groupes en fonction de A, et à partir de ces groupes on créera des sous-groupes en fonction de B.

Testez cette requête :

### Code : SQL

```
SELECT com_newsId -- comme c'est l'expression qui sert au  
regroupement, je peux sélectionner cette colonne sans crainte  
FROM dev_comment  
GROUP BY com_newsId;
```

Elle nous retourne trois lignes, chaque ligne contenant l'id de la news à laquelle se rapportent les groupes.

Maintenant, essayons un regroupement avec plusieurs critères : d'abord en fonction de l'id de la news à laquelle se rapporte le commentaire, et ensuite en fonction du pseudo de son auteur...

**Code : SQL**

```
SELECT com_newsId, com_author
FROM dev_comment
GROUP BY com_newsId, com_author;
```

Comme vous le voyez, cette fois-ci nous n'avons plus trois lignes, mais bien six. Bien évidemment, un GROUP BY A, B ne donne pas du tout le même résultat qu'un GROUP BY B, A .

Essayez donc de voir ce que donnerait la requête précédente avec un GROUP BY com\_author, com\_newsId. 😊

La première fonction d'agrégation que vous allez connaître est une fonction qui permet de... calculer une somme. Imaginons que je sois stupide (les mauvaises langues diront qu'il n'est même pas nécessaire de l'imaginer 🤪) et que je veuille faire la somme des id des commentaires relatifs à chaque news.

On sait par avance que la requête ressemblera à ceci :

**Code : SQL**

```
SELECT [...]
FROM dev_comment
GROUP BY com_newsId;
```

Mais une fonction d'agrégation, comment ça fonctionne ?

C'est simple : comme une fonction, tout ce qu'il y a de plus « normal » ! Rappelez-vous, une fonction travaille sur une colonne d'une ligne. Or, un groupe peut être considéré comme une ligne et de plus, le groupe est constitué des mêmes colonnes que les lignes à partir desquelles il a été créé. En conséquence de quoi, une fonction d'agrégation s'utilise exactement de la même façon qu'une fonction « normale ». Du côté de MySQL, il y a bien évidemment une différence de traitement puisqu'il faut travailler sur plusieurs valeurs, mais pour nous pauvres utilisateurs, c'est totalement transparent.

Voici donc la requête tant attendue :

**Code : SQL**

```
SELECT com_newsId, SUM(com_id)
FROM dev_comment
GROUP BY com_newsId;
```

Vous pouvez vérifier à la main, le compte est bien juste. 😊

Pour finir, voici la requête SELECT avec toutes les clauses possibles actuelles ordonnées comme il faut, hé oui, il ne faut pas oublier qu'on ne met pas ce qu'on veut où on veut.

**Code : SQL**

```
SELECT Expr[, Expr[, ...]]
FROM t_table
[WHERE Expr]
[GROUP BY Expr[, Expr[, ...]]]
[ORDER BY Expr [ASC | DESC] [, Expr [ASC | DESC] [, ...]]]
[LIMIT X [OFFSET Y]];
```

## Les fonctions d'agrégation

Il existe un peu plus de dix fonctions d'agrégation, mais je n'en présenterai que six. Toutefois, la liste de ces fonctions ainsi que leur description est bien évidemment accessible sur la [documentation de MySQL](#).

## Calculer une somme avec SUM(Expr)

Comme on l'a vu, la fonction SUM() permet de calculer la somme d'une expression dans un groupe. Cette expression peut être

une colonne comme on l'a vu précédemment, mais ça peut être bien d'autres choses. Exemple :

#### Code : SQL

```
SELECT SUM(com_newsId*IF(com_newsId > 9, com_newsId MOD 9,
com_newsId))
FROM dev_comment
GROUP BY com_newsId;
```

## Calculer une moyenne avec AVG(Expr)

AVG(), contraction de *average* qui se traduit en français par *moyenne*, permet de calculer la moyenne d'une expression dans un groupe. La somme étant définie par la formule : (somme des expressions) / (nombre d'expressions). Exemple :

#### Code : SQL

```
SELECT com_newsId, AVG(com_id)
FROM dev_comment
GROUP BY com_newsId;
```

## Trouver le maximum et le minimum avec MAX(Expr) et MIN(Expr)

MIN() et MAX() sont des fonctions d'agrégation servant respectivement à obtenir le minimum et le maximum d'une expression dans un groupe. Exemple :

#### Code : SQL

```
SELECT com_newsId, MIN(com_id), MAX(com_id)
FROM dev_comment
GROUP BY com_newsId;
```

Attention toutefois, ces quatre fonctions **ignorent** les lignes où Expr (leur paramètre) est nul ! En effet, imaginez que vous donnez la possibilité aux utilisateurs de donner une note à vos news en plus des commentaires. Comment feriez-vous pour différencier les commentaires avec une note et ceux sans ? Simple, vous mettriez NULL dans la colonne prévue pour la note. Et si ces fonctions n'ignoraient pas les NULL, il faudrait convertir NULL en entier : or, la valeur la plus naturelle serait zéro. Et à moins que vous ne soyez très mauvais en maths, vous devriez savoir que  $(4+5) / 2$  est très différent de  $(4+5+0) / 3$ . 😊

De plus, sachez que vous pouvez choisir d'ignorer les doublons en utilisant le mot-clé DISTINCT. Ainsi, si vous avez une table **t\_table** avec une colonne **t\_col** dans laquelle vous avez deux lignes ayant la valeur **2** pour **t\_col** :

#### Code : SQL

```
SELECT SUM(t_col) FROM t_table; -- retournera 4 car 2 + 2 = 4
SELECT SUM(DISTINCT t_col) FROM t_table; -- retournera 2 car 2 +
rien = 2 car on a dit à MySQL d'ignorer les doublons, il n'a donc
compté qu'une seule fois la valeur 2
```

Enfin, il est bon de dire que ces fonctions retourneront NULL s'il n'y aucune ligne dans la table ou si l'expression en paramètre de ces fonctions est nul pour toutes les lignes. Mais heureusement, on a vu comment traiter ce fameux NULL.

La fonction suivante est un peu plus complexe dans sa syntaxe, mais tout aussi utile que les précédentes. 😊

En effet, elle permet de concaténer toutes les valeurs d'une expression dans un groupe. Avec cela, il est très facile d'obtenir la liste des pseudos ayant posté dans chaque news !

Cette fonction est GROUP\_CONCAT(), voici sa syntaxe :

#### Code : SQL

```
SELECT GROUP_CONCAT([DISTINCT] Expr[, Expr[, ...]] [ORDER BY Expr
[ASC | DESC] [, Expr [ASC | DESC] [, ...]]] [SEPARATOR Expr])
FROM t_table
GROUP BY Expr[, Expr[, ...]];
```

Tout de suite, ça rigole moins. 😊

Ne vous en faites pas, c'est beaucoup plus simple que ça en a l'air.

Comme vous le voyez, on peut ajouter le mot-clé DISTINCT, ce qui évitera d'avoir des doublons si plusieurs lignes ont des valeurs identiques.

Expr[, Expr[, ...]] est la liste des expressions que vous souhaitez concaténer : une colonne, le retour d'une fonction, etc.

Après, vous pouvez mettre un ORDER BY pour que les valeurs soient concaténées dans un certain ordre.

Enfin, SEPARATOR permet de spécifier ce qui séparera les différentes valeurs concaténées. Par exemple, je veux grouper mes commentaires en fonction de l'id de la news à laquelle se rapportent les commentaires. Je veux que ma requête me retourne l'id des news ainsi que la concaténation des pseudos ordonnés de façon alphabétique inverse, séparés par un point-virgule :

#### Code : SQL

```
SELECT com_newsId, GROUP_CONCAT(DISTINCT com_author ORDER BY
com_author DESC SEPARATOR ';' ) AS com_pseudoList
FROM dev_comment
GROUP BY com_newsId;
-- ça rend vraiment mal dans la console :-°
SELECT com_newsId, GROUP_CONCAT(com_author, '(', com_content, ')'
ORDER BY com_author DESC SEPARATOR '\n') AS com_pseudoList
FROM dev_comment
GROUP BY com_newsId;
```

## La dernière des fonctions : COUNT()

Cette fonction sert à compter le nombre de lignes dans un groupe, mais elle est un peu spéciale : elle possède deux syntaxes ayant des comportements différents !

La première syntaxe est identique à celle de MAX(), MIN(), SUM() et AVG() : COUNT([DISTINCT] Expr). Cette fonction a le même comportement que les quatre autres, elle peut compter les lignes distinctes et ignore les NULL.

On dit que cette première syntaxe compte les colonnes.

La seconde syntaxe est bien plus simple : COUNT(\*) .

On dit que cette seconde syntaxe compte les lignes.

Chacune de ces syntaxes a son utilité, toutefois, sachez que dans une situation donnée, l'une des deux syntaxes est généralement plus performante que l'autre :

#### Code : SQL

```
-- Cette requête est plus performante...
SELECT COUNT(*)
FROM dev_comment
GROUP BY com_newsId;
-- ... que celle-ci :
SELECT COUNT(com_id)
FROM dev_comment
GROUP BY com_newsId;
```

## ROLLUP et HAVING

Il existe encore deux possibilités de GROUP BY que vous ignorez. La première est l'ordonnancement des lignes. Pour ordonner

vos lignes, vous avez utilisé la clause ORDER BY, mais en fait, il existe un cas où vous ne l'utiliserez pas : quand vous ordonnerez les lignes *via* une expression utilisée dans GROUP BY.

Prenons un exemple :

**Code : SQL**

```
SELECT t_col  
FROM t_table  
GROUP BY t_col;
```

Si vous observez ce que retourne cette requête, vous constaterez que les lignes sont ordonnées en fonction de la valeur de **t\_col** dans l'ordre croissant. En effet, quand vous utilisez GROUP BY, MySQL ordonne vos lignes en fonction des critères de regroupement. Toutefois, la clause ORDER BY prévaut sur l'ordre défini par GROUP BY.

Ainsi, si vous vouliez ordonner les lignes en fonction de la valeur de **t\_col**, mais dans l'ordre décroissant, vous seriez tentés de faire ceci :

**Code : SQL**

```
SELECT t_col  
FROM t_table  
GROUP BY t_col  
ORDER BY t_col DESC;
```

Mais c'est sans compter sur la puissance de GROUP BY ! En effet, comme je l'ai dit, les lignes sont ordonnées par GROUP BY en l'absence d'un ORDER BY. Mais GROUP BY, en plus d'ordonner les lignes, permet de choisir le sens de l'ordonnancement, c'est-à-dire l'ordre croissant ou décroissant. Ainsi, la requête précédente peut être écrite de cette façon :

**Code : SQL**

```
SELECT t_col  
FROM t_table  
GROUP BY t_col DESC;
```

Si vous avez plusieurs expressions de regroupement, vous pouvez spécifier un sens d'ordonnancement différent pour chacune d'elles. Exemple :

**Code : SQL**

```
SELECT [...]  
FROM t_table  
GROUP BY t_a DESC, t_b ASC;
```

La dernière possibilité de GROUP BY est un peu plus complexe, mais tout aussi pratique.  
Reprendons une ancienne requête :

**Code : SQL**

```
SELECT com_newsId, COUNT(*)  
FROM dev_comment  
GROUP BY com_newsId;
```

Comme vous le savez, cette requête retournera le nombre de commentaires pour chaque news, ainsi que l'id de ladite news.

Mais essayez ceci :

**Code : SQL**

```
SELECT com_newsId, COUNT(*)
FROM dev_comment
GROUP BY com_newsId WITH ROLLUP;
```

Si vous ouvrez bien les yeux, vous verrez que cette requête nous retourne une ligne de plus que la précédente. Mais ouvrez un peu plus grand vos yeux et regardez un peu les valeurs de cette nouvelle ligne : NULL et neuf(9). Neuf, ça alors, cela ne vous rappelle-t-il rien ? Eh oui : jusqu'à preuve du contraire, si vous faites la somme des valeurs des trois premières lignes (6, 2 et 1), vous obtenez 9 !

À partir de cette constatation, pouvez-vous deviner, même grossièrement, ce que produit l'option WITH ROLLUP ?

Nous avons vu que GROUP BY permet de créer des groupes (ou agrégats), mais l'option WITH ROLLUP, elle, crée ce qu'on appellera des **super-agrégats**. C'est-à-dire des agrégats générés à partir de ceux créés par GROUP BY. Cette option peut se révéler utile dans certains cas. Dans l'exemple qui nous occupe, cette option permet d'obtenir le nombre de commentaires par news, mais également le nombre total de commentaires. Si vous faisiez des sondages pour connaître l'avis des visiteurs à propos de votre site *via* une note sur 20 par exemple, ça vous permettrait aisément d'avoir la moyenne totale des notes en même temps que les moyennes des notes par mois, par semaine, par année, par trimestre ou je ne sais quoi d'autre. 😊

Attention toutefois, il y a deux choses à signaler à propos de WITH ROLLUP.

La première, c'est que son utilisation est récursive si vous utilisez plusieurs critères de regroupement. Un exemple sera plus parlant qu'un long discours :

**Code : SQL**

```
SELECT com_newsId, com_author, COUNT(*)
FROM dev_comment
GROUP BY com_newsId, com_author WITH ROLLUP;
```

Comme vous le voyez, cette requête nous retourne dix lignes tandis que la même requête sans l'option WITH ROLLUP nous en retournerait six.

Essayons de comprendre ces résultats.

Premièrement, MySQL groupe ces lignes en fonction de nos critères, nous avons six lignes de résultats.

Deuxièmement, pour chaque ensemble de lignes ayant un **com\_newId** donné, MySQL va ajouter une *ligne-résumé*. Ces lignes contiendront l'id de la news dans la colonne **com\_newsId**, NULL dans la colonne **com\_author**, et enfin le nombre de commentaires pour chaque news dans la colonne **COUNT(\*)**.

Finalement, MySQL va considérer les *lignes-résumés* créées précédemment comme un groupe et créer un résumé de ce groupe ! Ainsi, une nouvelle ligne sera ajoutée contenant NULL dans les colonnes **com\_newsId** et **com\_author**, tandis que la colonne **COUNT(\*)** contiendra la somme des commentaires par news, ce qui est également le nombre total de commentaires.

Vous pouvez trouver d'autres exemples de l'utilisation de WITH ROLLUP sur la [documentation de MySQL](#). 😊

La deuxième chose à savoir, c'est que l'option WITH ROLLUP peut poser problème quand on utilise des clauses ORDER BY et LIMIT.

Pour ORDER BY, c'est simple : vous ne devez jamais l'utiliser avec l'option WITH ROLLUP. D'ailleurs, si vous tentiez de le faire, vous seriez confrontés à une jolie erreur que voici 😊 :

**Code : Console**

```
ERROR 1221 (HY000) : Incorrect usage of CUBE/ROLLUP and ORDER BY
```

Ce qui se traduit par :

**Code : Console**

```
ERREUR 1221 (HY000) : Utilisation incorrecte de CUBE/ROLLUP et de ORDER BY
```

Pour ce qui est de LIMIT, vous n'aurez pas d'erreur de la part de MySQL, mais les résultats que vous obtiendrez ne seront pas forcément exploitables. Reprenons la requête précédente et limitons le nombre de lignes à huit :

**Code : SQL**

```
SELECT com_newsId, com_author, COUNT(*)
FROM dev_comment
GROUP BY com_newsId, com_author WITH ROLLUP
LIMIT 8;
```

Comme vous le voyez, nous avons perdu deux super-agrégats dans la bataille. 😞

Et si nous avions mis LIMIT 7, nous aurions aussi perdu la ligne qui représente l'agrégat relatif à la news ayant l'id 3.

Pour expliquer ce résultat, il suffit de connaître **l'ordre d'exécution des clauses d'une requête**. En effet, vous ne pouvez pas ordonner, grouper et limiter des lignes en même temps : ces opérations sont faites l'une après l'autre et il est évident qu'en fonction de l'ordre dans lequel ces clauses sont exécutées, le résultat différera. Mais nous verrons cela plus tard car pour le moment, c'est une nouvelle clause qui nous intéresse : HAVING.

Pourriez-vous m'écrire une requête qui me sélectionne l'id des news ayant au moins deux commentaires dans la table **dev\_comment** ?

Malheureusement, la réponse est **non** ! Peut-être avez-vous pensé à ceci :

**Code : SQL**

```
SELECT com_newsId
FROM dev_comment
WHERE COUNT(*) >= 2
GROUP BY com_newsId;
```

Si c'est le cas, bravo, il y avait de l'idée et c'est presque ça ! Mais ceux qui auront testé auront été confrontés à cette erreur :

**Code : Console**

```
ERROR 1111 (HY000) : Invalid use of group function
```

Qui se traduit par :

**Code : Console**

```
ERREUR 1111 (HY000) : Utilisation invalide des fonctions d'agrégation
```

En effet, vous devez savoir et retenir que vous ne pouvez en aucune façon utiliser des fonctions d'agrégation dans une clause WHERE. Et la raison est encore l'ordre d'exécution des clauses. 😊

C'est pour cela qu'une autre clause existe : HAVING. Elle a exactement le même rôle que WHERE : filtrer des lignes. Toutefois, vous pouvez utiliser des fonctions d'agrégation avec HAVING, contrairement à WHERE. Et encore une fois, la raison est l'ordre d'exécution des clauses. Voici donc la requête qui me permet d'obtenir ce que je veux :

**Code : SQL**

```
SELECT com_newsId
FROM dev_comment
GROUP BY com_newsId
HAVING COUNT(*) >= 2;
```

## L'ordre d'exécution des clauses

Avec toutes ces clauses et toutes leurs options, nous arrivons à une requête générique, assez monstrueuse, que voici :

Code : SQL

```
SELECT Expr[, Expr[, ...]]  
FROM t_table  
[WHERE Expr]  
[GROUP BY Expr [ASC|DESC] [, Expr [ASC|DESC] [, ...]] [WITH ROLLUP]]  
[HAVING Expr]  
[ORDER BY Expr [ASC|DESC] [, Expr [ASC|DESC] [, ...]]]  
[LIMIT X [OFFSET Y]];
```

Nous avons donc un total de cinq clauses hormis SELECT et FROM qui feront à nouveau parler d'elles un peu plus tard.



Parlons de l'ordre d'exécution des clauses, maintenant. La première chose à dire, c'est qu'il est facile à retenir.

En effet, les clauses sont exécutées dans l'ordre dans lequel elle apparaissent !

MySQL exécute donc d'abord le WHERE, puis le GROUP BY, ensuite le HAVING, suivis de l'ORDER BY et enfin le LIMIT.

En voyant cet ordre, vous avez sûrement compris pourquoi on ne peut pas utiliser de fonction d'agrégation dans la clause WHERE. Eh bien oui, puisque le WHERE est exécuté avant le GROUP BY, les groupes n'existent pas pour le WHERE ! Mais comme le HAVING est exécuté après le GROUP BY, les fonctions d'agrégation sont utilisables dans celui-ci puisque les groupes ont déjà été créés.

À propos de cela, ne voyez-vous rien d'étrange dans cette requête ?

Code : SQL

```
SELECT [...]  
FROM t_table  
GROUP BY t_col DESC  
HAVING t_col > 50 AND COUNT(*) < 5;
```

Imaginez qu'il y a un million de lignes dans cette table. MySQL va les lire, puis va grouper les lignes en fonction de la valeur de `t_col` pour enfin en filtrer quelques-unes en fonction de la valeur de `t_col` et du nombre de lignes qui ont généré le groupe.

Vous ne voyez toujours rien d'étrange ?



Bon, et si maintenant je vous dis que parmi ce million de lignes, seules mille lignes ont une valeur dans `t_col` supérieure à 50, vous ne voyez toujours pas la bizarrerie ?

À votre avis, est-ce plus rapide de grouper un million de lignes, ou mille ?

Eh oui, grouper mille lignes sera beaucoup plus rapide !

En conséquence, vous devez toujours, absolument toujours, filtrer le maximum de lignes dans la clause WHERE : moins il y a de lignes à traiter, plus ça ira vite. Ainsi, utilisez le HAVING avec parcimonie et n'y mettez pas de condition qui pourrait être insérée dans le WHERE. Cette requête produirait donc le même résultat que la précédente, mais bien plus rapidement :

Code : SQL

```
SELECT [...]  
FROM t_table  
WHERE t_col > 50  
GROUP BY t_col DESC  
HAVING COUNT(*) < 5;
```

Comme quoi, le code le plus court à écrire n'est pas forcément le plus rapide.



Cet ordre d'exécution montre aussi pourquoi on ne peut pas utiliser ORDER BY s'il y a un GROUP BY avec l'option WITH ROLLUP. Comme le ORDER BY est exécuté après l'insertion des lignes résumées, les résultats seraient tout simplement inexploitables étant donné qu'on ne saurait pas ce que résument ces lignes, puisqu'elles ne suivaient pas forcément l'agrégat qu'elles résument.

Cette explication est également valable pour l'utilisation conjointe de GROUP BY avec l'option WITH ROLLUP et LIMIT. GROUP BY étant exécuté avant, LIMIT coupe dans les lignes déjà groupées et les résumés des groupes, ce qui peut rendre inexploitables ces résultats car on peut « perdre » des groupes ou des résumés.

Et voilà, c'est bel et bien fini ! 😊

Vous aimeriez que ce soit vrai, n'est-ce pas ? 🤔

Ne vous en faites pas, ça le sera dans quelques lignes. En effet, il reste un dernier détail que vous ignorez sur les lignes et les agrégats.

Vous devez savoir que sans clause GROUP BY, la table elle-même est un agrégat ! Ainsi, pour compter le nombre total de commentaires, je peux tout à fait faire ceci :

#### Code : SQL

```
SELECT COUNT(*)
FROM dev_comment;
```

Attention, toutefois ! Avec GROUP BY, si vous tentiez de sélectionner la valeur d'une colonne non constante dans un groupe, vous aviez une valeur même si vous ne pouviez pas savoir à quelle ligne elle se rapporte. Sans GROUP BY, essayez de sélectionner la valeur d'une colonne non constante, comme ceci par exemple :

#### Code : SQL

```
SELECT com_id, COUNT(*)
FROM dev_comment;
```

Et vous tomberez nez à nez avec cette erreur :

#### Code : Console

```
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP
```

Ce qui se traduit par :

#### Code : Console

```
ERREUR 1140 (42000): Le mélange de colonnes groupées (MIN(), MAX(), COUNT(), ...) a
```

Donc, si vous rencontrez cette erreur, pensez à GROUP BY. 😊

Et voilà, ce chapitre est maintenant réellement terminé. 😊

À présent, vous devriez être à même d'écrire la requête qui donnera ce que j'ai demandé dans l'introduction de ce chapitre.

Étant donné la masse théorique apportée par ce chapitre, point de QCM, mais le prochain chapitre est un TP ! 🍪

## Troisième TP : un livre d'or

Quand on aime un site web, on a souvent envie d'y laisser une trace, un petit mot.

Il est donc assez important pour un site web de proposer un espace où les visiteurs pourront s'exprimer et laisser un petit message : cet espace est l'objet de ce TP. 😊

### Les spécifications

Si vous vous souvenez bien du dernier TP, vous devriez savoir que le point de départ d'un projet quel qu'il soit est le cahier des charges, ou encore les spécifications du projet. Il est donc logique de commencer par là. 😊

Le but est de créer deux scripts :

- le premier permettra aux visiteurs de poster un message et de visionner les messages postés,
- le second exploitera les messages du livre d'or pour établir des statistiques.

Passons aux spécifications. 😊

Le premier script permettra au visiteur de poster un message *via* un formulaire. Ce formulaire sera composé de trois champs :

- le premier permettra au visiteur de saisir son nom (ou un pseudo),
- le second permettra au visiteur d'attribuer une note au site,
- et le troisième permettra au visiteur de saisir son message.

Le pseudo du visiteur sera considéré comme valide s'il respecte ces conditions :

- il doit être composé de 3 à 50 caractères,
- les caractères qui le composent doivent être soit des lettres majuscules ou minuscules, soit des chiffres.

Il faut laisser la possibilité au visiteur de ne pas attribuer de note ; mais s'il en attribue une, ce doit être un nombre entier compris entre 1 (la pire) et 5 (la meilleure).

Le message du visiteur n'est soumis à aucune restriction.

Vous devrez mettre un lien vers le script de statistiques.

Le message d'un visiteur doit être accompagné de plusieurs données que voici :

- le nom (ou le pseudo) de la personne qui l'a écrit,
- la date à laquelle le message a été rédigé,
- l'IP du visiteur qui a rédigé le message,
- et la note si le visiteur en a attribué une.

Vous afficherez tous les messages déjà postés sur la page, du plus récent au plus ancien (le plus récent est donc le premier de la liste). Pour chaque message, vous afficherez ces informations :

- la date à laquelle il a été posté,
- le nom du visiteur qui l'a posté,
- le contenu du message,
- et un lien pour supprimer ce message.

En voyant cette dernière information, vous avez dû vous dire « Il est fou, n'importe qui pourra supprimer les messages ! ». Eh bien non, je ne suis pas fou. 😊

Mais pour éviter que n'importe qui supprime vos messages, vous devrez pouvoir définir si le visiteur a le droit ou non de supprimer le message.

Le lien de suppression transmettra l'id du message à supprimer *via* l'adresse, donc *via* la superglobale \$\_GET. Si un id est transmis, s'il est valide et si on a le droit de supprimer les messages, le message sera supprimé de la base de données.

Et voici qui termine les spécifications du premier script. Mais voilà déjà les spécifications du second !

Ce script permettra de visualiser des informations statistiques sur les messages du livre d'or.

Voici les informations que vous devrez afficher :

- le nombre total de messages dans le livre d'or,
- la date du message le plus ancien,
- la date du message le plus récent,
- le nombre de notes attribuées,
- la moyenne des notes attribuées,
- un audit de la période des fêtes 2007-2008,
- et des statistiques par année et par mois.

L'audit est assez simple. Imaginez que pour les fêtes de Noël, vous avez publié un article, ou bien fait de la publicité pour votre

site, ou je ne sais quoi d'autre.

Il serait **très intéressant** d'avoir une idée de l'impact de cette action sur les visiteurs. Pour cela, vous allez regarder l'évolution du nombre de messages dans votre livre d'or, l'évolution du nombre de notes attribuées et l'évolution de la note moyenne.

Concrètement, vous devrez afficher le nombre de messages postés chaque semaine, le nombre de notes attribuées et la moyenne de ces notes entre ces deux dates :

- une semaine avant le 25 décembre 2007,
- et une semaine après le 5 janvier 2008.

Les statistiques par année et par mois sont à peu de choses près identiques à celles de l'audit, sauf que cette fois, vous afficherez le nombre de messages postés, le nombre de notes attribuées et la moyenne de ces dernières pour chaque mois depuis la date à laquelle le premier message a été créé.

Fiou, finies ces spécifications ! 😊

## La boîte à outils !

Comme certaines parties des spécifications ne sont pas forcément claires, voici quelques précisions supplémentaires. 😊

Pour le formulaire du premier script et la validité du pseudo, c'est du déjà vu.

Pour la note, c'est du neuf. On sait que la note doit pouvoir prendre six valeurs :

- 1,
- 2,
- 3,
- 4,
- 5,
- et l'absence de note.

Voici une façon très simple de réaliser ceci :

### Code : PHP

```
<?php

if(isset($_POST['note'])) {
    if((int)$_POST['note'] >= 1 && (int)$_POST['note'] <= 5) {
        echo 'La note attribuée est : ' . (int)$_POST['note'] . ' sur 5';
    }
    elseif((int)$_POST['note'] == -1) {
        echo 'Vous n\'avez pas attribué de note';
    }
    else {
        echo 'La note saisie n\'est pas valide';
    }
}

?>
<form action=". ./index.php" method="post">
    Note :
    <select name="note">
        <option value="-1" selected="selected">Pas de note</option>
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
        <option value="4">4</option>
        <option value="5">5</option>
    </select>
    <input type="submit" value="Envoyer" />
</form>
```

On a donc créé une liste déroulante avec les cinq notes possibles et une valeur pour marquer l'absence de note, et pour vérifier si le visiteur en a attribué une, il suffit de vérifier si la valeur de \$\_POST['note'] est égale à la valeur qui marque l'absence de note.

Et pour vérifier si la note est valide, il suffit de vérifier si elle est comprise entre 1 et 5. 😊

Au niveau de la base de données, c'est très facile. Vous avez juste à vous souvenir du comportement de COUNT(), AVG() etc. avec NULL.

Pour ce qui est du message, vous allez avoir un problème. 🤦

Si vous tapez votre message dans le <textarea></textarea> et que vous affichez ce message, vous verrez que les retours à la ligne ne sont pas pris en compte !

En effet, dans un <textarea></textarea>, c'est le caractère \n qui dit « retour à la ligne ». Mais en HTML, ce caractère n'affiche pas de retour à la ligne, il faut utiliser la balise <br />.

Il est très facile de créer une fonction qui met les <br /> là où il faut, tellement facile qu'elle existe déjà : nl2br().

Vous devrez utiliser cette fonction à l'affichage seulement : pas question de polluer votre base de données avec des <br />, on verra pourquoi plus tard. 😊

Mais il reste une chose à signaler : si l'utilisateur rentre un caractère problématique, une apostrophe par exemple, un problème se pose. Imaginons que je rentre dans le formulaire la chaîne **pouet ' pouet** ; la requête SQL générée sera par exemple celle-ci :

#### Code : SQL

```
SELECT col FROM t_table WHERE col = 'pouet ' pouet';
```

Et là, c'est le drame : la chaîne de caractères est fermée avant l'heure, la requête plante, appellons les pompiers ! Pour éviter cela, quand vous utiliserez une chaîne de caractères dans vos requêtes SQL, vous devrez toujours l'échapper correctement avec la fonction mysqli\_real\_escape\_string(mysqli \$link, string \$query). Nous le verrons par la suite, mais échapper correctement les chaînes de caractères vous évitera d'être sujet à des failles de sécurité nommées injections SQL.

Étant donné les informations qu'on aura à stocker, vous devriez pouvoir créer la table vous-mêmes, mais si vous n'y arrivez pas, voici sa structure :

#### Secret (cliquez pour afficher)

```
table dev_gbook :  
gbook_id, entier positif non nul, auto-incrémenté  
gbook_date, date non nulle  
gbook_ip, entier non nul,  
gbook_login, 50 caractères, non nul  
gbook_txt, texte, non nul,  
gbook_note, entier positif plus petit que 256, peut être nul  
Vous ne pensiez quand même pas que j'allais tout faire non plus ? 🍪
```

Pour ce qui est de la possibilité ou non de supprimer des messages, nous allons faire vraiment très simple : il n'est pas question de différencier les visiteurs (c'est pour plus tard, ça 😞).

On va simplement définir une constante et l'utiliser pour savoir si l'on peut ou non supprimer des messages. Concrètement, ça ressemblera à ceci :

#### Code : PHP

```
<?php  
  
define('IS_ADMIN', false); // je définis que le visiteur ne peut  
pas supprimer de message  
  
if(IS_ADMIN) {  
    echo '<a href=".index.php?supprimer=23">Supprimer le message  
n°23</a>';  
}  
  
?>
```

Et en changeant la valeur de la constante, vous pourrez « piloter » les permissions des visiteurs.



Vous devez également vérifier qu'on peut supprimer des messages dans la procédure de suppression ! Si vous ne le vérifiez pas, il suffit qu'un visiteur trouve le bon lien et il pourra supprimer vos messages, même s'il n'en a pas le droit !

Pour le second script, les spécifications paraissent compliquées, mais si vous avez bien compris les chapitres précédents, c'est

vraiment simple. 😊

Pour les statistiques globales (nombre de messages, nombre de notes, date du premier message, etc.), c'est juste un SELECT avec des fonctions d'agrégation (MAX(), MIN(), COUNT(), AVG(), etc.).

L'audit est un peu plus compliqué, il faut mélanger pas mal de choses.

Premièrement, il faut restreindre les lignes qui nous intéressent avec un WHERE. Le critère de restriction est la date des messages, il faut que celle-ci soit comprise entre deux dates (vous devriez penser à BETWEEN, là 😊).

Vous avez les deux dates, mais je vous dit également qu'il faut prendre une semaine en plus pour chaque date.

Vous pourriez faire le calcul et trouver la bonne date par vous-mêmes (il suffit de faire 25-7 et 5+7), mais rappelez-vous du chapitre sur les dates et de la magie des INTERVAL !

Pour le regroupement par semaine, vous devez bien évidemment penser à GROUP BY. Vous devrez regrouper les lignes selon deux critères :

- l'année,
- et la semaine.

Ces deux informations sont très faciles à obtenir avec les fonctions de manipulation des dates. 😊

Les statistiques par mois sont très proches de l'audit ; la seule différence, c'est qu'il n'y aura pas de WHERE et qu'on regroupera les lignes en fonction de l'année et du mois.

Avant de vous laisser, voici un dernier outil : le squelette des scripts.

#### Code : PHP - Premier script

```
<?php
/* Déclaration des constantes */
// Vous définirez des constantes pour la connexion SQL,
// pour la taille (minimale et maximale) du pseudo,
// pour savoir si on peut supprimer des messages,
// et pour l'IP du visiteur
/* Déclaration des fonctions */
// Vous définirez cinq fonctions :
// une pour vérifier si le pseudo est valide,
// une pour vérifier si la note est valide,
// une pour insérer un message (avec la note, le pseudo, etc.),
// une pour supprimer un message,
// et une qui vous retournera l'objet MySQLi_Result (ce que
// retourne mysqli_query) qui contient les messages du livre d'or
/* Traitement des données */
// Connexion à la base
// Si on veut supprimer un message, on le supprime
// Sinon si on veut insérer un message et que les données sont
// valides, on l'insère
// On récupère la liste des messages
/* Affichage des données */
// Si des erreurs sont survenues, on les affiche
// Formulaire d'ajout de message
// Lien vers la page de statistiques
// Affichage des messages du livre d'or
?>
```

#### Code : PHP - Second script

```
<?php
/* Déclaration des constantes */
// Vous définirez des constantes pour la connexion SQL
/* Déclaration des fonctions */
// Vous définirez trois fonctions :
// une qui retournera un tableau contenant les statistiques
// globales,
// une qui retournera l'objet MySQLi_Result (ce que retourne
// mysqli_query) des statistiques de l'audit,
// et une qui retournera l'objet MySQLi_Result (ce que retourne
// mysqli_query) des statistiques par mois
```

```

/* Traitement des données */
// Récupération des statistiques globales
// Récupération des statistiques de l'audit
// Récupération des statistiques par mois
/* Affichage des données */
// Affichage des statistiques globales
// Affichage des statistiques de l'audit
// Affichage des statistiques par mois
?>

```

Bon courage ! 😊

## Correction

Code : SQL

```

CREATE TABLE dev_gbook (
    gbook_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    gbook_date DATETIME NOT NULL,
    gbook_ip INT NOT NULL,
    gbook_login VARCHAR(50) NOT NULL,
    gbook_txt TEXT NOT NULL,
    gbook_note TINYINT NULL
);

```

Code : PHP - Premier script

```

<?php

error_reporting(E_ALL);

/* Déclaration des constantes */
define('SQL_HOST', 'localhost');
define('SQL_USER', 'root');
define('SQL_PASS', '');
define('SQL_DB', 'dev');
define('LOGIN_MIN_LENGTH', 3);
define('LOGIN_MAX_LENGTH', 50);
define('IS_ADMIN', true);
define('USER_IP', ip2long($_SERVER['REMOTE_ADDR']));

/* Déclaration des fonctions */
function isValidLogin($login)
{
    return strlen($login) >= LOGIN_MIN_LENGTH && strlen($login) <=
    LOGIN_MAX_LENGTH && ctype_alnum($login);
}

function isValidNote($note)
{
    return (int)$note === -1 || ((int)$note >= 1 && (int)$note <=
    5);
}

function insertMessage($sqlLink, $message, $login, $note)
{
    if ($note == -1)
    {
        $note = 'NULL';
    }
    else
    {
        $note = (int)$note;
    }
    $query = 'INSERT INTO dev_gbook (gbook_date, gbook_ip,
    gbook_login, gbook_txt, gbook_note)

```

```
VALUES
(NOW(), ' . USER_IP . ', '' . mysqli_real_escape_string($sqlLink,
$login) . '',
'' . mysqli_real_escape_string($sqlLink, $message) . '', " . $note
.'');
    return mysqli_query($sqlLink, $query) or
exit(mysqli_error($sqlLink));
}

function deleteMessage($sqlLink, $id)
{
    return mysqli_query($sqlLink, 'DELETE FROM dev_gbook WHERE
gbook_id = ' . (int)$id);
}

function getMessageList($sqlLink)
{
    $msgQuery = "SELECT gbook_id, DATE_FORMAT(gbook_date, '%d/%m/%Y
&grave; %Hh%i') AS gbook_date,
gbook_ip, gbook_login, gbook_txt
FROM dev_gbook
ORDER BY gbook_id DESC";
    $msgResult = mysqli_query($sqlLink, $msgQuery) or
exit(mysqli_error($sqlLink));
    return $msgResult;
}

/* Traitement des données */
$sqlLink = mysqli_connect(SQL_HOST, SQL_USER, SQL_PASS);
mysqli_select_db($sqlLink, SQL_DB);

$errList = array();
$msgResult = null;

// Suppression d'un message
if(isset($_GET['del'])) && ctype_digit($_GET['del']) && IS_ADMIN)
{
    deleteMessage($sqlLink, $_GET['del']);
}
// Ajout d'un message
if(isset($_POST['message']) && isset($_POST['login']) &&
isset($_POST['note']))
{
    if(!isValidLogin($_POST['login']))
    {
        $errList[] = 'Le login n\'est pas valide.';
    }
    if(!isValidNote($_POST['note']))
    {
        $errList[] = 'La note n\'est pas valide.';
    }
    if(trim($_POST['message']) == '')
    {
        $errList[] = 'Veuillez saisir un message.';
    }

    if(empty($errList))
    {
        if(insertMessage($sqlLink, $_POST['message'],
$_POST['login'], $_POST['note']))
        {
            header('Location: index.php');
            exit;
        }
        else
        {
            $errList[] = 'Une erreur interne est survenue, veuillez
renvoyer votre message.';
        }
    }
}
```

```
}

// Récupération de la liste des messages
$msgResult = getMessageList($sqlLink);

/* Affichage des données */
?>
<html>
    <head>
        <title>Livre d'or</title>
    </head>

    <body>
        <?php if(!empty($errList)) { ?>
        <h3>Erreur(s)</h3>
        <p style="color: red;">
            <?php foreach($errList as $error) { ?>
            <?php echo $error; ?>
            <br />
            <?php } ?>
        </p>
        <?php } ?>

        <h3>Ajouter un message</h3>

        <form action=".index.php" method="post">
            <p>
                Login : <input type="text" name="login" value="<?php
if(isset($_POST['login'])) { echo htmlspecialchars($_POST['login']);
} ?>" />
                <br />
                Note :
                <select name="note">
                    <option value="-1" selected="selected">Pas de
note</option>
                    <option value="1">1 sur 5</option>
                    <option value="2">2 sur 5</option>
                    <option value="3">3 sur 5</option>
                    <option value="4">4 sur 5</option>
                    <option value="5">5 sur 5</option>
                </select>
                <br />
                Message :
                <br />
                <textarea name="message"><?php
if(isset($_POST['message'])) { echo
htmlspecialchars($_POST['message']); } ?></textarea>
                <br />
                <input type="submit" value="Signer le livre d'or" />
            </p>
        </form>

        <h3><a href=".stats.php">Statistiques</a></h3>

        <h3>Liste des messages</h3>

        <?php
if($message = mysqli_fetch_assoc($msgResult))
{
    do { ?>
    <p>
        Écrit le <?php echo $message['gbook_date']; ?> par <?php
echo htmlspecialchars($message['gbook_login']); ?>
        <?php if(IS_ADMIN) { ?>
        (<?php echo long2ip($message['gbook_ip']); ?>, <a
href=".index.php?del=<?php echo $message['gbook_id']; ?
>">Supprimer</a>)
        <?php } ?>
        <blockquote>
            <?php echo
nl2br(htmlspecialchars($message['gbook_txt'])); ?>
```

```

        </blockquote>
    </p>
<?php  } while($message = mysqli_fetch_assoc($msgResult));
} ?>
</body>
</html>

```



Pour tester ce second script, exécutez ce code, ça créera 5 000 lignes plus ou moins aléatoires dans votre table, ce qui permettra d'avoir une meilleure idée de ce que peut faire un tel script. Si vous n'avez pas les mêmes noms de table ou colonnes, il faudra un peu l'adapter.

#### Code : PHP

```

<?php

$data = array();
$interval = array('month', 'week', 'day');
for($i = 0; $i < 50; $i++)
{
    for($j = 0; $j < 100; $j++)
    {
        $data[] = '(NOW() - interval ' . mt_rand(0, 50) .
$interval[mt_rand(0, 2)] . ', ' .
            . mt_rand() . ", 'login' . ((($i+1)*($j+1)) . '',
'message' . ((($i+1)*($j+1))
            . " , " . (($j*$i)%10 === 0 ? 'NULL' : mt_rand(1,
5)) . ')';
    }
    mysqli_query($sqlLink, 'insert into dev_gbook(gbook_date,
gbook_ip, gbook_login, gbook_txt, gbook_note) values' . implode(',', $data)) or exit(mysqli_error($sqlLink));
    $data = array();
}

?>

```

#### Code : PHP - Second script

```

<?php

/* Déclaration des constantes */
define('SQL_HOST', 'localhost');
define('SQL_USER', 'root');
define('SQL_PASS', '');
define('SQL_DB', 'dev');

/* Déclaration des fonctions */
function getGlobalStats($sqlLink) {
    $result = mysqli_query($sqlLink, 'SELECT COUNT(*),
COUNT(gbook_note), AVG(gbook_note), MIN(gbook_date), MAX(gbook_date)
FROM dev_gbook') or exit(mysqli_error($sqlLink));
    return mysqli_fetch_row($result);
}

function getAuditStats($sqlLink) {
    $query = "SELECT YEAR(gbook_date) AS group_year,
MONTH(gbook_date) AS group_month,
WEEK(gbook_date) AS group_week, COUNT(*) AS nb_mess,
COUNT(gbook_note) AS nb_note, AVG(gbook_note) AS avg_note
FROM dev_gbook
WHERE gbook_date BETWEEN ('2007-12-25' - INTERVAL 1 WEEK) AND
('2008-01-05' + INTERVAL 1 WEEK)
GROUP BY group_year, group_week
ORDER BY group_year DESC, group_week DESC";

```

```
$result = mysqli_query($sqlLink, $query) or
exit(mysqli_error($sqlLink));
return $result;
}

function getMonthStats($sqlLink) {
    $query = 'SELECT YEAR(gbook_date) AS group_year,
MONTH(gbook_date) AS group_month,
COUNT(*) AS nb_mess, COUNT(gbook_note) AS nb_note, AVG(gbook_note)
AS avg_note
FROM dev_gbook
GROUP BY group_year, group_month
ORDER BY group_year DESC, group_month DESC';
    $result = mysqli_query($sqlLink, $query) or
exit(mysqli_error($sqlLink));
return $result;
}

/* Traitement des données */
$sqlLink = mysqli_connect(SQL_HOST, SQL_USER, SQL_PASS);
mysqli_select_db($sqlLink, SQL_DB);

// Statistiques globales
list($msgCount, $noteCount, $noteAvg, $dateMin, $dateMax) =
getGlobalStats($sqlLink);

// Audit de la période des fêtes 2007-2008
$testResource = getAuditStats($sqlLink);

// Statistiques par mois et par année
$yearResource = getMonthStats($sqlLink);

/* Affichage des données */
?>
<html>
    <head>
        <title>Statistiques du livre d'or</title>
    </head>

    <body>
        <h3>Général</h3>
        <p>
            <?php echo $msgCount; ?> messages ont été postés entre le
<?php echo $dateMin; ?> et le <?php echo $dateMax; ?>.
            <br />
            <?php echo $noteCount; ?> notes ont été postées, la note
moyenne est <?php echo $noteAvg; ?>
        </p>

        <h3>Statistiques des fêtes 2008</h3>
        <?php while($test = mysqli_fetch_assoc($testResource)) { ?>
            <h4><?php echo $test['group_month']; ?>/<?php echo
$test['group_year']; ?>, semaine <?php echo $test['group_week']; ?
></h4>
            <p>
                Nombre de messages : <?php echo $test['nb_mess']; ?>
                <br />
                Nombre de notes : <?php echo $test['nb_note']; ?>
                <br />
                Moyenne des notes : <?php echo $test['avg_note']; ?>
            </p>
            <?php } ?>

            <h3>Statistiques par années et par mois</h3>
            <?php while($year = mysqli_fetch_assoc($yearResource)) { ?>
                <h4><?php echo $year['group_month']; ?>/<?php echo
$year['group_year']; ?></h4>
                <p>
                    Nombre de messages : <?php echo $year['nb_mess']; ?>
                    <br />
```

```

Nombre de notes : <?php echo $year['nb_note']; ?>
<br />
Moyenne des notes : <?php echo $year['avg_note']; ?>
</p>
<?php } ?>
</body>
</html>

```

## Discussion

Alors, comment c'était ? 😊

Personnellement, j'ai trouvé ça amusant. 🎉

Généralement, un TP ne fait que tester vos connaissances, mais celui-ci est différent !

Regardez ma correction : elle apporte une nouveauté, mais surtout beaucoup de problèmes... 😬

La nouveauté est dans le second script et se trouve à la ligne n° 42 que voici :

### Code : PHP

```

<?php
list($msgCount, $noteCount, $noteAvg, $dateMin, $dateMax) =
getGlobalStats($sqlLink);
?>

```

`list()` est un objet très pratique, ça permet de déclarer des variables et de leur affecter des valeurs venant d'un array. Regardez ce que produit ce code :

### Code : PHP

```

<?php

$date = '24-03-2008'; // la date actuelle
list($jour, $mois, $annee) = explode('-', $date); // je la
décompose pour avoir le jour, le mois et l'année

echo 'Cette ligne a été écrite le ' . $jour . '/' . $mois . '/'
. $annee;

?>

```

Comme vous le voyez, `list()` affecte les valeurs de l'array aux variables passées en paramètres. Ces variables n'ont pas besoin d'être déclarées avant.

Dans l'exemple, `$jour` prend la valeur de l'index 0 de l'array retourné par `explode()`, `$mois` la valeur de l'index 1 et `$annee` la valeur de l'index 2.

`list()` ne peut utiliser que des index numériques, ne soyez donc pas étonnés d'avoir une erreur si votre array ne contient que des index non-numériques. 😊

Mais à votre avis, pourquoi `list()` permet d'utiliser des variables qui n'ont pas été déclarées précédemment ?

La réponse est simple : `list()` n'est pas une fonction mais une structure du langage, au même titre que `isset()` ou `empty()`. Et comme vous le savez sans doute déjà, les structures du langage peuvent faire certaines choses qui nous sont interdites, comme utiliser des variables non-déclarées.

Passons maintenant aux problèmes. Le plus évident est celui lié aux constantes. Jusqu'à présent, elles nous permettaient de ne pas avoir à changer du code en plein milieu du fichier, de ne pas avoir à changer plusieurs fois la même chose.

Mais ne trouvez-vous pas fastidieux de devoir déclarer les mêmes constantes dans plusieurs scripts ? 😊

D'autant plus que de cette façon, vous devrez éditer tous vos fichiers si vous changez une constante... Plutôt ennuyeux !

Mais heureusement pour nous, les créateurs de PHP sont très gentils et ont pensé à nous. Ainsi, il existe une méthode, appelée inclusion, qui nous permettra de nous affranchir de ce problème.

Concrètement, on déclarera nos constantes à un endroit, et grâce à une structure de PHP, les constantes seront accessibles très facilement sur tous nos scripts ! Et de cette façon, nous n'aurons plus qu'à modifier une seule fois les constantes. 😊

Le second problème, c'est nl2br(), htmlspecialchars(), ip2long() et les while(\$var = mysqli\_fetch\_assoc(\$struc)).

Mais pourquoi donc ?

Regardez les commentaires du squelette des scripts, j'ai une partie *traitement des données* et une partie *affichage des données*. Et pourtant, je retrouve nl2br(), htmlspecialchars(), ip2long() et des while(\$var = mysqli\_fetch\_assoc(\$struc)) dans la partie *affichage des données*.

Je mélange donc traitement et affichage, ce qui est **très grave** !

Totalement impardonnable, pendez-le !

Heureusement pour moi et mon petit cou, il existe des solutions. Sauf que vous n'en connaissez aucune pour le moment... Ne vous en faites pas, bientôt les *tableaux (arrays) multi-dimensionnels* et les *itérateurs* n'auront plus de secret pour vous et me sauveront de la pendaison.

Enfin, dernier petit détail : vous pouvez voir que pour vérifier si le formulaire a été soumis, j'ai utilisé trois fois la fonction isset(). Il faut savoir que isset() n'est pas limité à un paramètre, cette fonction peut en fait prendre autant de paramètres qu'on le veut. Ainsi, ces deux instructions sont strictement équivalentes :

#### Code : PHP

```
<?php  
isset($_POST['message']) && isset($_POST['login']) &&  
isset($_POST['note']);  
isset($_POST['message'], $_POST['login'], $_POST['note']);  
?>
```

Et voilà, vous avez réalisé un petit script qui vous permettra de récolter l'avis des visiteurs sur votre site et d'établir des statistiques, que demander de plus ?

En prime, vous avez appris quelques détails et avez entendu parler de concepts encore inconnus.

Le prochain chapitre sera le premier d'une petite série qui ne traitera que du SQL et qui marquera la fin de cette partie.

## Les index et sous-requêtes

S'il y a bien une chose à gérer correctement pour obtenir de bonnes performances de la part de votre SGBDR, c'est l'indexation. Toutefois, bien qu'indispensable, la gestion des index n'est pas des plus simples.

Pour ce qui est des sous-requêtes, posez-vous la question suivante : que retourne une requête de type SELECT ?

### Un index, pourquoi ?

Imaginez que vous possédez un livre et que pour une sombre raison, on vous demande de lister les pages qui parlent d'un point précis, les index par exemple. Bien que vous ayez déjà lu et relu ce livre qui vous passionne et vous tient en haleine des heures durant, il est plus qu'improbable que vous soyez capables de faire ce *listing* directement.

Pour accomplir cette tâche ô combien pénible, vous devrez lire la totalité des pages du livre une à une pour trouver toutes celles qui parlent des index. Après quelques heures, vous terminez enfin ce *listing*, bien épuisés. Et là, vous êtes frappés d'une illumination : pourquoi diable n'y a-t-il rien dans ce livre qui nous indique à quelle(s) page(s) trouver les informations qui nous intéressent ?

Cette page manquante et fort utile est un index, et voilà l'utilité d'un index : accéder très rapidement à des données en particulier. Il est en effet beaucoup plus rapide de lire deux ou trois pages d'index que de lire les 600 pages du livre. 

Pour les bases de données, c'est exactement pareil. Sans index, le SGBDR devra lire la totalité des lignes de la table pour trouver celles qui correspondent à notre demande, ce qui prend du temps et consomme des ressources ; c'est assurément très mauvais pour les performances.

### Les différents index et leur gestion

Maintenant qu'on sait ce qu'est un index, il faut savoir quels sont les index que MySQL nous fournit et quelles sont leurs caractéristiques.

MySQL nous fournit précisément trois types d'index :

- les index simples ;
- les index uniques ;
- les clés primaires.

Les index simples, comme leur nom l'indique, sont de simples index : ils n'ont aucune caractéristique, ils nous permettent seulement d'accéder très rapidement à des données précises.

Les index uniques sont des index simples auxquels on ajoute une contrainte d'unicité, c'est-à-dire que les données indexées ont toutes des valeurs différentes : il ne peut y avoir plusieurs données qui ont la même valeur.

Les clés primaires sont quant à elles des index uniques auxquels on ajoute une contrainte de non-nullité. Rappelez-vous, on pouvait marquer l'absence de valeur avec NULL. La clé primaire empêche d'avoir des données sans valeur. **De plus, il ne peut y avoir qu'une seule clé primaire par table !** La raison de cette limitation est la valeur **sémantique** de la clé primaire : elle indique que les colonnes qu'elle indexe permettent d'identifier une et une seule ligne de la table, c'est pourquoi on associe l'auto-incrémentation à une clé primaire.

Résumons-nous : nous savons à quoi servent les index, quels sont les types d'index et nous savons quelles sont les caractéristiques de chaque type d'index.

Mais ne serait-il pas intéressant de se demander ce que l'on peut indexer ? 

Vous ne pouvez pas indexer n'importe quoi ; indexer une base de données ne veut rien dire, par exemple.

Il n'y a qu'une chose que vous pouvez indexer : un groupe de colonnes. Mais bien évidemment, ce groupe peut n'être constitué que d'une seule colonne. 

Passons maintenant à la création des index. Il y a deux façons de créer des index, soit à la création de la table, soit à la modification de la table. Voici un exemple de table inutile qui montre la création des trois types d'index :

#### Code : SQL

```
CREATE TABLE dev_inutile (
    in_a INT,
    in_b INT,
    in_c CHAR(40),
    KEY index_in_a (in_a),
    UNIQUE KEY unique_in_b (in_b),
    PRIMARY KEY (in_c)
);
```

Les quatre premières lignes de cette requête doivent vous être familières, on crée simplement une table avec trois colonnes. Les trois suivantes sont beaucoup plus intéressantes, par contre.

La première de ces trois lignes crée un index simple, la seconde crée un index unique et la troisième crée une clé primaire. Voici la syntaxe plus générale pour créer ces index :

#### Code : SQL

```
KEY [index_name] (col_list)
UNIQUE KEY [index_name] (col_list)
PRIMARY KEY (col_list)
```

Pour les index simples et uniques, vous voyez qu'on peut leur donner un nom. Ce nom sert simplement à identifier l'index et il est facultatif (MySQL en donnera un par défaut).

**col\_list** indique la liste des colonnes sur lesquelles porte l'index, le nom des colonnes est séparé par une virgule. Ainsi, si je souhaite créer un index sur les trois colonnes de ma table, j'écrirai ceci :

#### Code : SQL

```
KEY (in_a, in_b, in_c)
```

 L'ordre dans lequel vous écrivez le nom des colonnes est très important, on verra pourquoi par la suite.

La seconde façon de créer des index passe par une modification de la table, et donc par une requête de type ALTER TABLE. Voici la syntaxe pour créer chaque type d'index :

#### Code : SQL

```
ALTER TABLE table_name ADD KEY [index_name] (col_list); -- ajoute un
index simple
ALTER TABLE table_name ADD UNIQUE KEY [index_name] (col_list); -- ajoute un index unique
ALTER TABLE table_name ADD PRIMARY KEY (col_list); -- ajoute une clé
primaire
```

 Il est possible que vous voyiez parfois ADD INDEX au lieu de ADD KEY. Sachez que INDEX et KEY sont bien souvent des synonymes.

La suppression d'index est à peu près aussi simple que leur création, on passe encore par un ALTER TABLE que voici :

#### Code : SQL

```
ALTER TABLE table_name DROP KEY index_name; -- supprime l'index
simple ou unique qui porte le nom index_name
ALTER TABLE table_name DROP PRIMARY KEY; -- supprime la clé
primaire
```

Enfin, dernier outil pour la gestion des index : la liste des index. Cette liste s'obtient avec cette requête :

#### Code : SQL

```
SHOW INDEX FROM table_name;
```

Pour le moment, ça ne vous servira pas à grand-chose étant donné que vous ne savez pas ce que ces données veulent dire, mais

nous allons vite rectifier cela.

À propos des colonnes à longueur variable, c'est-à-dire CHAR, VARCHAR, BINARY, VARBINARY, TEXT et BLOB, sachez que vous pouvez spécifier la taille de l'index.



Pour les colonnes de type TEXT et BLOB, vous **devez** spécifier la taille de l'index

Si vous pouvez filtrer efficacement avec les deux premiers caractères par exemple, pourquoi indexer tous les caractères ? Ça rendra l'index plus volumineux, pour rien.

Pour spécifier la taille de l'index, la syntaxe est la suivante :

#### Code : SQL

```
ALTER TABLE table_name ADD KEY  
[index_name] (col_name(taille_index) [,col_name2[,...]]);
```

## Où placer des index ?

La question peut paraître stupide, mais où, sur quelles colonnes placer des index ?

Si la question paraît stupide, la réponse le paraîtra aussi : ça dépend. En effet, il n'y a pas de méthode miracle qui décrive précisément sur quelles colonnes placer des index. Tout ce qu'on peut faire, c'est considérer un cas particulier (un livre d'or par exemple), appliquer quelques principes généraux et tester nos index.

Peut-être vous demandez-vous pourquoi, si les index améliorent les performances, ne pas en mettre partout ?

S'il est vrai que les index améliorent les performances, il faut également signaler qu'ils les dégradent. En effet, les index améliorent bien les performances, mais uniquement lors des lectures de données. Eh oui, si vous modifiez des données ou en supprimez, il va falloir modifier les index en conséquence, ce qui prend un peu de temps. Et si vous avez beaucoup d'index, ça peut prendre beaucoup de temps.

Alors, où mettre des index ? 😊

D'une manière générale, les colonnes susceptibles de nécessiter un index sont celles qui sont souvent utilisées pour :

- du filtrage ;
- des regroupements ;
- du tri.

Bien évidemment, il faut que ces index soient utilisés. Si vous avez une requête qui utilise une colonne pour du filtrage mais que cette requête n'est exécutée qu'une fois toutes les six lunes et qu'elle est la seule à utiliser ladite colonne pour du filtrage, y mettre un index n'est pas forcément une bonne idée.

Prenons un exemple : le livre d'or du précédent TP. Si nous regardons les requêtes, on trouve :

- une insertion ;
- une suppression avec un WHERE portant sur la colonne **gbook\_id** ;
- une sélection avec un ORDER BY portant sur la colonne **gbook\_id**.

La colonne **gbook\_id** est donc très souvent utilisée pour du tri et parfois pour un filtrage, c'est un candidat **idéal** pour un index. Mais en réalité, nous n'aurons pas à le faire, et pour cause, cette colonne est déjà dotée d'un index puisqu'on lui a mis une clé primaire. 🍪

Dans certains cas, en utilisant ces quelques généralités, vous serez amenés à pouvoir mettre plusieurs index sur une même table. Toutefois, vous devez savoir que lors de l'exécution d'une requête, MySQL n'utilisera qu'un et un seul index. Dans la plupart des cas, c'est l'index le plus pertinent qui sera utilisé. Pour juger de la pertinence d'un index, on regarde le nombre de lignes qui seront retournées en l'utilisant. Si vous avez le choix entre un index qui retourne 1000 lignes sur 1 000 000 et un index qui retourne 10 lignes sur 1 000 000, vous choisirez bien évidemment le second, car vous aurez bien moins de lignes à traiter par la suite. Par exemple, en considérant que vous avez un index sur **col1** et un index sur **col2**, cette requête utilisera l'index qui sera jugé le plus pertinent :

#### Code : SQL

```
SELECT col FROM t_table WHERE col1 = 34 AND col2 = 12345;
```

Si par contre vous avez un index portant sur (**col1, col2**), celui-ci sera utilisé pour trouver directement les lignes car cet index sur deux colonnes serait bien meilleur qu'un index sur une seule des deux colonnes.

Mais attention, car si nous avions écrit la requête comme ceci :

#### Code : SQL

```
SELECT col FROM t_table WHERE col2 = 12345 AND col1 = 34;
```

Alors l'index portant sur nos deux colonnes n'aurait pas pu être utilisé ! Et pour cause : nous avons un index qui porte sur (**col1, col2**), mais pas sur (**col2, col1**).

De façon analogue, considérons ces requêtes :

#### Code : SQL

```
SELECT col FROM t_table WHERE col1 = 3;
SELECT col FROM t_table WHERE col2 = 12345;
```

Dans le cas de la première requête, l'index portant sur (**col1, col2**) pourra être utilisé car (**col1**) est un motif qui correspond à (**col1, col2**).

Mais dans le cas de la seconde requête, (**col2**) n'est pas un motif qui correspond à (**col1, col2**), l'index ne sera donc pas utilisé. Soyez donc bien vigilants avec les index portant sur plusieurs colonnes, l'ordre des colonnes est important aussi bien à la création de l'index que dans les requêtes qui devront s'en servir.

Parfois, le SGBDR n'est pas capable de déterminer quel index est le plus pertinent, ou bien il peut utiliser un index qui n'est pas celui que vous voudriez qu'il utilise. Dans pareille situation, vous pouvez forcer le SGBDR à utiliser l'index que vous voulez. Pour MySQL, regardez du côté des commandes USE INDEX, IGNORE INDEX et FORCE INDEX.

Il peut aussi arriver que le SGBDR décide de n'utiliser aucun index. Par exemple, on peut imaginer que les index disponibles soient tellement peu pertinents que lire la table sera plus rapide que d'utiliser un index.

### Surveiller les index

Tantôt, je vous ai montré la requête SHOW INDEX, il est temps de détailler un peu ce qu'elle nous retourne. Elle nous retourne un jeu de résultats dans lequel chaque ligne représente une colonne sur laquelle porte un index.

Je ne vais pas détailler toutes les informations données, mais seulement les plus utiles.

Colonne	Information donnée
Table	Indique le nom de la table sur laquelle porte l'index.
Non_unique	Indique si l'index impose une contrainte d'unicité (0) ou non (1).
Key_name	Indique le nom de l'index.
Column_name	Indique le nom de la colonne sur laquelle porte l'index.
Cardinality	Indique la pertinence de l'index (en réalité, une estimation du nombre de valeurs uniques dans l'index) : plus la valeur est grande, plus il y a de chances que l'index soit utilisé.
Sub_part	Dans le cas d'un index portant sur une colonne à taille variable, indique la taille de l'index.
Null	Indique si la colonne peut contenir NULL (YES) ou non (chaîne vide).

Prenons un exemple, cette table et ces index :

#### Code : SQL

```
CREATE TABLE test_index (
    col1 INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
```

```

        col2 INT NULL,
        col3 INT NOT NULL
    );

ALTER TABLE test_index ADD KEY (col1, col2, col3);

```

Avec un SHOW INDEX, nous obtenons ceci :

#### Code : Console

```

+-----+-----+-----+-----+-----+-----+
| Table      | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Car
+-----+-----+-----+-----+-----+-----+
| test_index |          0 | PRIMARY  |           1 | col1        | A          |
| test_index |          1 | col1     |           1 | col1        | A          |
| test_index |          1 | col1     |           2 | col2        | A          |
| test_index |          1 | col1     |           3 | col3        | A          |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

```

Pour surveiller l'utilisation de vos index, une commande **fondamentale** existe : EXPLAIN. Cette commande vous permet d'obtenir des informations sur des SELECT, et parmi ces informations, on retrouve notamment les index qui auraient pu être choisis par MySQL, l'index qui a été choisi par MySQL et la taille de cet index. C'est grâce à cette commande que vous pourrez savoir si vos index sont utilisés comme ils le devraient, si vous avez créé des index inutiles, si vous n'avez pas créé d'index sur certaines colonnes fortement sollicitées, etc.

Pour l'utiliser, il vous suffit d'ajouter le mot-clé EXPLAIN devant le SELECT :

#### Code : SQL

```
EXPLAIN SELECT col FROM t_table WHERE [...] ;
```

Cette commande vous retournera un jeu de résultats dans lequel chaque ligne représente une table utilisée dans le SELECT (nous verrons ça plus tard).

Je ne vais pas détailler toutes les informations, seulement celles qui sont utiles à la gestion des index.

Colonne	Information donnée
table	Indique sur quelle table porte le SELECT.
possible_keys	Indique quels index sont susceptibles d'être utilisés. Si la colonne est vide, aucun index ne peut être utilisé, peut-être faudrait-il envisager d'en créer un.
key	Le nom de l'index utilisé ou NULL si aucun index n'a été utilisé.
key_len	La taille de l'index utilisé ou NULL si aucun index n'est utilisé. Cette colonne vous permet, dans le cas d'un index portant sur plusieurs colonnes, de savoir quelles parties de l'index ont vraiment été utilisées.
rows	Indique une estimation du nombre de lignes que MySQL devra examiner pour exécuter la requête. Plus le nombre est faible par rapport au nombre de lignes dans la table, mieux c'est.
extra	Donne des informations supplémentaires sur la façon dont la requête sera exécutée, <a href="#">voir la documentation</a> .

Les index doivent être entretenus, donc si vous procédez à de très nombreuses modifications sur la table, pensez à lancer une analyse de la table avec la requête ANALYSE TABLE :

#### Code : SQL

```
ANALYZE TABLE table_name;
```



## Les sous-requêtes

Sous-requête, sous-requête, l'idée de base des sous-requêtes est d'utiliser ce que retournent des requêtes SELECT dans d'autres requêtes. En gros, on imbrique des requêtes à la manière des poupées russes.

Mais avant de faire des sacs de nœuds (non, vous ne ferez pas ça !) avec nos requêtes, intéressons-nous à ce que les requêtes SELECT retournent. Non, ne criez pas au scandale, je sais que je vous ai toujours dit qu'une requête de ce type renvoyait un jeu de résultats organisé en lignes composées de colonnes. Mais cette affirmation, bien que vraie, est trop généraliste.

Considérons la table suivante et ces trois requêtes SELECT :

Code : SQL

```
CREATE TABLE dev_sreq (
    col1 INT NOT NULL,
    col2 INT
);
ALTER TABLE dev_sreq ADD UNIQUE KEY dev_sreq_col1_uk (col1);
SELECT col2 FROM dev_sreq WHERE col1 = 45; -- requête 1
SELECT col1 FROM dev_sreq WHERE col1 > 45; -- requête 2
SELECT col1, col2 FROM dev_sreq WHERE col1 >= 45; -- requête 3
```

De par la contrainte d'unicité, nous pouvons garantir que la première requête ne renverra qu'une et une seule ligne ne contenant qu'une et une seule colonne.

La seconde requête renverra un ensemble de lignes ne contenant qu'une et une seule colonne.

La dernière requête, quant à elle, renverra un ensemble de lignes composées d'une ou plusieurs colonnes.

Je ne vous apprends sûrement rien en vous disant cela.

Ce qu'il faut savoir, c'est qu'on peut considérer ces trois « types » de valeur de retour comme ceci :

- une ligne contenant une colonne peut être considérée comme une valeur unique ;
- un ensemble de lignes contenant une colonne peut être considéré comme un ensemble de valeurs uniques, une liste ;
- un ensemble de lignes peut être considéré comme une table.

Et le plus beau, c'est que MySQL considère ça exactement de cette façon.

Prenons un exemple concret : vous avez une table d'utilisateurs et une table de messages. Vous souhaitez obtenir la liste des messages écrits par un utilisateur dont vous connaissez le nom d'utilisateur, mais dans la table des messages, ce n'est pas le nom d'utilisateur mais son identifiant, obtenu *via* l'auto-incrémation par exemple, que vous avez stocké.

Pour obtenir la liste tant désirée, vous seriez sûrement amenés à utiliser une solution en deux temps :

- vous récupérez d'abord l'identifiant de l'utilisateur *via* une requête ;
- et ensuite, vous utilisez le résultat de la requête précédente pour filtrer la table des messages.

Cela se traduirait par exemple par ces deux requêtes :

Code : SQL

```
SELECT user_id FROM t_user WHERE user_name = 'nom';
SELECT msg_text, ... FROM t_message WHERE msg_userId = X;
-- X étant la valeur obtenue via la requête précédente
```

Un des problèmes de cette solution, c'est que vous allez devoir envoyer deux requêtes à MySQL et que vous recevrez donc deux réponses, ce qui fait transiter des données pour rien en plus d'alourdir le code de vos applications puisqu'il faut gérer le possible échec de la première requête.

La seconde requête nécessite un paramètre de filtre, l'identifiant de l'utilisateur. Cet identifiant est un entier, c'est-à-dire une valeur constante, unique. Comme dans la première requête, on ne sélectionne qu'une colonne, et qu'il n'y a normalement qu'un utilisateur associé à un nom, on sait que cette requête retournera une et une seule ligne contenant une et une seule colonne et que MySQL considérera ça comme une valeur unique.

Et grâce à cela, on va pouvoir résumer ces deux requêtes en une seule :

#### Code : SQL

```
SELECT msg_txt, ...
FROM t_message
WHERE msg_userId = (SELECT user_id FROM t_user WHERE user_name =
'nom');
```

De cette façon, vous n'avez plus qu'une requête à exécuter, c'est-à-dire que vous n'enverrez plus qu'une requête à MySQL, que vous ne ferez pas transiter de données inutilement et que vous n'aurez plus qu'à vérifier le bon déroulement d'une requête au lieu de deux.

Elle est pas belle, la vie ? 😊

Vous pouvez utiliser des sous-requêtes renvoyant une valeur unique un peu partout, aussi bien dans le WHERE que dans le GROUP BY, l'ORDER BY, les valeurs à sélectionner et partout où vous pouvez utiliser une valeur constante ! Exemple :

#### Code : SQL

```
SELECT col * (SELECT truc FROM t_truc WHERE id = x),
       (SELECT lala FROM t_lala WHERE id = z)
  FROM t_table
 GROUP BY col2 % (SELECT bidule FROM t_bidule WHERE id = y);
```

Intéressons-nous maintenant aux requêtes SELECT renvoyant un ensemble de lignes ne contenant qu'une et une seule colonne, c'est-à-dire une liste. Imaginons que vous ayez deux tables, une table contenant vos utilisateurs et une table contenant une liste d'utilisateurs qui sont par exemple sanctionnés pour l'une ou l'autre faute. Vous souhaitez obtenir la liste des utilisateurs non-sanctionnés, comment feriez-vous ?

Actuellement, la solution la plus élégante serait de faire un SELECT sur la table des sanctionnés, de récupérer le résultat dans votre script et d'injecter le tout dans une seconde requête qui filtrera à grand coup de NOT IN. Souvenez-vous, IN est un opérateur qui permet de vérifier si une valeur est égale à une valeur des valeurs d'une liste.

Oh, une liste, ça ne vous rappelle rien ? Eh si, ça vous rappelle qu'une requête SELECT qui renvoie un ensemble de lignes ne contenant qu'une seule colonne est considérée comme une liste. Vous pouvez donc utiliser une sous-requête renvoyant ce type de résultat dans le IN.

L'écriture de la requête renvoyant l'ensemble des utilisateurs non-sanctionnés devient triviale :

#### Code : SQL

```
SELECT user_name
  FROM t_user
 WHERE user_id NOT IN (SELECT sanction_userId FROM t_sanction);
```

Courage, il ne reste plus qu'un type de résultat possible pour les requêtes SELECT : l'ensemble de lignes contenant plusieurs colonnes, c'est-à-dire une table. À l'heure actuelle, le seul endroit où vous pourrez utiliser ce type de sous-requête est la clause FROM. Exemple :

#### Code : SQL

```
SELECT col1, col2
  FROM (SELECT col1, col2 FROM t_table) AS maTable;
```

Quand vous utilisez une sous-requête comme une table, vous devez donner un surnom au résultat de la sous-requête. Si vous ne le faites pas, vous serez confrontés à cette erreur :

**Code : Console**

```
ERROR 1248 (42000): Every derived table must have its own alias
```

Ce qui signifie, en français, que toutes les tables dérivées doivent avoir leur propre surnom.

Nous verrons par la suite d'autres clauses qui permettront d'utiliser des sous-requêtes retournant une table. 😊

À présent, vous devriez être en mesure de vous débrouiller avec les index. Bien sûr, vous devrez toujours faire attention à vos index : vérifier qu'ils sont pertinents, qu'il n'en manque pas, utiliser EXPLAIN sur les requêtes qui semblent lentes, les entretenir, etc.

Aussi, soyez bien conscients que lorsque vous testerez vos index, il est **indispensable** que la table corresponde à ce que vous aurez en conditions réelles.

Si vous estimatez que vous aurez cinq milliards de lignes dans votre table, ne testez pas vos index avec seulement cent lignes dans la table, ça ne serait absolument pas représentatif des conditions réelles, et vos tests n'auraient donc aucune valeur.

Les sous-requêtes, quant à elles, vous permettront d'éviter des allers-retours entre votre SGBDR et vos applications, tout en évitant parfois de lourds traitements du côté de vos applications.

Au terme de ces 12 chapitres, vous devez être à même d'administrer partiellement vos bases de données, d'utiliser différents types de requêtes dont les insertions, les suppressions et les sélections.

Toutefois, la vraie puissance des bases de données, celle qui réside dans le « R » de SGBDR, vous échappe encore !

Il reste quelques chapitres pour terminer cette partie. À l'affiche : les dernières clauses utilisées dans les requêtes SELECT, les **opérateurs ensemblistes**, les **jointures** internes, externes, croisées et naturelles, les **clés étrangères**, vrais piliers du relationnel, et enfin tous les petits plus indispensables tels que les **transactions** ! 😊

## Partie 3 : PHP, MySQL, allons plus loin

Pour le moment, les seuls chapitres de cette partie concernent la programmation orientée objet. Malheureusement, je n'ai pas encore eu la motivation d'en écrire certains nécessaires à la compréhension de ceux-ci.

Ainsi, je parle dans ces chapitres de certaines notions (les références par exemple) comme si j'en avais déjà parlé. Il est donc nécessaire de vous documenter sur ces notions parallèlement à la lecture de ces chapitres.

J'espère pouvoir vous fournir rapidement les chapitres manquants !

### Retour sur les fonctions et les tableaux

Dans ce chapitre, nous allons revenir sur deux points abordés dans la première partie de cours : les fonctions et les tableaux. Bien que déjà consistants, ces points n'avaient pas été complètement traités. Ce chapitre va tenter d'y remédier.

#### Des fonctions à nombre de paramètres variable

Nous avons vu qu'il est possible de passer des paramètres à une fonction ; malheureusement, nous sommes limités aux paramètres définis par la déclaration de la fonction. Si nous désirons créer une fonction qui peut faire la moyenne d'un nombre indéterminé de valeurs par exemple, nous sommes un peu coincés.

Toutefois, ce n'est pas impossible à réaliser grâce à un comportement du PHP : il ne génère pas d'erreur si nous donnons trop de paramètres à une fonction. Exemple :

**Code : PHP**

```
<?php  
  
function avg()  
{  
}  
  
avg(5, 6, 7, 34);
```

Passer un nombre indéterminé de paramètres à la fonction ne pose donc pas de problème, mais encore faut-il que nous puissions les récupérer. Il se trouve que PHP met à notre disposition trois fonctions qui permettent de gérer les paramètres :

- `func_num_args()` qui permet d'obtenir le nombre de paramètres passés à la fonction ;
- `func_get_arg()` qui permet d'obtenir un paramètre passé à la fonction ;
- `func_get_args()` qui permet d'obtenir tous les paramètres passés à la fonction.

La chose intéressante, c'est que ces fonctions ne se limitent pas aux paramètres que nous avons déclarés dans la fonction mais bien à tous les paramètres donnés à la fonction. Exemple :

**Code : PHP**

```
<?php  
  
function avg()  
{  
    var_dump(func_num_args());  
}  
  
avg(5, 6, 7, 34);
```

Dès lors, il devient très aisément d'implémenter cette fonction, je vous laisse faire. 😊

**Secret (cliquez pour afficher)**

**Code : PHP**

```
<?php  
  
function avg($a, $b)
```

```
{  
    return array_sum(func_get_args()) / func_num_args();  
}  
  
var_dump(avg(5, 6, 7, 34));
```

Couplé avec array\_sum(), fonction native qui calcule la somme d'un tableau, le code est très concis. Si j'ai donné deux paramètres dans la déclaration de ma fonction, c'est d'abord parce que calculer la moyenne de moins de deux nombres est un peu inutile, et ensuite pour éviter le cas où l'utilisateur ne donnerait aucun paramètre. Si je n'avais pas forcé ces deux paramètres et que l'utilisateur n'avait pas donné de paramètre, une erreur de division par zéro aurait été émise puisque func\_num\_args() aurait retourné zéro. En forçant ces deux paramètres, j'évite de devoir traiter ce cas dans la fonction et une erreur sera malgré tout émise si moins de deux paramètres sont donnés.

Puisque les paramètres non-déclarés dans la fonction ne sont pas associés à des variables, il n'est pas possible d'utiliser des noms pour les identifier. C'est pour cela que func\_get\_arg() et func\_get\_args() utilisent des indices numériques basés sur la position du paramètre lors de l'appel de la fonction. Dans l'exemple donné, 5 est le paramètre 0, 6 est le paramètre 1, 7 est le paramètre 2 et 34 est le paramètre 3. Vérifions cela :

#### Code : PHP

```
<?php  
  
function avg()  
{  
    var_dump(func_get_arg(2));  
    var_dump(func_get_args());  
}  
  
avg(5, 6, 7, 34);
```

Plusieurs fonctions natives permettent de passer un nombre variable de paramètres — printf() par exemple — et il peut être très pratique de pouvoir créer des fonctions acceptant un nombre variable de paramètres.

#### [function\\_exists\(\)](#)

Au fil des versions du PHP, des fonctions peuvent apparaître et disparaître : c'est l'évolution, nous n'y pouvons rien. Or, par souci de compatibilité avec les différentes versions du PHP, il peut être nécessaire de créer soi-même ces fonctions manquantes. Toutefois, si nous tentons de définir une fonction qui existe déjà, une jolie erreur pointera son minois :

#### Code : Console

```
Fatal error: Cannot redeclare strlen() in C:\wamp\www\TutoOO\index.php on line 3
```

L'idée est donc d'encapsuler la déclaration de cette fonction dans une condition qui ne sera vraie que si la fonction n'existe pas. Pour tester l'existence d'une fonction, nous utiliserons la fonction function\_exists(). Son utilisation est très simple : nous lui donnons le nom d'une fonction sous forme de chaîne de caractères et elle nous renvoie true ou false pour nous indiquer si la fonction existe ou non. Exemple :

#### Code : PHP

```
<?php  
  
if(false == function_exists('strlen'))  
{  
    echo 'declaration strlen';  
    function strlen() {}  
}  
else  
{  
    echo 'strlen existe déjà';  
}
```

```
}

if(false == function_exists('pouet'))
{
    echo 'declaration pouet';
    function pouet() {}
}
else
{
    echo 'pouet existe déjà';
}
```

## Les fonctions, des valeurs

Dans certains langages, en OCaml par exemple ou même en mathématiques, les fonctions peuvent être considérées comme des valeurs, exactement comme des entiers : un tableau ou autre est une valeur.

Concrètement, nous pourrions utiliser une fonction exactement comme n'importe quelle valeur ; nous pourrions par exemple :

- la stocker dans une variable ;
- la donner en paramètre d'une fonction ;
- et je ne sais quoi d'autre.

Nous avons déjà eu l'occasion de voir cette possibilité à l'œuvre, souvenez-vous :

### Code : PHP

```
<?php

$f = 'strlen';
echo $f('lili');
```

Puisque chaque valeur possède un type, les fonctions doivent également en avoir un. En PHP, ça n'est pas très au point, malheureusement. Ainsi, si vous faites un var\_dump() de \$f, tout ce que vous aurez c'est du string. Dans la documentation du PHP, vous trouverez un type particulier du nom de **callback**. Une valeur callback est une valeur qu'il est possible d'appeler, qui peut prendre des paramètres et retourner une valeur. Pour déterminer si une valeur est de type callback, nous pouvons utiliser la fonction is\_callable() :

### Code : PHP

```
<?php

function f() {}
var_dump(is_callable('f'));
var_dump(is_callable('strlen'));
```

Utiliser des fonctions comme valeurs peut être intéressant par exemple pour changer facilement d'algorithme. Prenons un exemple : un traitement pour chaque élément d'un tableau. Si nous désirons ajouter 2 à toutes les valeurs, nous écririons ceci :

### Code : PHP

```
<?php

function addTwo($array)
{
    foreach($array as $key => $value)
    {
        $array[$key] = $value + 2;
    }

    return $array;
}
```

Si nous désirons maintenant échapper tous les éléments d'un tableau avec la fonction `htmlspecialchars()`, nous ferions ceci :

#### Code : PHP

```
<?php

function htmlEscape($array)
{
    foreach($array as $key => $value)
    {
        $array[$key] = htmlspecialchars($value);
    }

    return $array;
}
```

Comme vous le constatez, le code est exactement le même, à l'exception du traitement que subissent les valeurs. Une façon très élégante d'éviter d'écrire 36 000 fois la même chose est de passer un deuxième argument, une fonction qui s'occupera du traitement :

#### Code : PHP

```
<?php

function map($array, $func)
{
    foreach($array as $key => $value)
    {
        $array[$key] = $func($value);
    }

    return $array;
}

function addTwo($value)
{
    return $value + 2;
}

var_dump(map(array(4, 6), 'addTwo'));
var_dump(map('lili<>', 'meuh'), 'htmlspecialchars');
```

#### Code : Console

```
array(2)  {
[0]=>
int(6)
[1]=>
int(8)
}
array(2)  {
[0]=>
string(12) "lili<>"
[1]=>
string(10) "meuh"
```

Ça a tout de suite plus fière allure, vous ne trouvez pas ? 😊 En fait, il faut que je vous avoue quelque chose ; nous venons de récréer une fonction que nous avions entraperçue précédemment : `array_map()`. En effet, les fonctions natives du PHP profitent bien évidemment de cette possibilité d'utiliser les fonctions comme des valeurs.

Pour information, cette fonction appartient à une catégorie de fonctions : les **fonctions d'ordre supérieur**. Pour qu'une fonction soit d'ordre supérieur, elle doit soit recevoir une fonction en paramètre, soit retourner une fonction.

## Les fonctions anonymes

Le code précédent est certes élégant, mais nous pouvons faire mieux. Nous avons dû déclarer une fonction pour remplir un rôle, ajouter 2 à une valeur. Il est dommage de monopoliser un nom de fonction pour une fonction qui n'est utilisée qu'une seule fois, car si par la suite nous avons besoin d'une fonction que nous utiliserions un peu partout et dont le nom idéal serait addTwo(), eh bien nous devrions changer de nom de fonction. En utilisant des noms de fonctions pour si peu de chose, d'après le jargon, nous *polluons l'espace de noms global*. Fort heureusement, il est possible de créer des fonctions qui n'ont pas de nom, des fonctions anonymes, donc.

Les fonctions anonymes ne peuvent être appelées par leur nom puisqu'elles n'en ont pas, ça n'a donc pas de sens de créer une fonction anonyme si elle n'est pas utilisée comme une valeur. Les fonctions anonymes seront donc toujours utilisées comme des valeurs. Pour définir une fonction anonyme, rien de bien compliqué, la syntaxe est exactement la même que pour une fonction nommée. Exemple :

### Code : PHP

```
<?php  
  
$f = function() {  
    echo 'bonjour';  
};  
$f();
```



Si une erreur est émise, mettez votre version du PHP à jour. Cette syntaxe a en effet été introduite avec la version 5.3 du PHP.

Comme c'est une affectation, il ne faut pas oublier de faire suivre l'accolade fermante par un point-virgule. Il est bien entendu possible de passer des paramètres à une fonction anonyme, nous pourrions donc réécrire notre code avec la fonction map() comme ceci :

### Code : PHP

```
<?php  
  
function map($array, $func)  
{  
    foreach($array as $key => $value)  
    {  
        $array[$key] = $func($value);  
    }  
  
    return $array;  
}  
  
$f = function($value) {  
    return $value + 2;  
};  
var_dump(map(array(5, 6), $f));  
// ou plus élégamment :  
var_dump(map(array(5, 6), function($value) { return $value + 2; }));
```

Vous pourriez être amenés à rencontrer une fonction un peu particulière : create\_function(). Avant la version 5.3 du PHP, créer des fonctions anonymes était déjà possible grâce à cette fonction. Avec les versions récentes du PHP, cette fonction n'est plus à utiliser car les vraies fonctions anonymes ont plusieurs avantages :

- coloration syntaxique lors de l'écriture du code ;
- aucun risque de collision de nom : create\_function() ne créait pas vraiment de fonction anonyme mais leur donnait un nom plus ou moins aléatoire et unique ;
- erreurs de syntaxe détectables : avec create\_function(), les erreurs de syntaxe n'étaient détectées que lors de l'appel à create\_function(), tandis qu'avec les vraies fonctions anonymes, l'erreur est détectée avant même l'exécution du script.

### Les fermetures

Pour commencer, je vais tirer les oreilles de la documentation PHP.  En effet, dans [la documentation](#), ils considèrent que fermeture et fonction anonyme sont équivalents. C'est totalement faux, mais voyons d'abord ce qu'est une fermeture. Considérons ce script :

#### Code : PHP

```
<?php  
  
$var = 54;  
$f = function() { echo $var; };  
$f();
```

Pour que notre fonction anonyme ne génère pas d'erreur, il est nécessaire qu'elle puisse utiliser la variable \$var. Une fermeture est une fonction qui nécessite des variables définies dans l'environnement dans lequel elle est elle-même définie. Une fermeture n'a donc **strictement rien** à voir avec une fonction anonyme.

Dans certains langages, la fermeture a directement accès aux variables définies dans son environnement, mais pas en PHP. Si vous testez ce code, une erreur sera émise disant que la variable \$var n'existe pas. Pour pouvoir accéder à ces variables, nous allons devoir les capturer, les enfermer. Pour cela, nous utiliserons le mot-clé `use()` :

#### Code : PHP

```
<?php  
  
$var = 54;  
$f = function() use($var) { echo $var; };  
$f();
```

Il est à noter que la valeur de \$var dans la fermeture ne variera jamais même si la variable originale est modifiée. Le comportement de la fermeture ne sera donc jamais altéré quoi qu'il arrive à la variable originale. Exemple :

#### Code : PHP

```
<?php  
  
$var = 54;  
$f = function() use($var) { echo $var; };  
$f();  
$var = 1000;  
$f();
```

Dans une fermeture, vous pouvez capturer n'importe quelle variable, y compris les paramètres passés à une fonction :

#### Code : PHP

```
<?php  
  
function getAdditionner($value)  
{  
    return function($add) use($value) {  
        return $value + $add;  
    };  
}  
  
$f = getAdditionner(5);  
echo $f(2);  
echo $f(4);
```

La fonction getAdditionner() retourne une fermeture qui prend un paramètre et capture le paramètre passé à getAdditionner(). Ainsi, le premier appel à la fermeture effectue le calcul  $5 + 2$ , tandis que le second appel effectue le calcul  $5 + 4$ .

### Fonction anonyme récursive

Lorsque une fonction est récursive, elle s'appelle elle-même. Or, les fonctions anonymes n'ayant pas de nom, il est difficile de les appeler. Pour pouvoir user de la récursivité, une petite astuce est nécessaire : la fonction doit se capturer elle-même.

#### Code : PHP

```
<?php

$f = function($i) use($f) {
    if($i > 0)
    {
        echo $i;
        $f(--$i);
    }
};

$f(4);
```

Toutefois, cela ne fonctionne pas car la variable \$f est déclarée après que la fonction anonyme a été créée puisque l'expression de droite est évaluée avant d'effectuer l'affectation. Le problème peut être contourné moyennant une autre astuce :

#### Code : PHP

```
<?php

$f = function($i) use(& $f) {
    if($i > 0)
    {
        echo $i;
        $f(--$i);
    }
};

$f(4);
```

Ce petit symbole devant \$f indique que nous avons affaire à une **référence**, ce qui sera vu au chapitre suivant. 😊

## Les tableaux

Les tableaux, voilà bien un sujet qui peut faire couler beaucoup d'encre. Dans ce petit chapitre de rien du tout, nous allons aborder les tableaux à plusieurs dimensions, les opérateurs de tableaux ainsi que quelques fonctions utiles.

Un tableau à plusieurs dimensions est simplement un tableau dont les valeurs peuvent être des tableaux. Jusqu'à présent, nous avions une valeur — par exemple l'entier 5 ou le booléen true — associée à une clé. Maintenant, ces valeurs pourront être des tableaux. Que les valeurs du tableau soient des entiers, des booléens, des tableaux ou n'importe quoi d'autre, rien ne change au niveau de la syntaxe. Ce que nous voyons ici n'a donc rien de neuf, c'est juste une possibilité supplémentaire. Exemple :

#### Code : PHP

```
<?php

$tab = array(
    'cle' => array(5, 6),
    'autreCle' => range(1, 7),
    5 => 'meuh'
);
print_r($tab);
```

Puisqu'à l'index cle j'ai un tableau ; si j'écris <?php \$tab['cle']; , j'ai... un tableau. Je peux donc à nouveau utiliser

l'opérateur [] dessus :

**Code : PHP**

```
<?php

$tab = array(
    'cle' => array(5, 6),
    'autreCle' => range(1, 7),
    5 => 'meuh'
);
var_dump($tab['cle'][0]); // affiche 5
```

Si j'avais eu un tableau dans un tableau dans un tableau, j'aurais pu écrire <?php  
\$tab['index'] ['autreIndex'] ['encoreUnIndex']; . Il n'y a donc vraiment aucune différence entre les tableaux et les autres valeurs.

Pour parcourir un tableau, nous utilisons la boucle foreach(). Celle-ci ne va parcourir qu'un seul tableau : si un des éléments du tableau est lui-même un tableau, il ne sera pas parcouru. Exemple :

**Code : PHP**

```
<?php

$tab = array(
    'cle' => array(5, 6),
    'autreCle' => range(1, 7),
    5 => 'meuh'
);

foreach($tab as $value)
{
    echo $value;
}
```

Ce code affiche deux fois la chaîne « Array » car quand un tableau est converti en chaîne de caractères, il devient cette chaîne. Pour parcourir la totalité de notre tableau, nous devrons imbriquer des foreach().

**Code : PHP**

```
<?php

$tab = array(
    'cle' => array(5, 6),
    'autreCle' => range(1, 7),
    5 => 'meuh'
);

foreach($tab as $value)
{
    if(is_array($value))
    {
        foreach($value as $sValue)
        {
            echo $sValue;
        }
    }
    else
    {
        echo $value;
    }
}
```

Grâce à la fonction `is_array()`, nous pouvons déterminer si la valeur est un tableau ou non ; si c'est le cas, nous faisons un nouveau `foreach()`. Maintenant, si notre tableau contient un tableau qui lui-même contient un tableau, nous retrouverons la chaîne « `Array` ». Pour parcourir un tableau intégralement sans connaître sa profondeur, son nombre de dimensions, nous allons devoir utiliser une fonction récursive. Le code est presque similaire mis à part qu'un `foreach()` va disparaître et que le `foreach()` principal sera encapsulé dans une fonction. À vous de jouer. 😊

### Secret (cliquez pour afficher)

#### Code : PHP

```
<?php

$tab = array(
    'cle' => array(
        array(5, 45),
        6
    )
);

$tab[] = array(array(array(5)));

function printArray($array)
{
    foreach($array as $value)
    {
        if(is_array($value))
        {
            printArray($value);
        }
        else
        {
            echo $value;
        }
    }
}

printArray($tab);
```

### L'ordre, une notion relative

De par leur nature, en PHP, l'ordre dans les tableaux n'est pas forcément logique pour nous. Concrètement, si j'ajoute une clé (5 par exemple), une autre clé à la suite (0) et que je parcours le tableau avec un `foreach()`, l'index 5 sera le premier à être visité. Le seul ordre qu'il y a de base dans un tableau est donc l'ordre dans lequel nous définissons des clés.

Si nous voulons une autre notion d'ordre, c'est à nous de la définir. Pour cela, PHP propose plusieurs fonctions qui vont nous permettre de changer réellement l'ordre des éléments dans le tableau. Testons :

#### Code : PHP

```
<?php

$tab = array(5, 7, 0, 'lala' => 1, 91, 8);
sort($tab);
print_r($tab);
```

D'abord, nous définissons un tableau dont les valeurs sont totalement mélangées ; ensuite, nous trions ce tableau. Dans ce cas, trier signifie que nous trions les valeurs de façon croissante. Si vous regardez la sortie, les clés que nous avions définies, qu'elles soient implicites ou explicites, ont été totalement remaniées. L'ordre que nous avons instauré a donc plus ou moins détruit le tableau de départ.

Voici une bonne partie des fonctions que le PHP nous offre pour trier des tableaux :

- `natsort()` ;

- natcasesort();
- sort();
- usort();
- ksort();
- krsort();
- uksort();
- asort();
- arsort();
- et uasort().

Il y en a beaucoup, mais ne craignez rien, certaines se ressemblent. Comme nous venons de le voir, sort() trie le tableau de façon croissante en fonction des valeurs et redéfinit les clés. asort() fait la même chose mais en conservant les clés :

#### Code : PHP

```
<?php  
  
$tab = array(5, 7, 0, 'lala' => 1, 91, 8);  
asort($tab);  
print_r($tab);
```

arsort() fait la même chose que asort(), mais de façon décroissante. ksort() et krsort() font la même chose que asort() et arsort(), mais trient les éléments en fonction des clés et non en fonction des valeurs :

#### Code : PHP

```
<?php  
  
$tab = array(5, 7, 0, 'lala' => 1, 91, 8);  
ksort($tab);  
print_r($tab);  
krsort($tab);  
print_r($tab);
```

Vous voyez, ce n'est pas bien compliqué. Pour les trois dernières, ça se corse légèrement. Parfois, en faisant simplement une comparaison « plus grand que » ou « plus petit que » le tri ne suffit pas. Dans ce cas, il faudra utiliser usort(), uasort() ou uksort(). Ces fonctions fonctionnent de la même manière que leur homologue sans « u » mis à part qu'elles ne se basent pas sur une comparaison « plus grand que » ou « plus petit que ». Mais alors, sur quoi se basent-elles ?

Vous ne trouvez pas ? N'avons-nous pourtant pas vu, il y a quelques instants, une méthode qui permet de changer d'algorithme facilement ? Eh oui, ces fonctions utilisent des callback. La fonction de callback devra prendre deux paramètres, les valeurs à comparer, et retourner :

- zéro si les valeurs sont égales ;
- un nombre inférieur à zéro si le premier paramètre de la fonction de callback est considéré comme plus petit que le second ;
- un nombre supérieur à zéro si le premier paramètre de la fonction de callback est considéré comme plus grand que le second.

À partir de cela, nous pouvons très facilement construire la fonction sort() à partir de usort() :

#### Code : PHP

```
<?php  
  
$tab = array(5, 7, 0, 'lala' => 1, 91, 8);  
$tab2 = $tab;  
sort($tab);  
print_r($tab);  
// $a - $b = 0 si $a == $b  
// = -x si $a < $b  
// = +x si $a > $b  
usort($tab2, function($a, $b) { return $a - $b; });
```

```
print_r($tab2);
```

Pratiques, ces fonctions d'ordre supérieur, n'est-ce pas ? Couplées avec les fonctions anonymes, c'est très élégant. 

Enfin, pour `natsort()` et `natcasesort()`, elles trient les associations en les comparant « naturellement » et conservent les clés. L'ordre « naturel » est propre au tri des chaînes de caractères. À cause de la façon dont l'ordinateur effectue le tri, la chaîne « `lala10` » est considérée comme plus « petite » que « `lala2` ». Le tri naturel permet d'éviter cela et d'obtenir un tri conforme à notre perception, c'est-à-dire que « `lala10` » sera considérée comme plus « grande » que « `lala2` ». La différence entre `natsort()` et `natcasesort()` est que `natcasesort()` ne tient pas compte de la casse lors de la comparaison, c'est-à-dire qu'elle considère que minuscules et majuscules sont équivalentes.

Vous avez sans doute remarqué qu'aucune de ces fonctions ne retournent de valeur. C'est parfaitement normal : elles modifient directement le tableau que nous leur donnons. Si vous regardez leur prototype, vous verrez la petite esperluette que nous avons rencontrée précédemment avec les fermetures récursives. Nous verrons ce que ce symbole signifie au chapitre prochain, comme promis.

## Les opérateurs de tableaux

Il existe cinq opérateurs que vous connaissez déjà et qui sont applicables aux tableaux :

- `+`
- `==`
- `==`
- `!=`
- `!=`

L'opérateur `+` permet d'obtenir l'union des deux tableaux. Concrètement, si j'écris `<?php $a + $b;`, j'obtiendrai un tableau contenant toutes les clés de `$a` et toutes les clés de `$b` qui n'existent pas dans `$a`. L'ordre est important, `$a + $b` n'est donc pas équivalent à `$b + $a`. Exemple :

### Code : PHP

```
<?php
$a = array('lili', 'lala');
$b = array('lulu');
var_dump($a + $b);
var_dump($b + $a);
```

Pour l'opérateur `==`, il considère que deux tableaux sont égaux si et seulement si ils possèdent les mêmes associations de clé et de valeur.

### Code : PHP

```
<?php
$a = array('lili', 'lala');
$b = array(1 => 'lala', 0 => 'lili');
var_dump($a == $b);
```

L'opérateur `==` vérifie en plus que le type des valeurs est identique dans les deux tableaux et que l'ordre des associations dans les tableaux est le même.

### Code : PHP

```
<?php
$a = array();
$b = array('0');
var_dump($a === $b); // false car types différents
$a = array('lili', 'lala');
```

```
$b = array(0 => 'lili', '1' => 'lala');
var_dump($a === $b); // true car ne vérifie pas le type des clés

$a = array('lili', 'lala');
$b = array('1' => 'lala', 0 => 'lili', );
var_dump($a === $b); // false car l'ordre est différent
```

Les opérateurs != et !== fonctionnent exactement comme == et ===, à la différence près qu'ils renvoient le résultat opposé.

## Fonctions utiles

Il existe un grand nombre de fonctions pour gérer les tableaux, quelques-unes seront présentées ici mais toutes sont disponibles dans [la documentation](#).

### [array\\_merge\(\)](#)

Cette fonction prend un nombre variable de tableaux en argument et retourne l'union de ceux-ci. Si des clés **non-numériques** sont communes à plusieurs tableaux, la valeur du tableau ayant la position la plus élevée dans la liste des arguments sera conservée. Les clés numériques sont **ignorées** et les éléments associés à ces clés en reçoivent une nouvelle. Exemple :

#### Code : PHP

```
<?php

var_dump(array_merge(array(5, 'lili' => 'val'), array(50 => 3,
'lili' => 'lulu')));
//array(0 => 5, 'lili' => 'lulu', 1 => 3)
```

### [array\\_diff\(\)](#)

Cette fonction prend un nombre variable de tableaux en argument et retourne un tableau contenant toutes les valeurs qui n'apparaissent que dans un et un seul des tableaux passés en argument. Exemple tiré de la documentation :

#### Code : PHP

```
<?php

$array1 = array('a' => 'green', 'red', 'blue', 'red');
$array2 = array('b' => 'green', 'yellow', 'red');
var_dump(array_diff($array1, $array2)); // array(0 => 'blue')
```

Il existe plusieurs variantes de cette fonction, par exemple `array_diff_key()` qui fait la même chose mais qui utilise les clés et non les valeurs pour effectuer la comparaison, et `array_diff_assoc()` qui utilise les clés et les valeurs pour effectuer la comparaison. Il est également possible d'utiliser un callback pour faire la combinaison avec, par exemple, les fonctions `array_diff_ukey()` et `array_udiff()`.

### [array\\_intersect\(\)](#)

La famille `array_diff()` permet de trouver les éléments qui n'apparaissent que dans un des tableaux. La famille `array_intersect()` fait le contraire et retourne les éléments qui apparaissent dans tous les tableaux.

#### Code : PHP

```
<?php

var_dump(array_intersect(array('lili', 'lulu'), array('pouet',
'lili')));
// array(0 => 'lili')
```

Il existe également des fonctions permettant de faire cela en se basant sur les clés et les valeurs et utilisant des callback.

### array\_combine()

Cette fonction permet de créer un tableau en utilisant les valeurs du premier argument en tant que clés et les valeurs du second argument en tant que valeurs :

#### Code : PHP

```
<?php  
  
var_dump(array_combine(array('pouet', 6), array(true, false)));  
// array('pouet' => true, 6 => false)
```

### shuffle()

Cette fonction permet de mélanger un tableau de façon plus ou moins aléatoire :

#### Code : PHP

```
<?php  
  
$tab = range(1, 5);  
shuffle($tab);  
print_r($tab);
```

Attention, cette fonction modifie directement le tableau de départ. Il est vraiment temps de parler des références !

### array\_rand()

Cette fonction permet de tirer des clés au hasard dans un tableau. Très pratique si vous voulez par exemple faire un tirage au sort dans une liste quelconque !

#### Code : PHP

```
<?php  
  
$tab = array('Pierre', 'Paul', 'Jason', 'Cédric');  
foreach(array_rand($tab, 2) as $index)  
{  
    echo $tab[$index];  
}
```

### array\_reverse()

Cette fonction permet d'inverser l'ordre d'un tableau. Son deuxième paramètre, un booléen, indique si nous désirons conserver les clés ou non. Exemple :

#### Code : PHP

```
<?php  
  
var_dump(array_reverse(range(1, 5))); // clés redéfinies car par  
défaut, false  
var_dump(array_reverse(range(1, 5), true)); // clés non redéfinies
```

Nous n'allons pas passer en revue toutes ces fonctions, je vous laisse le reste pour vous amuser avant d'aller dormir. 

## Un pointeur interne

Souvenez-vous : quand nous avons parlé des jeux de résultats issus d'une base de données, nous avions dit qu'ils possédaient un pointeur interne. En changeant la valeur de ce pointeur, nous pouvions nous déplacer dans le jeu de résultats et obtenir la ligne courante, c'est-à-dire la ligne sur laquelle le pointeur — oserais-je le dire ? — pointe.

Les tableaux possèdent également un pointeur interne qui fonctionne de la même façon. Pour gérer ce pointeur interne, le PHP met à notre disposition quelques fonctions :

- reset() pour remettre le pointeur interne au début du tableau ;
- next() pour avancer le pointeur interne d'un élément ;
- prev() pour reculer le pointeur interne d'un élément ;
- end() pour mettre le pointeur interne à la fin du tableau ;
- current() pour obtenir la valeur de la ligne actuellement pointée ;
- key() pour obtenir la clé de la ligne actuellement pointée.

Exemple :

### Code : PHP

```
<?php

$tab = array('Pierre', 'Paul', 'Jason', 'Cédric');
end($tab);
echo current($tab);
prev($tab);
echo key($tab);
reset($tab);
next($tab);
echo current($tab);
```

Lorsqu'une association est créée dans un tableau, elle est placée tout à la fin de celui-ci. Cela explique pourquoi lorsque nous parcourons un tableau avec foreach(), les associations apparaissent dans l'ordre dans lequel nous les avons ajoutées puisque foreach() utilise ce pointeur interne. À ce propos, sachez que foreach() remet le pointeur interne au début du tableau avant de commencer à le parcourir et déplace ledit pointeur à chaque itération.

Restent deux petites fonctions bien utiles avec les tableaux : each() et list(). each() nous retourne un tableau contenant l'index et la valeur de la ligne actuellement pointée et déplace le pointeur vers la prochaine ligne. Nous pouvons donc utiliser cette fonction pour parcourir aisément un tableau :

### Code : PHP

```
<?php

$tab = range(1, 5);
while($line = each($tab))
{
    echo $line[0] . ':' . $line[1];
}
```

each() retourne false si le pointeur interne pointe vers une ligne qui n'existe pas, nous pouvons donc utiliser la même technique qu'avec mysqli\_fetch\_assoc().

Quant à list(), eh bien souvenez-vous du TP du livre d'or : ce n'est pas une fonction mais une structure de langage qui permet d'extraire et d'affecter aisément à des variables les valeurs d'un tableau. Cette structure prend en paramètre un nombre variable de variables et y affecte la valeur correspondante dans le tableau. La valeur correspondante est la valeur dont la clé est la même que la position de la variable dans la liste de paramètres, exactement comme func\_get\_arg(), ce qui explique pourquoi list() ne fonctionne qu'avec les clés numériques.

### Code : PHP

```
<?php  
  
$tab = array(5);  
list($key, $val) = each($tab);  
echo $key . ':' . $val;
```

Avec cela, nous pouvons réécrire notre parcours d'un tableau avec each() :

#### Code : PHP

```
<?php  
  
$tab = range(1, 5);  
while(list($key, $val) = each($tab))  
{  
    echo $key . ':' . $val;  
}
```

Nous obtenons ainsi un foreach() à un reset() près. 😊

Il est à noter que les variables sont affectées en commençant par la fin de la liste des paramètres. Nous pouvons le prouver facilement :

#### Code : PHP

```
<?php  
  
list($tab[0], $tab[1]) = array(5, 2);  
print_r($tab);
```

L'association 1 => 2 apparaît avant l'association 0 => 5, preuve qu'elle a été insérée dans le tableau en premier.

## Tableaux et chaîne de caractères

Lorsque nous avons traité des chaînes de caractères, nous avons vu que lorsqu'une chaîne de caractères est délimitée par des guillemets, les variables qui y sont mentionnées sont remplacées par leur valeur. Exemple :

#### Code : PHP

```
<?php  
  
$prenom = 'Paul';  
echo "Bonjour $prenom";
```

Avec les tableaux, cela peut poser problème :

#### Code : PHP

```
<?php  
  
$prenoms = array('Dupont' => 'Paul');  
echo "Bonjour $prenoms['Dupont']";
```

Pour éviter que ce code émette une erreur, il nous suffit de délimiter la variable avec des accolades :

#### Code : PHP

```
<?php  
  
$prenoms = array('Dupont' => 'Paul');  
echo "Bonjour {$prenoms['Dupont']}";
```

Il est également possible de ne pas mettre ces accolades, mais il faudra alors omettre le délimiteur de chaîne de caractères autour de l'index :

**Code : PHP**

```
<?php  
  
$prenoms = array('Dupont' => 'Paul');  
echo "Bonjour $prenoms[\"Dupont\"]";
```

Toutefois, je vous recommande la syntaxe précédente avec les accolades : cette syntaxe est en effet utilisable dans d'autres cas tandis que la dernière syntaxe ne fonctionne qu'avec les tableaux.

J'espère que vous avez trouvé la liste des fonctions portant sur les tableaux moins ennuyeuse que je ne l'ai trouvée à écrire. 😊

Ne vous arrêtez pas trop aux fonctions sur les tableaux et les opérateurs, les parties sur les fonctions et le pointeur interne du tableau sont autrement plus importantes !

## Au cœur des variables

Lorsque nous avons parlé des variables dans la première partie de ce cours, plusieurs points ont été omis. Ce chapitre comblera ces petits trous et vous plongera au cœur même de la gestion des variables par le PHP.

### Des noms à dormir dehors

Lorsque nous avons parlé de la déclaration des variables, je vous avais donné une règle de nommage que voici : « Vous pouvez utiliser n'importe quel caractère alphabétique (minuscules et majuscules comprises) ou numérique, ou bien un *underscore* (\_). Toutefois, le nom de la variable ne peut pas commencer par un chiffre. »

Il s'avère en réalité que cette règle est complètement fausse.  Toutefois, deux choses sont à signaler :

- une syntaxe particulière est nécessaire ;
- et **je vous déconseille fortement d'utiliser cette méthode.**

La syntaxe pour nommer une variable comme on le souhaite est la suivante : \${expr}. Nous pouvons donc utiliser n'importe quelle expression pour nommer une variable. Testons :

#### Code : PHP

```
<?php

${'lalaèsè'} = 4; // $lalaèsè
${${'lalaèsè'}} = 'bonjour'; // $4 puisque $lalaèsè = 4 et que
(string)4 -> '4'
${__FILE__ . 'lili'} = 'ahah'; // $nomDuFichierlili

var_dump(${'lalaèsè'});
var_dump(${${'lalaèsè'}});
var_dump(${__FILE__ . 'lili'});
```

#### Code : Console

```
int(4) string(7) "bonjour" string(4) "ahah"
```

Il est à noter que lorsque nous n'utilisons qu'une variable pour nommer une variable, comme à la ligne 4 du code précédent, il est possible d'abréger la syntaxe :

#### Code : PHP

```
<?php

$s = 'truc';
${$s} = 'lili'; // $truc puisque $s = 'truc'

var_dump($$s); // nous n'utilisons qu'une variable, nous pouvons
omettre les accolades
```

Cette méthode qui utilise des expressions pour nommer une variable porte généralement le nom de « variable dynamique » ou encore de « variable variable ».

Mais je vous conseille de fuir cette méthode comme la peste. La première raison, c'est la lisibilité du script. Sérieusement, à quoi cela peut-il bien servir d'avoir une variable qui s'appelle lala!çè ? En quoi le nom ménage est sensiblement plus parlant que le nom menage ?

Bien souvent, l'utilisation qui est faite de cette méthode est celle-ci :

#### Code : PHP

```
<?php

$perso1 = 'luc';
$perso2 = 'pierre';
```

```
$perso3 = 'jason';

for($i = 1; $i <= 3; $i++)
{
    echo ${'perso' . $i};
}
```

Et que venons-nous de réinventer ? Les tableaux ! Eh oui, bien souvent, des codeurs utilisent cette méthode parce qu'ils ignorent que les tableaux existent ou qu'ils ne veulent pas apprendre à les utiliser. Or, remplacer les tableaux par cette méthode pose un énorme problème : il faut réinventer toutes les fonctions natives du PHP en rapport avec les tableaux.

En pratique, jamais, au grand jamais, je n'ai vu de script qui avait **réellement besoin** des variables dynamiques, et c'est pourquoi je vous déconseille fortement de les utiliser. Mais si vous avez un exemple de code qui nécessite absolument l'emploi de cette méthode, je serais ravi que vous m'en fassiez part.

Avant de terminer, une dernière chose concernant les tableaux. Considérons la déclaration suivante : `<?php $var['index'];`. Déclarons-nous une variable dont le nom est l'expression `$var['index']`, ou bien accédons-nous à l'index `index` de la variable dont le nom est l'expression `$var` ?

La bonne réponse est la première proposition. Toutefois, dans pareil cas, il est plus que recommandé de lever l'ambiguïté ; premièrement pour être sûrs que le PHP fera ce qu'on pense qu'il fera, et deuxièmement pour ne pas rendre le script encore plus illisible. Pour cela, nous utiliserons la syntaxe complète : `<?php ${$var['index']}`.

## Les références

Avant de parler des références, nous devons introduire un concept qui siège au cœur même de la gestion des variables par le PHP : la **table des symboles**.

Qu'est-ce qu'une variable ? Qu'est-ce qu'une affectation ? Les réponses à ces questions se trouvent dans la table des symboles. L'idée est la suivante : oubliez le terme « variable » et pensez aux termes « symbole » et « contenu ».

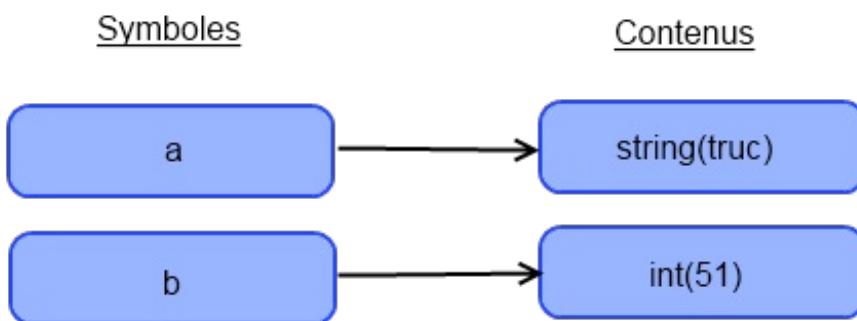
En PHP, une variable est en réalité un symbole auquel est associé un contenu. Concrètement, en interne, le PHP possède une table des symboles qui à chaque symbole (variable) associe un contenu (valeur). Considérons le script suivant :

### Code : PHP

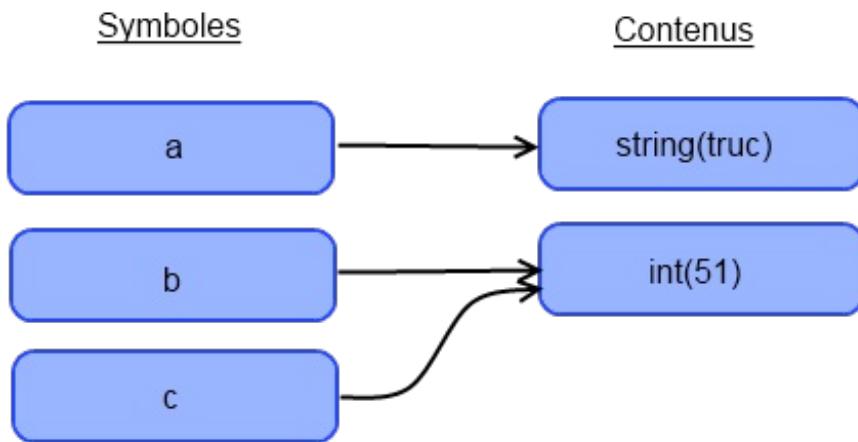
```
<?php

$a = 'truc';
$b = 51;
```

La table des symboles correspondante est la suivante :



Revenons-en aux références. Une référence, ou plutôt **des références**, sont des symboles qui sont associés au même contenu. Voici ce que ça donnerait :



Nous avons bien deux symboles qui sont associés au même contenu, ces deux symboles sont des références. Puisque le mécanisme de référence permet d'associer plusieurs symboles à un contenu, il n'est pas possible d'avoir de référence s'il n'y a qu'un symbole associé à un contenu, il en faut au minimum deux. En code, voici ce que ça donnerait :

#### Code : PHP

```
<?php
$a = 'truc';
$b = 51;
$c =& $b;
?>
```

Attention, la syntaxe pour déclarer des références est très trompeuse. De prime abord, nous pourrions lire la ligne 5 comme ceci : \$c est une référence sur \$b. Mais cette lecture est erronée, la lecture correcte est la suivante : au contenu de \$b, nous associons un nouveau symbole \$c.

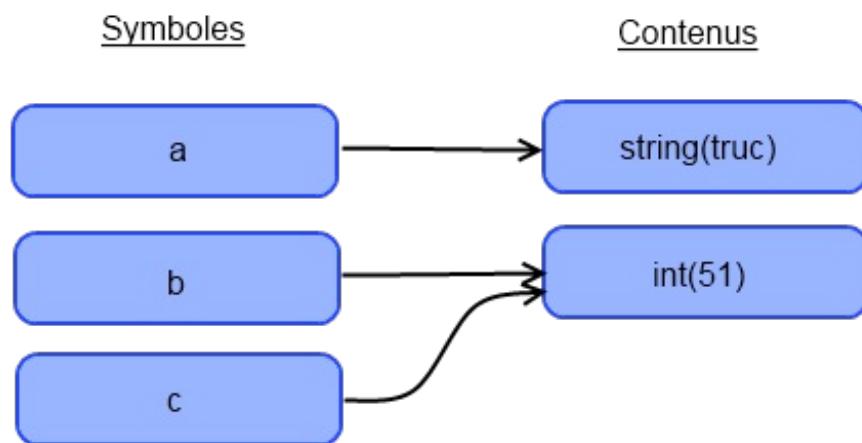
Il est donc totalement faux de dire que \$c est une référence sur \$b ou que \$b est référencé par \$c, ou je ne sais quoi d'autre : il n'y a pas le moindre lien, pas le moindre rapport entre \$b et \$c. Le seul et unique point commun entre ces deux symboles, c'est qu'ils sont associés au même contenu. Il nous est très facile de prouver cela :

#### Code : PHP

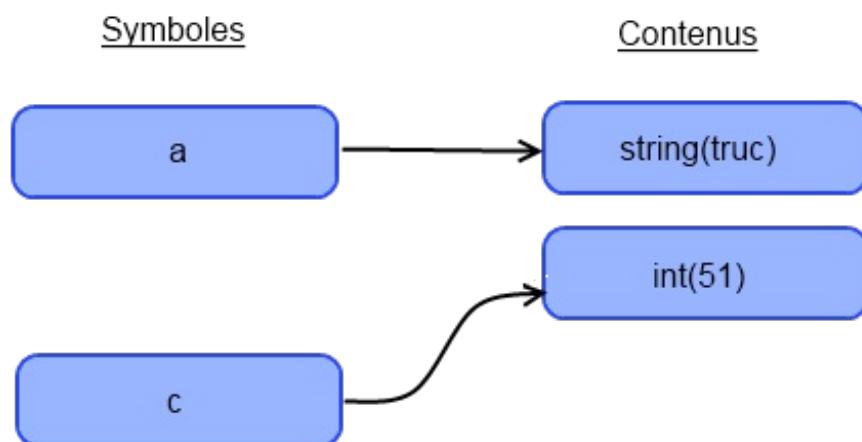
```
<?php
$a = 'truc';
$b = 51;
$c =& $b;
unset($b);
var_dump($c);
```

La structure de langage `unset()` sert à détruire un symbole. Or, même après avoir détruit \$b, nous pouvons accéder au contenu de \$c, il est donc évident qu'il n'y avait pas de lien entre \$b et \$c.

Regardons l'évolution de la table des symboles de plus près pour voir exactement ce qui arrive. Avant l'appel à `unset()`, la table des symboles est la suivante :



La structure de langage `unset()` détruit un symbole, mais ne touche en aucun cas au contenu. Donc, après l'appel à cette fonction, la table des symboles est la suivante :



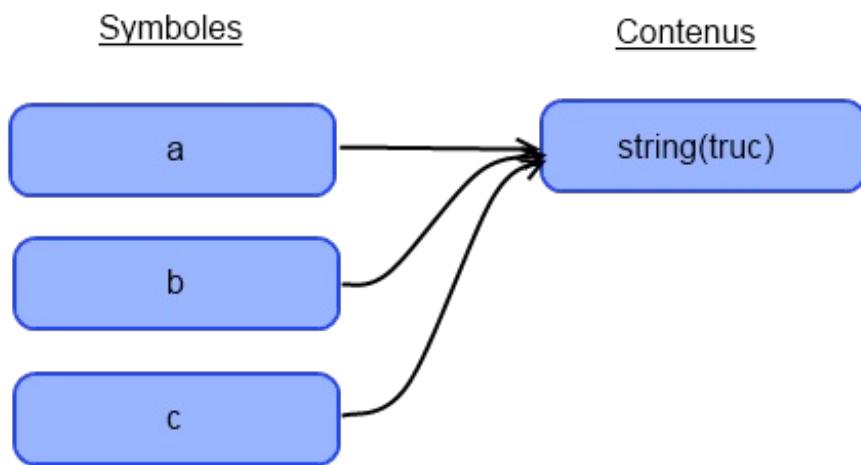
Le contenu est bel et bien toujours associé à \$c, nous pouvons donc toujours y accéder.

Nous pouvons bien évidemment avoir plus que deux références :

**Code : PHP**

```
<?php  
$a = 'truc';  
$b =& $a;  
$c =& $a;
```

La table des symboles serait alors la suivante :



Petite question pour être sûr que vous avez bien suivi : au niveau de la table des symboles, en quoi le code précédent est-il différent de celui-ci ?

#### Code : PHP

```
<?php
$a = 'truc';
$b = & $a;
$c = & $b;
```

#### Secret (cliquez pour afficher)

Les deux codes sont parfaitement équivalents. 🍪 Si vous y avez vu une différence, c'est que vous ne lisez pas correctement les lignes `<?php $truc = & $machin;`.

Reprends ligne par ligne. À la ligne 3, nous créons un symbole auquel nous associons un contenu. À la quatrième ligne, nous associons le symbole b au contenu associé au symbole a, a et b sont donc associés au même contenu. À la ligne 5, nous associons le symbole c au contenu associé au symbole b. Or, a et b sont associés au même contenu, donc que nous utilisions a ou b, le contenu auquel nous associons c est exactement le même, les deux codes sont donc parfaitement équivalents.



Si `unset()` n'efface pas le contenu d'une variable, que devient le contenu quand il n'est plus associé à aucun symbole ?



Mais comment ce ramasse-miettes sait-il qu'il n'y a plus aucun symbole associé à un contenu ?

En réalité, j'ai légèrement simplifié le contenu dans la table des symboles. 😊 Le contenu est en fait un conteneur, générique, qui peut contenir n'importe quel type de variable. Ce conteneur possède quatre informations :

- le type du contenu : string, int, etc. ;
- le contenu à proprement parler ;
- un compteur de références ;
- et un indicateur dont nous parlerons par la suite.

Le compteur de références est l'élément qui permet de savoir à combien de symboles est associé le contenu. Ce compteur est incrémenté de 1 quand nous associons un symbole à ce contenu, et décrémenté de 1 quand une association à ce contenu est perdue.

#### Code : PHP

```
<?php  
  
$a = 'truc'; // le compteur de références est à 1  
$b =& $a; // le compteur de références est à 2  
$c =& $b; // le compteur de références est à 3  
unset($a); // le compteur de références est de nouveau à 2
```

## La portée des variables

À quoi peuvent bien servir les références, en pratique ? Cette question est légitime, j'espère bien que vous vous l'êtes posée, car la réponse ne va pas tarder.

Si vous vous souvenez, dans le chapitre sur les fonctions, je vous avais dit que nous ne pouvons utiliser dans une fonction une variable déclarée en dehors de celle-ci, et inversement. Ce script provoquera une erreur :

### Code : PHP

```
<?php  
  
function f()  
{  
    var_dump($s);  
}  
  
$s = 'lala';  
f();
```

La raison de cette erreur est simple : il n'y a pas une table des symboles mais **des** tables des symboles qui sont toutes indépendantes. La fonction f() possède une table des symboles et le script possède une table des symboles, ces deux tables sont indépendantes l'une de l'autre. Quand nous sommes dans une fonction, c'est sur la table des symboles de la fonction, aussi appelée *table des symboles locale*, que nous travaillons. Quand nous sommes en dehors d'une fonction, au niveau global du script, nous travaillons sur la table des symboles du script (ou *table des symboles globale*).

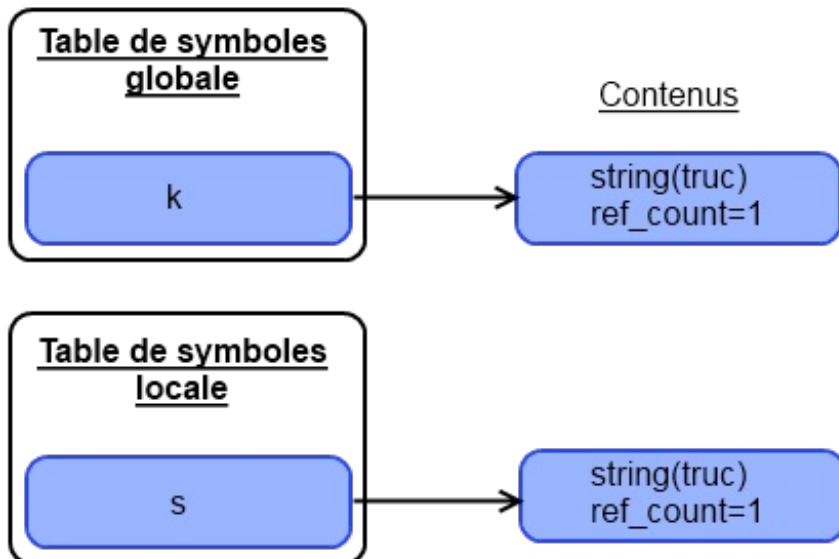
Dans le code précédent, nous déclarons un symbole dans la table des symboles globale. Or, quand nous utilisons \$s, nous sommes dans une fonction, nous utilisons donc la table des symboles locale. Et comme aucun symbole s n'existe dans cette table, l'erreur est inévitable. Sachez également que lorsqu'une fonction se termine, c'est-à-dire lorsqu'un return est rencontré ou qu'il n'y a plus de code à exécuter, la table des symboles locale est purement et simplement vidée.

Bien évidemment, puisque nos fonctions ont besoin d'informations pour travailler, il est nécessaire de pouvoir leur donner ces informations. Pour cela, nous utilisons les paramètres. En effet, les paramètres vont venir remplir la table des symboles locale.

### Code : PHP

```
<?php  
  
function f($s)  
{  
    var_dump($s);  
}  
  
$k = 'truc';  
f($k);
```

Dans la table des symboles locale, un symbole s sera ajouté et associé au contenu string(truc). Nous avons donc ceci :



Ce schéma nécessite quelques explications. 😊

Premièrement, vous constatez que j'ai sorti les contenus des tables des symboles. En effet, les contenus ne sont pas liés à une table des symboles en particulier : n'importe quelle table des symboles peut associer un de ses symboles à n'importe quel contenu.

Deuxièmement, il y a deux contenus. En PHP, comme dans beaucoup d'autres langages, les paramètres des fonctions fonctionnent par copie : la fonction ne reçoit pas le contenu qu'on lui donne mais une copie de celui-ci. Passer \$k en paramètre donne le même résultat que si nous avions écrit <?php \$s = \$k; .

Troisièmement, j'ai intégré le compteur de références aux contenus. Comme il n'y a à chaque fois qu'un symbole associé au contenu, le compteur de références est de 1 dans les deux cas.

Pour en revenir à l'utilité de nos références, considérons une fonction qui prend une chaîne de caractères en entrée, effectue un certain travail dessus (par exemple remplace tous les 'u' par des 'U'), et la retourne.

#### Code : PHP

```

<?php

function f($str)
{
    for($i = 0, $length = strlen($str); $i < $length; $i++)
    {
        if($str[$i] == 'u')
        {
            $str[$i] = 'U';
        }
    }

    return $str;
}

$str = 'truc';
$str = f($str);
var_dump($str);

```

Si la chaîne passée en paramètre est conséquente, quelques kilo-octets par exemple, en faire une copie pour ensuite supprimer cette copie est une pure perte de temps. L'idée serait donc de travailler directement sur le contenu associé au symbole k de la table des symboles globale. Associer deux symboles à un même contenu, n'est-ce pas ce que nous venons de faire avec les références ? Eh bien si, exactement, nous pouvons passer des paramètres par référence. Pour cela, rien de bien sorcier, il suffit d'ajouter une esperluette (&) devant le nom du paramètre :

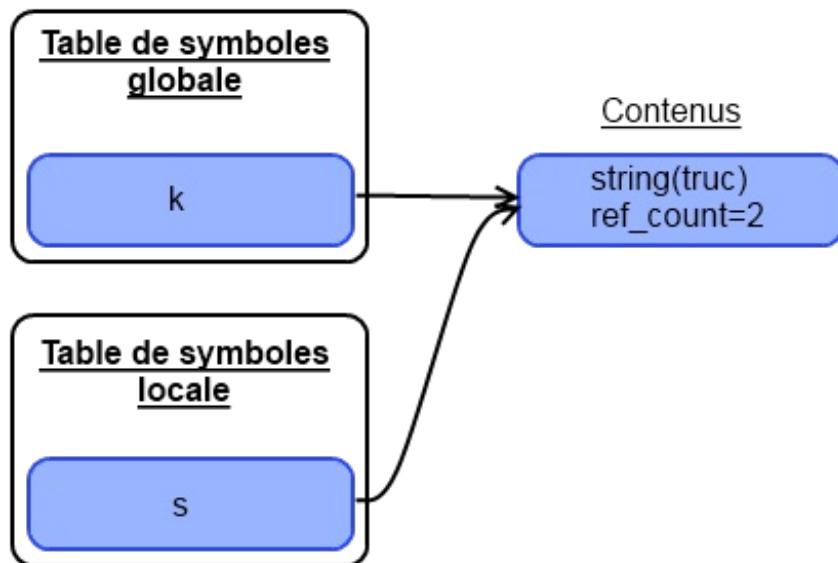
#### Code : PHP

```
<?php

function f(& $str)
{
    for($i = 0, $length = strlen($str); $i < $length; $i++)
    {
        if($str[$i] == 'a')
        {
            $str[$i] = 'A';
        }
    }
}

$str = 'abracadabra';
f($str);
var_dump($str);
```

Les tables des symboles seront de cet acabit :



Les références permettent donc d'économiser temporairement de la mémoire et d'éviter de réserver et libérer de la mémoire inutilement. Voilà pourquoi plusieurs fonctions relatives aux tableaux travaillent directement sur le tableau que nous leur donnons.

Il est possible que vous rencontriez une autre syntaxe pour le passage de paramètre par référence :

#### Code : PHP

```
<?php

function f($s)
{
    // ...
}

$var = 'truc';
f(& $var);
```

Cette pratique est une ancienne méthode, dépréciée, qui ne doit par conséquent plus être employée. Ce code génère d'ailleurs une erreur dans les versions récentes du PHP.

## Des variables non-déclarées ou non-initialisées

Considérons le code suivant :

**Code : PHP**

```
<?php

function f($var)
{
}

f($truc);
$machin;
f($machin);
```

Ce code provoque deux erreurs car nous utilisons une variable non déclarée et une variable à laquelle aucun contenu n'a été associé. Le passage de paramètre par référence permet toutefois de passer des variables non déclarées ou auxquelles aucun contenu n'a été associé. Testons :

**Code : PHP**

```
<?php

function f(& $var)
{
}

f($truc);
$machin;
f($machin);
```

Cela permet par exemple d'initialiser des variables dans une fonction sans devoir au préalable leur affecter une valeur.

En plus de passer un paramètre par référence à une fonction, il est possible à une fonction de retourner une référence. Mais avant de vous parler de cela, nous allons devoir refaire un tour sur les tables de symboles. 😊

## Au cœur du PHP

Avant d'aller plus loin, nous allons faire un petit détour par les entrailles du PHP et parler de quelque chose que beaucoup ignorent : la copie à l'écriture.

Nous savons qu'en interne, le PHP utilise des tables des symboles, qu'à chaque symbole est associé un conteneur et qu'un mécanisme, les références, permet d'associer plusieurs symboles de différentes tables à un même conteneur. Maintenant, je vous pose la question suivante : dans ce code, combien de conteneurs différents y a-t-il ?

**Code : PHP**

```
<?php

$a = 'truc';
$b = $a;
```

Contre toute attente, la réponse est **un**. En effet, les deux symboles sont associés au même conteneur. En théorie, il devrait y avoir deux conteneurs, un pour chaque symbole, mais en pratique il n'y en a qu'un seul. La raison est très simple : le PHP n'est pas idiot et optimise les choses. Pourquoi deux symboles qui ne sont pas des références doivent chacun avoir un conteneur ? Parce que sinon, si nous affectons une valeur à l'une des variables, l'autre serait également altérée. Mais pourquoi diable devrions-nous avoir deux conteneurs si nous ne faisons aucune affectation ? Puisque le contenu des deux symboles reste le même, ça n'a aucune importance qu'ils soient associés au même conteneur, étant donné qu'ils ont la même valeur. De plus, cela évite de perdre du temps et de la mémoire. L'idéal serait donc que tant qu'aucune affectation n'est faite, le conteneur soit partagé par les symboles et que dès qu'une affectation est demandée, un nouveau conteneur soit créé et associé au symbole auquel on veut affecter la valeur. Le PHP implémente ce système, c'est la copie à l'écriture.

Nous allons vérifier cela en utilisant une charmante fonction : `debug_zval_dump()`. Cette fonction permet d'afficher le contenu d'un conteneur sous forme de chaîne de caractères.

**Code : PHP**

```
<?php
$a = 'lala';
$b = $a;
$c = $b;

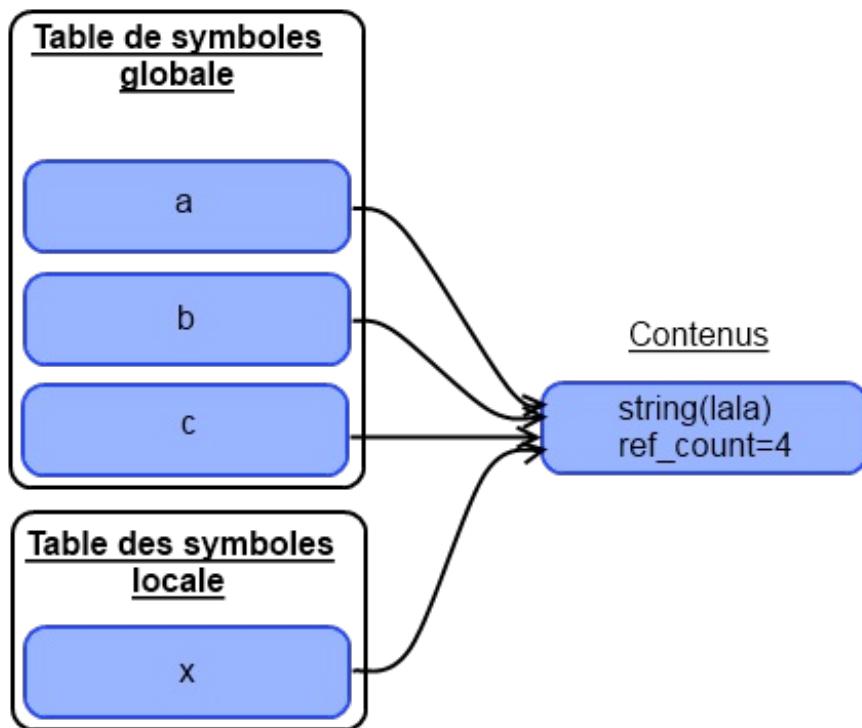
debug_zval_dump($c);
```

**Code : Console**

```
string(4) "lala" refcount(4)
```

Le *refcount* est de 4, ce qui indique que plusieurs symboles partagent le conteneur. Ces symboles sont a, b, c et un quatrième dû au paramètre donné à la fonction `debug_zval_dump()`.

La table des symboles est donc la suivante :



Nous allons maintenant affecter une valeur à \$c. Puisqu'il va y avoir affectation, un conteneur va être créé avec un compteur de références qui vaudra 1. Puisque le symbole sera associé à un autre conteneur, le conteneur précédent ne sera associé qu'à deux symboles, a et b, son compteur de références vaudra donc 2. Vérifions cela :

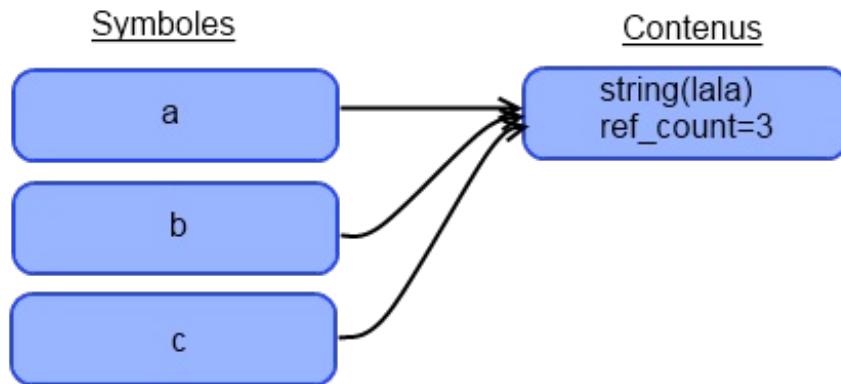
**Code : PHP**

```
<?php
$a = 'lala';
$b = $a;
$c = $b;
debug_zval_dump($a);
debug_zval_dump($c);
$c = 'truc';
debug_zval_dump($a);
debug_zval_dump($c);
```

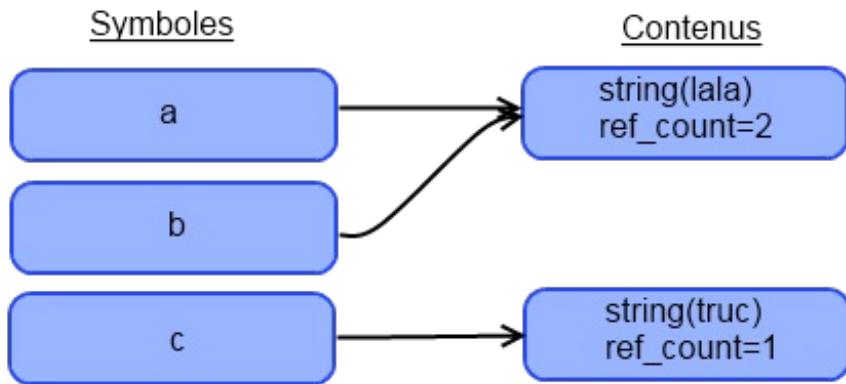
**Code : Console**

```
string(4) "lala" refcount(4) car symbole a, b, c et paramètre
string(4) "lala" refcount(4) car symbole a, b, c et paramètre
string(4) "lala" refcount(3) car symbole a, b et paramètre
string(4) "truc"  refcount(2) car symbole c et paramètre
```

La sortie est exactement ce à quoi nous nous attendions. La table des symboles avant l'affectation est la suivante :



Et voici la table après l'affectation :



Mais si un conteneur est créé lorsqu'une affectation est faite, pourquoi les références fonctionnent-elles ?

C'est grâce à la quatrième information du conteneur, le petit indicateur. Ce petit indicateur peut prendre deux valeurs : ces deux valeurs indiquent si les symboles associés sont des références ou non. Si l'indicateur indique que ce ne sont pas des références, le scénario décrit ci-dessus se produit, tandis que si l'indicateur indique que ce sont des références, le contenu sera simplement mis à jour. Mais avant de vous le prouver, nous devons voir un petit quelque chose :

**Code : PHP**

```
<?php
$a = 'lala';
$b =& $a;
$c = $b;
```

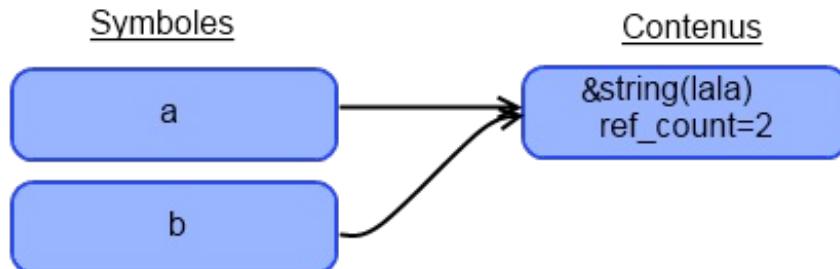
Qui peut me donner l'évolution de la table des symboles ?

\$a et \$b sont des références, donc nous savons avec certitude que ces symboles seront associés au même conteneur. Ce sont des références, donc l'indicateur de références l'indiquera. Grâce à la copie à l'écriture, aucun conteneur ne devrait être créé

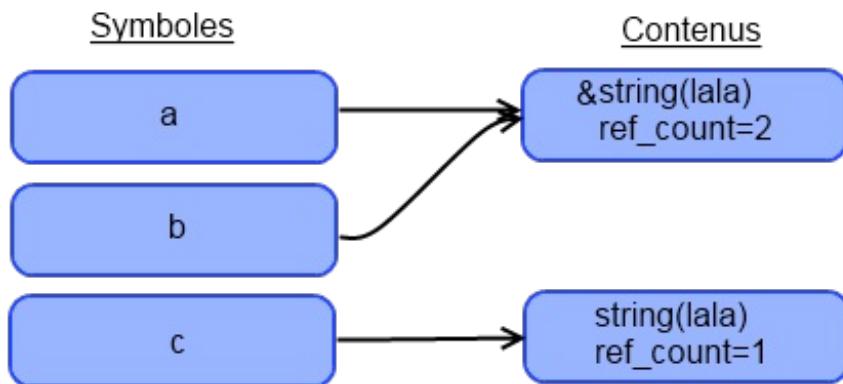
puisque nous n'affectons rien à \$c.

Mais que se passe-t-il si par la suite nous faisons une affectation à \$a ou \$b ? L'indicateur de références indique que ce sont des références, donc une simple mise à jour devrait être faite. Or, \$c est associé au même conteneur que les deux autres, donc si une mise à jour est faite, le contenu de \$c changera, ce qui n'est pas le comportement désiré. Mais bien que le conteneur indique le nombre de symboles auquel il est associé, il n'indique pas à quels symboles il est associé ; impossible donc de créer un conteneur pour \$c à ce moment.

La seule solution est donc, au moment même où l'on affecte \$b à \$c, de créer un conteneur. Avant l'affectation de \$c, la table des symboles est la suivante :



Notez la présence d'une esperluette pour indiquer qu'il s'agit de références. Après l'affectation de \$c, la table des symboles est la suivante :



Nous pouvons le vérifier :

#### Code : PHP

```
<?php
$a = 'lala';
$b = &$a;
$c = $b;
debug_zval_dump($a);
debug_zval_dump($c);
```

#### Code : Console

```
string(4) "lala" refcount(1)
string(4) "lala" refcount(2)
```

La seconde ligne est logique, nous avons le symbole c et le paramètre de la fonction. La première l'est moins : pourquoi un compteur de références ayant la valeur 1 ?

Souvenez-vous, je vous ai dit que passer un paramètre était équivalent à écrire `<?php $s = $k;`. Or, ce que nous affectons est une référence, par conséquent, un nouveau conteneur est créé lors du passage en paramètre. Comme ce conteneur n'est associé qu'au paramètre, son compteur de références a très logiquement la valeur 1. Le comportement observé est donc exactement le comportement attendu. Et nous constatons au passage qu'il est impossible, avec la fonction `debug_zval_dump()`, d'obtenir le nombre de symboles associés à un conteneur si ces symboles sont des références.

Un schéma similaire se produit si nous associons un symbole à un conteneur qui n'est pas associé à des références :

#### Code : PHP

```
<?php
$a = 'truc'; // un conteneur
$b = $a; // toujours un conteneur car copie à l'écriture
$c =& $b;
// $c et $b sont des références et sont donc associées au même
conteneur
// un conteneur &string est créé et ces deux symboles y sont
associés
// $a est désormais le seul symbole associé au conteneur de départ
```

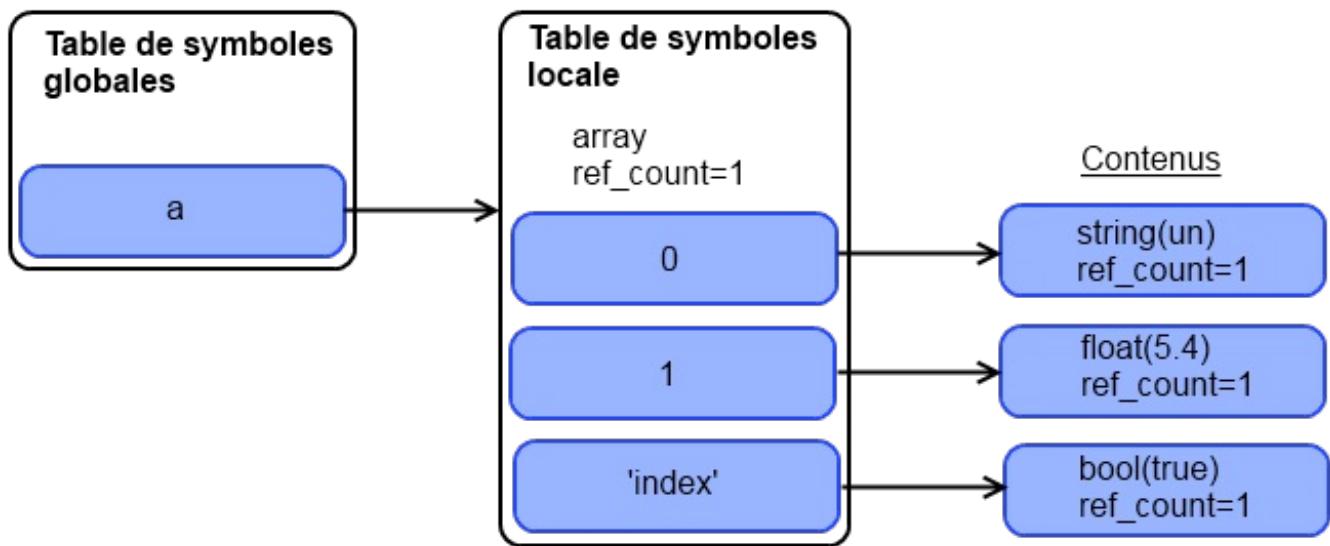
## Les tableaux

Les tableaux en PHP constituent un cas assez particulier dans ces tables des symboles. En fait, ce que je vais vous dire va changer votre façon de voir les tableaux : un tableau est un mélange entre une table des symboles et un conteneur. Soit la déclaration suivante :

#### Code : PHP

```
<?php
$a = array('un', 5.4, 'index' => true);
```

Les tables des symboles peuvent être représentées comme ceci :



Puisque les tableaux sont gérés exactement de la même façon que les variables, nous pouvons bien évidemment créer une référence avec un indice d'un tableau. Après, tout ce que nous avons vu précédemment reste valable.

#### Code : PHP

```
<?php
$a = array('un', 5.4, 'index' => true);
$b = $a; // ref_count de table array incrémenté de 1
$b = 'pouet'; // ref_count de table array décrémenté de 1 et
création d'un conteneur
```

```
$c =& $a; // table array devient table &array et ref_count
incrémenté de 1
$x =& $a['index'] // conteneur a->index devient &bool(true) et
ref_count incrémenté de 1
```

## La pile

Précédemment, je vous ai dit que les fonctions possèdent toutes une table des symboles. Cela est faux, elles en ont **deux**. Cette deuxième table des symboles est la pile, c'est-à-dire les paramètres réellement passés à la fonction et uniquement cela. En effet, si nous déclarons des variables dans une fonction, la table des symboles locale va les accueillir ; mais pas la pile, qui ne regroupe que les paramètres et rien d'autre. De plus, considérons ce code :

**Code : PHP**

```
<?php

function f ($s)
{
    $s = 'blabla';
}

$s = 'lala';
f($s);
```

Après l'affectation dans la fonction, est-ce que la valeur passée en paramètre est définitivement perdue ? Non, elle se cache encore dans cette fameuse pile. Pour accéder à cette pile, nous allons utiliser les fonctions que nous avons vues précédemment :

- `func_get_arg()`;
- `func_get_args()`;
- `func_num_args()`.

La pile est comparable à un tableau indexé numériquement : la première fonction permet d'obtenir un argument ; la seconde, d'obtenir un tableau contenant tous les arguments et la troisième, d'obtenir le nombre d'arguments réellement passés à la fonction. Vérifions que les paramètres sont bien dans la pile quoi qu'il arrive :

**Code : PHP**

```
<?php

function f ($s)
{
    debug_zval_dump($s);
    $s = 'blabla';
    var_dump(func_get_arg(0));
}

$s = 'lala';
f($s);
```

**Code : Console**

```
string(4) "lala" refcount(4)
string(4) "lala"
```

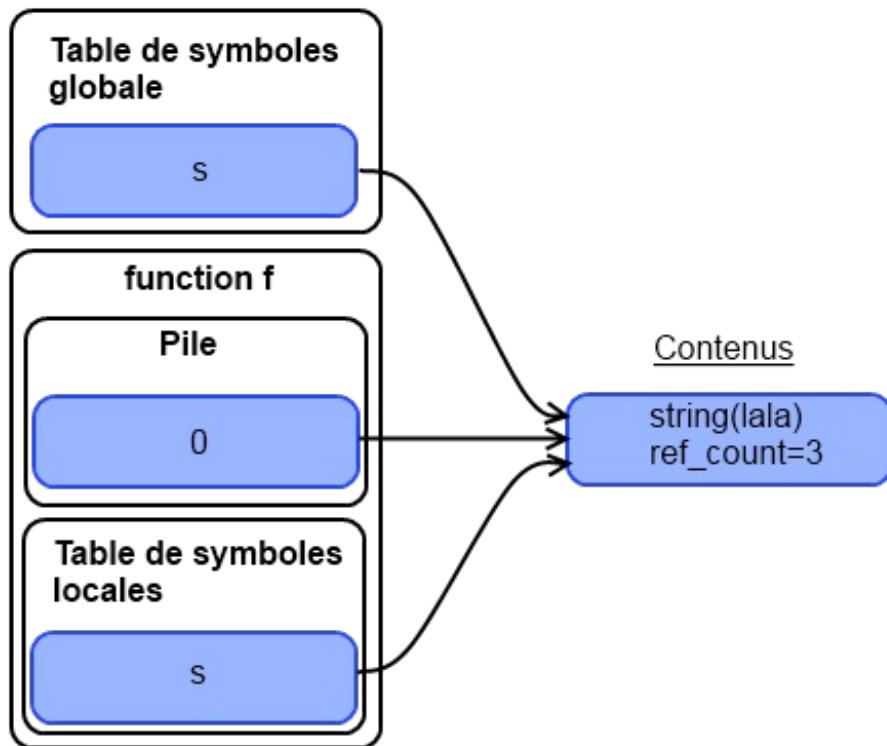
Le compteur de références indique 4. Les quatre symboles associés au conteneur sont les suivants :

- le symbole `s` de la table globale ;
- le symbole `s` de la table locale ;
- le symbole `d` du passage par paramètre à `debug_zval_dump()` ;

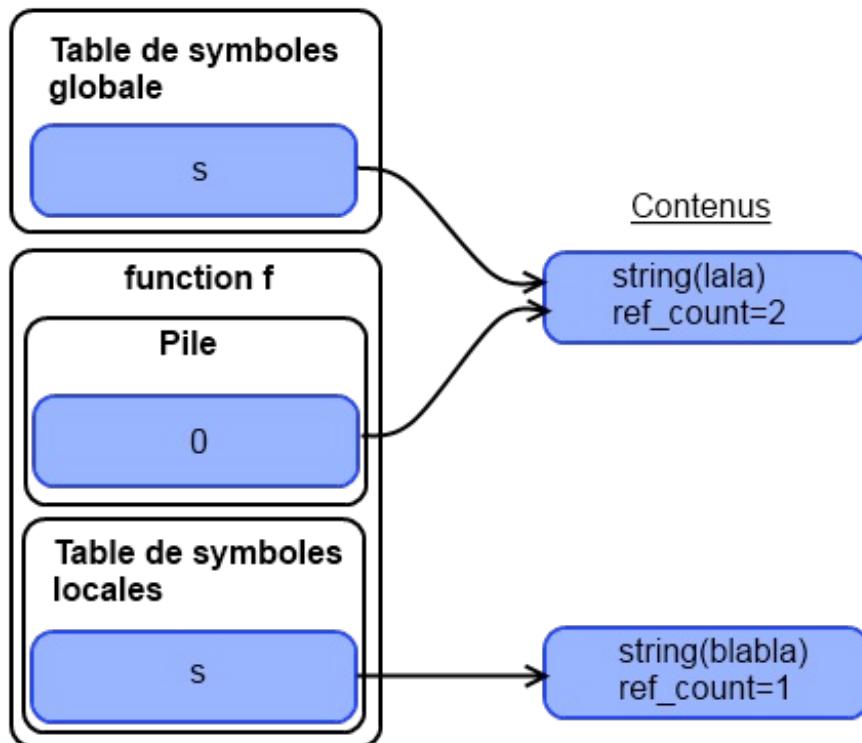
- et le symbole de la pile.

Ne trouvez-vous pas étrange que le compteur de références indique 4 et non 5 ? Je n'ai compté qu'un symbole pour `debug_zval_dump()` et pourtant chaque fonction en introduit deux. Je n'ai pas trouvé de réponse à cette question, mais je suppose que les fonctions natives du PHP fonctionnent différemment puisqu'elles ne sont pas écrites en PHP. Si quelqu'un possède des informations fiables à ce sujet, merci de m'en faire part. 😊

Au final, lors de l'appel de cette fonction, avant l'affectation de `$s`, les tables des symboles sont les suivantes :



Et après l'affectation, voici les tables des symboles :



Maintenant que nous savons tout cela, nous pouvons déterminer très exactement si une copie sera faite ou non, ce qui va me permettre de vous montrer des exemples de retour par référence et de vous expliquer pourquoi le passage par référence n'est pas

toujours la bonne solution pour gagner en performance.

## Pour terminer

Quittons un peu les entrailles du PHP et revenons à nos scripts. Je vous parlais d'un exemple de retour par référence. Pour indiquer qu'une fonction retourne une référence, il suffit d'ajouter une espace devant son nom. Pour le reste, c'est comme d'habitude :

### Code : PHP

```
<?php

function & getReferenceByIndex(& $array, $index)
{
    return $array[$index];
}

$tab = array(4, 5);
$index = & getReferenceByIndex($tab, 1);
$index = 0;
var_dump($tab);
```

### Code : Console

```
int(0) array(2) { [0]=> int(4) [1]=> &int(0) }
```

Pouvez-vous m'expliquer pourquoi cela ne fonctionne pas si je ne passe pas le tableau par référence ? 

#### Secret (cliquez pour afficher)

Imaginons que nous ne le passions pas par référence. PHP est censé agir comme si \$array possédait son propre conteneur, même si, par souci d'optimisation, ça n'est pas immédiatement le cas. Ce qui veut dire que sans référence, nous retournerions normalement une référence... sur la copie. Donc un nouveau conteneur pour un tableau serait effectivement créé, et c'est au conteneur d'un des symboles de ce conteneur que serait associé \$index. Or, nous ne voulons pas associer le symbole à une copie, mais à l'original. Nous sommes donc obligés de passer le tableau par référence.

Il est à noter que les seules expressions utilisables dans un contexte de référence sont les suivantes :

- une variable ;
- une fonction qui retourne une référence ;
- linstanciation d'une classe.

Vous ne savez pas ce qu'est une classe, alors oubliez cela pour le moment. Vous ne pourrez donc pas écrire des choses comme ceci :

### Code : PHP

```
<?php

function f(& $var) {}
f('lala'); // incorrect
f($a = 'pouet'); // incorrect
f($a); // correct
f(array()); // incorrect
f(realpath(__FILE__)); // incorrect
```

## Dépasser la portée

Il peut arriver qu'une fonction ait besoin d'accéder à une variable globale. Actuellement, dans ce genre de situation, nous serions contraints de passer cette information comme paramètre de notre fonction. Toutefois, il est possible de faire autrement grâce à une variable superglobale : **\$GLOBALS**. Cette superglobale est un tableau contenant des références vers toutes les variables qui se trouvent dans la table des symboles globale. Exemple :

**Code : PHP**

```
<?php

function f()
{
    var_dump($GLOBALS['lala']);
}

$lala = 'truc';
f();
```

Si nous devons utiliser plusieurs fois cette variable globale dans notre fonction, nous pourrions être amenés à écrire ceci :

**Code : PHP**

```
<?php

function f()
{
    $lala =& $GLOBALS['lala'];
}

$lala = 'truc';
f();
```

Il faut savoir que le PHP propose une syntaxe strictement équivalente mais plus condensée :

**Code : PHP**

```
<?php

function f()
{
    global $lala;
}

$lala = 'truc';
f();
```

Bien évidemment, si vous désirez avoir non pas une référence mais une copie du contenu de la variable globale, vous ne pourrez pas utiliser cette syntaxe abrégée. Aussi, si jamais le symbole lala n'a pas été déclaré dans la table des symboles globale, il le sera automatiquement.

Enfin, je vous conseille d'éviter d'utiliser des variables globales dans une fonction. Une fonction est une entité qui se veut indépendante de son contexte, nous lui donnons des paramètres, elle effectue ce qu'il faut et nous retourne une valeur. Or, imaginons que nous utilisions des variables globales : si vous souhaitez réutiliser cette fonction et que par mégarde vous oubliez de déclarer la variable globale, votre fonction ne pourra pas remplir son rôle. En utilisant des variables globales dans une fonction, nous la contextualisons et la conséquence directe est qu'elle devient inutilisable dans un contexte autre que celui pour lequel elle a été prévue.

## Des types à part

Nous avons vu que le contenu de nos variables était stocké dans un conteneur. En réalité, ça n'est pas toujours vrai : deux types de contenu ne sont pas stockés dans le conteneur.

Quand nous avons vu l'utilisation de MySQLi, nous étions confrontés à des types particuliers : des objets MySQLi et MySQLi\_Result. Lorsque nous affectons un objet à une variable, ce qui est mis dans le conteneur n'est pas l'objet à proprement parler, mais l'**identifiant** unique de l'objet. L'objet à proprement parler est stocké ailleurs, l'endroit exact n'a aucune importance : tout ce qu'il faut savoir, c'est que dans le conteneur, nous trouvons un identifiant qui permet d'accéder à l'objet.

Puisque c'est l'identifiant qui est stocké dans le conteneur, qu'est-ce qui est copié si un nouveau conteneur est créé à partir du précédent ? L'identifiant, et rien d'autre. Nous avons donc deux symboles associés à deux conteneurs différents dont le contenu est le même. Puisque l'identifiant stocké dans le conteneur est le même, les deux variables se réfèrent au même objet. Par conséquent, il est inutile de passer les objets par référence si l'on désire seulement les utiliser. Exemple :

#### Code : PHP

```
<?php

function f($resultSet)
{
    // on « détruit » le jeu de résultats, il ne sera plus possible
    // d'accéder
    // aux données qui s'y trouvaient
    mysqli_free_result($resultSet);
}

$link = mysqli_connect('localhost', 'root');
$result = mysqli_query($link, 'SELECT 1');
f($result);
mysqli_fetch_assoc($result));
```

Puisque l'objet est partagé, l'objet représenté par \$file devrait également être altéré par mysqli\_free\_result(). C'est bien le cas puisque mysqli\_fetch\_assoc() génère une erreur indiquant qu'il ne peut récupérer les données dans l'objet :

#### Code : Console

```
Warning: mysqli_fetch_assoc() [function(mysqli-fetch-assoc): Couldn't fetch mysqli_result in C:\wamp\www\TutoOO\index.php on line 11
```

Attention, n'allez pas dire que quand nous travaillons sur des objets, c'est comme avec des références. Ce ne sont pas des références, il est très simple de le prouver :

#### Code : PHP

```
<?php

function f($resultSet, & $resultSet2)
{
    $resultSet = null;
    $resultSet2 = null;
}

$link = mysqli_connect('localhost', 'root');
$result = mysqli_query($link, 'SELECT 1');
$result2 = mysqli_query($link, 'SELECT 1');
f($result, $resultSet2);
var_dump($result === null);
var_dump($resultSet2 === null);
```

Dans le premier cas, le paramètre est passé par copie, il y a donc un conteneur pour \$result et un conteneur pour \$resultSet. Or, nous ne mettons null que dans le conteneur qui a été associé à \$resultSet, et uniquement dans celui-ci. Le conteneur associé à \$result demeure inchangé, ce qui explique pourquoi \$result === null renvoie false.

Dans le second cas, le paramètre est passé par référence, \$result2 et \$resultSet2 sont donc associés au même conteneur. Il est donc logique que le \$result2 === null nous retourne true.

Les objets ne sont donc pas des références, et il est parfois nécessaire de les passer par référence.

Le second type de contenu qui n'est pas stocké dans le conteneur est le type resource, ou ressource en français. Une ressource est un type générique qui permet de manipuler diverses choses, comme par exemple des flux de fichiers ou des images. Les objets et les ressources seront bien évidemment vus en détail par la suite.

## foreach et les références

Imaginons que nous souhaitions ajouter 2 à tous les éléments d'un tableau ; nous ferions quelque chose comme ceci :

**Code : PHP**

```
<?php

$tab = range(2, 5);
foreach($tab as $key => $value)
{
    $tab[$key] += 2;
}
var_dump($tab);
```

Nous pourrions toutefois alléger ce code car foreach() permet d'utiliser des références :

**Code : PHP**

```
<?php

$tab = range(2, 5);
foreach($tab as /* $key => */ & $val)
{
    $val += 2;
}
var_dump($tab);
```

Puisque les clés des tableaux sont également des symboles, nous ne pouvons évidemment pas les passer par référence.

Si vous avez attentivement regardé ce que renvoie var\_dump(), quelque chose a dû vous titiller.

**Code : Console**

```
array(4) {
    [0]=>
    int(4)
    [1]=>
    int(5)
    [2]=>
    int(6)
    [3]=>
    &int(7)
}
```

Pourquoi l'indicateur du conteneur associé au symbole 3 indique qu'il s'agit de références ? Il ne peut y avoir référence que si au moins deux symboles sont associés au conteneur, mais quel est ce second symbole ?

Ce second symbole vient du foreach(), c'est \$val qui est une référence. En effet, foreach() ne détruit pas la variable qu'il utilise : après avoir parcouru le tableau, elle existe toujours. Vérifions cela :

**Code : PHP**

```
<?php

$tab = range(2, 5);
foreach($tab as & $val)
{
    $val += 2;
}
$val = 'meuh';
var_dump($tab);
```

**Code : Console**

```
array(4) {  
    [0]=>  
    int(4)  
    [1]=>  
    int(5)  
    [2]=>  
    int(6)  
    [3]=>  
    &string(4) "meuh"  
}
```

Mais il y a pire : non seulement foreach() ne détruit pas le symbole utilisé, mais en plus, si ce symbole existe déjà, foreach() écrira dedans sans nous prévenir. Exemple :

**Code : PHP**

```
<?php  
  
$tab = array('a', 'c', 'z');  
$c = 'lala';  
$var =& $c;  
foreach($tab as $var) {}  
var_dump($c);
```

**Code : Console**

```
string(1) "z"
```

À cause de cela, nous pourrions être confrontés à des bogues étranges et difficilement détectables ; exemple :

**Code : PHP**

```
<?php  
  
$tab = array('a', 'c', 'z');  
foreach($tab as & $var) {}  
var_dump($tab);  
foreach($tab as $var) {}  
var_dump($tab);
```

**Code : Console**

```
array(3) {  
    [0]=>  
    string(1) "a"  
    [1]=>  
    string(1) "c"  
    [2]=>  
    &string(1) "z"  
}  
array(3) {  
    [0]=>  
    string(1) "a"  
    [1]=>  
    string(1) "c"
```

```
[2] =>
&string(1) "c"
}
```

Après le premier foreach(), \$var est une référence sur la dernière case du tableau. À la première itération du second foreach(), le PHP y écrit la valeur contenue dans la première case : 'a'. À la seconde itération, il y écrit la valeur contenue dans la seconde case : 'c'. Et à la troisième et dernière itération, il y inscrit son propre contenu, d'où ce 'c' dans la dernière case du tableau.

Pour éviter ce genre de soucis, je vous conseille :

- soit d'utiliser des symboles uniques dans un foreach() ;
- soit de détruire le symbole utilisé, **aussi bien avant qu'après** le foreach(), avec unset() ;
- ou alors d'encapsuler le foreach() dans une fonction.

## Les variables statiques

Parler de « staticité » sur les variables, qui sont censées être dynamiques, voilà tout un concept, n'est-ce pas ? 😊

Nous avons vu que lorsqu'une fonction se termine, sa table des symboles se vide, ce qui implique que tous les conteneurs associés peuvent éventuellement être supprimés. Cela explique pourquoi les valeurs des variables sont réinitialisées à chaque appel de fonction. L'idée des variables statiques est d'éviter cela : une variable statique est une variable qui ne sera pas affectée par la fin de la fonction. Exemple :

### Code : PHP

```
<?php

function f()
{
    static $count = 0;
    $count++;
    var_dump($count);
}
f();
f();
```

D'un appel à l'autre, la valeur est conservée, c'est pourquoi nous obtenons d'abord un 1, puis un 2. Il est à noter que détruire le symbole count n'affectera pas la valeur, comme nous pourrions le supposer. Pourtant, count est le seul symbole associé au contenu, donc si nous le supprimons, le conteneur associé devrait être supprimé également. La solution se cache ici :

### Code : PHP

```
<?php

function f()
{
    static $count = 0;
    debug_zval_dump($count);
}
f();
```

### Code : Console

```
long(0) refcount(1)
```

Si debug\_zval\_dump() nous donne 1 comme valeur pour le compteur de références, c'est que le symbole est... une référence. Oui, \$count est une référence. Nous n'avons pas accès à l'autre symbole, il est probablement caché dans une autre table des symboles propre au PHP.

Tout ce que nous avons vu précédemment reste bien évidemment valable, puisqu'il s'agit d'un symbole et d'un conteneur comme les autres. Nous pouvons donc par exemple retourner une référence sur la variable statique :

**Code : PHP**

```
<?php

function & f()
{
    static $count = 0;
    $count++;
    var_dump($count);
    return $count;
}
$a =& f();
$a = 50;
f();
```

**Code : Console**

```
int(1) int(51)
```

## Ne pas abuser des références

Après avoir vu tout ça, vous êtes à même de comprendre pourquoi, dans certains cas, les références sont à éviter. Premier exemple :

**Code : PHP**

```
<?php

function f(& $var)
{
    var_dump($var);
}

$s = 'lala';
f($s);
```

Au début, nous avons un symbole et un conteneur. Lors de l'appel de la fonction, un nouveau symbole est créé et associé au conteneur précédent. Lors de l'appel à var\_dump(), le paramètre n'est pas passé par référence, mais l'indicateur du seul conteneur présent indique que les symboles auxquels il est associé sont des références ; le PHP va donc... créer un nouveau conteneur. Si nous n'avions pas utilisé de référence, comme la copie n'est faite qu'à l'écriture et qu'il n'y en a aucune, aucune copie n'aurait été faite.

Dans ce cas, les références provoquent donc une chute de performance.

Voici le second exemple :

**Code : PHP**

```
<?php

function & f()
{
    $var = 'lala';
    return $var;
}

$s = & f();
```

Cette fois-ci, il n'y a pas de duplication de contenu. Toutefois, avant que la fonction ne soit terminée, deux symboles, var et \$, seront des références vers un même conteneur. L'indicateur de références va donc devoir changer d'état. Une fois la fonction terminée, il ne restera qu'un symbole associé au conteneur, l'indicateur de références va donc devoir à nouveau changer d'état, puisque pour qu'il y ait référence, il faut au moins deux symboles. Si nous n'avions pas utilisé les références, le résultat aurait été le même et ces opérations n'auraient pas dû être faites.

À cause de cet abus de références, nous avons fait deux opérations parfaitement inutiles.

Sachez donc modérer l'utilisation des références, car le PHP n'est pas idiot et optimise lui-même la copie de conteneur. Si vous êtes arrivés ici en ayant tout lu, tout compris, d'une traite et sans mal de tête, félicitations ! Il ne faut bien sûr pas retenir tout ce chapitre sur le bout des doigts, mais je trouvais assez intéressant de vous expliquer les choses dans le détail. 

## Fragmenter son code

Dans ce chapitre, nous allons voir comment fragmenter notre code. Attention, « fragmenter » peut avoir deux sens :

- fragmentation réelle, physique du code ;
- fragmentation logique du code.

Les fonctions constituent une façon logique de fragmenter le code ; ici, nous allons parler d'une manière de fragmenter physiquement le code et d'une autre en le fragmentant logiquement.

### Des scripts dans le script

Dans une application réelle, le nombre de lignes de code peut devenir vraiment conséquent. Or, se retrouver avec des fichiers de 100 000 lignes de code, ça n'est pas très sexy. Par ailleurs, si vous avez des éléments, des fonctions par exemple, utilisés dans différents scripts, ça n'est carrément pas sexy du tout de faire du copier-coller.

Prenons un exemple pratique : vous avez une magnifique fonction qui fait quelque chose de vraiment exceptionnel, et vous avez besoin de l'utiliser dans deux scripts différents. Actuellement, il vous faut faire un copier-coller, avec tous les inconvénients que ça apporte.

Le premier, c'est la duplication de code. Dupliquer du code, ça veut dire avoir globalement plus de lignes de code, mais pas seulement. Dupliquer du code, ça veut dire que s'il faut modifier le code, il faut faire cette modification sur toutes les copies. Le risque d'oublier des copies n'est pas nul et croît avec la taille du projet, c'est donc difficilement gérable.

Le second inconvénient, c'est la perte de temps. Retrouver la partie utile du code, faire un copier-coller, toutes les modifications à faire en cas d'édition, ça peut prendre beaucoup de temps.

L'idéal serait, exactement comme pour les fonctions, de définir une fois le code utile quelque part, dans un script par exemple, et de pouvoir appeler ce script quand nous le désirons. Cela tombe bien car le PHP le permet grâce à la structure de langage `include`. Soit un fichier `fonctions.php` dont nous aurons besoin dans plusieurs scripts :

#### Code : PHP - fonctions.php

```
<?php  
  
function f()  
{  
    echo 'ça fonctionne';  
}
```

Et le script qui a besoin de cette douce fonction :

#### Code : PHP - index.php

```
<?php  
  
include 'fonctions.php';  
  
f();
```

Le fonctionnement de cette structure est très simple à comprendre. Quand une inclusion est faite, le PHP va exécuter le script inclus puis continuer à exécuter le script principal. Les deux codes précédents sont donc strictement équivalents à ceci :

#### Code : PHP

```
<?php  
  
function f()  
{  
    echo 'ça fonctionne';  
}  
  
f();
```

Bien évidemment, ce script possède tous les défauts que nous avons énumérés précédemment, il faut donc préférer la version avec inclusion. Vous pourrez par exemple avoir un script dans lequel vous définirez les constantes utiles à votre application, un autre dans lequel vous implémenterez les fonctions relatives à la gestion du livre d'or, un autre dans lequel vous implémenterez les fonctions relatives aux news, etc. Ainsi, quand nous aurez besoin de l'un ou l'autre de ces éléments, une simple instruction et c'est fini.

Puisque les deux codes précédents sont équivalents, il va de soi que tout ce qui a été défini dans le script appelant est accessible dans le script appelé, et inversement. L'instruction include peut être placée un peu partout, dans les fonctions par exemple. Dans pareille situation, tout se passe exactement comme si nous avions fait un copier-coller dans la fonction. Les variables déclarées dans le script inclus ne seront donc pas accessibles depuis l'extérieur de la fonction ; de plus, toutes les fonctions, constantes et autres éléments globaux seront accessibles de partout après le premier appel à la fonction. Exemple :

#### Code : PHP - fonctions.php

```
<?php

define('POUET', 'pouet');
$a = 'lala';
function f()
{
    echo 'ça fonctionne';
}
```

#### Code : PHP - index.php

```
<?php

function testInclude()
{
    include 'fonctions.php';
    var_dump($a); // string lala
    var_dump(POUET); // string pouet
    f(); // affiche 'ça fonctionne'
}

var_dump(POUET); // erreur constante indéfinie
testInclude();
var_dump(isset($a)); // bool false
f(); // affiche 'ça fonctionne'
var_dump(POUET); // string pouet
```

Vous pouvez bien évidemment utiliser des chemins absous ou relatifs dans l'include ; exemple :

#### Code : PHP

```
<?php

include '../lala.php';
include 'C:/truc.php';
```

## Une boucle infinie

Que se passerait-il si dans un script **a.php** nous inclusions un script **b.php**, et que dans ce script **b.php** nous inclusions le script **a.php** ?

A inclut B, B inclut A, A inclut B, B inclut A, et ainsi de suite jusqu'à la fin des temps : c'est la boucle infinie. Toutefois, il arrive que ce genre de schéma se produise, par exemple lorsqu'un script C inclut un script B, qui lui-même inclut un script A, qui à son tour inclut le script C. Ce schéma n'est pas impossible, et il serait bien de pouvoir éviter cette boucle infinie. C'est pour cela qu'en plus d'include, le PHP propose une autre instruction d'inclusion : include\_once. Le comportement et l'utilisation sont rigoureusement identiques à une exception près : si le code a déjà été inclus, include\_once n'inclura pas à nouveau le code. Cela permet d'éviter la boucle infinie :

**Code : PHP - index.php**

```
<?php  
  
echo 'inclus test.php';  
include_once 'test.php';
```

**Code : PHP - test.php**

```
<?php  
  
echo 'inclus index.php';  
include_once 'index.php';
```

La sortie sera la suivante :

**Code : Console**

```
inclus test.php  
inclus index.php
```

## Retourner une valeur

Si vous faites un `var_dump()` d'un include, vous verrez que la valeur est le booléen `true`. Il est toutefois possible de changer cela et de retourner n'importe quelle valeur. Pour le faire, rien de bien sorcier, il suffit d'utiliser l'instruction `return` au niveau global du script inclus. Exemple :

**Code : PHP - test.php**

```
<?php  
  
return 'pouet';
```

**Code : PHP - index.php**

```
<?php  
  
var_dump(include 'test.php');
```

**Code : Console - Sortie de index.php**

```
string(5) "pouet"
```

## L'`include_path`

Dans le dossier où se trouvent les fichiers `index.php` et `test.php`, créez un sous-dossier nommé `test` et mettez-y le fichier `test.php`. Maintenant, essayons ceci :

**Code : PHP - test/test.php**

```
<?php  
echo 'test/test.php';
```

#### Code : PHP - index.php

```
<?php  
include 'test.php';
```

Le PHP essaie d'inclure un fichier qui devrait se trouver dans le même répertoire que le script appelant ; ce fichier n'existe pas, une erreur est donc générée. L'*include\_path* est une chaîne de caractères contenant un ensemble de chemins vers des dossiers séparés par un caractère : le point-virgule (;) sous Windows et le deux-points (:) sous Linux. Lorsque nous faisons une inclusion et que le fichier n'est pas trouvé, le PHP va tenter de l'inclure en utilisant un par un les dossiers de l'*include\_path*. Exemple :

#### Code : PHP - index.php

```
<?php  
set_include_path(__DIR__ . '/test');  
include_once 'test.php';
```

Le fichier **test.php** n'existe pas dans le même répertoire que le script appelant, le PHP va donc regarder dans l'*include\_path*. Il y a un chemin, le PHP tente donc d'inclure **\_\_DIR\_\_**.**'/test/test.php'** ; or, la constante **\_\_DIR\_\_** donne le chemin d'accès au dossier courant. Le PHP tente donc d'inclure le fichier **test.php** se trouvant dans le dossier **test** que nous venons de créer : le fichier y est bien, le PHP l'inclut.

Si je crée un fichier **test.php** dans le même dossier que le script appelant avec ce contenu :

#### Code : PHP - test.php

```
<?php  
echo 'test.php';
```

Qu'est-ce qui sera affiché à l'écran ?

Il suffit de faire par ordre : est-ce qu'un fichier **test.php** existe dans le dossier du script appelant ? Oui, ce fichier est donc inclus, et à l'écran sera affiché « **test.php** ».

Si ce fichier n'avait pas existé, nous nous serions retrouvés dans le cas précédent et « **test/test.php** » aurait été affiché à l'écran.

Donc, en résumé, quand nous tentons d'inclure un fichier :

- le PHP tente d'inclure le fichier directement ;
- si le fichier n'est pas directement trouvé, le PHP utilise l'*include\_path* ;
- le PHP va, pour chaque dossier contenu dans l'*include\_path*, tenter d'inclure le fichier en faisant précéder son nom du chemin du dossier courant dans le parcours de l'*include\_path* ;
- si le fichier peut être inclus, la recherche s'arrête, sinon la recherche continue ;
- si l'*include\_path* ne permet pas d'inclure le fichier, une erreur est émise.

Toutefois, je vous déconseille d'abuser de l'*include\_path* : tester les dossiers un par un prend du temps. Il est généralement utilisé pour de grosses bibliothèques qui y définissent uniquement l'emplacement du dossier racine de la bibliothèque, de sorte que vous puissiez la mettre où vous le souhaitez.

## Des fichiers critiques

Il se peut que, dans votre application, vous ayez à inclure des fichiers vitaux à votre application : si l'inclusion échoue, l'application ne pourra pas fonctionner. Dans ce genre de situation, il serait bien de pouvoir terminer le script et éviter d'afficher une ribambelle d'erreurs à vos visiteurs.

include et include\_once possèdent chacun une fonction jumelle : require et require\_once. Le fonctionnement et l'utilisation sont exactement les mêmes, à une différence près : le comportement si l'inclusion ne peut être faite.  
Dans le cas d'include, si l'inclusion ne peut être faite, une erreur est émise.  
Dans le cas de require, si l'inclusion ne peut être faite, une erreur fatale est émise, le script se termine donc. Exemple :

**Code : PHP**

```
<?php

include 'fichierQuiNExistePas';
echo 'toujours vivant !';
require 'fichierQuiNExistePas';
echo 'eh bah non je suis fini';
```

La chaîne « toujours vivant » sera bel et bien affichée tandis que l'autre ne le sera pas puisque l'inclusion a échoué et que require a émis une erreur fatale.

## Les dangers de l'inclusion

Beaucoup d'applications utilisent les inclusions d'une façon particulière : l'utilisateur choisit la page à inclure. Concrètement, l'utilisateur, d'une façon ou d'une autre, indique à un script (*via* \$\_GET ou autre) quel script exécuter ; exemple :

**Code : PHP**

```
<?php

if(isset($_GET['file']))
{
    include $_GET['file'];
}
```

Si l'on rajoute à l'url un petit &file=test.php, le script test.php sera inclus. Mais que se passerait-il si j'ajoutais ceci : &file=http://monserveur/unscriptmechant.php ?

include inclurait le script et l'exécuterait donc. Si ce script est un script malveillant, il pourrait par exemple récupérer toutes les variables déclarées, et par la même occasion peut-être des mots de passe ; il pourrait également supprimer des fichiers. En fait, il pourrait absolument tout faire. Si vous laissez un élément extérieur altérer vos inclusions, il est nécessaire de vérifier ce que vous allez inclure. Vous pouvez par exemple vérifier que ce que vous allez inclure n'est pas un lien *via* substr() ou strpos(), qu'il s'agisse bien d'un fichier sur votre serveur, vous pouvez aussi utiliser un système de liste blanche qui permettrait de filtrer les inclusions non-autorisées, etc. Il y a beaucoup de façons de faire : peu importe celle que vous choisissez, mais soyez bien prudents. 😊

**Code : PHP**

```
<?php

$includeAllowed = array(
    'test.php' => 1,
    'lala.php' => 1,
    // ...
);

if(isset($includeAllowed[$_GET['file']]))
{
    include $_GET['file'];
}
else
{
    // pas dans la liste, erreur
}
```

## \$\_GET et les inclusions

Considérons ce script :

**Code : PHP**

```
<?php  
var_dump($_GET['lala']);
```

Si ce script est inclus et que \$\_GET['lala'] n'a pas été défini, une erreur sera émise. La question est donc : comment passer des paramètres via \$\_GET ou \$\_POST à un script inclus ?

C'est extrêmement simple, il suffit de les déclarer dans le script appelant :

**Code : PHP**

```
<?php  
$_GET['lala'] = 'bidule';  
include 'codePrecedent.php';
```

Il est aussi possible de donner non pas un chemin vers le fichier, mais une URL :

**Code : PHP**

```
<?php  
include 'http://localhost/tuto/codePrecedent.php?lala=bidule';
```

Par contre, ceci ne fonctionnera pas :

**Code : PHP**

```
<?php  
include 'codePrecedent.php?lala';
```

En effet, dans pareil cas, le PHP essaie d'inclure un fichier dont le nom est **codePrecedent.php?lala**.

## Les espaces de noms

Sur votre disque dur, il est assez probable que vous ayez deux fichiers portant le même nom. Cela est possible car ces deux fichiers se trouvent dans deux dossiers différents. Ainsi, porter le même nom n'est pas un problème car le dossier dans lequel se trouve chaque fichier permet de le différencier de l'autre. Nous aurions par exemple ceci :

- /home/lala/index ;
- /home/lili/index .

Imaginons maintenant que vous écriviez deux bibliothèques de fonctions : une qui traite des nombres complexes et une autre qui traite des nombres réels. Nous pouvons supposer que chacune de ces bibliothèques intégrera une fonction en vue d'additionner deux nombres. Il serait agréable de pouvoir appeler ces fonctions sum(), puisqu'elles font effectivement une somme.

Malheureusement, de la même façon qu'il est impossible de créer deux fichiers portant le même nom dans un même dossier, il est impossible de créer deux fonctions portant le même nom. Fort heureusement, le PHP nous propose une fonctionnalité qui permet de créer des sortes de « dossiers » virtuels dans lesquels nous pourront ranger des éléments, des fonctions par exemple. Cette fonctionnalité, ce sont les espaces de noms, ou *namespace*. Exemple :

**Code : PHP - functions.php**

```
<?php
```

```
function sum()
{
    return 'Complex\sum';
}

function sum()
{
    return 'Real\sum';
}
```

Ce code génère une erreur, c'est le comportement habituel. Maintenant, nous allons ranger ces deux fonctions dans des espaces de noms propre à chacune :

**Code : PHP - functions.php**

```
<?php

namespace Complex
{
    function sum()
    {
        return 'Complex\sum';
    }
}

namespace Real
{
    function sum()
    {
        return 'Real\sum';
}
```



Les espaces de noms ont été intégrés à partir de la version 5.3 du PHP, vérifiez donc votre version du PHP si une erreur est générée.

Incluons ce fichier dans un script et testons :

**Code : PHP**

```
<?php

include 'functions.php';
var_dump(sum());
```

Une jolie erreur, chouette alors. 😊 Lorsque vous explorez votre disque dur, peu importe où vous vous trouvez, vous êtes toujours dans l'un ou l'autre dossier, car la racine du disque dur est elle-même un dossier. Lorsque vous êtes dans un dossier, vous pouvez accéder aux fichiers en utilisant uniquement leur nom. Si vous n'êtes pas dans le même dossier mais dans un dossier parent, vous devrez faire précéder le nom du fichier par le nom du dossier dans lequel se trouve le fichier. Et si vous n'êtes pas dans un dossier parent, vous devrez spécifier le nom du fichier précédent de l'arborescence complète. Les espaces de noms fonctionnent exactement de la même façon. Lorsque nous ne disons pas au PHP dans quel espace de noms nous sommes, il considère que nous sommes à la racine des espaces de noms, l'espace de noms global. Ainsi, il est logique que le code précédent provoque une erreur : nous appelons une fonction sum() ; or, nous avons déclaré ces fonctions dans un espace de noms particulier et non dans l'espace de noms global. Pour accéder à la fonction tant désirée, nous allons donc utiliser son nom précédé de l'espace de noms :

**Code : PHP**

```
<?php
```

```
include 'functions.php';
var_dump(Complex\sum());
var_dump(Real\sum());
```

Bien évidemment, si l'élément dont vous avez besoin se trouve dans le même espace de noms que celui dans lequel vous êtes, vous n'aurez pas à spécifier l'espace de noms, exactement comme avec les fichiers.

#### Code : PHP - functions.php

```
<?php

namespace Complex
{
    function sum()
    {
        return 'Complex\sum';
    }

    function something()
    {
        return sum();
    }
}

namespace Real
{
    function sum()
    {
        return 'Real\sum';
    }
}
```

#### Code : PHP

```
<?php

include 'functions.php';
var_dump(Complex\something());
```

## Créer des espaces de noms

Pour le moment, oublions l'utilisation des espaces de noms et concentrons-nous sur leur création. La syntaxe que je vous ai montrée précédemment permet de déclarer plusieurs espaces de noms dans un seul fichier. Toutefois, il est conseillé, par souci de lisibilité, de n'avoir qu'un seul espace de noms par fichier. Si vous avez deux espaces de noms, créez deux fichiers. Si vous en avez trois, créez-en trois, la lisibilité n'en sera que meilleure.

Aussi, sachez que les accolades ne sont pas obligatoires. Si je les ai mises dans l'exemple, c'est pour délimiter aisément les espaces de noms en se basant sur l'indentation, mais j'aurais tout aussi bien pu les omettre. Si nous ne déclarons qu'un espace de noms par fichier, il est évident que ces accolades ne servent à rien puisque l'ensemble du fichier sera dans le même espace de noms.

Une des limitations du PHP est que vous ne pouvez placer aucun code avant la déclaration de l'espace de noms, mis à part des commentaires et l'instruction declare. Si vous essayez de mettre quelque instruction que ce soit, vous aurez droit à cette erreur :

#### Code : Console

```
Fatal error: Namespace declaration statement has to be the very first statement in
```

Lorsque vous déclarez un espace de noms, ce que vous déclarez se réfère à l'espace de noms global. Concrètement, si j'écris ceci : `<?php namespace Complex;`, je déclare un espace de noms Complex qui se trouve dans l'espace de noms global. Le nom « pleinement qualifié » (c'est-à-dire le nom complet depuis l'espace de noms global) de l'espace de noms est donc \Complex, le *backslash* représentant l'espace de noms global.

Peu importe comment ce script sera utilisé, peu importe qu'il soit inclus dans un script qui déclare un autre espace de noms : lorsque vous déclarez un espace de noms, il se réfère toujours à l'espace de noms global.

De la même façon que vous pouvez créer des sous-dossiers, c'est-à-dire des dossiers contenus dans des dossiers, vous pouvez déclarer des sous-espaces de noms. Pour ce faire, il suffit de séparer les différents niveaux d'espace de noms par un *backslash*. Ainsi, `<?php namespace Library\IO\Stream;` déclare un sous-sous-espace de noms dont le nom pleinement qualifié est \Library\IO\Stream.

Aussi, il est tout à fait possible d'avoir un même espace de noms dans plusieurs fichiers différents. Vous pouvez déclarer un espace de noms Pouet dans un fichier **a.php**, et un même espace de noms Pouet dans un fichier **b.php**. Si vous incluez les deux fichiers dans un même script, ces deux espaces de noms n'en formeront qu'un. Exemple :

#### Code : PHP - a.php

```
<?php  
  
namespace Pouet;  
  
function a  
{  
    echo 'a';  
}
```

#### Code : PHP - b.php

```
<?php  
  
namespace Pouet;  
  
function b  
{  
    echo 'b';  
}
```

#### Code : PHP

```
<?php  
  
include 'a.php';  
include 'b.php';  
  
Pouet\ a();  
Pouet\ b();
```

Sachez qu'un espace de noms est généralement un espace « déclaratif ». Exécuter du code dans un espace de noms n'est bien sûr pas incorrect syntaxiquement, mais en général, vous ne ferez que déclarer des éléments dans un espace de noms. Les éléments que vous pouvez déclarer dans un espace de noms sont les suivants :

- fonctions ;
- constantes ;
- classes.

Nous verrons ce qu'est une classe par la suite, mais arrêtons-nous un instant sur les constantes. Nous avons vu que pour définir une constante, la fonction `define()` était employée. Or, cette fonction agit sans tenir compte des espaces de noms. C'est pour cela qu'une syntaxe tenant compte des espaces de noms a été introduite :

**Code : PHP - a.php**

```
<?php  
  
namespace Lala;  
  
const NomConstante = 51;
```

**Code : PHP**

```
<?php  
  
include 'a.php';  
var_dump(Lala\NomConstante);
```

Toutefois, cette syntaxe est plus limitée que define() : vous ne pouvez utiliser que des expressions constantes. Vous ne pourrez donc pas écrire ce genre de chose :

**Code : PHP**

```
<?php  
  
namespace Lala;  
  
const A = long2ip(0); // expression non constante, fonction  
const B = 5 + 2; // expression non constante, expression complexe
```

Il faut également signaler que si vous tentez d'utiliser une constante non définie, deux cas sont possibles :

- soit vous ne spécifiez pas d'espace de noms, Constante par exemple, et une erreur est émise ;
- soit vous spécifiez un espace de noms, \Constante par exemple, et le script se termine car l'erreur émise est une erreur fatale.

Pour terminer, il faut signaler que toute classe, constante ou fonction déclarée par le PHP se trouve dans l'espace de noms global. Le nom pleinement qualifié de la fonction ip2long(), par exemple, est donc \ip2long().

## Utiliser les espaces de noms

Après la déclaration vient l'utilisation. En réalité, l'utilisation n'est pas bien complexe, mais la façon dont le PHP utilise et résout les espaces de noms l'est un peu plus.

Premièrement, il faut savoir qu'il y a trois « façons » de nommer un élément. Soient les scripts suivant :

**Code : PHP - a.php**

```
<?php  
  
namespace Un\Exemple;  
  
function f() {}
```

**Code : PHP**

```
<?php  
  
include 'a.php';  
  
Un\Exemple\f();  
\Un\Exemple\f();  
f();
```

- Un \Exemple\f() est un nom dit « qualifié » car des espaces de noms sont spécifiés.
- \Un\Exemple\f() est un nom dit « pleinement qualifié » ou « absolu » car nous partons de l'espace de noms global.
- f() est un nom dit « non qualifié » car il n'y a aucune mention d'espace de noms.

Lorsqu'il s'agit d'un nom non-qualifié, voici comment le PHP résout le problème :

- le PHP cherche un élément correspondant dans l'espace de noms courant ;
- si aucun élément correspondant n'est trouvé dans l'espace de noms courant, le PHP cherche un élément correspondant dans l'espace de noms global ;
- si aucun élément correspondant n'est trouvé dans l'espace de noms global, une erreur est émise.

Vérifions cela :

**Code : PHP - a.php**

```
<?php

namespace Test
{
    function strlen()
    {
        return 500000;
    }

    function a()
    {
        return strlen('lala');
    }
}
```

**Code : PHP**

```
<?php

include 'a.php';
var_dump(Test\a());
```

Dans la fonction \Test\ a(), nous utilisons un nom non qualifié. Une fonction strlen() existe dans l'espace de noms courant, elle est donc utilisée.

Si nous supprimions la fonction \Test\strlen(), le PHP ne trouverait pas d'élément correspondant dans l'espace de noms courant, il appellerait donc la fonction strlen() du PHP qui se trouve dans l'espace de noms global.

Dans les deux autres cas, soit le PHP trouve l'élément dans l'espace de noms et l'utilise, soit une erreur est émise. Exemple avec un nom pleinement qualifié :

**Code : PHP**

```
<?php

namespace Test
{
    function a()
    {
        return \Test\strlen('lala');
    }
}
```

L'élément \Test\strlen() n'existe pas, Le PHP émet bien une erreur, fatale qui plus est.

Il peut arriver que nous redéfinissions dans un espace de noms un élément qui existe déjà dans l'espace de noms global. Si nous utilisons un nom non qualifié, le PHP pourrait être amené à aller chercher dans l'espace de noms global. Or, nous souhaiterions que ce ne soit pas le cas et qu'une erreur soit directement émise plutôt que d'aller chercher dans l'espace de noms global. Pour ce faire, nous pouvons utiliser le nom pleinement qualifié de l'élément ; toutefois, cela peut être relativement lourd si nous imbriquons beaucoup d'espaces de noms. Mais par chance, le PHP a introduit une façon très simple d'obtenir le nom pleinement qualifié de l'espace de noms courant.

**Code : PHP**

```
<?php

namespace Test
{
    function a()
    {
        return namespace\strlen('lala');
    }
}
```

Ce code est strictement équivalent au précédent et nous aurait épargné bien des caractères si nous avions imbriqué beaucoup d'espaces de noms. Puisque `namespace` représente l'espace de noms pleinement qualifié, le nom de l'élément est également un nom pleinement qualifié.

Il existe également une constante dite « magique » qui représente, sous forme de chaîne de caractères, l'espace de noms courant pleinement qualifié : `__NAMESPACE__`.

**Code : PHP**

```
<?php

namespace Test
{
    function a()
    {
        var_dump(__NAMESPACE__);
    }
}
```

Vous avez sans doute remarqué qu'il n'y a pas de `backslash` au début du nom de l'espace de noms ; c'est dommage, mais ça reste le nom pleinement qualifié, bien que le PHP n'en fasse qu'à sa tête.

Lorsque vous aurez de nombreuses imbriques d'espaces de noms, il deviendra fastidieux d'utiliser les noms pleinement qualifiés. Qu'à cela ne tienne, le PHP, dans son éternelle bienveillance, nous propose de donner des alias aux espaces de noms. Concrètement, nous pourrions dire que « Alias » correspond à l'espace de noms « `\Un\Exemple\Qui\Est\Tordu` ». Exemple :

**Code : PHP - a.php**

```
<?php

namespace Un\Exemple\Qui\Est\Tordu;

function a()
{
    var_dump(__NAMESPACE__);
}
```

**Code : PHP**

```
<?php

include 'test.php';
use Un\Exemple\Qui\Est\Tordu as Alias;
```

```
Un\Exemple\Qui\Est\Tordu\a();
Alias\a();
```

Comme Alias est un alias de l'espace de noms Un\Exemple\Qui\Est\Tordu, les deux appels sont strictement identiques. De la même façon que l'instruction namespace, use se réfère toujours à l'espace de noms global.

Attention, créer un alias ne garantit pas que l'espace de noms pour lequel l'alias est créé existe. Ce code ne génère aucune erreur :

#### Code : PHP

```
<?php
use lilalala\lala as lala;
```

Dans ce code, quelque chose de particulier a été fait : l'alias est équivalent à l'espace de noms le plus profond. En effet, l'espace de noms le plus profond dans \lilalala\lala est lala, et l'alias est également lala. Dans pareil cas, le as est superflu. Ainsi, le code précédent équivaut à celui-ci :

#### Code : PHP

```
<?php
use lilalala\lala;
```

Toutefois, quand le as est omis, le comportement peut différer lorsque l'alias porte sur un espace de noms non composé, c'est-à-dire un espace de noms sur un seul niveau. Exemple :

#### Code : PHP

```
<?php
use lala as lala;
use lili;
```

Le premier use ne génère aucune erreur, alors que le second en émet une.

Pour information, les alias ne changent strictement rien à la façon dont le PHP résout le nom des éléments.

Voilà une bonne chose de faite. 😊 Les inclusions sont fondamentales, tous les scripts un tant soit peu sérieux les utilisent. Les espaces de noms ne sont pas forcément utiles de prime abord, mais nous verrons une astuce très sexy qui les utilise lorsque nous parlerons de la programmation orientée objet, donc comprenez-les !

## Les nombres

Les nombres, ces nombres qui paraissent si anodins, sont pourtant un sujet aussi vaste qu'intéressant. Dans ce chapitre, nous allons parler de ce que sont les nombres, comment les écrire et surtout de quelles manières le PHP permet de le faire.

### Comment écrire un nombre ?

Avant de nous attaquer à l'écriture des nombres, posons-nous les questions suivantes : qu'est-ce qu'un nombre ? À quoi servent-ils ?

Un nombre est un concept qui permet de représenter une quantité et de la comparer à d'autres quantités. Prenons un exemple : si je vous dis que j'ai douze pommes, vous pouvez vous représenter dans votre tête ce que je possède, c'est-à-dire la quantité de pommes que je possède. De même, en vous disant que j'ai douze pommes, vous savez que j'aurai plus à manger qu'une personne qui vous dirait qu'elle ne possède qu'une pomme, car vous pouvez comparer ces nombres, et donc la quantité qu'ils représentent.

Un nombre, par exemple douze (ou plutôt 12), est composé de chiffres. Un chiffre est tout simplement un symbole arbitraire utilisé pour écrire des nombres. Dans le système le plus utilisé, les chiffres employés sont les suivants : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. Pour faire une analogie, si vous comparez un nombre à un mot, alors vous pouvez comparer un chiffre à une lettre.

Pour écrire un nombre, nous utilisons une notation particulière, la « notation positionnelle ». Cette notation utilise deux notions :

- la position du chiffre dans le nombre ;
- la base.

Prenons un nombre, 1423 par exemple. Ce nombre est composé de quatre chiffres ; nous allons définir leur position. Disons que le chiffre le plus à droite, 3, est à la position dite **0**. Le chiffre directement à sa gauche, 2, sera à la position **1**. L'idée est donc de définir que la position du chiffre à l'extrême droite est la position 0, et ensuite d'incrémenter cette position en passant d'un chiffre à l'autre vers la gauche. La position du chiffre 4 dans le nombre 1423 est donc 2, tandis que la position du chiffre 1 est 3. Si **c** représente un chiffre quelconque et que nous mettons sa position en indice, nous pouvons écrire un nombre composé de **n** chiffres de cette façon :  $c_{n-1}c_{n-2}\dots c_3c_2c_1c_0$ .

Avant de parler de la base, remanions un peu notre nombre. Vous serez sûrement d'accord si je vous dis que 1423 est égal à **1000 + 423**. De même, 423 est égal à **400 + 23**, et 23 est égal à **20 + 3**. Nous pouvons donc réécrire notre nombre comme ceci : **1000 + 400 + 20 + 3**. Allons encore un peu plus loin, et remplaçons :

- 1000 par **1 \* 1000**;
- 400 par **4 \* 100**;
- 20 par **2 \* 10**;
- et 3 par **3 \* 1**.

Notre nombre devient donc : **1 \* 1000 + 4 \* 100 + 2 \* 10 + 3 \* 1**.

Savez-vous ce qu'est l'opération **puissance** ? C'est une opération qui consiste à multiplier un nombre par lui-même un certain nombre de fois. La puissance **p** d'un nombre **n** se note comme ceci : **n<sup>p</sup>**. Plutôt qu'un long discours, voici quelques exemples :

- $n^1 = n$ ;
- $n^2 = n * n$ ;
- $n^3 = n * n * n$ ;
- $n^4 = n * n * n * n$ .

Par convention, un nombre à la puissance zéro vaut toujours un.

- $10^0 = 1$ ;
- $5^0 = 1$ ;
- $n^0 = 1$ .

Revenons à notre nombre. Êtes-vous d'accord si je vous dis que 100 est égal à **10 \* 10** ? Si vous ne l'êtes pas, il est temps de revoir vos tables de multiplications. 😊 Si par contre vous l'êtes, vous serez donc également d'accord si je vous dit que 100 est égal à **10<sup>2</sup>**, puisque nous venons de voir que **10<sup>2</sup>** était égal à **10 \* 10**. De la même façon, 1000 est égal à **100 \* 10 = 10 \* 10 \* 10 = 10<sup>3</sup>**, et 10 est égal à **10<sup>1</sup>**. Nous avons également vu que tout nombre à la puissance zéro est égal à 1, nous pouvons donc dire que 1 est égal à **10<sup>0</sup>**. Remplaçons ces valeurs par leur équivalent en puissance de 10 dans notre nombre ; il devient donc : **1 \* 10<sup>3</sup> + 4 \* 10<sup>2</sup> + 2 \* 10<sup>1</sup> + 3 \* 10<sup>0</sup>**.

Un nombre revient dans chaque multiplication : 10. Ce nombre n'est ni le fruit du hasard, ni anodin : c'est cette fameuse base.

Finalement, remplaçons les chiffres par **c<sub>i</sub>** avec **i** leur position dans le nombre, et la base par la lettre **b** :

$c_3 * b^3 + c_2 * b^2 + c_1 * b^1 + c_0 * b^0$ . En notation positionnelle, un nombre est donc une somme de multiplications particulières : la multiplication du chiffre à la position  $p$  par la base à la puissance  $p$ .

En fonction de la base que vous utilisez pour exprimer un nombre, vous utiliserez une certaine quantité de chiffres. En base 10 (ou décimale) par exemple, nous utilisons... 10 chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. En base 8, ou octale, nous utiliserons.. 8 chiffres : 0, 1, 2, 3, 4, 5, 6 et 7. En général, si la base est  $b$ , nous utiliserons  $b$  chiffres différents.

Bien évidemment, avec une base supérieure à 10, nous aurons besoin d'autres symboles que ces traditionnels 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. En base 16 ou hexadécimale, ces chiffres supplémentaires sont des lettres de l'alphabet ; l'ensemble des chiffres utilisés est : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. En base 64, l'ensemble des chiffres est composé des 26 lettres de l'alphabet en minuscules et en majuscules, ce qui fait 52 chiffres, en plus des douze caractères suivants : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, + et /.

## Des nombres équivalents

Comparer des nombres exprimés dans la même base est immédiat : si un nombre est composé de davantage de chiffres que l'autre et que ces chiffres supplémentaires ne sont pas 0 ou un équivalent, c'est le plus grand des deux. Si les deux nombres sont composés du même nombre de chiffres, il suffit de comparer les chiffres composant les deux nombres, un par un, en partant de l'extrême gauche, et ce jusqu'à ce que des chiffres à la même position soient différents. À ce moment-là, le nombre le plus grand est celui dont le chiffre est le plus grand.

Si la base est différente, cette méthode n'est plus possible. Fort heureusement, nous pouvons convertir nos nombres d'une base à l'autre. En effet, une quantité, c'est une quantité : elle ne va pas changer en fonction de la manière dont nous l'exprimons. Il est donc logique de se dire que si j'ai un nombre  $n$  en base  $p$ , ce que je noterai dorénavant  $n_p$ , alors il doit exister un nombre  $m_q$  qui représente la même quantité.

L'idée est donc de prendre un des deux nombres et de trouver son équivalent dans la base de l'autre nombre. Pour trouver l'équivalent d'un nombre dans une autre base, plusieurs méthodes existent. Pour choisir la méthode à utiliser, la question à se poser est la suivante : dans quelle base est-ce que je sais calculer, c'est-à-dire faire des additions, des soustractions, des multiplications et des divisions ? Si vous savez calculez dans la base de départ, la méthode pour parvenir à obtenir l'équivalent d'un nombre dans une autre base est de faire des divisions entières, les chiffres composant le nombre équivalent constituant le reste de chaque division. Pour comprendre, rien de mieux qu'un exemple : je veux convertir  $42_{10}$  en base 4.

- $42/4 = 10$ , reste 2, c'est  $c_{n-1}$ .
- $10/4 = 2$ , reste 2, c'est  $c_{n-2}$ .
- $2/4 = 0$ , reste 2, c'est  $c_{n-3}$ .

À chaque itération, j'ai bien fait une division entière, et le résultat de cette division était réutilisé dans l'itération suivante. Le nombre est composé par tous les restes,  $42_{10}$  est donc équivalent à  $222_4$ , ce qui se note  $42_{10}222_4$ . Nous pouvons le vérifier :  $222_4 \cdot 2 \cdot 4^2 + 2 \cdot 4^1 + 2 \cdot 4^0 = 2 \cdot 16 + 2 \cdot 4 + 2 \cdot 1 = 32 + 8 + 2 = 42_{10}$

Faisons maintenant l'inverse et trouvons l'équivalent de  $333_4$  en base 10. Cette fois-ci, nous ne savons plus calculer dans la base de départ mais dans la base d'arrivée. Dans pareille situation, nous n'allons plus faire des divisions, mais simplement calculer la forme que nous avons vue précédemment, à savoir

$$c_{n-1} * b^{n-1} + c_{n-2} * b^{n-2} + \dots + c_2 * b^2 c_1 * b^1 + c_0 * b^0, \text{ et donc}$$

$$333_4 \cdot 3 \cdot 4^2 + 3 \cdot 4^1 + 3 \cdot 4^0 = 3 \cdot 16 + 3 \cdot 4 + 3 = 63_{10}$$

Si par le plus grand des malheurs vous ne savez calculer ni dans la base de départ, ni dans la base d'arrivée, il vous faudra passer par une base intermédiaire dans laquelle vous savez calculer. Pour trouver l'équivalent en base 13 de  $133_{16}$ , nous commencerons par trouver l'équivalent en base 10 de  $133_{16}$ , et ensuite nous trouverons l'équivalent du résultat en base 13. Je vous sens bouillonnants d'envie, faites-le donc. 😊

### Secret (cliquez pour afficher)

$$133_{16} \cdot 1 \cdot 16^2 + 3 \cdot 16^1 + 3 \cdot 16^0 = 1 \cdot 256 + 3 \cdot 16 + 3 \cdot 1 = 307_{10}$$

$$307/13 = 23, \text{ reste } 8.$$

$$23/13 = 1, \text{ reste } 10.$$

$$1/13 = 0, \text{ reste } 1.$$

Si nous posons que 10 est équivalent à A, alors  $307_{10}1A8_{13}133_{16}$

## Le binaire : la base 2

En tant que mordus d'informatique, vous avez dû entendre parler à un moment ou à un autre du fameux « binaire ». Ce binaire

n'est en fait qu'une base, la base 2, et permet donc d'exprimer des nombres. Mais pourquoi la base 2 est-elle si importante en informatique ?

Une base définit un ensemble de chiffres, cet ensemble peut être assimilé à un ensemble d'états. Si nous prenons l'exemple d'un ascenseur, il peut être dans quatre états différents :

- en train de monter ;
- en train de descendre ;
- à l'arrêt ;
- ou en panne.

Si nous choisissons de représenter ces différents états par des nombres, une base sera très pratique : la base 4. L'état « monter » correspondrait à  $0_4$ , l'état « descendre » correspondrait à  $1_4$ , l'état « immobile » correspondrait à  $2_4$  et l'état « en panne » correspondrait à  $3_4$ . Avec la base 4, tous les états possibles de l'ascenseur peuvent être exprimés avec un seul chiffre, et tous les chiffres de la base correspondent à un état : c'est donc la base idéale pour exprimer ces différents états.

Les ordinateurs, comme vous le savez sans doute, fonctionnent avec de l'électricité. Avec cette électricité, quels sont les états les plus facilement identifiables ? C'est tout bête : soit le courant passe, soit le courant ne passe pas. Dans un ordinateur, nous trouvons donc deux états, ce qui explique pourquoi la base 2 est si largement utilisée en informatique.

En base binaire, les chiffres sont souvent appelés « bits ». Puisque nous sommes en base 2, ces bits peuvent prendre deux valeurs : 0 et 1.

## Les opérateurs bits à bits

Dans beaucoup de langages, il existe une famille d'opérateurs, les opérateurs dits « bits à bits » ou *bitwise*. La particularité de ces opérateurs est d'effectuer une opération sur chaque chiffre ou bit d'un nombre exprimé en base 2.

Prenons par exemple deux nombres en base 2,  $0110_2$  et  $1101_2$ , et effectuons un **ET bits à bits** de ces deux nombres. Puisque c'est un opérateur bits à bits, nous allons, pour chaque bit du premier nombre, faire un **ET** avec le bit correspondant du second nombre. Commençons par le premier couple de bits, c'est-à-dire les bits à la position 0. Nous devons donc faire un **ET** entre 0 et 1. Si nous posons que 0 est équivalent à false et que 1 est équivalent à true, cela revient à trouver le résultat de l'opération false **&&** true, ce qui vaut false (et donc 0, puisque false est équivalent à 0). Passons aux bits en position 1 : nous avons 1 et 0. Nous devons donc trouver le résultat de true **&&** false, ce qui vaut false, et donc à nouveau 0. Pour les bits en position 2, nous avons 1 et 1 ; nous déterminons que le résultat de true **&&** true est true, et donc 1. Enfin, pour les bits en position 3, nous avons 0 et 1, ce qui nous donne 0.

Sachant que l'espérance représente l'opérateur **ET bits à bits**, nous pouvons donc écrire que  $0110_2 | 1101_2 = 0100_2$ .

Si nous faisons un **OU bits à bits** au lieu d'un **ET bits à bits**, sachant que le *pipe* () représente l'opérateur **OU bits à bits**, nous écririons que  $0110_2 | 1101_2 = 1111_2$ , car dans tous les couples de bits, il y a au moins un des deux bits du couple qui vaut 1.

Dans vos programmes, vous écrivez généralement vos nombres en base 10, mais en interne, tous vos nombres sont stockés en base 2, puisque votre ordinateur, de par son fonctionnement, ne peut utiliser que cette base. C'est pour cela que vous pouvez utiliser ces opérateurs bits à bits indépendamment de la base avec laquelle vous avez exprimé vos nombres. Exemple :

### Code : PHP

```
<?php  
var_dump(6 & 13);
```

### Code : Console

```
int(4)
```

Si nous cherchons les équivalents en base 2 de  $6_{10}$  et  $13_{10}$ , nous trouvons  $0110_2$  et  $1101_2$ . Nous avons vu juste au-dessus que  $0110_2 | 1101_2 = 0100_2$ , ce qui est équivalent à  $4_{10}$ . Nous avons donc bel et bien fait un **ET bits à bits** entre nos deux nombres. 😊

Le PHP propose en tout six opérateurs bits à bits :

- le **ET bits à bits** dont l'opérateur est **&** ;
- le **OU bits à bits** dont l'opérateur est **|** ;
- le **OU EXCLUSIF bits à bits** dont l'opérateur est **^** ;

- le **NON bits à bits** dont l'opérateur est `~` ;
- le **décalage à gauche** dont l'opérateur est `<<` ;
- le **décalage à droite** dont l'opérateur est `>>`.

Pour les trois premiers opérateurs, rien de bien sorcier, vous devriez déjà savoir tout ça puisque vous savez faire un **ET**, un **OU** et un **OU EXCLUSIF** entre deux booléens. Pour le quatrième, vous connaissez également l'opération qu'il effectue : c'est une simple inversion, comme avec l'opérateur `!`. Toutefois, si vous le testez, vous risquez de ne pas comprendre le résultat. Exemple :

#### Code : PHP

```
<?php  
var_dump (~4);
```

#### Code : Console

```
int (-5)
```

Mais que s'est-il passé ? D'où vient ce nombre négatif ? Nous avons fait le **NON bits à bits** de  $4_{10}$ , c'est-à-dire  $100_2$ , cela devrait donc nous donner  $011_2$ , c'est-à-dire  $3_{10}$ .

Souvenez-vous des premiers chapitres de la partie sur le SQL. Lors de la création de notre première table, je vous avais parlé d'une histoire d'entier signé et non signé. Pour rappel, dans le cas d'un entier signé, le bit à l'extrême gauche indiquait le signe. Si nous obtenons ce  $-5$ , c'est parce qu'en PHP, les nombres sont des nombres signés. En faisant un **NON bits à bits**, nous allons inverser le bit de signe, ce qui explique pourquoi nous obtenons un nombre négatif. Pour aller plus loin, sachez qu'en PHP, les nombres sont exprimés avec 32 chiffres en base 2, la plage de valeur des nombres entiers en PHP s'étend donc de  $00000000000000000000000000000000_2$  à  $11111111111111111111111111111111_2$ . Lorsque nous écrivons  $4_{10}$  dans notre programme, en interne, c'est donc  $000000000000000000000000000000100_2$  qui est stocké. Lorsque nous faisons un **NON bits à bits**, tous les 0 deviennent des 1, et inversement ; nous obtenons donc  $11111111111111111111111111111111011_2$ , ce qui est équivalent à  $-5_{10}$ . Nous pouvons le vérifier très simplement avec une petite fonction que nous offre le PHP et qui permet d'obtenir la représentation en base 2 d'un nombre : `decbin()`.

#### Code : PHP

```
<?php  
var_dump(decbin(4));  
var_dump(decbin(-5));
```

#### Code : Console

```
string(3) "100" string(32) "11111111111111111111111111011"
```

Pour les deux derniers opérateurs, c'est nouveau, mais pas très compliqué. Imaginons un nombre en base 2,  $1101_2$  par exemple, et incrémentons la position de chaque bit : le bit 0 devient le bit 1, le bit 1 devient le bit 2, le bit 2 devient le 3 et le bit 3 devient le bit 4. À la position 0, un nouveau bit de valeur 0 est ajouté. Après cela, nous avons donc le nombre  $11010_2$ . Ce que nous venons de faire, c'est décaler tous les bits vers la gauche une fois. Exemple en PHP :

#### Code : PHP

```
<?php  
var_dump(decbin(13));  
var_dump(decbin(13 << 1));
```

**Code : Console**

```
string(4) "1101" string(5) "11010"
```

Si nous avions fait un décalage de 2 et non de 1, le bit 0 serait devenu le bit 2, le bit 1 serait devenu le bit 3, le bit 2 serait devenu le bit 4 et le bit 3 serait devenu le bit 5. Aux positions 0 et 1, des bits de valeur 0 auraient été injectés.

Il est à noter que les bits à gauche peuvent se perdre dans la nature, exemple :

**Code : PHP**

```
<?php  
var_dump(decbin(2 << 31));
```

Puisque je décale le nombre  $10_2$  de 31 vers la gauche, je devrais obtenir  $10000000000000000000000000000000_2$ . Malheureusement, comme je l'ai dit, en PHP, les nombres sont représentés sur 32 bits ; or, ce chiffre en comporte 33. Le bit à l'extrême gauche est donc purement et simplement tronqué et par conséquent perdu, ce qui explique pourquoi nous obtenons  $0_2$ . Le décalage à droite est identique au décalage à gauche, mis à part le fait que le décalage se fait... vers la droite. De la même façon qu'avec le décalage à gauche, si nous faisons un décalage trop important vers la droite, les bits de droite seront tronqués et donc perdus.

## Les flags

Imaginons que vous ayez implémenté une fonction qui peut activer ou désactiver de nombreuses options, par exemple comme ceci :

**Code : PHP**

```
<?php  
function f($isDef, $isAllowed, $isOn, $isRunning)  
{  
}
```

Chacune de ces options peut être soit activée, soit désactivée. Tiens, deux états, ça ne vous rappelle rien ? Eh oui, du binaire. Chaque option peut être représentée par un 1 si elle est activée et un 0 si elle est désactivée ; l'ensemble des options peut donc être paramétré avec quatre bits. Nous pourrons donc passer l'état de ces quatre options avec un seul nombre, puisque ceux-ci sont exprimés avec 32 bits en PHP. Par exemple, le cas où toutes les options sont désactivées est représenté par  $0000_2$ , tandis que le cas où les options isAllowed et isRunning sont activées est représenté par  $0101_2$ .

Passons maintenant à la pratique. Pour commencer, nous allons définir quatre constantes qui représenteront nos options :

**Code : PHP**

```
<?php  
define('RUNNING', 1); // 0001 en base 2  
define('ON', 2); // 0010 en base 2  
define('ALLOWED', 4); // 0100 en base 2  
define('DEF', 8); // 1000 en base 2
```

Ces valeurs n'ont bien sûr pas été choisies complètement au hasard : il faut que chaque option définisse un seul bit à 1 et tous les autres à zéro, et que la position du bit défini à 1 soit différente pour chaque option. En choisissant ces valeurs particulières pour les constantes, la combinaison des options devient un jeu d'enfant. Vous voulez que isAllowed et isRunning soient activées ? Faites un **OU bits à bits** entre les deux. En effet,  $0100_2 | 0001_2 = 0101_2$ . Vous voulez activer l'option isDef ? Faites un **OU bits à bits** entre les options actuelles et la constante DEF. Exemple :

**Code : PHP**

```
<?php

define('RUNNING', 1); // 0001 en base 2
define('ON', 2); // 0010 en base 2
define('ALLOWED', 4); // 0100 en base 2
define('DEF', 8); // 1000 en base 2

$options = 0; // toutes les options désactivées
$options = RUNNING | ALLOWED; // isRunning et isAllowed activées
$options = $options | DEF; // activons l'option isDef
```

Il est à noter que de façon analogue aux opérateurs arithmétiques, il existe des opérateurs bits à bits contractés :

#### Code : PHP

```
<?php

$options = $options | DEF;
// équivalent à :
$options |= DEF;
// &=, ^=, <<= et >>= existent également
```

Pour être sûrs qu'une option est activée, nous utiliserons donc l'opérateur **OU bits à bits**.

#### *Connaître l'état d'une option*

Pour cela, rien de plus simple, il suffit de faire un **ET bits à bits** entre les options actuelles et l'option à tester. En effet, comme dans la constante représentant l'option à tester, tous les bits mis à part celui qui nous intéressent sont à zéro, soit l'option est désactivée et le résultat vaudra 0, soit l'option est activée et le résultat sera différent de 0.

#### Code : PHP

```
<?php

define('RUNNING', 1); // 0001 en base 2
define('ON', 2); // 0010 en base 2
define('ALLOWED', 4); // 0100 en base 2
define('DEF', 8); // 1000 en base 2

$options = RUNNING | ALLOWED | DEF;
var_dump((bool)($options & RUNNING)); // bool(true)
var_dump((bool)($options & ON)); // bool(false)
```

#### *Désactiver une option*

Pour désactiver une option, il va falloir combiner deux opérateurs bits à bits : `~` et `&`. Actuellement, les options sont représentées par **1101<sub>2</sub>** et nous souhaitons désactiver l'option isAllowed. Pour ce faire, nous allons commencer par utiliser le **NON bits à bits** sur la constante ON :  $\sim 0100_2 = 1011_2$ . Ensuite, il reste à faire un **ET bits à bits** entre les options actuelles et ce résultat : **1101<sub>2</sub>1011<sub>2</sub> = 1001<sub>2</sub>**.

#### Code : PHP

```
<?php

define('RUNNING', 1); // 0001 en base 2
define('ON', 2); // 0010 en base 2
define('ALLOWED', 4); // 0100 en base 2
define('DEF', 8); // 1000 en base 2

$options = RUNNING | ALLOWED | DEF;
```

```
var_dump(decbin($options));
$options &= ~ALLOWED;
var_dump(decbin($options));
```

Cette méthode peut aussi être utilisée pour d'autres choses, par exemple pour gérer des droits. Si le bit correspondant au droit est à 1, le droit est accordé, sinon il est refusé. En fait, cela est utilisable pour toute situation où il n'y a que deux états possibles : activé ou désactivé, autorisé ou refusé, allumé ou éteint, etc.

Le PHP utilise d'ailleurs cette méthode dans certaines de ces fonctions, `set_error_handler()` par exemple (que nous verrons bientôt dans le détail).

## Des bases plus pratiques

Dans nos programmes, nous écrivons bien souvent nos nombres en base 10. Or, ce n'est pas une très bonne base pour écrire un nombre tel que **1110110101<sub>2</sub>**, car cela nous oblige à faire de nombreux calculs pour trouver l'équivalent en base 10.

Heureusement pour nous, le PHP, comme beaucoup d'autres langages, permet d'exprimer des nombres dans deux bases supplémentaires : la base 8 ou octale, et la base 16 ou hexadécimale. Testons :

### Code : PHP

```
<?php

var_dump(42);
var_dump(0x42);
var_dump(042);
```

### Code : Console

```
int(42)
int(66)
int(34)
```

Lorsque le préfixe **0x** est ajouté à un nombre, le PHP considère que le nombre est écrit en base 16. Cela est facilement vérifiable : **42<sub>16</sub> \* 16 + 2 = 64 + 2 = 66**. Si le nombre est préfixé par un **0**, le PHP considère cette fois que le nombre est écrit en base 8. Vous pouvez faire la vérification, **42<sub>8</sub>** est bien équivalent à **34<sub>10</sub>**.

Si une écriture native pour ces deux bases a été implémentée, c'est parce qu'elles ont une caractéristique intéressante : 16 et 8 sont respectivement égaux à **2<sup>4</sup>** et **2<sup>3</sup>**. Or, il se trouve que lorsque nous voulons passer d'une base **b** à une base **b<sup>p</sup>**, il n'est pas nécessaire d'utiliser les méthodes précédentes, il suffit de remplacer des groupes de **p** chiffres par le chiffre équivalent dans la base d'arrivée. Prenons un exemple : le passage de la base 2 à la base 8. 8 est égal à 2 à la puissance 3, nous pouvons donc passer d'un nombre en base 2 à un nombre en base 8 en groupant les chiffres trois par trois. Prenons le nombre **111010<sub>2</sub>**, il est composé de deux groupes de trois chiffres : 111 et 010. **111<sub>2</sub>** est équivalent à **7<sub>8</sub>** et **010<sub>2</sub>** est équivalent à **2<sub>8</sub>**, et donc **111010<sub>2</sub> 72<sub>8</sub>**. Vous pouvez vérifier, c'est correct. Personnellement, je préfère devoir retenir et manipuler le nombre 72 que le nombre 111010. 😊 Dans certains cas, l'intérêt est encore plus flagrant. Les adresses IP version 6, par exemple, sont des nombres nécessitant 128 chiffres en base 2 ou 32 chiffres en base 16. Il est évident que retenir et manipuler 32 chiffres est bien plus simple que d'en retenir et d'en manipuler 128.

Pour reprendre l'exemple des flags, imaginons que nous ayons 30 options ; les deux dernières seront donc **00010000000000000000000000000000<sub>2</sub>** et **00100000000000000000000000000000<sub>2</sub>**. Pour trouver l'équivalent en base 10 en un instant, à part en connaissant ces valeurs par cœur, vous aurez bien du mal. L'équivalent en base 16 est bien plus facile à trouver. En effet, dans le cas du premier nombre, nous avons un premier groupe de 4 bits qui est 0001. Le nombre équivalent en base 16 est 1. Ensuite, nous avons 7 groupes de 4 bits qui sont tous 0000, c'est-à-dire 0 en base 16. L'équivalent en base 16 de ce nombre est donc 10000000. Pour le second nombre, seul le premier groupe de 4 bits change : l'équivalent de **0010<sub>2</sub>** en base 16 est 2, l'équivalent en base 16 de ce nombre est donc 20000000.

Si nous voulons faire l'inverse, c'est-à-dire passer d'une base **b<sup>p</sup>** à une base **b**, il suffit de remplacer un chiffre par le groupe de **p** chiffres correspondant dans la base d'arrivée. Soit le nombre **45FE<sub>16</sub>**, pour trouver l'équivalent en base 2, nous allons remplacer chaque chiffre par le groupe de quatre bits équivalent :

- **4<sub>16</sub> 0100<sub>2</sub>**;

- $5_{16}0101_2$ ;
- $F_{16}15_{10}1111_2$ ;
- $E_{16}14_{10}1110_2$ .

Et donc, finalement :  **$45FE_{16}01000101111110_2$** . Pour éviter de devoir repasser par la base 10, il est assez utile de connaître ce petit tableau qui montre les équivalences en base 2, 8 et 16 des 16 premiers nombres exprimés en base 10.

	Base 10	Base 2	Base 8	Base 16
00	0000	0	0	0
01	0001	1	1	1
02	0010	2	2	2
03	0011	3	3	3
04	0100	4	4	4
05	0101	5	5	5
06	0110	6	6	6
07	0111	7	7	7
08	1000		8	8
09	1001		9	9
10	1010		A	A
11	1011		B	B
12	1100		C	C
13	1101		D	D
14	1110		E	E
15	1111		F	F

## Fonctions de changement de base

En plus de la possibilité d'écrire nativement des nombres en base 8 et 16, le PHP nous propose quelques fonctions permettant de passer d'une base à une autre. Nous en avons déjà vu une, `decbin()` qui permet de passer de la base 10 à la base 2. Il existe son opposé, `bindec()`, qui prend en paramètre une chaîne de caractères représentant le nombre en base 2 et qui retourne le nombre équivalent en base 10. Attention, contrairement à `decbin()`, `bindec()` retourne bien un entier et non une chaîne de caractères.

### Code : PHP

```
<?php
var_dump(bindec('11110'));
var_dump(bindec(decbin(-1)));
```

Une chose étrange survient : la seconde ligne nous affiche **float(4294967295)** et non **int(-45)**, comme nous pourrions nous y attendre ; pourquoi ? La raison est simple : `bindec()` ne tient pas compte du bit de signe, il travaille sur des nombres non signés. En effet, **-1<sub>10</sub>** est équivalent à **11111111111111111111111111111111<sub>2</sub>** ; si nous considérons qu'il n'y a pas de bits de signe, ce nombre est équivalent à **4294967295<sub>10</sub>** ; `bindec()` travaille donc bien avec des nombres non signés.

En plus de `bindec()` et `decbin()`, deux paires de fonctions similaires existent :

- `decoct()` et `octdec()` ;
- `dechex()` et `hexdec()`.

Ces fonctions permettent respectivement de passer des bases 8 et 16 à la base 10 et inversement ; de plus, elles fonctionnent exactement de la même façon que bindec() et decbin().

Si vous voulez travailler avec des bases plus exotiques, n'ayez crainte, vous n'aurez pas à créer votre propre fonction de changement de base, base\_convert() est là pour vous. Cette fonction permet de passer d'une base **P** à une base **Q** à la seule condition que **P** et **Q** soient compris entre 2 et 36 inclus. Si vous demandez pourquoi la limite est 36, comptez le nombre de chiffres en base 10, ajoutez-y le nombre des lettres de l'alphabet, et vous devriez comprendre. 😊 En effet, pour représenter les nombres, base\_convert() utilise l'ensemble des chiffres de la base 10 et les 26 lettres de l'alphabet. Il est à noter que cette fonction reçoit et retourne les nombres uniquement sous forme de chaîne de caractères, et ce même si nous utilisons la base 8, 10 ou 16.

#### Code : PHP

```
<?php  
  
// vérifions que 133 en base 16 est équivalent à 1A8 en base 13  
var_dump(base_convert('133', 16, 13));
```

#### Code : Console

```
string(3) "1a8"
```

Nous avons vu que dès que vous utilisez des données provenant d'un utilisateur, il fallait s'assurer que ces données soient valides, c'est-à-dire qu'elles soient du type attendu et éventuellement dans une plage de valeurs acceptées. Il vous sera peut-être nécessaire un jour de permettre aux utilisateurs de saisir des nombres dans une base autre que la base 10. Quand nous considérons que nous travaillons uniquement en base 10, nous avons vu que pour garantir que l'utilisateur nous donnait bel et bien un nombre, nous pouvions transtyper la valeur reçue avec (int). Or, il s'avère que ce transtypage ne fonctionne que pour la base 10. Si vous souvenez-vous du chapitre où nous avons parlé du transtypage, vous vous souvenez peut-être d'intval(). Cette fonction permet en effet d'obtenir un nombre en base 10 à partir de n'importe quoi, et je vous avais dit à l'époque que je préférerais utiliser le transtypage pour soulager mon clavier. Toutefois, intval() possède un avantage énorme sur le transtypage : elle peut tenir compte de la base. Exemple :

#### Code : PHP

```
<?php  
  
var_dump(intval('42')); // par défaut en base 10  
// nous allons dire à intval() que le paramètre est exprimé en base  
8  
var_dump(intval('42', 8));  
// la fonction nous a bien retourné 34, ce qui est l'équivalent en  
base 10 de 42 en base 8
```

Il y a bien d'autres choses dont nous pourrions discuter à propos des nombres, mais cela suffira pour ce que nous voulons faire. Retenez bien cette technique des flags, elle nous sera rapidement utile et bon nombre de programmes l'utilisent d'une façon ou d'une autre. 😊

## PHP et les erreurs

Les erreurs, voilà bien des éléments qui peuvent nous pourrir la vie. 😱

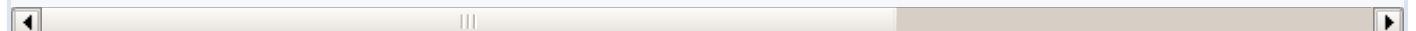
Pourtant, ces erreurs ne sont pas inutiles, et en assurer une gestion correcte est crucial tout au long de la vie d'une application. Ce sera donc l'objet de chapitre.

### Les erreurs

Bien souvent, nous avons été confrontés à de charmants messages de la part du PHP. En voici quelques exemples :

#### Code : Console

```
Warning: Division by zero in C:\wamp\www\TutoOO\index.php on line 3
Notice: Undefined variable: s in C:\wamp\www\TutoOO\index.php on line 4
Warning: fopen(lala) [function.fopen]: failed to open stream: No such file or direc
Fatal error: Call to undefined function f() in C:\wamp\www\TutoOO\index.php on line
Parse error: parse error, expecting ''('' in C:\wamp\www\TutoOO\index.php on line 3
```



Vous ne devez pas percevoir ces erreurs comme un ennui, une gêne ou autre, mais comme un **outil**. Les erreurs vous permettent de détecter des problèmes dans votre application, elles vous sont donc très utiles pour améliorer vos codes et les rendre plus robustes. Prenons un exemple tout simple, nous créons une fonction pour diviser deux nombres :

#### Code : PHP

```
<?php
function divide($a, $b)
{
    return $a/$b;
}
```

Si un petit malin s'amuse à donner la valeur 0 en deuxième paramètre, cette fonction retournera false, car une division par 0 n'est pas possible. Si la suite du script se base sur ce résultat, de gros bogues peuvent s'entasser, et même rendre votre application inutilisable.

De la même façon, si vous tentez d'utiliser une fonction qui n'existe pas, quelles conséquences cela aura-t-il sur votre application ? Qu'est-ce que le PHP fera si votre script n'est pas correct au niveau syntaxique ?

Il est donc vital d'avoir un mécanisme qui va nous prévenir quand ces problèmes surviennent : les erreurs. En PHP, il y a deux phases lors de l'exécution d'un script, si je puis dire :

- l'analyse et la traduction de votre code ;
- l'exécution de votre code.

Les erreurs les plus courantes lors de la première phase sont généralement dues à une erreur de syntaxe. Exemple :

#### Code : PHP

```
<?php
$
```

#### Code : Console

```
Parse error: parse error, expecting 'T_VARIABLE' or ''$'' in C:\wamp\www\TutoOO\ind
```

La syntaxe du PHP est clairement définie : nous savons par exemple qu'après un signe \$, nous pouvons trouver ceci :

- une suite de caractères alphanumériques pouvant contenir des *underscores* ;

- une variable, dans le cas des hideuses variables variables.

Dans le code précédent, nous trouvons un signe \$, mais il est suivi par... un point-virgule. Cette syntaxe n'est pas valide, le PHP nous le signale donc en émettant une erreur de type E\_PARSE, E\_PARSE étant l'une des constantes qui indiquent une catégorie d'erreur. Les erreurs de type E\_PARSE terminent également le script.

Dans la plupart des cas, les messages d'erreur de ce type indiquent ce que le PHP attendait ou n'attendait pas comme syntaxe. Dans l'exemple, l'erreur nous dit que le PHP attendait (*expecting*) un « T\_VARIABLE » ou un « \$ ». Mais qu'est-ce que T\_VARIABLE ?

Lorsque le PHP va analyser votre script, il va commencer par le découper sous forme de *token*. Ainsi, l'opérateur && est par exemple représenté par le *token* T\_BOOLEAN\_AND. Ainsi, si nous avons ce script :

#### Code : PHP

```
<?php
$a && $b;
?>
```

... l'équivalent en *token* de la troisième ligne est ceci :

#### Code : Console

```
T_VARIABLE T_BOOLEAN_AND T_VARIABLE
```

Nous pouvons le vérifier :

#### Code : PHP

```
<?php
var_dump(token_get_all('<?php $a&&$b;'));
echo token_name(308) . ' ' . token_name(278);
```

La fonction token\_get\_all() nous donne la représentation sous forme de *token* de la chaîne qui lui est donnée, et la fonction token\_name() permet de traduire le code du *token* en un nom un peu plus parlant. Certains symboles, le point-virgule et l'opérateur d'affectation par exemple, ne possèdent pas de *token*, ils sont tels quels. La liste de tous les *tokens* utilisés se trouve dans [la documentation](#). En découpant de la sorte votre script, le PHP peut aisément vérifier la syntaxe et vous avertir s'il trouve une erreur de syntaxe, de sorte que vous puissiez la corriger.

Lors de la phase d'exécution, des erreurs peuvent également être générées : ce sont des erreurs dites de *runtime*. Par exemple, utiliser une variable ou une fonction non déclarée émettra ce genre d'erreur.

Voici les niveaux d'erreur que vous serez souvent amenés à rencontrer :

Type d'erreur	Signification
E_ERROR alias Fatal error	Erreur <i>runtime</i> qui provoquera l'arrêt du script : l'appel à une fonction qui n'existe pas, par exemple
E_WARNING alias Warning	Erreur <i>runtime</i> qui ne provoque pas l'arrêt du script, une division par zéro par exemple
E_NOTICE alias Notice	Erreur <i>runtime</i> qui ne provoque pas l'arrêt du script, l'utilisation d'une variable qui n'est pas déclarée par exemple
E_PARSE alias Parse error	Erreur d'analyse qui provoque l'arrêt du script.

La différence entre les types E\_NOTICE et E\_WARNING est assez « ridicule » à mes yeux. D'après la documentation du PHP, un type E\_NOTICE indique quelque chose qui peut être une erreur mais qui peut également être normal. Sérieusement, comment peut-on considérer qu'utiliser une variable non déclarée est un évènement normal dans la vie d'un script ?

Il en existe d'autres, la liste complète se trouve dans [la documentation](#), mais celles-ci sont les plus courantes. Les types E\_USER\_ERROR, E\_USER\_WARNING et E\_USER\_NOTICE sont pratiquement équivalents à E\_ERROR, E\_WARNING et E\_NOTICE, la seule différence étant que les erreurs E\_USER\_\* sont générées par vous-mêmes *via* une fonction prévue à cet effet : trigger\_error().

Dorénavant, la première chose à faire quand vous rencontrerez une erreur, c'est de la **lire**. Quand quelqu'un vient demander sur un forum pourquoi son script plante et ce que signifie telle erreur, il devient vite lassant de répéter mille fois la même chose.

Après avoir lu l'erreur, il convient donc de la **comprendre**. D'abord, vous **déterminez le type d'erreur** et ensuite vous **regardez précisément** ce qu'elle vous veut. Si tout le monde savait lire une erreur, presque aucun sujet ne demanderait ce que signifie l'une ou l'autre.

## Gérer les erreurs

Lorsqu'une erreur est émise, elle est interceptée par un gestionnaire d'erreurs. Nativement, ce gestionnaire d'erreurs ne fait pas grand-chose si ce n'est afficher les erreurs qu'il intercepte, comme nous l'avons vu si souvent.

La première chose, extrêmement importante à signaler, c'est que ce n'est pas parce que le gestionnaire d'erreurs n'affiche pas une erreur qu'elle n'a pas été émise. La raison est simple : ce gestionnaire d'erreurs est configurable. Vous pouvez par exemple décider quels types d'erreurs ce gestionnaire doit intercepter, ou si celui-ci doit les afficher ou non. Pour définir les types d'erreurs que le gestionnaire doit intercepter, nous utiliserons une fonction que vous avez déjà aperçue plusieurs fois tout au long de la partie I : error\_reporting().

Cette fonction prend un paramètre : une combinaison de constantes. Souvenez-vous, quand nous avons vu les différentes bases et les opérateurs bits à bits, nous avons parlé de flags, c'est-à-dire la combinaison de plusieurs nombres avec les opérateurs bits à bits. error\_reporting() utilise cette méthode des flags. Ainsi, si nous désirons que les erreurs de type E\_WARNING et E\_NOTICE soient interceptées, nous utiliserons la combinaison E\_WARNING | E\_NOTICE. Tous les autres types d'erreurs seront ignorés par le gestionnaire d'erreurs ; toutefois, **cela n'empêche pas le script de se terminer** si l'erreur émise provoque la fin du script. Testons :

### Code : PHP

```
<?php  
  
error_reporting(0); // n'intercepte aucune erreur  
  
1/0; // E_WARNING, pas intercepté, rien ne s'affiche  
  
error_reporting(E_NOTICE); // intercepte les E_NOTICE  
  
1/0; // E_WARNING, pas intercepté, rien ne s'affiche  
  
error_reporting(E_WARNING | E_NOTICE); // intercepte les E_NOTICE et  
// E_WARNING  
  
1/0; // E_WARNING, intercepté
```

Si vous désirez intercepter absolument toutes les erreurs, la combinaison à utiliser est E\_ALL | E\_STRICT. Je vous recommande très vivement d'intercepter absolument toutes les erreurs.

Pour définir si les erreurs interceptées doivent être affichées ou non, nous utiliserons la fonction ini\_set() qui permet de définir la configuration du PHP sans pour autant modifier le fichier de configuration du PHP, ce qui s'avère pratique si votre hébergeur ne vous permet pas de le modifier. Le fichier de configuration du PHP, **php.ini**, est un fichier assez simple qui associe des clés à des valeurs, exactement comme un tableau. La fonction ini\_set() prendra donc deux arguments : la clé à modifier et la valeur à injecter. La clé qui définit si les erreurs interceptées doivent être affichées est **display\_errors**. Les trois valeurs possibles sont celles-ci :

- **On** pour afficher les erreurs ;
- **Off** pour ne pas afficher les erreurs ;
- **stderr** pour envoyer les erreurs sur un flux particulier dédié aux erreurs.

Nous n'utiliserons que les valeurs **On** et **Off** :

### Code : PHP

```
<?php  
  
error_reporting(E_ALL | E_STRICT);  
  
ini_set('display_errors', 'Off');  
1/0; // n'affiche rien  
ini_set('display_errors', 'On');  
1/0; // affiche l'erreur
```



Pour vérifier si `ini_set()` a pu changer la valeur de la configuration, vérifiez si elle ne retourne pas false avec l'opérateur strict de comparaison.

Tout au long de la vie d'un projet, celui-ci peut être dans deux états :

- en développement ;
- ou en production.

Pendant le développement, seuls les développeurs et les testeurs seront amenés à visiter le site pour le tester, repérer des bogues et autres. Il faut donc évidemment que toutes les erreurs soient reportées **et** qu'elles soient affichées pour que les rapports de bogues et les corrections puissent être rapidement faits.

En production, monsieur-tout-le-monde peut accéder au site. Il est évident que les visiteurs n'ont aucune utilité des erreurs, il faudra donc les leur cacher. Par ailleurs, certaines erreurs peuvent donner des informations sur votre site, des mots de passe par exemple, ce qui peut conduire à de graves problèmes de sécurité. Il est donc plus que nécessaire de cacher tout cela aux yeux de tous.

Toutefois, bien qu'en production, votre application peut générer des erreurs pour l'une ou l'autre raison. Or, si les visiteurs n'ont que faire de vos erreurs, les développeurs, eux, en ont besoin. Il faut donc pouvoir sauvegarder ces erreurs quelque part, dans un fichier par exemple, sans pour autant qu'elles soient visibles par les visiteurs. Pour se faire, le PHP propose deux options de configuration :

- `log_errors`, qui indique si oui (**On**) ou non (**Off**) les erreurs doivent être sauvegardées, *logguées* dans un fichier ;
- `error_log`, qui indique le chemin du fichier dans lequel seront sauvegardées les erreurs.

Testons :

#### Code : PHP

```
<?php  
  
error_reporting(E_ALL | E_STRICT);  
  
ini_set('display_errors', 'Off'); // en production, nous n'affichons  
pas les erreurs  
ini_set('error_log', 'test.log');  
ini_set('log_errors', 'Off');  
var_dump($f); // E_NOTICE, non logguée  
ini_set('log_errors', 'On');  
1/0; // E_WARNING, logguée
```

Allez jeter un œil dans le dossier où se trouve votre script, un fichier **test.log** doit maintenant s'y trouver et contenir l'erreur.

Si vous générez une erreur avec `fopen()` par exemple, le PHP va inclure dans l'erreur un lien HTML vers la page de documentation associée à `fopen()`. Du HTML dans des fichiers de *log*, c'est inutile et ça rend la lecture des *logs* bien plus difficile. Le PHP propose donc d'activer ou non cet ajout d'HTML *via* l'option de configuration `html_errors`. Exemple :

#### Code : PHP

```
<?php  
  
error_reporting(E_ALL | E_STRICT);  
  
ini_set('error_log', 'test.log');
```

```
ini_set('log_errors', 'On');
ini_set('display_errors', 'Off');

ini_set('html_errors', 'On');
fopen('lala', 'r'); // code HTML activé
ini_set('html_errors', 'Off');
fopen('lala', 'r'); // code HTML désactivé
```

Il peut arriver que dans un script, vous ayez besoin de connaître la dernière erreur émise. Vous pouvez obtenir les informations sur cette éventuelle erreur grâce à la fonction `error_get_last()` qui vous retourne un tableau contenant les si précieuses informations, ou bien grâce à une variable un peu particulière : `$php_errormsg`. Cette variable sera accessible uniquement là où l'erreur a été émise si l'option de configuration correspondante, `track_errors`, est activée. Exemple :

#### Code : PHP

```
<?php

error_reporting(E_ALL | E_STRICT);

ini_set('display_errors', 'Off');
ini_set('track_errors', 'On');

1/0;
echo $php_errormsg;
```

En résumé, les choses les plus importantes à retenir sont les suivantes :

- vous devez toujours intercepter tous les types d'erreurs ;
- en développement, vous devez afficher les erreurs ;
- en production, vous ne devez jamais afficher les erreurs, mais les *logguer*.

Je n'ai présenté que les options de configuration les plus intéressantes à mon sens, mais il en existe quelques autres, vous les trouverez dans [la documentation](#).

### Affiner la gestion

Les options de configuration nous permettent de changer le comportement par défaut du PHP lorsqu'une erreur est émise, mais tout n'est pas possible. Il peut arriver que nous souhaitions gérer les erreurs de telle sorte que le gestionnaire natif du PHP ne nous suffise pas. Par chance, il est possible de remédier à cela en définissant nous-mêmes un gestionnaire d'erreurs. L'idée est la suivante : entre les erreurs et l'interception par le gestionnaire natif du PHP, nous allons venir intercaler notre gestionnaire d'erreurs. Ainsi, quand une erreur est émise, soit notre gestionnaire d'erreurs prend en charge le type d'erreur et la traite, soit il ne la prend pas en charge et c'est le gestionnaire natif qui s'en chargera.

Un gestionnaire d'erreurs personnalisé est en fait une fonction que nous définissons comme étant un gestionnaire d'erreurs personnalisé *via* la fonction `set_error_handler()`. Cette fonction prend deux paramètres :

- la fonction qui représente notre gestionnaire d'erreurs ;
- une combinaison de constantes pour définir quels types d'erreur notre gestionnaire interceptera.

La valeur par défaut de ce second paramètre est `E_ALL | E_STRICT`. Par conséquent, par défaut, notre gestionnaire interceptera toutes les erreurs. Toutes ? Non. En effet, un gestionnaire d'erreurs personnalisé ne peut intercepter les types d'erreur suivants :

- `E_ERROR` ;
- `E_PARSE` ;
- `E_CORE_ERROR` ;
- `E_CORE_WARNING` ;
- `E_COMPILE_ERROR` ;
- `E_COMPILE_WARNING` ;
- et la plupart des erreurs `E_STRICT`.

En pratique, ce n'est pas vraiment un problème, puisque ces erreurs devraient soit ne pas survenir en production, soit nécessiter un traitement tout particulier. Il est toutefois possible d'appeler une fonction personnalisée en cas d'erreur de type `E_ERROR`, nous verrons cela par la suite.

Cette fonction si magnifique prendra quatre ou cinq paramètres que voici :

- le niveau de l'erreur ;
- le message d'erreur ;
- le fichier dans lequel l'erreur a été émise ;
- la ligne à laquelle l'erreur a été émise ;
- un tableau reprenant toutes les variables accessibles depuis l'endroit où l'erreur a été émise.

**Code : PHP**

```
<?php

set_error_handler(function($type, $msg, $file, $line, $context =
array()) {
    echo 'une erreur s\'est produite : (' . $type . ') ' . $msg .
"\n"
    . 'dans le fichier ' . $file . ' à la ligne ' . $line .
"\n"
    . 'Contexte : ' . print_r($context, true);
});

$s = 'lala';
1/0;
```

Si nous changeons le filtre d'erreurs pour n'attraper que les erreurs E\_NOTICE, nous reverrons le gestionnaire d'erreurs natif :

**Code : PHP**

```
<?php

set_error_handler(function($type, $msg, $file, $line, $context =
array()) {
    echo 'une erreur s\'est produite : (' . $type . ') ' . $msg .
"\n"
    . 'dans le fichier ' . $file . ' à la ligne ' . $line .
"\n"
    . 'Contexte : ' . print_r($context, true);
}, E_NOTICE);

$s = 'lala';
1/0;
```

Une chose importante à signaler : toutes les options de configuration que nous avons vues précédemment, à l'exception de **track\_errors**, n'ont aucun effet sur notre gestionnaire d'erreurs. Peu importe la valeur de l'`error_reporting` par exemple, toute erreur dont le type est géré par notre gestionnaire sera interceptée par celui-ci.

**Code : PHP**

```
<?php

set_error_handler(function($type, $msg, $file, $line, $context =
array()) {
    echo 'une erreur s\'est produite : (' . $type . ') ' . $msg .
"\n"
    . 'dans le fichier ' . $file . ' à la ligne ' . $line .
"\n"
    . 'Contexte : ' . print_r($context, true);
});

error_reporting(0);

$s = 'lala';
1/0;
```

Quant à **track\_errors**, si vous désirez que la variable \$php\_errormsg soit remplie, votre gestionnaire d'erreurs **doit** renvoyer false.

Par contre, petit bémol : si votre gestionnaire d'erreurs retourne false, le gestionnaire d'erreurs natif sera appelé. Les deux gestionnaires d'erreurs s'appliqueront donc l'un à la suite de l'autre.

Si vous désirez supprimer votre gestionnaire d'erreurs personnalisé, rien de plus simple : il suffit d'appeler la fonction `restore_error_handler()`. Lorsque vous appelez cette fonction, deux cas sont possibles :

- si vous n'avez utilisé qu'une fois `set_error_handler`, votre gestionnaire d'erreurs est supprimé et il ne reste que le gestionnaire d'erreurs natif ;
- si vous avez appelé `set_error_handler(A)` suivi de `set_error_handler(B)`, le gestionnaire d'erreurs B est supprimé et le gestionnaire d'erreurs A redévient le gestionnaire d'erreurs personnalisé.

## *trigger\_error()*

La fonction `trigger_error()` permet, comme nous l'avons dit précédemment, d'émettre nous-mêmes une erreur. Cette fonction prend deux paramètres :

- le message d'erreur ;
- le type d'erreur.

Vous ne pouvez émettre que des erreurs de type E\_USER\_NOTICE, E\_USER\_WARNING et E\_USER\_ERROR ; et par défaut, c'est une erreur de type E\_USER\_NOTICE qui est émise. Exemple :

### Code : PHP

```
<?php  
trigger_error('pouet', E_USER_WARNING);
```

## *debug\_backtrace()*

Lorsqu'une erreur est émise, il peut être intéressant pour le développeur de connaître la pile d'appel des fonctions. La pile d'appel des fonctions représente l'imbrication des fonctions appelées. Exemple :

### Code : PHP

```
<?php  
  
function a()  
{  
    b();  
}  
  
function b()  
{  
    print_r(debug_backtrace());  
}  
  
a();
```

La pile permet d'établir précisément le chemin du code parcouru qui a abouti à l'erreur. Pour chaque élément de la pile, vous avez :

- le nom du fichier où l'élément a été appelé ;
- la ligne à laquelle l'élément a été appelé ;
- le nom de la fonction si l'élément est une fonction ;
- les arguments passés à l'élément.

Il est à noter que les include apparaissent dans la pile d'appel. Nous pouvons donc maintenant améliorer notre gestionnaire d'erreurs :

**Code : PHP**

```
<?php

set_error_handler(function($type, $msg, $file, $line, $context =
array()) {
    echo 'une erreur s\'est produite : (' . $type . ') ' . $msg .
"\n"
    . 'dans le fichier ' . $file . ' à la ligne ' . $line .
"\n"
    . 'Contexte : ' . print_r($context, true);
    debug_print_backtrace();
});

function a()
{
    b(0);
}

function b($arg)
{
    trigger_error('lala', E_USER_NOTICE);
}

a();
```

**Code : Console**

```
une erreur s'est produite : (1024) lala
dans le fichier C:\wamp\www\TutoOO\index.php à la ligne 17
Contexte : Array
(
    [arg] => 0
)
#0 {closure}(1024, lala, C:\wamp\www\TutoOO\index.php, 17, Array ([arg] => 0))
#1 trigger_error(lala, 1024) called at [C:\wamp\www\TutoOO\index.php:17]
#2 b(0) called at [C:\wamp\www\TutoOO\index.php:12]
#3 a() called at [C:\wamp\www\TutoOO\index.php:20]
```

C'est plutôt joli, et pour exterminer les erreurs, c'est très pratique. 😊

***error\_log()***

Je vous ai dit qu'afficher des erreurs en production, c'est très mal. Or, notre gestionnaire d'erreurs fait exactement ce qu'il ne faut pas faire. Pour sauvegarder vos erreurs, vous pourriez utiliser des fonctions de votre cru à grand renfort de base de données ou de fichiers. Mais si vous désirez sauvegarder les erreurs dans un fichier ou envoyer les erreurs par mail, le PHP propose une fonction : `error_log()`. En général, un administrateur est heureux quand il est rapidement prévenu d'une erreur grave, nous allons donc modifier notre gestionnaire de configuration pour qu'il enregistre les erreurs dans un fichier, sauf si c'est une erreur de type `E_USER_ERROR`, auquel cas nous enverrons un mail. `error_log()` peut prendre quatre paramètres :

- le message d'erreur ;
- la façon dont l'erreur est sauvegardée, par exemple par mail ou dans un fichier ;
- la destination, que ce soit un fichier ou un mail ;
- des en-têtes additionnels dans le cas d'une sauvegarde par mail.

Exemple :

**Code : PHP**

```
<?php

error_log('lala', 1, 'lala@lala.com'); // par mail
```

```
error_log('lili', 3, 'fichier.log'); // par fichier
```

Pour obtenir la pile d'appel sous forme de chaîne de caractères, utilisez ceci :

**Code : PHP**

```
<?php

function debugBacktrace()
{
    ob_start();
    debug_print_backtrace();
    return ob_get_clean();
}
```

À vous de jouer. 😊

**Secret (cliquez pour afficher)**

**Code : PHP**

```
<?php

function debugBacktrace()
{
    ob_start();
    debug_print_backtrace();
    return ob_get_clean();
}

set_error_handler(function($type, $msg, $file, $line, $context =
$array()) {
    $s = 'une erreur s\'est produite : (' . $type . ') ' . $msg .
"\n"
        . 'dans le fichier ' . $file . ' à la ligne ' . $line .
"\n"
        . 'Contexte : ' . print_r($context, true) .
debugBacktrace();

    if($type == E_USER_ERROR)
    {
        $method = 1;
        $dest   = 'lala@lala.lala';
    }
    else
    {
        $method = 3;
        $dest   = 'fichier.log';
    }

    error_log($s, $method, $dest);
});
```

### Intercepter les erreurs fatales

Je vous avais promis une astuce pour intercepter les erreurs fatales ; chose promise, chose due. L'idée est d'utiliser quelque chose d'inchangé aux erreurs fatales : elles terminent le script. Le tout est donc de pouvoir, lorsque le script se termine, détecter si c'est à cause d'une erreur. Grâce à la fonction register\_shutdown\_function(), nous pourrons enregistrer une fonction qui sera appelée lorsque le script se terminera, et ce, quelle que soit la raison de cet arrêt. L'astuce est donc d'enregistrer une fonction qui vérifiera s'il y a eu une erreur fatale grâce à error\_get\_last(), et qui agira en conséquence. Malheureusement, les fonctions relatives à la pile d'appel sont inutilisables sur les fonctions enregistrées comme telles, mais c'est toujours mieux que rien.

**Code : PHP**

```
<?php

register_shutdown_function(function() {
    $error = error_get_last();
    if($error !== null)
    {
        echo 'erreur fatale ?';
    }
}) ;

f();
```

Maintenant que vous êtes rompus à la gestion des erreurs, si je vois une seule erreur sur l'un de vos sites, je vous tire les oreilles. 

Ce chapitre n'est ni long ni compliqué, mais il est assez important, soyez donc sûrs de le maîtriser.

## Partie 4 : Annexes

Dans cette partie, vous trouverez des articles un peu à part, qui n'ont pas de place bien précise.

### Histoire de ce tutoriel

Bonjour chers lecteurs,

Dans ce chapitre un peu à part, vous trouverez des informations sur ce tutoriel comme l'historique des mises à jour, la liste des personnes qui m'ont aidé à le réaliser, qui l'ont relu, corrigé, etc.

#### Remerciements

Merci à toutes les personnes qui m'ont permis de rédiger ce tutoriel ou qui m'ont aidé.

Merci à [strucky](#) et [Dentuk](#) pour leurs relectures, leurs conseils et leurs critiques.

Merci à [nicodd](#) pour ses relectures et pour m'avoir donné un avis de débutant en PHP.

Merci à [delphiki](#) pour ses relectures et pour m'avoir supporté.

Merci à [Tortue facile](#), [Mickael Knight](#) et [Lpu8er](#) pour s'être occupés de la validation de mon tutoriel !

Merci à [ptipilou](#) pour toutes les corrections qu'il a apportées au tutoriel, ses conseils et son oreille attentive [*comprendra qui pourra* 😊].

Merci à [nicofrand](#) et [vincent1870](#) pour les corrections qu'ils ont apportées.

Et enfin, merci au [Site du Zéro](#) et à [toute son équipe](#) ainsi qu'aux [zCorrecteurs](#) pour m'avoir offert la possibilité de publier ce tutoriel qui sera, je l'espère, utile à certaines personnes.



[Si j'ai oublié de citer quelqu'un, envoyez-moi un MP. =D]

#### Historique des mises à jour

- 20 juillet 2007 : point de départ du tutoriel, création de la partie I : *Les bases du langage*
- 27 juillet : rédaction du chapitre 1, *Prémices*
- 28 juillet : rédaction du chapitre 2, *Premiers pas...*
- 29 juillet : rédaction du chapitre 3, *Les variables, le cœur du PHP*
- 30 juillet : rédaction du chapitre 4, *Les opérateurs : partie I*
- 31 juillet : modification du chapitre 4
- 1<sup>er</sup> août : rédaction du chapitre 5, *Les opérateurs : partie II*
- 2 août : rédaction du chapitre 6, *Les structures de contrôle*, modification des chapitres 1, 2 et 3
- 16 août : rédaction du chapitre 7, *Les fonctions*, et du chapitre 8, *Les chaînes de caractères, un casse-tête*
- 17 août : rédaction du chapitre 9, *Le type manquant : les arrays*, du chapitre 10, *Bons plans*, et du chapitre 11, *Transmettre des variables*
- 18 août : modification des chapitres 6, 7 et 10
- 19 août : rédaction du chapitre 12, *Premier TP : une calculatrice*, modification du chapitre 11
- 20 août, 00h00 : envoi du tutoriel à la validation !
- 26 août : rédaction de la première sous-partie du premier chapitre de la partie II, *MySQL, qui es-tu ?*
- 1<sup>er</sup> septembre : tutoriel validé !
- 3 septembre : correction d'une erreur dans le code du chapitre *Premier TP : une calculatrice* (mauvais nom de variable)
- 4 septembre : grosse mise à jour pour corriger des fautes d'orthographe dans les chapitres 1 à 10 de la partie I, et nouvel envoi à la validation
- 5 septembre : correction d'une erreur dans un code de la sous-partie *Évaluation* du chapitre *Les structures de contrôle*, nouvel envoi à la validation et fin de la rédaction du chapitre I de la partie II, *MySQL, qui es-tu ?*
- 8 septembre : rédaction de la moitié du chapitre II de la partie II, *Créons une base de données et une table pas-à-pas*
- 9 septembre : rédaction de la fin du chapitre *Créons une base de données et une table pas-à-pas*
- 10 septembre : rédaction du chapitre III, *Gérons nos tables !*
- 27 octobre jusqu'au 29 octobre : reprise du tutoriel, rédaction du chapitre IV, *SELECT, ou comment récupérer des données*
- 27 novembre : rédaction du chapitre V, *Supprimer, modifier et insérer des données*
- 28 novembre : rédaction du chapitre VI, *PHP et MySQL : des alliés de poids*
- 12 et 13 janvier 2008 : rédaction du chapitre VII, *Second TP : un visionneur de bases de données et de leurs tables*
- 15 janvier : rédaction du chapitre VIII, *Fonctions, conditions et quelques opérateurs SQL*

- 16 janvier : rédaction du chapitre IX, *La gestion des dates*
- 23 janvier : envoi du tutoriel à la validation !
- 14 février : rédaction du chapitre X, *Bataillons, à vos rangs... fixe !* (la fin s'est faite dans la sueur et les larmes, comprendra qui pourra 😊)
- 24 mars : rédaction du chapitre XI, *Troisième TP : un livre d'or*
- 19 avril : corrections mineures et uniformisation de la coloration des codes suite au passage du Site du Zéro à Pygments
- 08 mai : rédaction du chapitre A de la partie III, *Premiers pas avec la Programmation Orientée Objet*
- 10 mai : rédaction du chapitre B de la partie III, *Implémentation en PHP*
- 11 mai : envoi du tutoriel à la validation !
- octobre 2008 : suppression de mon compte et du tutoriel pour des raisons personnelles
- mars 2009 : réinscription sur le Site du Zéro et remise en ligne du tutoriel sous licence Creative Commons BY-SA
- 4 novembre 2009 : changement du style d'indentation en faveur de BSD/Allman
- 5 novembre 2009 : passage à l'extension MySQLi
- 6 novembre 2009 : rédaction du chapitre XII, *Les index et sous-requêtes*
- janvier 2010 : début de la rédaction de plusieurs chapitres sur l'orienté objet
- février 2010 : rédaction de plusieurs chapitres portant sur les références, les fonctions anonymes, les espaces de noms et bien d'autres

## Suite du tutoriel

Actuellement, je travaille sur la « suite » de ce tutoriel. Concrètement, j'aimerais faire un tutoriel qui accompagnera le lecteur dans la création d'un site complet du style du Site du Zéro. Je ne m'attends pas à ce que le lecteur soit capable de reproduire un site de ce genre après la seule lecture de ce tutoriel, mais s'il parvient déjà à comprendre le cheminement qui m'amène à tel ou tel choix dans le développement, j'en serai content.

Certaines parties de la version actuelle devront être réécrites, elles ne me plaisent pas, mais ça n'est pas spécialement dramatique, donc ça attendra.

J'espère pouvoir vous offrir de nouvelles parties rapidement.