
*PHP

Introduction au langage PHP

1. Introduction au langage PHP
2. Migration de PHP 4 vers PHP 5

Les environnements de travail pour développer en PHP

1. Installation et prise en main de MAMP
2. Installer un environnement LAMP6 sur Debian

Les bases du langage PHP

1. Premier programme : affichage du traditionnel « Hello World »
2. Les différents types de commentaires
3. Les constantes
4. Les variables
5. Les tableaux (ou arrays)
6. Les opérateurs
7. Les structures de contrôle : les conditions
8. Les structures de contrôle : les boucles
9. Les procédures et fonctions utilisateurs
10. Traitement des formulaires avec $_GET et $_POST
11. Les cookies
12. Les sessions
13. Les importations de fichiers avec require() et include()

Traitement d'images avec PHP

1. Imagefilter() : les effets spéciaux

La programmation orientée objet (POO)

1. Les classes et objets
2. Visibilité des propriétés et des méthodes
3. Méthodes magiques : __set() et __get()
4. Méthodes magiques : __call()
5. Méthodes magiques : __clone
6. Méthodes magiques : __sleep() et __wakeup()
7. Les classes abstraites et finales
8. Les exceptions - 1ère partie
9. Les exceptions - 2ème partie
10. Utiliser l'interface Iterator avec PHP 5

Les motifs de conception (design patterns)

1. Singleton : instance unique d'une classe

Sécurité des applications PHP

1. Introduction aux Cross Site Request Forgeries ou Sea Surf
2. Sécuriser les mots de passe avec les hashs et les salts

Les frameworks PHP

1. Génération de PDF avec le Zend Framework

Le langage PHP a été inventé par Rasmus LERDORF en 1995 pour son usage personnel (mise en ligne de son CV en l'occurence). Autrefois abbréviation de Personal HomePage devenue aujourd'hui Hypertext Preprocessor, PHP s'impose comme un standard dans le monde de la programmation web par ses performances, sa fiabilité, sa souplesse et sa rapidité.

Préambule

PHP a été inventé à l'origine pour le développement d'applications web dynamiques qui constituent encore le cas d'utilisation le plus courant et son point fort. Cependant, les évolutions qui lui ont été apportées jusqu'à aujourd'hui assurent à PHP une polyvalence non négligeagle. PHP est par exemple capable d'interragir avec Java, de générer des fichiers PDF, d'exécuter des commandes Shell, de gérer des objets (au sens programmation orientée objet), de créer des images ou bien de fournir des interfaces graphiques au moyen de PHP GTK.
Dans cette présentation du langage, nous introduirons tout d'abord les caractéristiques de PHP, puis nous verrons en quoi il est particulièrement adapté aux développements d'applications web. Nous synthétiserons ensuite les autres types d'applications possibles aec PHP avant de terminer sur les limites que l'on peut lui reprocher.

Les caractéristiques du langage PHP

License

PHP est tout d'abord un langage de script interprêté (en réalité précompilé en Opcode), gratuit, OpenSource et distribué sous une license autorisant la modification et la redistribution.

Portabilité

PHP est supporté sur plusieurs systèmes d'exploitation. C'est par exemple le cas des versions Microsoft Windows™, mais aussi des systèmes reposant sur une base UNIX (Apple MAC OS X™, distributions Linux ou encore Sun Solaris).
Il sera alors très facile de déplacer une application écrite en PHP d'un serveur Windows d'origine vers un serveur Linux sans avoir à la modifier (ou très peu).

Exécution

D'un point de vue exécution, PHP a besoin d'un serveur Web pour fonctionner. Toutes les pages demandées par un client seront construites par le serveur Web, en fonction des paramètres transmis, avant d'être retournées au client. Le schéma ci-dessous illustre le principe de fonctionnement de PHP.

***Note** : il aurait été possible d'ajouter un serveur de bases de données (local ou distant) sur cette illustration. Nous aurions eu alors deux étapes supplémentaires qui sont l'interrogation de la base de données par PHP et la récupération des résultats en provenance du serveur SQL.*

Apprentissage de PHP

PHP est un langage dit de « haut niveau » dont la syntaxe est très proche du langage C. Cette syntaxe proche du langage naturel lui assure un apprentissage rapide et peu de contraintes d'écriture de code. Néanmoins, la maîtrise rapide de sa syntaxe ne signifie pas la maîtrise de ses fonctionnalités et de ses concepts. Une bonne connaissance et une utilisation avancée de la programmation PHP nécessite un temps d'apprentissage relativement long.

Richesse du langage PHP

Une des forces du langage PHP est sa richesse en terme de fonctionnalités. En effet, il dispose à l'origine de plus de 3 000 fonctions natives prêtes à l'emploi garantissant aux développeurs de s'affranchir de temps de développement supplémentaires et parfois fastidieux. Ces fonctions permettent entre autre de traiter les chaînes de caractères, d'opérer mathématiquement sur des nombres, de convertir des dates, de se connecter à un système de base de données, de manipuler des fichiers présents sur le serveur...

PHP puise aussi sa richesse dans le dynamisme de sa communauté de développeurs. Celle-ci était estimée à 500 000 personnes environ en 2003 mais il est très probable qu'elle ait dépassé le million maintenant. Les profils de développeurs de la communauté sont très divers. Il y'a ceux qui apportent de nouvelles fonctionnalités et librairies de version en version, ceux qui traduisent la documentation en plusieurs langues ou encore les programmeurs ayant des compétences plus modestes qui réalisent des applications Opensources prêtes à l'emploi. Parmi les plus connues, nous pouvons citer les CMS (Joomla, SPIP, Dotclear, Wordpress...), les systèmes de ventes en ligne (OSCommerce), les forums (PHPBB, IPB, VBulletin), les frameworks (Zend Framework, Symfony, CakePHP, Jelix)...

PHP, un langage fiable et performant

Le langage est maintenant devenu un langage fiable, performant et viable. Il est capable de supporter des sites qui sollicent des millions de requêtes journalières. De nombreuses entreprises de renommée nationale et internationale lui font confiance pour le développement de leur site Internet. Nous pouvons parmi elles citer TF1, IBM, Le Monde, Le Figaro, Club-Internet, Orange, Pages Jaunes... Un récent rapport daté de novembre 2006 indique que 87% des entreprises du CAC40 utilisent PHP. Lire l'interview de Perrick Penet (AFUP).

PHP, un langage conçu pour les applications web dynamiques

Le langage PHP a la principale fonction d'être spécialement conçu pour la réalisation d'applications web dynamiques. Par définition, une « application (ou page) dynamique » est un programme capable de générer une page unique en fonction de paramètres qui lui sont transmis.
Un script PHP peut donc être intégré directement à l'intérieur d'un code html. Petit exemple pratique ci-dessous :

Premier programme PHP : "Hello World"

```
<html>
    <head>
        <title>Hello World en PHP</title>
    </head>
    <body>
        <p>
            <?php echo 'Hello World !'; ?>
        </p>
    </body>
</html>
```

Le script PHP, clairement identifié par les deux balises <?php (ouverture) et ?> (fermeture), provoquera l'écriture de la chaîne de caractères **Hello World** entre les balises HTML après son exécution sur le serveur.

Nous aurions également pu obtenir le même résultat en utilisant le script PHP suivant :

Autre version du "Hello World"

```
<html>

    <head>

        <title>Hello World en PHP</title>

    </head>

    <body>

        <?php echo '<p>Hello World !</p>'; ?>
    </body>
```

```
</html>
```

De ce fait, on en déduit que **PHP est capable de générer du code HTML** (ainsi que d'autres formats), ce qui fait tout son intérêt. La présentation du document généré est alors complètement dépendante des conditions passées et des paramètres initiaux fournis. Prenons l'exemple du site de vente par correspondance Amazon.fr. Les utilisateurs inscrits au site qui ont déjà passé plusieurs commandes auront la surprise de voir à chaque nouvelle visite sur leur page personnalisée, une liste de produits sélectionnés automatiquement qui correspondent aux critères de ses précédents achats. Chacune de ces actions de marketing direct ciblé est unique et générée en fonction des intérêts du consommateur.

PHP pour d'autres domaines d'application

Fort de sa richesse, PHP ne se limite pas forcément à l'édition de pages web dynamiques. Il peut par exemple être utilisé en ligne de commande via l'utilisation de l'exécutable *php*. Ce cas d'utilisation permet alors d'exécuter des scripts directement sur les machines. Un script PHP serait alors mis au profit de la machine. Nous pouvons très bien imaginer un programme PHP capable de supprimer un certain nombre de fichiers présents dans un dossier. Il est même possible de coupler l'utilisation de PHP avec un gestionnaire de tâches tel qu'un serveur *cron* sous Linux.

PHP c'est également la possibilité de créer des applications lourdes fonctionnant sans serveur ni navigateur. Autrement dit des applications traditionnelles, autonomes et munies de fenêtres. Tout cela se réalise au moyen de la librairie PHP GTK disponible à l'adresse : http://gtk.php.net

Un autre point fort de PHP est sa capacité à s'interfacer très facilement avec de nombreux systèmes de gestion de bases de données relationnelles (SGBDR). Parmi eux, nous pouvons retrouver *MySQL, Oracle, SQLite, MSSQL, PostgreSQL...* Grâce à ces systèmes couplés au langage PHP, il devient possible de distribuer les applications sur plusieurs serveurs (serveur Web + serveur de bases de données). Le second intérêt à cela est de pouvoir rendre une application encore plus dynamique. En effet, les données (contenu) de l'application se trouve à présent dans la base de données et PHP se charge de les récupérer puis de les manipuler (traitement des chaines de caractères, enregistrement dans des fichiers, génération de flux RSS...).

Les possibilités offertes par PHP sont donc très nombreuses et nous n'allons pas les détailler toutes car nous y resterions des heures. Retenons néanmoins une liste de ses principales capacités :

- Manipulation d'un système de fichiers (création, édition, suppression, droits d'accès...)
- Gestion des sessions utilisateurs
- Génération et parsing de documents XML grâce à la librairie SimpleXML
- Génération d'images avec GD2
- Génération de fichiers PDF
- Accès simplifié aux bases de données avec la librairie PDO
- Exécution de commandes Shell

- Gestion des e-mails en POP et IMAP
- Compression et décompression d'archives ZIP
- Cryptage MD5 et SHA1
- Gestion d'annuaires LDAP
- Manipulation des dates
- Manipulation des URL
- Envoi et lecture de cookies
- Dialogue avec Java
- Utilisation d'Ajax
- ...

La version actuelle de PHP apporte un grand vent de fraîcheur au langage et aux professionnels. Le principal manque jusque là était son modèle objet trop succint. Les développeurs se sont alors penchés sur cette problématique et ont finalement implémenté un modèle de programmation objet proche du langage Java. Grâce à un tel modèle, les professionnels encore hésitants à utiliser PHP deviennent de plus en plus nombreux. Des applications complètement objet voient également le jour et intègrent un design pattern (motif de conception) MVC. Les plus connues aujourd'hui sont les frameworks Zend etSymfony des sociétés respectives Zend Technologies et Sensio (agence française).

Les limites de PHP

Malgré toutes les qualités que nous pouvons attribuer au langage PHP, subsistent quelques défauts. Par exemple, dans le cas de très grosses applications, il peut présenter quelques faiblesses et devenir inadapté. Un langage tel que PERL deviendrait alors plus adéquat. Cet argument reste toutefois très subjectif dans la mesure où les développeurs de PHP améliorent la qualité et la robustesse du langage.
Le second défaut (mais qui paradoxalement fait son succès et sa qualité) que nous pouvons lui reprocher est sa grande simplicité d'utilisation. Cela a beaucoup terni l'image de PHP parceque tout webmaster (même très peu expérimenté) devient capable de créer du code et des applications facilement. Cependant, la plupart des codes produits par des développeurs amateurs n'est pas forcément "propre" ou bien conçu, souvent peu sécurisé, peu maintenable et même non optimisé. Les autres langages comme C++, .Net, Java, ASP, Perl, Python ou Ruby ne subissent pas cette mauvaise image dans la mesure où leur apprentissage n'est pas forcément très aisé.
Enfin, le dernier défaut reprochable à PHP est son manque de rigueur dans la nomenclature des fonctions et de la syntaxe (voir tutoriel sur l'utilisation des balises courtes). Tout d'abord, les fonctions ne sont pas sensibles à la casse, ce qui signifie par exemple que str_replace() et STR_REplaCe() sont identiques pour l'interprêteur PHP. Heureusement la nouvelle version en cours (PHP6) résoudra ce défaut. Par ailleurs, nous constatons un manque de standardisation des noms des fonctions :

- Utilisation du séparateur underscore : str_replace(), preg_match(), mysql_real_escape_string() ...

- Fonctions composées de plusieurs mots écrite en un seul : wordwrap(), htmlspecialchars() ...
- Traduction du *to* en litéral ou numéraire : bin2hex(), strtotime()...

Conclusion

Nous conclurons que PHP a encore de beaux jours devant lui et que son avenir sera encore très prometteur avec l'arrivée de sa nouvelle version. Celle-ci lui apportera d'ailleurs une touche plus professionnelle car elle se destine plus particulièrement aux utilisateurs confirmés et professionnels.

Migration de PHP 4 vers PHP 5

Le support de PHP 4 appartient au passé. Il devient donc urgent de migrer vers PHP 5 car en 2008 nulle nouvelle version de PHP 4 ne verra le jour (un support sera tout de même assuré sur les failles de sécurité jusqu'au 08/08/2008).
La compatibilité entre PHP 5 et PHP 4 a été une des préoccupations majeures durant le développement de PHP 5. Une grande majorité des applications devraient pouvoir être exécutées sur PHP 5 sans problèmes, ou ne nécessiter que des modifications mineures. Il existe cependant quelques différences et nous allons essayer de les résumer ici pour vous permettre une migration simple.

Pourquoi migrer ?

- Nouvelles fonctionnalités
- Meilleures performances
- Meilleure sécurité
- Meilleure stabilité
- PHP 5 est fortement supporté

Ce qui a changé avec PHP 5 ?

- la refonte du coeur de PHP qui permet une prise en charge complète de la programmation orientée objet
- la refonte de la prise en charge de XML
- l'intégration de la base de données embarquée SQLite
- l'intégration de nouvelles extensions (JSON, Filter, ZIP, ...)
- l'apparition d'un socle commun pour la gestion des appels aux bases de données : PHP Data Object (PDO)
- l'utilisation de la reflexion objet (introspection)
- les exceptions ont fait leur apparition en PHP5
- un niveau d'erreur E_STRICT à été ajouté
- apparition de la SPL (Standard PHP Library), un rassemblement de classes internes utiles

Bien que la plupart des scripts PHP 4 existants devraient fonctionner, il convient de noter quelques différences majeures pouvant entraîner des erreurs ou des comportements différents :

- la gestion des objets (passage par référence)
- la refonte du support de DOM avec l'abandon de l'extension DomXML
- l'extension MySQL n'est plus incluse par défaut
- nouveau mode d'erreur E_STRICT

Le nouveau modèle objet

La principale nouveauté de PHP 5 est certainement le nouveau modèle objet. Le traitement des objets a complètement été réécrit pour arriver à de meilleures performances et plus de fonctionnalités. A ce jour le modèle objet de PHP 5 est proche de celui de Java, il en résulte donc un certain nombre de nouveautés : méthodes magiques, visibilité, (était déjà présent en PHP4, la nouveauté c'est le destructeur ou plus généralement les méthodes magiques), encapsulation, clonage, interfaces, classes abstraites...

Les objets sont passés par référence

Dans PHP 4 l'objet en entier était copié lorsqu'il était assigné ou passé comme paramètre à une fonction. Dans PHP 5 les objets sont référencés par un pointeur et non pas leur valeur (on peut penser à un pointeur en tant qu'identifiant d'objet).

Passer un objet par référence

Les objets ne sont donc plus passés par valeur mais par référence. Il en résulte qu'une fois transmis à une fonction, un objet PHP 5 verra ses valeurs évoluer alors qu'en PHP 4 c'est une copie qui sera modifiée au sein de la fonction, l'objet originel restera inchangé.
En PHP 4 pour faire passer un objet en référence on pouvait le faire en préfixant la variable avec le signe « & ». Testez notre exemple Xxxx en enlevant le « & » dans la déclaration de la fonction « fctActionObjet() », vous verrez que dans un cas l'objet « $b » est modifié et dans l'autre il ne l'est pas (et pour cause la modification a été faite sur une copie temporaire).

Passage des objets par copie en PHP 4

```php
<?php

// Classe PHP 4
class ClassA
{
  var $nom;

  function ClassA($var){
    $this->nom = $var;
  }

  function changeNom($var){
    $this->nom = $var;
  }
}

// Fonction prenant un objet en paramètre
// et changeant une valeur.
function fctActionObjet(&$obj){
  $obj->changeNom('Pierre');
}

$b = new ClassA('Cyril');
fctActionObjet($b);

print_r($b);

?>
```

Passage des objets par référence en PHP 5

```php
<?php

// Classe PHP 5
class ClassA
{
  private $nom;

  function __construct($var){
    $this->nom = $var;
  }
```

```php
  function changeNom($var){
    $this->nom = $var;
  }
}

// Fonction prenant un objet en paramètre
// et changeant une valeur.
function fctActionObjet($obj){
  $obj->changeNom('Pierre');
}

$b = new ClassA('Cyril');
fctActionObjet($b);

print_r($b);

?>
```

Dupliquer un objet

Vu que les objets sont passés par référence en PHP 5, une méthode spécifique à été ajoutée afin de les dupliquer : **clone().**

Clonage d'objet en PHP 4

```php
<?php

// Classe PHP 4
class ClassA
{
  var $nom;

  function ClassA($var){
    $this->nom = $var;
  }

  function changeNom($var){
    $this->nom = $var;
  }
}

$b = new ClassA('Cyril');
$newObj = $b;

?>
```

Clonage d'objet en PHP 5

```php
<?php

// Classe PHP 5
class ClassA
{
```

```
  private $nom;

  function __construct($var){
    $this->nom = $var;
  }

  function changeNom($var){
    $this->nom = $var;
  }
}

$b = new ClassA('Cyril');
$newObj = clone($b);

?>
```

Heureusement grâce au travail du PHPGroup ces changements cassent que peu la compatibilité et PHP interprète souvent le code PHP 4 pour le rendre compatible avec PHP 5.

Quelques informations supplémentaires :

- PHP5 rajoute aussi la visibilité objet : protected / private / public, les interfaces, les classes abstraites, l'autoloading de classes et le typage fort objet
- Le mot clef « var » utilisé en PHP 4 fonctionne en PHP 5, il est traduit en « public ».
- Le constructeur de classe utilisé en PHP 4 (fonction ayant le même nom que la classe) fonctionne avec PHP 5.
- Il n'est pas nécessaire d'enlever les « & » qui étaient utilisés dans le code PHP 4 pour simuler le passage par référence. Vous aurez cependant un message d'erreur si vous êtes en mode « strict »

Nouveaux mots-clés réservés

PHP 5 a amené son lot de nouveaux mots réservés. Il s'agit d'identifiants prédéfinis en PHP qui ne doivent pas être utilisés comme constante, nom de classe, nom de fonction ou nom de méthode dans vos scripts.

interface	*implements*	*clone*	*try*
catch	*public*	*private*	*protected*
throw	*this*	*final*	*static*

En règle générale vous pourrez avoir des incompatibilités sur ce qui touche à la POO pour des noms de fonctions / méthodes que vous avez implémentés pour simuler des comportements objets avancés non supportés par PHP 4.
Il existe d'autres mots réservés, consultez la documentation en ligne pour en avoir la liste intégrale à l'adresse : http://www.php.net/manual/fr/reserved.php

XML

Les autres nouveautés concernent la gestion de XML. La version 4 de PHP impliquait une utilisation relativement lourde pour qui souhaitait manipuler des flux XML. Avec la version 5, deux nouveautés révolutionnent sa manipulation :

- L'intégration d'un nouveau gestionnaire XML : la bibliothèque libxml2, qui amène une implémentation DOM standard complète (ce qui n'était pas le cas en PHP 4) ;
- L'extension SimpleXML.

La première permet de traiter tous les aspects de la manipulation XML, avec la complexité que cela implique. La seconde s'adresse à tous les traitements XML simples. Il n'est plus obligatoire de passer par des opérations compliquées pour récupérer les données de fichiers XML.

Les incompatibilités peuvent venir des changements liés à l'implémentation DOM complète de PHP 5 qui sera incompatible avec celle utilisée en PHP 4. On peut noter une amélioration de la compatibilité de cette extension à partir de PHP 4.3 mais dans tous les cas tout ce qui n'est pas objet ne sera pas fonctionnel en PHP 5.
Si vous utilisiez l'extension DomXML pour parser du XML il est temps de réécrire votre code. Au pire des cas vous pouvez installer DomXML à partir du repository PECL.

Bases de données

Pour des raisons de licence l'extension MySQL n'est plus embarquée par défaut dans le package PHP 5. Ce n'est pas grand chose, cela veut juste dire que les extensions ne sont plus activées par défaut. Il vous faudra juste l'indiquer lors de la compilation. De façon générale cela ne vous concerne que si vous compilez vous même PHP : La majorité des auto-installeurs PHP (WAMP5, XAMP, MAMP, EasyPHP) embarquent MySQL chargé par défaut.

Divers

En PHP 5, la nouvelle constante de rapport d'erreurs E_STRICT a été introduite avec comme valeur 2048. Cela permet à PHP, lors de l'exécution, de faire des suggestions sur la compatibilité et le suivi de votre code. Ceci vous incite à toujours utiliser les meilleures méthodes de codage et les plus récentes : par exemple les messages stricts vous avertiront sur l'utilisation de fonctions obsolètes et l'utilisation de variables non déclarées.

Preparer et tester sa migration

Validateur de code

Sous Linux il est possible de tester la compatibilité de vos scripts en ligne de commande. Pour cela placez toute votre arborescence PHP sur un serveur

utilisant PHP 5. Connectez vous à ce serveur en root et tapez la ligne de commande suivante :

```
find /source/repertoire -name \*.php | xargs -n1
/chemin/vers/php -ddisplay_errors=1 -derror_reporting=8191 -l
```

Le niveau d'affichage d'erreur est tel (8191 : E_ALL | E_STRICT) que toutes les erreurs de code ou tous les appels à des fonctions/fonctionnalités dépréciées généreront une erreur. Ce que ce validateur vous indique :

- Si les fichiers sont exécutés sans erreurs
- Si vous n'utilisez pas de fonctionnalités / fonctions dépréciées.

Attention cette méthode ne vous donnera pas toutes les informations. Notamment vous n'aurez pas de message d'erreur quand un objet n'utilise pas la fonction clone() pour se copier.

Le code suivant vous affiche une erreur sur la ligne « $a = & new agent(); » car il n'est plus nécessaire d'utiliser le « & ».

Exemple de code PHP 5 forçant le passage par référence de l'objet

```php
<?php

class agent{
  public $nom;
}

$a = & new agent();

?>
```

Utiliser WampServer

WampServer est un outil qui vous permet de déployer une plateforme Apache PHP 5 MySQL sur votre poste de travail Windows sans coup férir. Une des grandes qualités de ce logiciel est de disposer d'un système d'add-ons. L'un d'eux, le plus connu, permet de switcher de PHP 5 vers PHP 4 et vice versa.
Ainsi une manière empirique de tester la migration de votre logiciel serait de l'installer en local avec WampServer sous PHP 4, de le tester puis de switcher en PHP 5 et de le tester à nouveau.

Pour automatiser les tests vous pouvez utiliser un outil tel que l'extension firefox « selenium-ide » qui permet de définir des scénarios de tests extrêmement

simplement.

Conclusion

Faire fonctionner vos applications développées pour PHP 4 dans un environnement PHP 5 ne devrait pas vous demander énormément de temps. Surtout si vous n'utilisez pas de programmation orientée objet.
En fait le réel intérêt de cette migration réside dans l'énorme potentiel que vous offre PHP 5 au travers de toutes les nouveautés qu'il offre. Alors n'hésitez pas, foncez vers PHP 5 !

Installation et prise en main de MAMP

Afin de pouvoir travailler avec le langage PHP, nous devons nous assurer des pré-requis nécessaires à l'exécution des programmes. Dans le cadre de développement d'applications web, un serveur Web muni de PHP est obligatoire. Le logiciel MAMP (abbréviation de Macintosh, Apache, MySQL and PHP) permet de lancer un serveur web local sur une machine fonctionnant sur le système d'exploitation Mac OS X. C'est un logiciel destiné aux utilisateurs de la marque à la pomme et qui contient les composants gratuits suivants : Apache (deamon serveur), PHP5, MySQL (SGBD, SQLite (SGBD), PHPMyAdmin et SQLiteManage (outils d'administration de BDD).
Nous allons montrer dans ce cours comment installer ce logiciel sous MacOS et l'utiliser pour nos premières applications PHP.

Téléchargement du logiciel

La première chose est bien entendu de télécharger le logiciel sur le site officiel de MAMP à l'adressehttp://www.mamp.info/
MAMP est disponible pour les Macintosh équipés de puce Motorola PowerPC ou Intel.

Installation de MAMP

Après avoir décompressé l'archive Zip, nous obtenons une capture semblable à celle-ci. C'est l'installateur du logiciel.

Il suffit de faire un « drag and drop » du dossier *MAMP* de la fenêtre et de le déposer dans le répertoire*Applications* de celle-ci. Ca y'est MAMP est correctement

installé. Nous remarquons le dossier MAMP (muni du logo gris) dans le répertoire *Applications* de MacOS (raccourci Pomme + Shift + A).

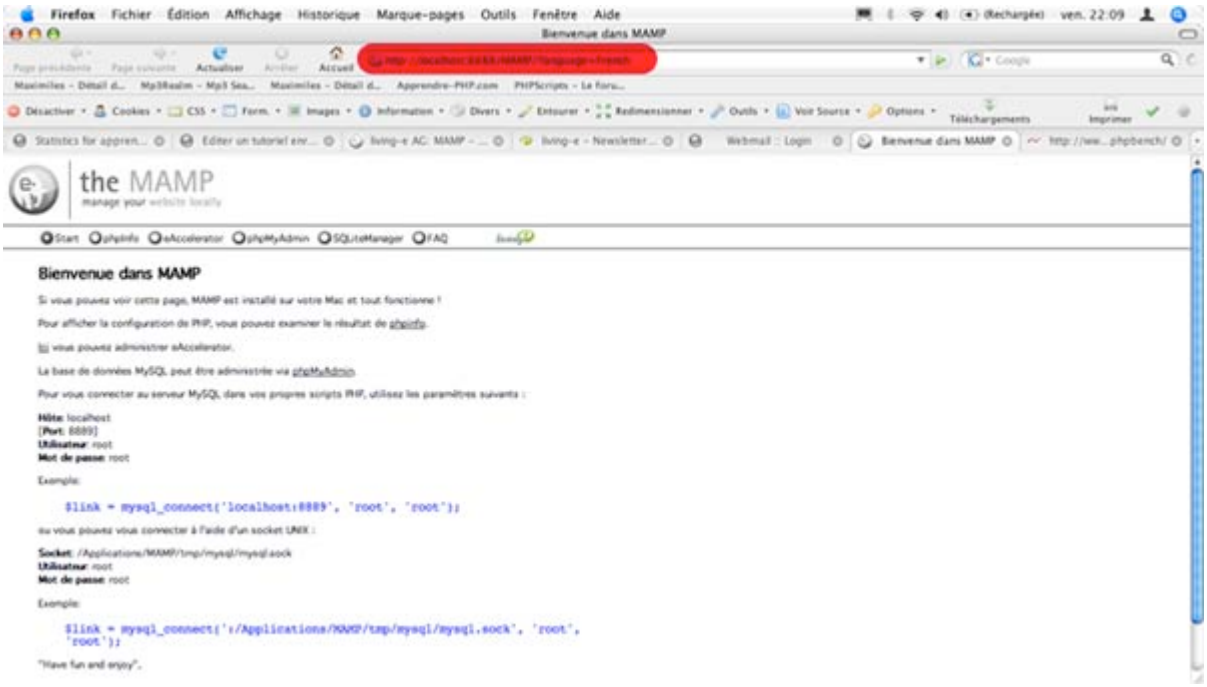
Premier démarrage du logiciel

Le lancement de MAMP s'effectue en cliquant sur l'icône *MAMP.app*. Une nouvelle fenêtre se lance.

Cette fenêtre permet de contrôler que le serveur Web (Apache) et le serveur de bases de données (MySQL) ont bien été lancés. Les voyants verts attestent du bon fonctionnement. L'onglet*Préférences...* permet de configurer MAMP.

Les paramètres modifiables sont par exemple les ports derrière lesquels les processus serveurs travaillent ou bien le répertoire dans lequel se situent les différents projets (sites web par exemple).

Il est recommandé de laisser la configuration imposée par MAMP pour des personnes n'ayant pas suffisamment de connaissances en informatique.
Le démarrage de MAMP implique aussi l'ouverture d'une nouvelle fenêtre dans le navigateur Web (Safari par défaut). C'est la page d'accueil de MAMP. Elle atteste aussi du bon fonctionnement du serveur Web local.



Notons tout d'abord que pour accéder au serveur Web local, nous utilisons l'url*http://localhost:8888/MAMP*. Le localhost peut aussi être remplacé par l'adresse IP *127.0.0.1*. Le serveur Web écoute les informations du réseau local sur le port 8888. Le port 80 étant réservé par défaut au Web par Internet.
Le menu situé en haut donne accès au phpinfo() (configuration de PHP), aux utilitaires de gestion de bases de données (PHPMyAdmin et SQLiteManager) ainsi qu' à une application d'optimisation des performances de PHP (eAccelerator).

Les répertoires importants de MAMP

Les répertoires importants de MAMP sont au nombre de six :

- **/bin** : répertoire contenant les exécutables d'Apache, PHP4, PHP5, MySQL5 et SQLite.
- **/conf** : répertoire contenant les fichiers de configuration d'Apache (httpd.conf), PHP (php.ini) et SQLiteManager (config.db).
- **/tmp** : répertoire contenant les fichiers temporaires créés par les exécutables. Le répertoire /tmp/php contient notamment les fichiers temporaires des sessions PHP.
- **/db** : répertoire contenant les bases de données SQLite et MySQL5.
- **/logs** : répertoire contenant les fichiers de logs d'erreurs de PHP, Apache et MySQL.
- **/htdocs** : répertoire contenant les différents projets de sites Web.

Ce dernier nous intéresse tout particulièrement car c'est dans celui-ci que nous déposerons nos sites Internet.

Premier test de PHP

Nous allons tester notre première application PHP : le traditionnel *"hello world !"*. Pour cela, nous commençons par créer un nouveau répertoire appellé *tests-php* dans le répertoire */htdocs*.

Puis nous plaçons dans ce répertoire un fichier nommé *hello.php* qui contient le code suivant :

Listing de hello.php : premier programme PHP

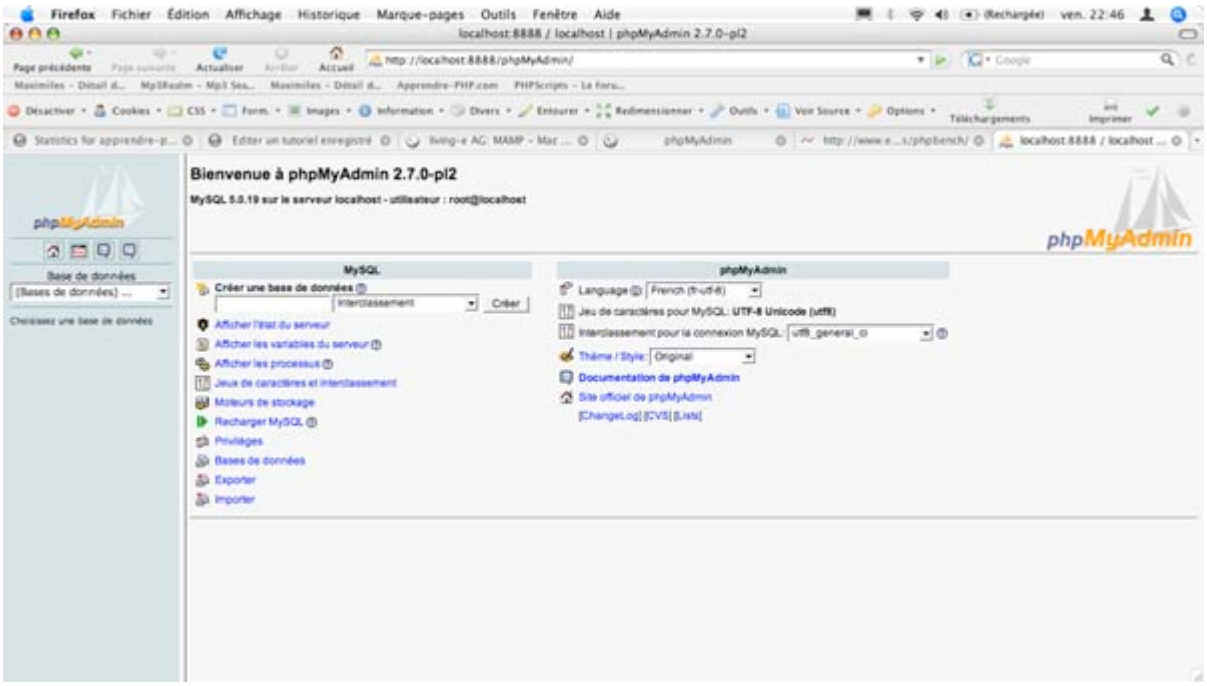
```
<html>

    <head>

        <title>Hello World en PHP</title>

    </head>

    <body>

        <p>

            <?php echo 'Hello World !'; ?>
        </p>
    </body>
</html>
```

Dans notre navigateur, nous appellons le fichier en entrant l'adresse suivante : *http://localhost:8888/tests-php/hello.php*. Le script PHP est exécuté et affiche à l'écran le texte *Hello World !*. PHP fonctionne donc parfaitement :)

Gestionnaires de bases de données : PHPMyAdmin et SQLiteManager

PHPMyAdmin

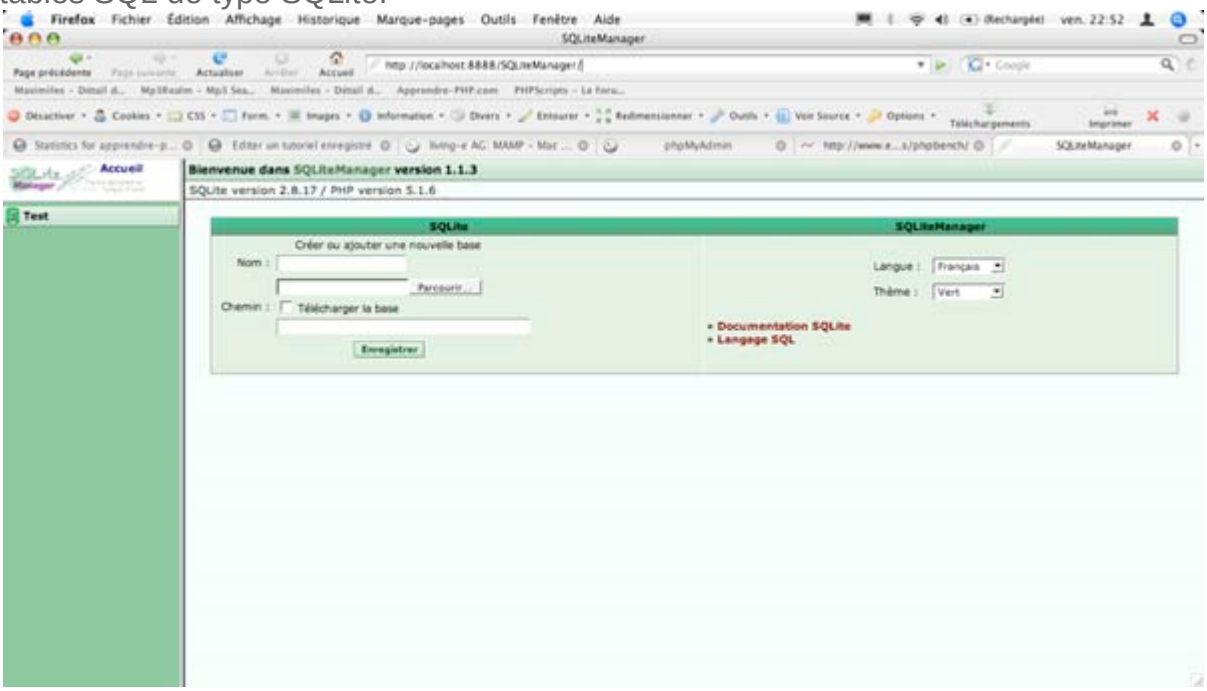
L'utilitaire gratuit PHPMyAdmin se trouve à l'adresse suivante : *http://localhost:8888/phpMyAdmin/*. Il permet de gérer les bases de données et tables SQL de type MySQL.



Nous reviendrons sur cette application lors de prochains tutorielss nécessitant l'utilisation de bases de données MySQL.

SQLiteManager

L'utilitaire gratuit SQLiteManager se trouve à l'adresse suivante : *http://localhost:8888/SQLiteManager/*. Il permet de gérer les bases de données et tables SQL de type SQLite.



Nous reviendrons sur cette application lors de prochains tutoriels nécessitant l'utilisation de bases de données SQLite.

Le widget MAMP

MAMP propose également un petit widget à placer dans le Dashboard de MacOS. Celui-ci permet de contrôler l'état des serveurs Web et SQL; et de les redémarrer si nécessaire.

En cliquant sur le petit point d'exclamation, il est possible de basculer en 1 seconde de PHP5 à PHP4. C'est très pratique lorsque l'on veut tester la compatibilité d'une application.

Conclusion

Nous venons d'apprendre à installer et utiliser le logiciel MAMP. Nous ne sommes en revanche pas arrêter sur la configuration de PHP. Celle-ci étant suffisante pour la majorité des applications web.

Premier programme : affichage du traditionnel « Hello World...

Dans les précédents tutoriels concernant les environnements de travail, nous avons montré que PHP était un langage de script dynamique précompilé et interprêté côté serveur. Il nous appartient maintenant de réaliser nos premiers programmes et de les exécuter sur le serveur Web (local ou distant).

En programmation informatique, il existe une *"tradition"*qui est de générer la **chaîne de caractères** (notez le terme au passage) *Hello World !* sur la sortie standard (dans notre cas c'est un écran d'ordinateur). Commençons donc par le tout premier script présenté ci-dessous.

Premier script PHP

Recopiez le code suivant dans un éditeur de texte (BlocNote, Wordpad ou Notepad++ font largement l'affaire) puis enregistrez ce fichier avec le

nom *hello_world_basic.php*.

Note : tous les fichiers comportant du code PHP doivent obligatoirement être enregistrés avec l'extension .php (ou .phpX où X est le numéro de version de PHP).

Premier programme PHP : le "Hello World"

```
<?php echo 'Hello World !'; ?>
```

Exécutez ce premier script dans un navigateur Web (Safari, Firefox, Opéra, Internet Explorer...). Vous constatez que le texte *Hello World !* s'affiche bien à l'écran. Nous obtenons donc le résultat escompté au départ. Passons aux explications.

Explication du code

Tout d'abord les balises. Tout script PHP doit être entouré par deux balises pour le délimiter d'un autre type de contenu se trouvant dans un même fichier (du code HTML par exemple). Ici nous utilisons les marqueurs *<?php* et *?>*. Il en existe d'autres mais ils sont fortement déconseillés à utiliser. Si vous souhaitez savoir quels sont-ils et pourquoi il ne faut pas les employer, nous vous invitons à consulter le tutoriel suivant : Pourquoi il est déconseillé d'utiliser les balises courtes (short-tags) ?.

Quoiqu'il en soit, vous devez toujours utiliser les marqueurs de ce premier programme. C'est LA première bonne pratique à adopter quand on code en PHP.
La seconde partie du code correspond à ce que l'on appelle en programmation **une instruction**. La fonction **echo()** (ou plutôt la structure de langage car c'est une fonction particulière de PHP) se charge d'écrire ce qu'on lui passe en paramètre sur la sortie standard. Ici le paramètre est une chaîne de caractère (type) dont la valeur est *« Hello World ! »*.
Notion importante à retenir : le script PHP est exécuté sur le serveur et le résultat de cette exécution qui est renvoyé (ici du code html) est interprêté par le navigateur Web.

Amélioration du Hello World

Jusque là rien de difficile. Passons alors à un niveau supérieur. Nous allons générer notre *Hello World !* au milieu d'un document HTML. Voici le code du fichier *hello_world_avance1.php* :

Génération d'un document HTML minimal

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Premier programme PHP !</title>
  </head>
  <body>
    <?php
```

```
      echo 'Hello World !';
    ?>
  </body>
</html>
```

Premier programme PHP !
Après exécution de ce fichier, on constate que le résultat à l'écran est exactement le même que précédemment. Oui mais uniquement à l'oeil nu ! Ici nous avons généré notre chaîne de caractères à l'intérieur de code HTML. Le principe de page dynamique commence donc à se faire sentir à partir de là. En effet, PHP va nous permettre de générer des pages à partir de modèles et de paramètres qu'on lui fournit.
Admettons que nous souhaitions afficher notre *Hello World !* en gras. Deux choix s'offrent à nous :

1. On place les balises et de part et d'autre du script.
2. On place les balises et directement dans l'instruction echo().

Quoiqu'il en soit le résultat produit sera le même. Mais le second exemple (voir code ci-dessous) vous montre alors qu'il est possible de générer du code HTML pour construire une page. L'intérêt de PHP devient alors évident et nous laisse imaginer toutes les possibilités qui s'offrent à nous par la suite. Par exemple : générer des tableaux HTML, des listes, des liens, des paragraphes, des documents XML...

Un "Hello World" en gras

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Premier programme PHP !</title>
  </head>
  <body>
    <?php
      echo '<strong>Hello World !</strong>';
    ?>
  </body>
</html>
```

Le code ci-dessous aura pour effet d'écrire à l'écran : **Hello World !**

Conclusion

Nous venons de voir, dans ce premier tutoriel des bases du langage PHP comment :

- intégrer du code PHP dans une page web.
- afficher du texte sur la sortie standard.
- générer du code HTML.

Dans les prochains cours, nous étudierons les différentes manières d'utiliser les chaînes de caractères, de déclarer des variables et des constantes, de tester des

conditions ou bien encore d'écrire des boucles... Mais chaque chose en son temps :-)

Les différents types de commentaires

Comme dans tout autre langage de programmation, PHP dispose de sa syntaxe de commentaires. Il en existe d'ailleurs plusieurs sortes que nous allons détailler. Les commentaires font partie des éléments triviaux dans la réussite d'un programme; et c'est pourquoi il est important de les utiliser avec intelligence.

Un commentaire, c'est quoi ?

Un commentaire, dans un langage de programmation, est une ligne écrite en langage naturel (langue maternelle du développeur par exemple) qui ne sera pas exécutée par l'interprêteur (ou le compilateur selon le langage employé). Sa fonction est de décrire ou bien d'expliquer une partie du code qui se révèlerai délicate à déchiffrer en cas de maintenance ou de travail collaboratif (plusieurs développeurs travaillant sur le même programme).

Les commentaires sont donc particulièrement utiles pour un développeur solitaire, mais ils le sont davantage lorsque c'est une équipe complète qui travaille sur un même projet. Ils permettent entre autre d'imposer des nomenclatures et une organisation dans l'écriture du code d'un projet collaboratif. De plus, les commentaires assurent une maintenance plus aisée du programme par son auteur ou une tierce personne.

Un autre point fort des commentaires est la ***génération de documentation*** technique. En effet, il existe des applications telles que PHPDocumentor qui s'appuie sur une syntaxe particulière des commentaires afin de générer la documentation d'une application. Cela assure un gain de temps non négligeable pour une équipe de développement.
Commenter un code fait aussi partie des bonnes pratiques à adopter en programmation. Mais il ne faut tout de même pas entrer dans l'excès inverse où chaque instruction du code deviendrait commentée. La clarté et la lisibilité du programme en seraient alors atteintes.

Le commentaire linéaire

Il existe deux sortes de commentaire. Le commentaire sur une seule ligne et le commentaire multilignes. Etudions ensemble les deux méthodes pour commenter un texte sur une seule ligne.

Les commentaires sur une seule ligne

```php
<?php

  // Ceci est un premier commentaire sur une ligne
  echo 'Hello World !';
```

```
  # Ceci est un second commentaire sur une ligne
  echo 'Bonjour le monde !';

?>
```

PHP propose deux manières de commenter un texte placé sur une ligne. La méthode la plus employée est celle du premier exemple avec le double slash (//) contrairement à la seconde utilisant un signe dièse (#).

Le commentaire multilignes

Il permet de commenter un texte écrit sur plusieurs lignes. Il est très fréquemment utilisé par les développeurs. Ces commentaires sont définis au moyen des symbôles **/*** et ***/**. L'exemple ci-après illustre leur emploi.

Les commentaires multilignes

```
<?php
  /*
    Ce programme a été écrit par Emacs

    Il affiche la chaîne 'Hello Word !' à l'écran
  */
  echo 'Hello World !';
?>
```

Conclusion

Nous avons défini ce qu'est un commentaire et comment il s'emploie selon qu'il est linéaire ou multilignes. Il faut donc penser à **toujours** les utiliser dans vos programmes afin d'assurer une relecture et une maintenance facilitées de votre code.

Les constantes

En programmation, il est souvent nécessaire de définir des structures de données dont la valeur ne doit pas changer au cours de l'exécution du programme. Ces structures de données sont typiquement ce que l'on appelle ***des constantes***. La plus connue des constantes mathématiques est par exemple le nombre PI dont la valeur est approximativement *3.1415926535898*. Notons que PHP intègre nativement la constante M_PI. Celle-ci pourra alors être utilisée pour tout calcul de circonférence par exemple. Commençons par étudier comment on déclare une constante dans un programme PHP.

La fonction native define()

La déclaration de constante se prépare au moyen de la fonction native define() qui prend trois paramètres dont les deux premiers sont obligatoires. Voici le détail de la documentation officielle de la fonction define() :

Prototype de la fonction define()

```
bool define ( string $name, mixed $value [, bool $case_insensitive] )
```

- Le premier paramètre de la fonction est une chaine de caractères qui définit le nom de la constante.Par convention, toutes les constantes doivent être écrites en majuscules.
- Le second est la valeur que l'on affecte à cette constante. Cela peut-être une chaîne de caractères, un entier, un réel ou bien encore un booléen.
- Enfin, le troisième argument est facultatif. C'est un booléen qui indique si l'interprêteur doit se soucier de la casse ou non de la constante.

Nous conseillons de ne jamais renseigner ce troisième argument afin d'imposer une rigueur dans l'écriture du code. Les deux premiers obligatoires suffisent. Passons à la déclaration de notre première constante.

Déclaration d'une constante

Comme tout le monde le sait (ou presque), l'eau bout à une température de 100°C (en théorie). Nous allons donc déclarer une première constante de nom **TEMPERATURE_EBULLITION_EAU** et qui renferme une information numérique de valeur **100**.

Déclaration d'une constante

```
<?php
  // Déclaration de la constante
  define('TEMPERATURE_EBULLITION_EAU',100);

  // Affichage de sa valeur
  echo 'L\'eau bout à ', TEMPERATURE_EBULLITION_EAU ,'°C';
?>
```

Si l'on teste ce script, on constate que la chaîne de caractères *« L'eau bout à 100°C »* s'affiche correctement à l'écran, ce qui prouve que la constante a bien été déclarée et initialisée à la bonne valeur.
Notes :

[1] A la déclaration, si la valeur est un nombre, on l'inscrit telle qu'elle dans la fonction.
[2] Pour lire la valeur d'une constante, il suffit de l'appeller par son nom.
D'un point de vue chimique, l'eau est aussi une molécule constituée de deux atomes d'hydrogène auxquels sont associés un atome d'oxygène. Sa formule chimique s'écrit donc **H2O**. Nous allons donc créer une nouvelle constante de nom **FORMULE_EAU** représentant une chaîne de caractères de valeur**H2O**.

Déclaration d'une nouvelle constante

```
<?php
  // Déclaration de la constante
  define('FORMULE_EAU','H2O');

  // Affichage de sa valeur
```

```
  echo 'Formule chimique de l\'eau : ', FORMULE_EAU;
?>
```

Note : à la déclaration, si la valeur est une chaîne de caractères, il faut l'entourer d'apostrophes ou de guillemets doubles.

Redéfinition d'une constante

La valeur d'une constante ne peut être redéfinie ! Des erreurs de syntaxe seront retournées en cas de tentative de rédéfinition de constante ou d'affectation d'une nouvelle valeur. Les deux scripts suivants présentent respectivement ces deux cas.

Erreurs générées en cas de redéfinition d'une constante

```
[Erreur générée] Notice: Constant TEMPERATURE_EBULLITION_EAU
already defined in /Applications/MAMP/htdocs/Tests-
PHP/constante.php on line 6

[Erreur générée] Parse error: syntax error, unexpected '=' in
/Applications/MAMP/htdocs/Tests-PHP/xml.php on line 6
```

Notons le signe **=** permettant l'affectation d'une valeur à une **variable** et non une constante. Pour plus d'information sur les variables, nous vous invitons à lire le tutoriel au sujet des variables.

Conclusion

Au terme de ce tutoriel, nous avons défini ce qu'est une constante et à quoi elle sert dans un programme informatique. De plus, nous avons présenté la fonction define() qui permet de déclarer des constantes. Enfin nous avons appris à lire le contenu d'une constante en l'appellant par son nom. Nous sommes prêts pour passer au tutoriel suivant concernant les variables, le pendant des constantes.

Les variables

Parmi les concepts les plus importants de la programmation figure la *notion de variable*. C'est une notion à assimiler et maîtriser pour pouvoir entreprendre les

premières applications de base. Nous verrons que les variables font partie des mécanismes qui permettent de rendre une application dynamique. Entrons dans le vif du sujet.

Qu'est-ce qu'une variable ?

Une variable est une structure de données de ***type primitif*** (entier, réel, caractère, chaîne de caractères, booléen ou null) ou bien de ***type structuré*** (tableau ou objet) qui permet de stocker une ou plusieurs valeurs. Chaque valeur d'une variable est susceptible d'être remplacée par une autre au cours de l'exécution du programme. D'où le terme « variable ». En programmation, une variable est définie suivant 4 informations essentielles listées ci-après :

- Un nom
- Un type
- Une / des valeurs
- Une sémantique (un sens) après les opérations effectuées sur cette variable. Plus concrètement, la valeur de la variable est-elle logique par rapport à son contexte initial ?

Schématiquement, une variable peut être assimilée à une boîte opaque sur laquelle est collée une étiquette portant un nom (le nom de la variable). A l'intérieur de ce paquet se trouve quelque chose (la valeur) d'un type précis (le type de la variable). Quand nous souhaitons obtenir cette valeur, nous ne nous adressons pas directement à elle mais nous y faisons référence en sélectionnant la boîte par son nom.

Prenons un exemple plus parlant. Un médecin souhaite obtenir une information concernant l'état de santé d'un de ses patients (*Mr Charles Dupont*). Pour l'obtenir, il devra demander à sa secrétaire de lui sortir le dossier *Charles Dupont* dans lequel se trouve cette information qu'il ne connait pas !

Déclaration et initialisation d'une variable

Après cette introduction plutôt théorique, nous entamons la partie pratique du sujet. Il s'agit de la déclaration et de l'initialisation d'une variable en langage PHP. Contrairement à des langages très typés comme C, C++ ou Java; PHP ne porte pas d'importance au typage des variables. Par exemple, pour une même variable, le programmeur est libre de lui affecter une valeur de type entier à un instant T1 puis de lui affecter une chaine de caractères à un instant T2. On dit que PHP est un langage de typage « faible et dynamique ». Cela rend son utilisation plus souple par les développeurs mais pas forcément plus assidue... C'est d'ailleurs quelque chose que l'on peut reprocher à PHP.
Afin d'adopter de bonnes pratiques dès le début, nous vous conseillons de déclarer toute vos variables avec un type et leur *affecter* une valeur par défaut. Retenez le terme *d'affectation* au passage.
Par ailleurs, il est nécessaire de nommer les variables avec des noms évocateurs afin de faciliter la lecture du code. Une variable mal nommée sera source de problème lors d'un débugging ou d'une maintenance du programme des semaines plus tard.

Toute variable doit être déclarée au moyen du signe dollars **$** suivi obligatoirement de lettres (en majuscules ou minuscules) ou d'un underscore (tiret souligné _). Ci-dessous, un tableau récapitulatif des syntaxes correctes et incorrectes de déclaration de variables.

Correct	Incorrect	Explications
$variable	variable	Une variable doit commencer par $
$Variable1	$Variable 1	Les espaces sont interdits
$variable_suite	$variable-suite	Le tiret est interdit
$_variable	$-variable	Le tiret est interdit
$variable2	$2variable	Il ne peut y avoir de chiffre après le $

Note : PHP est l'un des rares langages de programmation acceptant les caractères accentués dans les noms de variables. Cependant les employer est fortement déconseillé pour des raisons de maintenance et de portabilité.
Le script suivant présente la déclaration de 6 variables de types différents. La première et la seconde sont de type chaine de caractères, la troisième de type entier, la quatrième de type booléen, la cinquième de type tableau et la dernière de type Etudiant. Nous n'aborderons pas ces deux derniers dans ce tutoriel. D'autres tutoriels sont dédiés aux tableaux et à la programmation orientée objet.

Déclaration de variables de types différents

```
<?php
  $prenom = 'Hugo';  // Type string (chaine de caractères)
  $nom = "Hamon";    // Type string (chaine de caractères)
  $age = 19;    // Type entier
  $estEtudiant = true;  // Type booléen
  $cours =
array('physique','chimie','informatique','philosophie'); //
Type tableau
  $unEtudiant = new Etudiant(); // Objet de type Etudiant
?>
```

Pour déclarer une variable de type :

- *string :* on entoure la chaîne de caractères de guillemets ou d'apostrophes.
- *entier, réel ou flottant :* on écrit la valeur telle qu'elle. Pour les flottants, la virgule est remplacée par un point (écriture à l'américaine).
- *booléen* : on écrit *true* ou *false* directement.
- *sans type* : si l'on ne souhaite pas typer la variable, on lui affecte la valeur *null*.

Note : *le type d'une variable n'est pas déclaré explicitement comme en Java, C ou encore C++ mais implicitement au moment de l'affectation d'une valeur.*

Les noms de variables sont sensibles à la casse, ce qui signifie que l'interprêteur fera la différence entre deux variables écrites différemment. Par exemple, $nom et $NOM seront considérées comme deux variables complètement distinctes.

Par convention, les noms de variables composés de plusieurs mots (exemple : $estEtudiant) doivent avoir le premier mot en minuscules et les autres mots avec la première lettre en majuscule. Cette règle n'est pas à obligatoirerent à suivre mais fortement conseillée dans la mesure où elle fait partie des bonnes pratiques de programmation.

Note : il est recommandé de placer la directive error_reporting du php.ini à E_ALL sur toutes vos pages afin de faire remonter toute erreur générée. Les variables mal déclarées ou non initialisées peuvent être source d'erreur d'analyse.

L'affectation de valeur à une variable

L'affectation permet de fixer une valeur à une variable. Elle s'effectue au moyen du symbole égal **=**. Dans le script précédent, nous avons déclaré 6 variables et leur avons affecté à chacune une / plusieurs valeur(s) par défaut.
L'exemple ci-dessous montre comment changer la valeur d'une variable. Rien n'est compliqué puisque c'est exactement la même chose que précédemment.

```php
<?php
  $prenom = 'Hugo';
  $age = 19;

  echo $prenom;         // Affiche 'Hugo'
  echo '<br/>';
  echo $age;         // Affiche 19

  $prenom = 'Hadrien';
  $age = 18;

  echo $prenom;         // Affiche 'Hadrien'
  echo '<br/>';
  echo $age;         // Affiche 18

?>
```

Explications : nous avons deux variables initialisées *$prenom* et *$age* Nous affichons leur valeur pour les contrôler. Puis nous actualisons les variables en leur affectant à chacune une nouvelle valeur. Enfin, nous affichons les nouvelles valeurs. Nous remarquons alors que nous avons perdu les valeurs originales. Une affectation entraîne donc l'écrasement de l'ancienne valeur par la nouvelle.

La concaténation, c'est quoi ?

Sous ce terme un peu « barbare » se cache un principe fondamental lié des variables. La concaténation n'est ni plus ni moins que l'opération permettant d'assembler deux ou plusieurs informations dans une variable. Cette opération se réalise au moyen de l'opérateur de concaténation qui est le point (**.**). Illustrons cela avec un exemple :

La concaténation

```php
<?php
  // Déclaration des variables
  $prenom = 'Hugo';
  $nom = 'Hamon';
  $identite = '';
  // On concatène $nom et $prenom dans $identite
  $identite = $prenom .' '. $nom;
  // Affiche 'Hugo Hamon'
  echo $identite;
?>
```

Nous avons placé ici dans la variable $identite, le contenu de la variable $prenom suivi d'un espace par concaténation et enfin le contenu de la variable $nom.

Nous aurions également pu procéder de la manière suivante, ce qui nous évite d'employer une nouvelle variable $identite.

Autre exemple de concaténation

```php
<?php
  // Déclaration des variables
  $prenom = 'Hugo';
  $nom = 'Hamon';
  // On concatène $nom dans $prenom
  $prenom .= ' '. $nom;
  // Affiche 'Hugo Hamon'
  echo $prenom;
?>
```

La syntaxe de concaténation ci-dessus signifie que l'on ajoute l'espace et le contenu de la variable $nom à la suite du contenu de la variable $prenom. Ainsi, cette syntaxe est similaire à celle-ci :

Opérations mathématiques sur les variables

Les variables peuvent contenir des nombres. Il est donc logique que nous puissions opérer mathématiquement sur ces valeurs. PHP propose une série d'opérateurs mathématiques assurant cela. Il est donc possible d'additionner, diviser, multiplier, ou encore soustraire les valeurs des variables.
Les exemples ci-dessous illustrent l'emploi de ces différents opérateurs mathématiques.

Opérations mathématiques sur les variables

```php
<?php
  // Déclaration des variables mathématiques
  $a = 10;
  $b = 2;
  $c = 0;
  // $c vaut 10+2 = 12
  $c = $a + $b;
```

```
  // $c vaut 10x2 = 20
  $c = $a * $b;
  // $c vaut 10/2 = 5
  $c = $a / $b;
  // $c vaut 10-2 = 8
  $c = $a - $b;
  // $c vaut le reste de la division de $a par $b soit 0
  $c = $a % $b;
  // Incrémentation de $a
  $a++;
  // Décrémentation de $b
  $b--;
?>
```

L'opérateur *modulo* (%) retourne le reste de la division de deux nombres. L'opérateur d'incrémentation augmente de 1 la valeur de la variable. La syntaxe $a++ est identique à $a+=1 et $a = $a+1. L'opérateur de décrémentation diminue de 1 la valeur de la variable. La syntaxe $b-- est identique à $b-= 1 et $b = $b-1.
Pour plus d'informations concernant les opérateurs mathématiques, nous vous conseillons de vous reporter au tutoriel « les opérateurs arithmétiques, logiques et binaires ».

Les variables dynamiques

Elles sont apparues avec la version 4 de PHP et on les appelle aussi « variable variable ». Concrètement, une variable dynamique repose sur le fait que la valeur d'une variable devienne à son tour une variable. Prenons un exemple :

Utilisation des variables dynamiques

```
<?php
  // Définition des variables
  $poupee = 25;
  $voiture = 17;
  $console = 250;
  // Choix d'un jouet
  $jouet = 'console';
  // Affichage du prix du jouet sélectionné
  echo $jouet ,' : ', $$jouet ,'€';  // Retourne 'console : 250€'
?>
```

Dans cet exemple, nous déclarons 3 variables qui représentent les prix de 3 jouets (une poupée, une voiture et une console de jeux vidéo). Puis nous déclarons une variable $jouet dans laquelle on stocke le nom du jouet pour lequel on désire connaître le prix. Ici nous avons décidé de savoir le prix que vaut la console vidéo. Enfin, au moyen d'une instruction echo(), on affiche le jouet sélectionné (console vidéo) et son prix en faisant une variable variable. Ainsi, $jouet = 'console' mais $$jouet est similaire à $console. En effet, la valeur de $jouet est devenue le nom de la variable.

Note : *Bien que leur principe de fonctionnement est intéressant, il est vivement recommandé de ne pas les utiliser. Les tableaux, que nous verrons dans le chapitre suivant, remplissent cette fonction de variables dynamiques.*

Test de l'existence d'une variable

Dans un programme, il est très souvent utile de savoir si une variable existe avant de pouvoir l'exploiter. Le langage PHP met à disposition des développeurs la fonction (même structure du langage) isset() qui permet de savoir si oui ou non la variable passée en paramètre existe. Cette fonction retourne un booléen TRUE ou FALSE. Illustrons ceci avec un exemple :

Test d'existence de variables dans un programme

```php
<?php
  // Déclaration de la variable
  $prenom = 'Hugo';

  echo isset($prenom);  // Retourne TRUE -> affiche 1
  echo '<br/>';
  echo isset($nom);  // Retourne FALSE -> n'affiche rien
?>
```

Dans notre exemple, $prenom est bien déclarée donc la fonction isset() retourne TRUE. Quant à la variable $nom, elle n'est pas déclarée dans le programme donc la fonction isset() retourne FALSE.

Note : nous conseillons plutôt d'utiliser la fonction empty() qui permet de savoir si une variable est vide ou non. Ainsi en faisant ce test, elle permet de connaître deux choses :

- La variable existe ou non(elle est déclarée)
- La variable contient une valeur ou bien est vide (null par exemple)

Destruction d'une variable

Il est aussi parfois utile de détruire les variables qui encombrent le programme et qui ne servent plus. Par défaut, toutes les variables sont automatiquement effacées à la fin de l'exécution du programme mais il est néanmoins possible de forcer leur suppression en cours d'exécution. Pour se faire, PHP dispose de la fonction (ou plutôt la structure de langage) unset() qui prend en paramètre la variable à supprimer.

Suppression d'une ou plusieurs variables

```php
<?php
  // Déclaration des variables
  $prenom = 'Hugo';
  $nom = 'Hamon';
  $age = 19;
  $estEtudiant = true;
```

```php
  // Suppression d'une variable
  unset($prenom);
  // Suppression de plusieurs variables
  unset($nom, $age, $estEtudiant);
?>
```

Note : si l'on souhaite supprimer plusieurs variables d'un coup, il suffit de les lui préciser en les séparant par une virgule.

Conclusion

Ce cours nous a permis de définir les notions de variable, de déclaration, d'initialisation mais aussi les principes de concaténation, d'opérations mathématiques et de variables dynamiques. Nous sommes désormais à même de nous lancer dans des programmes plus complexes.

Les tableaux (ou arrays)

Nous avons principalement étudié dans les précédents cours les structures de données simples (constantes et variables). En plus de ces dernières, PHP propose des**types de données structurés** que l'on appelle plus communément des « *tableaux* ». A quoi servent-ils exactement ? Comment les manipule-t-on ? C'est ce dont nous allons étudier dans ce tutoriel.

Qu'est-ce qu'un tableau ?

Avant toute chose, il est bon de préciser qu'un tableau PHP et un tableau HTML sont deux choses complètement différentes. Un tableaux PHP a pour fonction de stocker et manipuler des informations tandis qu'un tableau HTML sert à présenter des données sur un écran.
Les tableaux, aussi appellés *arrays* en anglais, sont des types de données structurés permettant de grouper des informations ensemble. A la différence des types primitifs (entiers, réels, flottants, booléens, chaines de caractères), les tableaux peuvent stocker une ou plusieurs valeurs à la fois (de types différents).
Lors de la déclaration d'un tableau, il est inutile de préciser sa dimension et le type de données qu'il va contenir. PHP s'en charge automatiquement. Les tableaux sont dits *dynamiques*. A chaque nouvelle entrée enregistrée dans le tableau, PHP agrandit sa taille de 1 élément.
Le langage PHP propose également deux types distincts de tableaux : **les tableaux à index numériques** et **les tableaux associatifs**. Nous étudierons chacun de ces formats de tableaux plus loin dans ce cours.

Déclaration d'un tableau

La déclaration d'un tableau vide se fait de la même manière qu'une variable, c'est à dire avec un signe dollars ($) et un nom. Le format du nom doit respecter les mêmes règles de déclaration qu'une variable. Nous identifierons ensuite le tableau par le nom que nous lui avons attribué.

Pour déclarer un nouveau tableau, il suffit d'utiliser la structure de langage array(). Cette fonction prend en paramètres facultatifs (séparés par une virgule), les valeurs que l'on souhaite insérer dans le tableau pour l'initialiser. Si rien n'est précisé en paramètre, le tableau créé sera vide. Voici 3 exemples de déclaration et d'initialisation de tableaux.

Déclaration et initialisation de tableaux

```php
<?php

  // Déclaration d'un tableau vide
  $fruits = array();

  // Déclaration d'un tableau indexé numériquement
  $legumes = array('carotte','poivron','aubergine','chou');

  // Déclaration d'un tableau associatif
  $identite = array(

      'nom' => 'Hamon',
```

```
        'prenom' => 'Hugo',
        'age' => 19,
        'estEtudiant' => true
    );

?>
```

Explications :

- La première instruction crée un tableau vide appellé *$fruits*.
- La seconde déclare un tableau indexé numériquement de nom *$legumes* et rempli de 4 valeurs.
- Enfin le dernier tableau créé est un tableau associatif de nom *$identite* et composé de couples *clé => valeur*.

Ajout d'une nouvelle entrée dans un tableau

Pour ajouter une nouvelle valeur dynamiquement à la fin des tableaux précédents, il suffit de procéder comme expliqué dans l'exemple suivant :

Ajout d'élément dans un tableau

```
<?php

  // Ajout d'un légume au tableu indexé numériquement
  $legumes[] = 'salade';

  // Ajout de la taille de la personne dans le tableau associatif
  $identite['taille'] = 180;

?>
```

Explications :

- La première instruction ajoute dynamiquement la valeur 'salade' à la fin du tableau. Le tableau contient donc à présent : carotte, poivron, aubergine, chou, salade.
- La seconde instruction crée dynamiquement un nouveau couple clé ('taille') => valeur (180) à la fin du tableau.

Dans le cas du tableau indexé numériquement, il est aussi possible d'ajouter une valeur à un index précis en procédant de cette manière :

Ajout d'éléments dans un tableau à clés numériques

```
<?php

  // Ajout de légumes au tableu
  $legumes[12] = 'endive';
  $legumes[20] = 'piment';

?>
```

Explications :

- PHP agrandit dynamiquement le tableau $legumes et ajoute la valeur 'endive' à l'index 12.
- PHP agrandit encore dynamiquement le tableau $legumes et ajoute la valeur 'piment' à l'index 20.

Le tableau indexé numériquement

Un tableau indexé numériquement est tout simplement une liste d'éléments reperés chacun par un index numérique unique. Le premier élément du tableau sera reperé par l'index 0, le second par l'index 1, le troisième par l'index 2 et ainsi de suite.

Pour accéder à un élement du tableau, il suffit d'y faire référence de cette manière : **$tableau[0]**,**$tableau[1]**, **$tableau[2]**... Reprenons notre exemple précédent :

Lecture d'une valeur dans un tableau à clés numériques

```php
<?php

  // Déclaration d'un tableau indexé numériquement
  $legumes = array('carotte','poivron','aubergine','chou');
  // Ajout d'un légume au tableu
  $legumes[] = 'salade';
  // Affichage de l'aubergine
  echo $legumes[2];

?>
```

Le tableau associatif

Il est apparu pour pallier les faiblesses du tableau à index numériques. En effet pour ce dernier, il faut absolument connaître son emplacement pour atteindre la valeur et pour un programmeur ce n'est pas toujours le cas. De plus, une valeur repérée par un index à moins de sens que cette même valeur repérée par une clé chainée.

Un tableau associatif est un tableau composé de couples *clé chainée / valeur*. A chaque clé est référencée une valeur. Nous avons vu précédemment comment déclarer un tableau associatif et lui associer des valeurs référencées par des clés.
Pour accéder à l'une des valeurs du tableau, il suffit d'y faire référence de la manière suivante :**$tableau['cle']**. Dans notre exemple précédent, nous pourrions afficher l'identité de la personne de cette façon :

Lectures de valeurs dans un tableau associatif

```php
<?php

  // Affichage des valeurs du tableau associatif
  echo 'Nom : ', $identite['nom'] ,'<br/>';
  echo 'Prénom : ', $identite['prenom'] ,'<br/>';
  echo 'Age : ', $identite['age'] ,' ans<br/>';
```

```
  echo 'Taille : ', $identite['taille'] ,' cm';
?>
```

Cela aura pour effet d'afficher à l'écran :

Résultat de l'exécution du script

```
Nom : Hamon

Prénom : Hugo

Age : 19 ans

Taille : 180 cm
```

Note **:** *il est possible de mixer tableaux associatifs et tableaux indexés numériquement.*

Les tableaux multidimensionnels

Nous venons de voir comment créer des tableaux simples à une seule dimension. On les appelle aussi « vecteurs ». Mais il est églament possible de créer des tableaux à plusieurs dimensions. Ce sont **des tableaux de tableaux**. Prenons un exemple simple. Nous allons concevoir une *« matrice »* (tableau à 2 dimensions) représentant une partie gagnante d'un jeu de morpion. Un jeu de morpion se représente visuellement par un tableau de 3 lignes sur 3 colonnes. Notre matrice aura donc ses caractéristiques.

Exemple de création d'une matrice (tableau 2D)

```
<?php
  // Déclaration de la matrice
  $matrice = array();
  $matrice[0] = array('X','O','X');
  $matrice[1] = array('X','X','O');
  $matrice[2] = array('X','O','O');
?>
```

Explications :

Pour créer une matrice, nous devons mettre en place un tableau de tableaux. A chaque index numérique (ligne du tableau), nous associons un nouveau tableau de 3 cases (qui représente les 3 colonnes de la ligne). Si nous souhaitons accéder à la case du milieu du jeu de morpion, nous devons nous rendre à la ligne n°2 (index 1) et à la colonne n°2 (index 1). Ce qui donne :

Accès à la valeur de coordonnées (1,1)

```
<?php

  // Retourne X
  echo $matrice[1][1];
```

```
?>
```

Concrètement, pour accéder à une valeur d'une matrice, il faut y faire référence de cette manière :

Principe d'accès à une valeur d'une matrice

```
laMatrice[ numéroDeLigne ][ numéroDeColonne ]
```

Tableau particulier : la chaîne de caractères

Lorsque l'on déclare une variable stockant une chaine de caractères, nous faisons tout naturellement ceci :

Déclaration d'une chaine de caractères

```
<?php

  $chaine = 'Bonjour le monde !';
?>
```

Par cette syntaxe, PHP va en réalité déclarer un tableau indexé numériquement qui contient N cases de 1 caractère. Ainsi nous pourrons accéder directement à une lettre de la chaîne de caractères de cette manière :

Lecture d'un caractère dans le tableau de chaine de caractères

```
<?php

  $chaine = 'Bonjour le monde !';

  echo $chaine[3];  // Affiche la lettre 'j'
?>
```

En retenant ce concept astucieux, nous pourrons par la suite traiter les chaînes de caractères de façon plus aisée.

Parcours d'un tableau

Comme dans tout autre langage de programmation, le parcours de tableau se fait à l'aide de boucles. PHP dispose de sa propre structure de contrôle pour parcourir le contenu d'un tableau. Il s'agit de la structure foreach(). C'est une boucle particulière qui avance le pointeur du tableau à chaque itération. Celle-ci a été intégrée depuis la version 4 de PHP et se présente sous deux syntaxes possibles :

Parcours de tableaux avec foreach()

```
<?php



  // Affichage des valeurs d'un tableau
  foreach($leTableau as $valeur)
```

```
  {
    echo $valeur ,'<br/>';
  }

  // Affichage des couples clé / valeur
  foreach($leTableau as $cle => $valeur)
  {
    echo $cle ,' : ', $valeur ,'<br/>';
  }
?>
```

Cette structure prend en paramètre le nom du tableau à parcourir puis les données qu'il faut récupérer (valeurs uniquement ou bien valeurs et clés). Dans la première syntaxe, la valeur de l'élément courant du tableau est directement assignée à la variable $valeur. Dans la seconde, la clé courante de l'élément du tableau est affectée à la variable $cle et sa valeur stockée dans la variable $valeur.

Note : foreach() agit sur une copie du tableau d'origine.
Nous pourrions parcourir nos deux tableaux d'exemple ($legumes et $identite) en utilisant foreach(). Ainsi, nous obtiendrons le code suivant :

Parcours des tableaux $legumes et $identite

```
<?php
  // Affichage des légumes
  foreach($legumes as $valeur) {

    echo $valeur ,'<br/>';
  }

  // Affichage de l'identité de la personne
  foreach($identite as $cle => $valeur) {

    echo $cle ,' : ', $valeur ,'<br/>';
  }
?>
```

Remarquons également qu'il est tout à fait possible de parcourir un tableau au moyen d'une autre boucle. Prenons l'exemple de la boucle for() pour illustrer ce propos.

Parcours d'un tableau à indexes numériques contigus avec une boucle for()

```
<?php

  // Calcul de la taille du tableau $legumes
  $tailleLegumes = sizeof($legumes);

  // Parcours du tableau
  for($i=0; $i<$tailleLegumes; $i++)
  {
    echo $legumes[ $i ] ,'<br/>';
```

```
  }
?>
```

Cette méthode a néanmois un inconvénient. Dans le cas des tableaux associatifs ou des tableaux à indexes numériques non contigus, il nous est difficile d'obtenir la clé de l'élément courant parcouru. La solution est d'avoir recours à des fonctions particulières de manipulation des tableaux pour obtenir les clés.

Afficher le contenu d'un tableau

Lorsque l'on développe, il arrive très souvent que l'on veuille afficher le contenu d'un tableau dans le but de pouvoir débugguer un programme. Pour cela, PHP introduit la fonction print_r() qui assure cette fonction. Afin de respecter l'indentation à l'affichage, nous préfixons le résultat de cette fonction par les balises <pre> et </pre>. Le code qui suit affiche le contenu de notre tableau associatif $identite.

Affichage du contenu d'un tableau

```
<?php
  echo '<pre>';
  print_r($identite);
  echo '</pre>';
?>
```

Le résultat produit sera le suivant :

Résultat de l'exécution du script

```
Array
(
    [nom] => Hamon
    [prenom] => Hugo
    [age] => 19
    [estEtudiant] => 1
    [taille] => 180
)
1
```

Entre crochets figurent les clés et à droite des flèches les valeurs associées.

Opérations sur les tableaux

L'intérêt d'utiliser des tableaux pour structurer ses applications, c'est que cela permet ensuite d'opérer sur ces derniers. En effet, PHP propose une série de

fonctions natives capables de manipuler ces structures. En voici quelques unes fréquemment employées et particulièrement utiles :

- count() et sizeof() retournent toutes les deux la taille du tableau passé en paramètre.
- sort() trie les éléments d'un tableau du plus petit au plus grand.
- rsort() trie les éléments d'un tableau du plus grand au plus petit.
- in_array() permet de vérifier qu'une valeur est présente dans un tableau.
- array_rand() extrait une ou plusieurs valeurs du tableau au hasard.
- current() retourne la valeur de l'élément courant du tableau (où se trouve le pointeur)

Pour connaître toutes les fonctions relatives aux tableaux, nous vous invitons à parcourir le chapitre dédié à ces derniers de la documentation officielle de PHP : les tableaux / arrays

Conclusion

Nous sommes arrivés au terme de ce chapitre sur les tableaux. Nous avons passés en revue tous les aspects théoriques et pratiques importants qu'il faut assimiler. Nous avons défini ce que sont les tableaux et étudié les deux types de tableaux de PHP. Puis nous avons appris à concevoir des tableaux multidimensionnels avant de finir sur le parcours et les opérations de manipulation.

Les opérateurs

Le tutoriel qui va suivre n'est pas tout à fait un cours mais plutôt un mémo concernant les opérateurs PHP. Il en existe 12 types au total. Nous allons les passer en revue au moyen de tableaux de synthèse. Commençons tout d'abord par les opérateurs arithmétiques.

Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent de réaliser des opérations mathématiques sur les variables. Ce sont naturellement toutes les opérations conventionnelles telles que l'addition, la multiplication, la soustraction ou la division. Le tableau suivant résume toutes les opérations mathématiques possibles.

Opérateur	Opération	Exemple	Résultat
-	Négation	-$a	Opposé de $a
+	Addition	$a + $b	Somme de $a et $b
*	Multiplication	$a * $b	Produit de $a et $b
-	Soustraction	$a - $b	Différence de $a et $b
/	Division	$a / $b	Quotient de $a et $b
%	Modulo	$a % $b	Reste de $a / $b

Note *: l'opérateur modulo retourne le reste de la division entre deux nombres. Son utilisation permet par exemple de déterminer la parité d'un nombre ou bien de réaliser une alternance de couleurs des lignes d'un tableau HTML.*

Les opérateurs d'incrémentation / décrémentation

L'opérateur d'incrémentation (respectivement décrémentation) permet d'augmenter (respectivement diminuer) d'une unité la valeur de la variable. On les utilise

essentiellement dans les boucles pour mettre à jour la valeur du compteur à chaque itération.

Opérateur	Opération	Exemple	Résultat
++	Pré-Incrémentation	++$a	Incrémente $a, puis retourne $a
++	Post-Incrémentation	$a++	Retourne $a, puis incrémente $a
--	Pré-Décrémentation	--$a	Décrémente $a, puis retourne $a
--	Post-Décrémentation	$a--	Retourne $a, puis décrémente $a

Ces opérateurs peuvent être placés soit devant la variable ou bien après celle-ci. Cette nuance peut-être intéressante lors de l'utilisation de boucle.

En règle générale, les cas d'utilisation font que l'on emploie plus $a++ (respectivement $a--).

L'opérateur d'assignation

C'est sans doute l'opérateur le plus élémentaire et essentiel du langage PHP. C'est grâce à lui que l'on affecte une valeur à une variable, que l'on crée un tableau ou bien que l'on instancie une classe.

Opérateur	Opération	Exemple	Résultat
=	Assignation	$a = 3	Affecte la valeur 3 à $a

Les opérateurs de chaine de caractères

Ils sont au nombre de deux. Le premier est l'opérateur de concaténation réalisé par le point (.) tandis que le second est l'opérateur d'assignation concaténant par le .=

Opérateur	Opération	Exemple	Résultat
.	Concaténation	$a . $b	Concatène les valeurs de $a et $b
.=	Assignation concaténant	$a .= $b	Ajoute la valeur de $b à la suite de celle de $a

Les opérateurs de comparaison

Ils sont essentiellement utilisés dans les structures conditionnelles (if, elseif, else, for, while...) afin de comparer des valeurs entre elles. Ses tests renverront TRUE (vrai) si la comparaison est juste ou bien FALSE (faux) si la comparaison est fausse.

Opérateur	Opération	Exemple	Résultat
==	Egalité en valeur	$a == $b	Vérifie que les valeurs de $a et $b sont identiques
===	Egalité en valeur et type	$a === $b	Vérifie que les valeurs et types de $a et $b sont identiques
!=	Différence en valeur	$a != $b	Vérifie que les valeurs de $a et $b sont différentes
!==	Différence en valeur et type	$a !== $b	Vérifie que les valeurs et types de $a et $b sont différents
<>	Différence en valeur	$a <> $b	Alias de !=
<	Infériorité stricte	$a < $b	Vérifie que $a est strictement inférieur $b
<=	Infériorité ou égalité	$a <= $b	Vérifie que $a est strictement inférieur ou égal à $b
>	Supériorité stricte	$a > $b	Vérifie que $a est strictement supérieur $b
>=	Supériorité ou égalité	$a >= $b	Vérifie que $a est strictement supérieur ou égal à $b

Note : il ne faut pas confondre l'opérateur d'affectation qui s'emploie avec un seul signe égal (=) et qui permet de fixer la valeur d'une variable.
Lorsque l'on souhaite comparer à la fois les valeurs et les types des variables, on utilise l'opérateur === (respectivement !==). Cette vérification est conseillée pour deux raisons :

- On s'assure que les deux variables sont du même type.
- Cette opération est légèrement plus rapide à l'exécution que sa consoeur qui ne fait pas de vérification sur le type.

Illustrons la nuance entre les deux formes de comparaison :

Test de comparaison sur les variables

```php
<?php

  $a = '2';     // Type string
  $b = 2;       // Type entier
  $c = 2;       // Type entier

  // Comparaison sur les valeurs
  if($a == $b)
  {
    echo '$a et $b ont la même valeur !';
  }

  // Comparaison sur les valeurs et les types
  if($a === $c)
  {
    echo '$a et $c sont de même valeur et de même type !';
  }
?>
```

Ces deux exemples introduisent la notion de structures de contrôle par condition que nous étudierons au prochain tutoriel. Nous avons déclaré 3 variables de valeur identique 2. Seule $a est de type string (chaine de caractères) tandis que les deux autres sont de type entier (int). Lorsque l'on exécute ces deux tests, nous constatons que seul le premier affiche le message. C'est parce que le double signe égal ne s'assure pas que le type des deux variables est le même contrairement au triple signe égal.

En toute logique, une chaine de caractères ne devrait pas être "égale" à un entier. Ce serait comme comparer des torchons et des serviettes !!! On s'aperçoit ici d'une faiblesse de PHP en terme de typage des variables, et c'est donc pour cette raison que l'on parle de typage faible dans le langage PHP.
Nous vous conseillons donc dans la mesure du possible de vérifier vos valeurs à la fois sur les types et sur les valeurs.

***Astuce** : PHP introduit la fonction intval() qui permet de faire du transtypage (ou cast). Elle transforme par exemple un entier sous forme de chaine de caractères en entier naturel.*

Les opérateurs logiques

Les opérateurs logiques sont très souvent utilisés dans les structures de contrôle. Elles permettent de définir des expressions plus ou moins complexes qui renverront un booléen TRUE ou FALSE. Ci-dessous un tableau synthétique de ces opérateurs.

Opérateur	Opération	Exemple	Résultat

Opérateur	Opération	Exemple	Résultat
&&	ET	$a && $b	TRUE si $a ET $b sont vrais
AND	ET	$a AND $b	Alias de &&
||	OU	$a || $b	TRUE si $a OU $b est vrai
OR	OU	$a OR $b	Alias de ||
XOR	OU exclusif	$a XOR $b	TRUE si $a OU $b est vrai mais pas les deux
!	NON	!$a	TRUE si $a est faux

La différence entre AND et && (respectivement OR et ||) réside dans la priorité d'exécution. Les opérateurs && et || ont une priorité plus élevée par rapport à leur semblable litéral respectif.

Les opérateurs binaires (bitwise)

Ces opérateurs permettent de manipuler les bits dans un entier. Si les paramètres de gauche et de droite sont des chaines de caractères, l'opérateur de bits agira sur les valeurs ASCII de ces caractères. (extrait de la documentation officielle).

Opérateur	Opération	Exemple	Résultat
&	ET	$a & $b	Les bits positionnés à 1 dans $a ET $b sont positionnés à 1
|	OU	$a | $b	Les bits positionnés à 1 dans $a OU $b sont positionnés à 1
^	OU exclusif	$a ^ $b	Les bits positionnés à 1 dans $a OU dans $b mais pas dans les deux sont positionnés à 1
~	NON	$a ~ $b	Les bits qui sont positionnés à 1 dans $a sont positionnés à 0, et vice versa
<<	Décalage à gauche	$a << $b	Décale les bits de $a, $b fois sur la gauche (chaque décalage équivaut à une multiplication par 2)

Opérateur	Opération	Exemple	Résultat
>>	Décalage à droite	$a >> $b	Décalage des bits de $a, $b fois par la droite (chaque décalage équivaut à une division par 2)

Pour plus d'information concernant les opérateurs binaires, nous vous invitons à lire les exemples de la documentation officielle : les opérateurs binaires.

Les opérateurs combinés

Les opérateurs combinés sont des opérateurs mêlant opération arithmétique ou binaire avec une assignation. D'où la notion de combinaison. Le tableau suivant résume tous ces opérateurs.

Opérateur	Opération	Exemple	Résultat
+=	Addition	$a += 4	Ajoute 4 à la valeur de $a et stocke le résultat dans $a
-=	Soustraction	$a -= 4	Soustrait 4 à la valeur de $a et stocke le résultat dans $a
*=	Multiplication	$a *= 4	Multiplie par 4 la valeur de $a et stocke le résultat dans $a
/=	Division	$a /= 4	Divise par 4 la valeur de $a et stocke le résultat dans $a
%=	Modulo	$a %= 4	Calcule le reste de la division de $a par 4 et stocke le résultat dans $a
&=	ET binaire	$a &= $b	Egal à $a = $a & $b
|=	OU binaire	$a |= $b	Egal à $a = $a | $b
^=	XOR binaire	$a ^= $b	Egal $a = $a ^ $b
<<=	Décalage à gauche	$a <<= $b	Egal à $a = $a << $b

Opérateur	Opération	Exemple	Résultat
>>=	Décalagae à droite	$a >>= $b	Egal à $a = $a >> $b

En généralisant le tout, on obtient **$a (opérateur)= $b** qui est aussi équivalant à **$a = $a (opérateur) $b**.

L'opérateur de contrôle d'erreur

C'est l'opérateur arobase (@) qui permet de supprimer toutes les erreurs générées par une expression (fonction, variables, constantes...). On le place juste devant l'expression pour laquelle on souhaite masquer l'erreur qui est levée.

Opérateur	Opération	Exemple	Résultat
@	Erreur	@include('fichier.php')	Masque l'erreur générée par la fonction include()

Néanmoins cette pratique est vivement déconseillée car on ne devrait pas masquer une erreur générée mais au contraire il faut la traiter. Cependant il existe des cas très particuliers où l'emploi de cette technique se révèle utile. Par exemple, lorsque l'on utilise la fonctionfsockopen().

L'opérateur d'exécution

Cet opérateur, délimité par des apostrophes ou guillemets obliques, permet d'exécuter des commandes Shell. Il se comporte de la même manière que la fonction shell_exec(). Si celle-ci est désactivée ou si le safe mode est activé alors cet opérateur ne fonctionnera pas par mesure de sécurité (sur les hébergements mutualisés par exemple).

Opérateur	Opération	Exemple	Résultat
``	Commande Shell	`ifconfig`	Exécute la commande shell et retourne le résultat

Le résultat d'une commande shell exécutée peut-être restitué aussi bien sur la sortie standard que dans une variable. Illustration par un exemple issu de la documentation officielle de l'opérateur d'exécution.

Utilisation de l'opérateur d'exécution

```
<?php
  $output = `ls -al`;
```

```
  echo "<pre>$output</pre>";
?>
```

Les opérateurs sur les tableaux

Comme pour les chaines de caractères, PHP propose une série d'opérateurs permettant de manipuler les tableaux. Ces opérateurs sont les mêmes que les opérateurs de comparaison mais n'ont pas forcément la même fonction et la même signification.

Opérateur	Opération	Exemple	Résultat
+	Union	$a + $b	Union de $a et $b
==	Egalité	$a == $b	TRUE si $a et $b contiennent les mêmes paires clé / valeur
===	Identique	$a === $b	TRUE si $a et $b contiennent les mêmes paires clé / valeur dans le même ordre et de même type
!=	Inégalité	$a != $b	TRUE si $a et $b ne sont pas égaux
<>	Inégalité	$a <> $b	Alias de !=
!==	Non identique	$a !== $b	TRUE si $a et $b ne sont pas identiques

Pour plus d'informations au sujet des opérateurs sur les tableaux, nous vous invitons à consulter la documentation officielle : les opérateurs sur les tableaux.

Les opérateurs de type d'objet (instance of)

L'opérateur de type d'objet (*instanceof* = « instance de ») est utilisé en programmation orientée objet afin de déterminer si un objet, son parent et ses classes dérivées sont de même type ou non.

Opérateur	Opération	Exemple	Résultat
instanceof	Instance	$a instanceof B	TRUE si $a est une instance de la classe B

Prenons un exemple simple pour illustrer ce concept.

Utilisation de l'opérateur instanceof

```php
<?php

  // Déclaration de classes
  class Etudiant { }
  class Professeur { }

  // Instanciation d'un objet de type Etudiant
  $hugo = new Etudiant();

  // Test du type de l'objet
  if($hugo instanceof Etudiant) {

    echo 'Hugo est un étudiant !';
  }
    else
  {
    echo 'Hugo est un professeur !';
  }

?>
```

Dans notre exemple, on crée un objet *$hugo* de type *Etudiant* (= instance d'une classe *Etudiant*). Puis on teste si cet objet est de type *Etudiant* grâce à l'opérateur *instanceof*. Si le test renvoie TRUE alors *$hugo*est de type *Etudiant*, sinon il est de type *Professeur*.

La priorité des opérateurs

Comme en mathématiques, les opérateurs ont un ordre de priorité entre eux lorsqu'ils sont exécutés. Le tableau qui suit résume chacune des priorités de ces opérateurs.

Priorité	Opérateur
1	new
2	() et []
3	-- ++ et !
4	~ - (int) (float) (string) (array) (object) @
5	instanceof

Priorité	Opérateur
6	* / et %
7	+ - et .
8	<< et >>
9	< <= et >= >
10	== != et ===
11	&
12	^
13	|
14	&&
15	||
16	? et :
17	Affectation et opérateurs combinés
18	AND
19	XOR
20	OR

Conclusion

Ce tutoriel ou mémo s'achève ici. Nous avons fait le tour des opérateurs du langage PHP. Les opérateurs d'affectation, logique et de comparaison sont les trois qu'il faut connaître sur le bout des doigts car ce sont les plus utilisés.

Les structures de contrôle : les conditions

Ce chapitre présente 2 principes importants à assimiler : les condititions et les boucles. Ce premier concept est particulièrement important puisqu'il permet d'effectuer une série d'actions en fonction des conditions que l'on teste. Quant au second, il permet de répéter N fois une série d'actions, et ce dans des temps très courts (à l'échelle humaine). L'intérêt de l'arrivée de l'informatique est de pouvoir faciliter la vie des utilisateurs en exécutant des tâches que ces derniers ne peuvent remplir. Si l'on demande à une personne de compter de 0 jusqu'à 1 000, il lui faudra plusieurs minutes tandis qu'un ordinateur traitera cette opération en quelques millisecondes (voire quelques nanosecondes selon sa puissance de calcul). On comprend alors tout de suite les atouts de l'informatique.

Les structures conditionnelles

Une structure conditionnelle permet d'exécuter ou non une série d'instructions en fonction d'une condition d'origine (appellée aussi *expression* ou *prédicat*). Si le calcul de cette condition retourne TRUE alors le bloc d'instructions concerné est exécuté. Les expressions évaluées peuvent être plus ou moins complexes, c'est à dire qu'elles peuvent être constituées d'une combinaison d'opérateurs de comparaison, d'opérateurs logiques et même de fonctions.
Le langage PHP introduit 4 constructions conditionnelles : *if, elseif, else* et *switch*.

L'instruction conditionnelle if()

C'est la plus connue de toutes car on la retrouve dans tous les langages de programmation. Elle permet d'exécuter un bloc d'instructions uniquement si l'expression est vraie. Le mot clé *if()* signifie en anglais *si*. Sa syntaxe est la suivante :

Principe de fonctionnement de l'instruction if()

```
if(expression)
{
  bloc d'instructions;
}
```

Illustrons son fonctionnement avec un exemple simple.

Exemple d'utilisation de l'instruction if()

```
<?php
  // Déclaration d'une variable
  $vitesse = 60;

  // On teste la valeur de la variable
```

```
  if($vitesse > 50)
  {
    echo 'Je perds 1 point pour excès de vitesse !';
    echo 'Le policier me dit : "Roulez doucement maintenant
!"';
  }
?>
```

Le code ci-dessus modélise la vitesse d'un véhicule par une variable *$vitesse*. La condition quant à elle vérifie la vitesse du véhicule. Si cette valeur dépasse les 50 (Km/h), alors le conducteur est arrêté par les forces de l'ordre et perd un point sur son permis de conduire. Si sa vitesse est inférieure ou égale à 50 alors rien ne se passe car le conducteur n'est pas en infraction.

La clause else

La clause *else* (traduire par *sinon*), ajoutée après l'accolade fermante du bloc if(), permet de définir une série d'instructions qui seront exécutées si l'expression testée est fausse (renvoie FALSE). Reprenons notre exemple précédent pour illustrer son fonctionnement.

Utilisation de la clause else dans notre exemple précédent

```
<?php
  // Déclaration d'une variable
  $vitesse = 40;

  // On teste la valeur de la variable
  if($vitesse > 50)
  {
    echo 'Les policiers m\'arrêtent !';
    echo 'Je perds 1 point pour excès de vitesse !';
    echo 'Le policier me dit : "Roulez doucement maintenant
!"';
  }
    else
  {
    echo 'Les policiers me laissent suivre ma route !';
  }
?>
```

Nous avons ajouté la clause *else* dans notre code et actualisé la valeur de la variable *$vitesse* (40 Km/h). Lorsque l'on exécute ce code, la condition n'est pas franchie puisque 40 n'est pas supérieur à 50 donc on entre dans la clause *else* et on exécute la seule instruction echo().

L'instruction elseif()

Elle peut se traduire par *« ou si »*. Cette instruction se place obligatoirement après l'accolade fermante d'un bloc if(). Elle permet notamment d'éviter une imbrication

de blocs conditionnels if() et il est possible d'en cumuler plusieurs. Aggrémentons encore notre exemple avec cette instruction conditionnelle.

Exemple d'utilisation de l'instruction conditionnelle elseif()

```
<?php
  // Déclaration d'une variable
  $vitesse = 82;

  // On teste la valeur de la variable
  if($vitesse > 50 && $vitesse < 70)
  {
    echo 'Les policiers m\'arrêtent !';
    echo 'Je perds 1 point pour excès de vitesse !';
    echo 'Le policier me dit : "Roulez doucement maintenant
!"';
  }
    elseif($vitesse > 70 && $vitesse < 80)
  {
    echo 'Les policiers m\'arrêtent !';
    echo 'Je perds 2 points pour excès de vitesse !';
    echo 'Le policier me verbalise par une amende';
  }
    elseif($vitesse > 80 && $vitesse < 90)
  {
    echo 'Les policiers m\'arrêtent !';
    echo 'Je perds 3 points pour excès de vitesse !';
    echo 'Le policier me verbalise par une grosse amende !"';
  }
    elseif($vitesse > 90)
  {
    echo 'Les policiers m\'arrêtent !';
    echo 'Je perds 4 points pour excès de vitesse !';
    echo 'Le policier me retire mon permis !';
    echo 'Je finis à pied !';
  }
    else
  {
    echo 'Les policiers me laissent suivre ma route !';
  }
?>
```

Dans cet exemple, nous simulons la perte de points sur le permis de conduire en fonction de la vitesse. La vitesse maximale autorisée est toujours de 50 Km/h. Nous roulons à 82 Km/h, soit 32 Km/h de plus que la vitesse maximale autorisée. Nous allons donc entrer dans le second bloc d'instruction *elseif()* car 82 est compris entre 80 et 89 (90 exclus). Nous perdons alors 3 points sur notre permis de conduire et l'on récupère aussi une amende à payer.
D'un point de vue interprétation, PHP va tester une à une les conditions. Si la première est fausse, il passe à la seconde. Si la seconde est fausse, il passe à la troisième et ainsi de suite. S'il en rencontre une de vraie, alors il entre dedans et

exécute le bloc d'instructions. Si aucune n'est vraie, il finit par exécuté le bloc d'instructions de la clause *else*.

Syntaxe alternative sur une ligne

Note : dans le cas où nous n'avons qu'une seule instruction à executer par instruction conditionnelle, nous pouvons nous passer des accolades. Ainsi, nous écrirons par exemple :

Exemple d'utilisation des écritures d'instructions sur une seule ligne

```php
<?php
  // Déclaration d'une variable
  $vitesse = 82;

  // On teste la valeur de la variable
  if($vitesse > 50 && $vitesse < 70)
    echo 'Je perds 1 point pour excès de vitesse !';
  elseif($vitesse > 70 && $vitesse < 80)
    echo 'Je perds 2 points pour excès de vitesse !';
  elseif($vitesse > 80 && $vitesse < 90)
    echo 'Je perds 3 points pour excès de vitesse !';
  elseif($vitesse > 90)
    echo 'Je perds 4 points pour excès de vitesse !';
  else
    echo 'Je fais des appels de phares aux automobilistes !';
?>
```

Néanmoins, nous vous recommandons toujours d'utiliser les accolades qui rendent la relecture du code plus aisée.

L'instruction switch()

Il existe une autre alternative à la structure if() / elseif() / else ou bien aux imbrications de blocs if(). Elle se nomme *switch()* (traduire par *au cas où*). Cette instruction conditionnelle permet de tester toutes les valeurs possibles que peut prendre une variable. Voici sa syntaxe.

Syntaxe de l'instruction conditionnelle switch()

```
switch(expression)
{
  case valeur1 :
    instructions;
  break;

  case valeur2 :
    instructions;
  break;
```

```
  default :
    instructions;
  break;
}
```

Illustrons son utilisation avec un nouvel exemple. Le précédent n'étant pas adapté. Nous allons modélisé un jeu vidéo. Le joueur doit choisir son personnage (avatar).

Exemple d'utilisation de l'instruction conditionnelle switch()

```
<?php
  // Le joueur a choisi son personnage ('elfe')
  $personnage = 'elfe';

  // Test de la valeur du personnage
  switch($personnage)
  {
    case 'sorcier' :
      echo 'Vous avez choisi le sorcier !';
      echo 'Vous disposez de pouvoirs magiques.';
    break;

    case 'guerrier' :
      echo 'Vous avez choisi le guerrier !';
      echo 'Vous disposez d\'armes blanches.';
    break;

    case 'roi' :
      echo 'Vous avez choisi le roi !';
      echo 'Vous vivez dans un château entouré de gardes.';
    break;

    case 'elfe' :
      echo 'Vous avez choisi l\'elfe !';
      echo 'Vous pouvez voler et vous déplacer rapidement.';
    break;

    case 'ogre' :
      echo 'Vous avez choisi l\'ogre !';
      echo 'Vous avez beaucoup de force dans les bras.';
    break;

    default :
      echo 'Veuillez choisir votre personnage svp !';
    break;
  }
?>
```

Si la variable a une valeur non définie dans le switch(), alors un bloc *default* lance une suite d'instructions. Le mot-clé *case* définit la valeur à tester. Cette valeur succède ce mot-clé. Le mot-clé *break*, quant à lui, permet de sortir du bloc switch()

si l'on a exécuté un bloc d'instructions. Il est facultatif mais vivement conseillé dans la majorité des cas d'utilisation.
Il est également recommandé de toujours prévoir un bloc *default* pour s'assurer que l'on ait une action par défaut si la valeur de la variable n'est pas référencée dans le switch(). Cela pourrait vous éviter bien des surprises.

Conclusion

Ce cours nous a permis d'apprendre à utiliser les principales instructions conditionnelles utiles en programmation PHP. Il s'agit des blocs d'instruction if(), elseif(), else et switch(). PHP inclut également d'autres structures de contrôle qui sont les boucles. Nous les étudierons dans le chapitre suivant.

Les structures de contrôle : les boucles

Ce tutoriel fait suite à celui concernant les instructions conditionnelles. Les boucles constituent un principe trivial de l'informatique que tout développeur se doit de maîtriser. Elles font, elles aussi, partie des structures de contrôle. Le langage PHP en compte 4 ayant chacune ses spécificités : *for()*, *while()*, *do-while()* et *foreach()*.

Qu'est-ce qu'une boucle ?

On appelle « *boucle* » une structure de contrôle capable d'exécuter un certain nombre de fois une même série d'instructions, en fonction d'une ou plusieurs conditions à vérifier.
Il faudra toujours vérifier que les conditions deviennent fausses (FALSE) à un moment donné afin de pouvoir sortir de la boucle. Dans le cas contraire, la boucle deviendra infinie et fera planter le programme tant que le max_execution_time du php.ini ne sera pas atteint.

La boucle for()

C'est la plus connue des boucles car elle est présente dans la majorité des langages de programmation. Paradoxalement, ce n'est pas forcément la plus utilisée en programmation PHP. Pourquoi ? Dans la plupart des langages de programmation, l'instruction for() sert à parcourir un tableau, alors que PHP dispose de sa propre instruction de contrôle *foreach()* spécialement dédiée pour cette tâche. Nous l'avions déjà évoquée dans le tutoriel sur les tableaux mais nous y reviendrons plus loin dans ce cours.
La particularité de la boucle *for()* (traduire « *pour chaque itération* ») est qu'il faut connaître par avance la condition d'arrêt. Autrement dit, la valeur qui rendra la condition fausse et stoppera la boucle. Sa syntaxe est simple et prend 3 paramètres obligatoires :

Syntaxe de la boucle for()

```
for(initialisation; condition; incrémentation)
{
  bloc d'instructions;
}
```

- L'*initialisation* est l'expression qui permet d'initialiser la boucle (valeur de départ). Généralement, les boucles débutent à la valeur 0.
- Le second paramètre correspond à la *condition d'arrêt* de la boucle. Cette condition est recalculée à chaque itération (passage de boucle) afin de déterminer si l'on continue de boucler ou bien si l'on sort de la boucle.

- Enfin, le dernier paramètre détermine l'expression qui sera exécutée à la fin d'une itération. Généralement on prévoit ici une incrémentation (ou décrémentation) pour mettre à jour le compteur de la boucle.

L'exemple qui suit montre la génération de la table de multiplication du chiffre 9.

Génération de la table de multiplication du chiffre 9 avec for()

```
<?php

  // Boucle générant la table de multiplication du 9
  for($i=0; $i<=10; $i++)
  {
    echo '9 x ', $i ,' = ', (9*$i) ,'<br/>';
  }
?>
```

Explications :

- Nous initialisons notre boucle à 0. Ce sera la première valeur qui sera multipliée au chiffre 9.
- Nous arrêtons la boucle lorsque le compteur $i atteindra la valeur 11. La valeur 10 passera mais quand $i vaudra 11 alors le programme sortira de la boucle.
- Nous déterminons l'action à exécuter à la fin de chaque itération. Ici on demande d'incrémenter automatiquement le compteur $i en écrivant $i++; (Rappel : $i++ <=> $i = $i+1).
- Nous affichons les 10 valeurs de la table de multiplication du 9. Le programme calcule chaque valeur en faisant le produit du 9 par le compteur $i.

***Note** : la variable $i est totalement arbitraire. Cette lettre est utilisée par convention (i pour indice) mais rien n'empêche d'écrire $compteur par exemple.*

La boucle while()

La boucle *while()* (traduire par *« tant que »*) signifie que l'on va répéter un bloc d'instructions tant que la condition passée en paramètre reste vraie (TRUE). Lorsque celle-ci deviendra fausse (FALSE), le programme sortira de la boucle.
Sa syntaxe présentée ci-dessous :

Syntaxe de la structure conditionnelle while()

```
while(condition)

{

  bloc d'instructions;

}
```

On l'utilise particulièrement lorsque l'on extrait des informations d'une base de données. La particularité de cette instruction est que la condition n'est testée qu'après l'exécution complète du bloc d'instructions. Si la condition devient fausse avant la fin, le reste des instructions sera tout de même exécuté.

Reprenons notre exemple précédent mais cette fois-ci avec une boucle *while()*.

Génération de la table de multiplication du chiffre 9 avec while()

```
<?php

  // Déclaration et initialisation du compteur
  $i = 0;
  // Boucle générant la table de multiplication du 9
  while($i<=10)
  {
    // Affichage de la nouvelle ligne
    echo '9 x ', $i ,' = ', (9*$i) ,'<br/>';
    // Incrémentation du compteur
    $i++;
  }
?>
```

Explications :

- Contrairement à la boucle *for()*, nous devons déclarer et initialiser notre compteur avant d'entrer dans la boucle. La valeur par défaut reste la même, à savoir 0.
- Puis nous déterminons la condition d'arrêt. Lorsque $i vaudra 11 alors nous sortirons de la boucle. La conditon **$i<=$10** est toujours vraie tant que $i n'atteint pas la valeur 11.
- Enfin dans le corps de la boucle, nous calculons et affichons le produit de 9 par le compteur $i avant de finir par une incrémentation du compteur.

La boucle do { ... } while()

L'instruction do {...} while() (traduire par *« répéter / faire … tant que »* est une alternative à l'instruction*while()* qui permet de tester la condition qu'après la première itération et exécution du premier bloc d'instructions. Voici sa syntaxe :

Syntaxe de la structures de contrôle do - while()

```
do

{

  bloc d'instructions;

}

while(condition);
```

Avec notre exemple, cela donnerait :

Génération de la table de multiplication du chiffre 9 avec do - while()

```
<?php
```

```
  // Déclaration et initialisation du compteur
  $i = 0;
  // Boucle générant la table de multiplication du 9
  do
  {
    // Affichage de la nouvelle ligne
    echo '9 x ', $i ,' = ', (9*$i) ,'<br/>';
    // Incrémentation du compteur
    $i++;
  }
  while($i<=10);
?>
```

Explications :

- Nous commençons par initialiser le compteur à la valeur 0.
- Puis nous définissons la liste d'instruction à exécuter. C'est à dire, calcul, affichage du produit et incrémentation du compteur.
- Nous terminons par le test de la condition permettant de sortir ou non de la boucle.

Remarque : si nous avions initialisé le compteur $i à une valeur strictement supérieure à 10 (11, 12, 13...), alors le programme aurait calculé et affiché le produit de 9 par cette valeur puis serait sorti de la boucle car la condition est fausse dès le début.

La boucle foreach

Dans la plupart des langages de programmation, le parcours de tableau se réalise à l'aide de boucles. PHP dispose quant à lui de sa propre structure de contrôle permettant de parcourir le contenu d'un tableau. Il s'agit de la structure foreach(). C'est une boucle particulière qui avance le pointeur du tableau à chaque itération. Celle-ci a été intégrée depuis la version 4 de PHP et se présente sous deux syntaxes possibles :

Syntaxe de la structure de contrôle foreach()

```
<?php

  // Affichage des valeurs d'un tableau
  foreach($leTableau as $valeur)
  {
    echo $valeur ,'<br/>';
  }

  // Affichage des couples clé / valeur
  foreach($leTableau as $cle => $valeur)
  {
    echo $cle ,' : ', $valeur ,'<br/>';
  }
?>
```

Cette structure prend en paramètre le nom du tableau à parcourir puis les données qu'il faut récupérer (valeurs uniquement ou bien valeurs + clés). Dans la première syntaxe, la valeur de l'élément courant du tableau est directement assignée à la variable $valeur. Dans la seconde, la clé courante de l'élément du tableau est affectée à la variable $cle et sa valeur stockée dans la variable $valeur.

Note : foreach() agit sur une copie du tableau d'origine.
Prenons deux tableaux d'exemple. L'un indexé numériquement et l'autre associativement. Nous allons les parcourir tous les deux au moyen de deux instructions foreach() de syntaxe différente.

Parcours de tableaux avec foreach()

```php
<?php

  // Déclaration des deux tableaux
  $legumes =
array('salade','oignon','piment','carotte','aubergine');
  $couleurs = array('rouge'=>'#ff0000', 'vert' => '#00ff00',
'bleu' => '#0000ff');

  // Affichage des légumes
  foreach($legumes as $legume)
  {
    echo $legume ,'<br/>';
  }

  // Affichage des couleurs et leur code hexadécimal
  foreach($couleurs as $couleur => $codeHexadecimal)
  {
    echo $couleur ,' : ', $codeHexadecimal ,'<br/>';
  }
?>
```

Les instructions d'arrêt et de continuité

PHP introduit également deux instructions particulières des boucles. Il s'agit de continue et break. La première permet de forcer le passage à l'itération suivante en sautant tout ou partie du bloc d'instructions à exécuter. La seconde, quant à elle, permet de forcer à quitter une structure conditionnelle telle que *for(), while(), foreach()* ou *switch()*.
Ces deux instructions peuvent prendre un paramètre optionnel de type entier permettant de connaître le nombre de structures emboîtées qui ont été interrompues.

Illustrons brièvement ces deux concepts à l'aide d'exemples simples.

Exemple d'utilisation de continue

```php
<?php

  for($i=0; $i<=10; $i++)
```

```
  {
    // On n'affiche pas les 5 premiers nombres
    if($i<=5)
    {
      continue;
    }
    // Affichage des nombres de 6 à 10
    echo $i ,'<br/>';
  }
?>
```

Dans cet exemple, nous souhaitons commencer à itérer depuis la valeur 0 mais nous ne voulons afficher que les valeurs comprises de 6 à 10. Grâce à la condition et au mot-clé *continue*, nous parvenons à ne pas afficher les 6 premières valeurs.
Remarque : nous aurions pu obtenir le même résultat sans utiliser le mot-clé *continue* en remplaçant la condition du *if()* par **$i > 5** et en plaçant l'instruction *echo()* à la place du *continue*.

Exemple d'utilisation de break

```
<?php
  for($i=0; $i<=10; $i++)
  {
    // On affiche la valeur du compteur
    echo 'Compteur = ';
    // On sort de la boucle si on atteint le chiffre 5
    if($i>5)
    {
      break;
    }
    // Affichage des nombres de 0 à 5
    echo $i ,'<br/>';
  }
?>
```

Cet exemple illustre l'emploi du mot-clé *break*. Ici nous demandons de sortir de la boucle dès que l'on atteint le chiffre 5. Pour nous convaincre du résultat, voici le résultat généré :

Résultat de l'exécution du script

```
Compteur = 0

Compteur = 1

Compteur = 2

Compteur = 3

Compteur = 4

Compteur = 5
```

```
Compteur =
```

Nous remarquons ici que nous sommes entrés dans la 7ème itération. Le programme exécute la première instruction puis passe le test. La condition est vraie donc l'instruction *break* est exécutée, ce qui force la sortie de la boucle.

Conclusion

Cette seconde partie du cours au sujet des structures de contrôle a été l'occasion de découvrir le concept de boucle. Nous avons donc étudié les structures itératives *for(), while(), do-while()* et *foreach()* avant de terminer par les deux instructions particulières *continue* et *break*.

Les procédures et fonctions utilisateurs

Comme dans la plupart des langages de programmation, il est possible en PHP de créer ses propres fonctions qui viennent compléter les fonctions natives. Cela permet de remplir des opérations particulières qui sont redondantes dans l'application, et de résoudre un problème en le fragmentant en plusieurs petits problèmes et solutions.
Ce tutoriel présente les fonctions utilisateurs. Nous définirons tout d'abord ce qu'est une fonction et une procédure. Puis nous verrons comment déclarer une fonction et lui demander de renvoyer une ou plusieurs valeurs, avant d'aborder les principes de visibilité et de portée de variables. Enfin nous expliquerons les mécanismes de passage de paramètres (par copie et par référence) et terminerons par les fonctions à arguments infinis.

Qu'est-ce qu'une fonction / procédure utilisateur ?

En plus des **fonctions natives** qui ne nécessitent pas d'importation de bibliothèque avant leur appel, il existe **les fonctions utilisateurs**. Celles-ci sont créées par le programmeur lui même et enregistrées dans une librairie externe.
Plus concrètement, une fonction utilisateur se définit comme une suite d'instructions plus ou moins longue et complexe. Cette fonction peut-être perçue comme un **sous-programme** du **programme principal** qui**l'appelle**.
Il existe deux types de fonctions utilisateurs : **les fonctions** et **les procédures**. Quelle différence y-a-t-il entre les deux ? La différence qui les sépare est **la valeur de retour**. Une procédure est un groupe d'instructions qui ne renvoie pas de valeur après leur exécution. Une fonction, quant à elle, est en réalité une procédure qui retourne une valeur (de type primitif ou complexe) ou un objet.
Enfin, une fonction utilisateur peut (ou non) prendre des paramètres d'entrée au même titre qu'une fonction mathématique. Ces paramètres peuvent être de type primitif (int, float, string, boolean) ou structuré (array, object).

Déclaration d'une fonction

Principe de base

Une fonction utilisateur est définie par deux principaux éléments. Sa signature et son corps. La signature est elle même composée de deux parties : le nom et la liste de paramètres. Le corps, quant à lui, établit la suite d'instructions qui devront être exécutées.

La déclaration d'une nouvelle fonction se réalise au moyen du mot-clé ***function*** suivi du nom de la fonction et de ses arguments. Le code suivant illustre tout ça :

Exemple de déclaration d'une fonction

```php
<?php
  function nomDeLaFonction($param1, $param2, ...)
  {
    // Bloc d'instructions
  }
?>
```

Lors de la déclaration d'une fonction, son nom et le nom de ses paramètres sont totalement arbitraires. Il est possible de nommer la fonction et ses arguments dans un langage naturel. Exemple :*parcourirLeTableau($tableau)*.
Attention : une fonction ne doit être déclarée qu'une seule fois. Dans le cas contraire, un message d'erreur sera retourné par l'interpréteur PHP.
Prenons un exemple très simple d'une fonction (procédure en réalité) qui calcule la somme des deux paramètres. Le résultat est stocké dans une **variable locale** *$somme* puis affiché au moyen d'une instruction echo().

Déclaration d'une procédure

```php
<?php
  function calculerSomme($a, $b)
  {
    // Déclaration de la variable locale $somme
    $somme = $a + $b;
    // Affichage du résultat
    echo 'La somme de ', $a ,' et de ', $b ,' vaut ', $somme;
  }
?>
```

Valeur par défaut

Il est possible d'appliquer une valeur par défaut à un paramètre si celui-ci n'est pas indiqué lors de l'appel. Dans notre exemple précédent nous allons appliquer la valeur 5 par défaut au paramètre $b.

Affectation d'une valeur par défaut à un paramètre

```php
<?php
  function calculerSomme($a, $b=5)
  {
    // Déclaration de la variable locale $somme
    $somme = $a + $b;
    // Affichage du résultat
    echo 'La somme de ', $a ,' et de ', $b ,' vaut ', $somme;
  }
?>
```

Valeur de retour

Comme nous l'avions expliqué plus haut, il y'a deux sortes de fonctions utilisateurs : les fonctions et les procédures. Jusque là, notre exemple était une procédure puisqu'il ne renvoyait aucune valeur (soit *void*). Nous allons à présent modifier notre fonction pour qu'elle renvoie la somme des deux paramètres au lieu de l'afficher sur la sortie standard.

Déclaration d'une fonction

```php
<?php
  function calculerSomme($a, $b)
  {
    // Déclaration de la variable locale $somme
    $somme = $a + $b;
    // Renvoi de la somme au programme principal
    return $somme;
  }
?>
```

Nous remarquons ici la présence d'un nouveau mot clé. Il s'agit de ***return***. L'instruction *return* renvoie la valeur de la variable locale $somme au programme principal qui appelle la fonction. Nous verrons dans la partie suivante comment récupérer cette valeur. L'exécution de la fonction est stoppée immédiatement après l'exécution de cette instruction *return*.
Une fonction peut aussi comporter plusieurs instructions *return* dans son corps. Voici un exemple qui simule un mécanisme vérifiant qu'une personne est connectée.

Exemple d'une fonction ayant plusieurs instructions return

```php
<?php
  function estConnecte($connecte)
  {
    // Test du paramètre d'entrée
    if(1 === $connecte)
    {
      return true;
    }
    return false;
  }
?>
```

Ici nous renvoyons une valeur booléenne (*true* ou *false*) en fonction de l'état de la variable *$connecte*.
Important : une fonction ne peut retourner qu'une seule et unique valeur de type primitif à la fois. Pour renvoyer plusieurs valeurs d'un coup, il faut avoir recours au tableau. Le principe est expliqué juste après.

Retour de plusieurs valeurs

Dans certains cas particuliers d'applications il est demandé à une fonction de retourner plusieurs valeurs d'un coup. Nous avons vu précédemment que l'appel de plusieurs instructions *return* était impossible. Pour palier à cette faiblesse, le langage PHP autorise le renvoi d'un tableau de valeurs, soit un type structuré.
Prenons l'exemple d'une fonction qui renvoie les identifiants de connexion à une base de données. Ceci est un exemple totalement arbitraire.

Exemple de renvoi d'un tableau

```
<?php
  function getIdentifiantsSql()
  {
    return
array('sql.monsite.com','monLogin','monPassword','maBaseDeDonn
ees');
  }
?>
```

Nous aurions pu aussi d'abord déclarer le tableau local et lui affecter les valeurs avant de le retourner comme le montre le code suivant.

Déclaration puis renvoi du tableau

```
<?php
  function getIdentifiantsSql()
  {
    $arrayIdentifiants = array();
    $arrayIdentifiants[] = 'sql.monsite.com';
    $arrayIdentifiants[] = 'monLogin';
    $arrayIdentifiants[] = 'monPassword';
    $arrayIdentifiants[] = 'maBaseDeDonnees';

    return $arrayIdentifiants;
  }
?>
```

Appel d'une fonction

Jusqu'à maintenant nous déclarions des fonctions mais nous ne les utilisions pas. C'est ce que nous allons étudier maintenant avec ***l'appel de fonction***.
L'appel d'une fonction se fait dans le programme principal après avoir déclarer la fonction. Une fonction est appellée grâce à son nom suivi des paramètres. Reprenons notre toute première fonction qui calcule la somme de deux nombres et affiche le résultat sur la sortie standard. L'appel se réalise de la manière suivante :

Appel de la procédure calculerSomme()

```
<?php
  /**
```

```
   *  Ici la fonction a été déclarée
   */

  /**
   *  Appel de la procédure
   */
   calculerSomme(10,1);
   echo '<br/>';
   calculerSomme(7, 23);
   echo '<br/>';
   calculerSomme(130, 231);
   echo '<br/>';
   calculerSomme(98, 62);
   echo '<br/>';
?>
```

Après exécution des 4 fonctions, le résultat produit est le suivant :

Résultat de l'exécution des appels à calculerSomme()

```
La somme de 10 et de 1 vaut 11

La somme de 7 et de 23 vaut 30

La somme de 130 et de 231 vaut 361

La somme de 98 et de 62 vaut 160
```

Voyons maintenant ce que cela donne avec la même fonction qui renvoie la valeur. Celle dotée de l'instruction *return*. En exécutant le code précédent, nous remarquons que rien ne s'affiche alors que l'exécution s'est déroulée comme prévue. Pourquoi ? Tout simplement parceque les valeurs retournées n'ont pas été récupérées et stockées dans des structures de données (variables, tableau, objet...). Nous allons donc les stocker dans un tableau nommé *$resultatsSomme* avant d'afficher le contenu de celui-ci.

Exemple de récupération de valeur retournée par la fonction calculerSomme()

```
<?php
  /**
   *  Ici la fonction a été déclarée
   */

  // Déclaration du tableau
  $resultatsSomme = array();

  /**
   *  Appel de la fonction
   */
   $resultatsSomme[] = calculerSomme(10,1);
   $resultatsSomme[] = calculerSomme(7, 23);
```

```php
    $resultatsSomme[] = calculerSomme(130, 231);
    $resultatsSomme[] = calculerSomme(98, 62);

    // Affichage du contenu du tableau
    echo '<pre>';
    print_r($resultatsSomme);
    echo '</pre>';
?>
```

Ce qui produit le résultat suivant :

Résultat de l'exécution du script

```
Array
(
    [0] => 11
    [1] => 30
    [2] => 361
    [3] => 160
)
```

Il faut donc toujours penser à enregistrer la / les valeurs de retour dans une variable / tableau, ou bien l'utiliser directement dans une condition.

Visibilité des variables

Lorsqu'une variable est déclarée, sa visibilité dans le programme principal dépend de son contexte. En PHP il existe 3 niveaux de visibilité décrits ci-dessous :

- Les **variables superglobales** : ce sont en fait les tableaux tels que $_SESSION, $_COOKIE, $_GET, $_POST, $_SERVER... Il sont créés directement par PHP et sont accessibles (en lecture et écriture) depuis n'importe où dans le programme. Ils peuvent donc être appelés à l'intérieur d'une fonction sans être passés en paramètre de celle-ci.
- Les **variables globales** : ce sont toutes les variables, tableaux, objets et constantes que nous créons nous-même dans le programme principal. Ces structures de données ne sont visibles en théorie que dans le programme principal mais il est possible d'y avoir accès dans une fonction sans les passer en paramètre. Nous verrons cela plus loin dans ce cours.
- Les **variables locales** : ce sont toutes les variables d'une fonction (paramètres compris). Leur visibilité n'est que locale, c'est à dire dans la fonction elle-même. Le programme principal ne peut donc pas agir sur ces variables. C'est aussi d'ailleurs pourquoi leur nom peut-être totalement choisi arbitrairement sans risque de confusion pour PHP avec les variables globales du programme principal. Elles sont automatiquement supprimées de la mémoire lorsque la fonction a été complètement executée.

Si nous tentons d'appeler une variable qui ne se trouve pas dans le bon contexte, l'interprêteur PHP retournera une erreur de type *Notice: Undefined variable: monNom in /Applications/MAMP/htdocs/Tests-PHP/fichier.php on line 8*

Portée d'une variable

Nous venons de voir que les variables déclarées globales dans le programme principal et les variables locales déclarées à l'intérieur d'une fonction ne peuvent être accessibles. Les deux espaces étant complètement fermés et indépendants. En PHP il existe 3 niveaux de portée d'une variable qui sont *global*,*local* et *statique*. Le dernier est un peu particulier.

- Une variable déclarée dans la fonction et préfixée par le mot-clé *global* est considérée comme globale. Elle devient alors visible dans la fonction mais aussi à l'extérieur de celle-ci.
- Une variable déclarée traditionnellement dans la fonction prend par défaut une portée *locale*. Elle est alors visible uniquement dans la fonction et est détruite à la fin de son exécution.
- Une variable déclarée dans la fonction et préfixée par le mot-clé *static* est considérée comme une variable locale dont la valeur se conserve tout au long de l' exécution du programme principal. Cette valeur pourra alors être réutilisée en appelant la fonction de nouvelles fois.

Illustrons le premier et le troisième points à l'aide d'un exemple commun. La fonction suivante simule le remplissage d'une voiture en carburant. Le budget en carburant est une variable globale décrémentée du montant du plein d'essence.

Exemple d'illustration de la portée des variables dans une fonction

```php
<?php
  // Déclaration de la fonction
  function remplirReservoir($volume, $prixAuLitre)
  {
    // Déclaration des variables
    global $budgetEssence;
    static $volumeTotalEssence;
    $montant = 0;

    // Calcul du prix du remplissage de l'essence
    $montant = round($volume * $prixAuLitre, 2);
    // Retrait de $montant à $budgetEssence
    $budgetEssence -= $montant;
    // Ajout du volume au volume total enregistré
    $volumeTotalEssence += $volume;

    // Affichage des informations
    echo 'Vous avez mis ', $volume ,'L d\'essence.';
    echo 'Prix au Litre : ', $prixAuLitre ,' euros.</br>';
    echo 'Prix total : ', $montant ,' euros.';
    echo 'Budget essence restant : ', $budgetEssence ,'
euros.';
```

```
    echo 'Volume total d\'essence depuis le début : ',
$volumeTotalEssence ,'L.';
  }

  // Déclaration des variables globales
  $budgetEssence = 700;

  // Premier plein d'essence
  remplirReservoir(42, 1.25);
  // Second plein d'essence
  remplirReservoir(38, 1.19);
  // Troisième plein d'essence
  remplirReservoir(43, 1.23);
?>
```

Le resultat après exécution du programme est le suivant :

Résultat de l'exécution du programme

```
Vous avez mis 42L d'essence.

Prix au Litre : 1.25 euros.

Prix total : 52.5 euros.

Budget essence restant : 647.5 euros.

Volume total d'essence depuis le début : 42L.


Vous avez mis 38L d'essence.

Prix au Litre : 1.19 euros.

Prix total : 45.22 euros.

Budget essence restant : 602.28 euros.

Volume total d'essence depuis le début : 80L.


Vous avez mis 43L d'essence.

Prix au Litre : 1.23 euros.

Prix total : 52.89 euros.

Budget essence restant : 549.39 euros.

Volume total d'essence depuis le début : 123L.
```

Explications : la variable *$budgetEssence* est déclarée *globale* dans la fonction. Cela signifie qu'elle est accessible en lecture et écriture dans la fonction mais aussi à l'extérieur de celle-ci. La variable*$volumeTotalEssence* est déclarée statique. La variable *$montant* est une variable locale utile uniquement pendant l'exécution de la fonction. Elle est détruite lorsque la fonction est complètement exécutée. Elle compte le volume total d'essence acheté depuis le début. Sa valeur est donc conservée pour les appels suivants de la fonction. La fonction native round() arrondit la somme du plein d'essence à deux chiffres après la virgule. L'étape de calcul suivante utilise les opérateurs combinés arithmétiques. Plus d'informations sur le tutoriel des opérateurs de PHP.

Passage de paramètre par copie ou référence

Nous entamons l'avant dernier point important concernant les fonctions utilisateurs. Il s'agit du passage de paramètres (que nous avons déjà vu lors des appels de fonctions). Il existe deux types de passage de paramètres en PHP. Il s'agit du *passage par copie* ou bien le *passage par référence*.
Dans le premier cas, la valeur passée en paramètre de la fonction n'est pas modifiée. En réalité la fonction travaille sur une copie de cette valeur. C'est aussi le passage de paramètre par défaut de PHP. Voici un exemple simple qui présente le mécanisme :

Présentation du mécanisme de passage par valeur

```php
<?php
  function multiplierParDeux($nombre)
  {
    $nombre *= 2;
  }

  $unNombre = 5;
  multiplierParDeux($unNombre);
  echo $unNombre;    // Affiche 5
?>
```

Pour pouvoir modifier la valeur de *$unNombre*, il faut procéder à un *passage par référence*. Le passage par référence consiste à passer l'adresse mémoire de la variable et non une copie de sa valeur. Pour cela, il faut simplement ajouter le symbole & devant le paramètre dans la signature de la fonction. Reprenons notre exemple précédent :

Présentation du mécanisme de passage par référence

```php
<?php
  function multiplierParDeux(&$nombre)
  {
    $nombre *= 2;
  }

  $unNombre = 5;
  multiplierParDeux($unNombre);
```

```
  echo $unNombre;     // Affiche 10
?>
```

Note : les objets sont toujours passés par référence. Le passage par copie ne s'applique pas à ces structures de données.
Il est également possible de retourner des valeurs par référence mais cela n'a pas grand intérêt puisqu'en général nous renvoyons la valeur d'une variable locale que nous stockons ensuite dans une variable globale du programme principal. Dans certains cas très particuliers, cela peut se révéler intéressant. Pour signaler le retour par référence d'une valeur, nous devons placer le symbole & devant le nom de la fonction lors de sa déclaration.

Les fonctions à paramètres infinis

PHP permet également de créer des fonctions à paramètres infinis. C'est ce que l'exemple suivant illustre.

Exemple de déclaration et d'appel de fonction à paramètres infinis

```
<?php
  function avoirPortraitRobot()
  {
    // Stockage des paramètres dans un tableau
    $infosPortrait = func_get_args();
    // Affichage des informations
    echo 'Le portrait robot du suspect a ', func_num_args() ,'
critères :';
    foreach($infosPortrait as $element)
    {
      echo ' - ', $element ,'';
    }
  }

  avoirPortraitRobot
  (
    'homme',
    'yeux bleux',
    'brun',
    'cheveux coupés courts',
    '1m85',
    'nez crochu',
    'balafré à la joue gauche'
  );

?>
```

Notons que nous déclarons une fonction sans paramètre. Le résultat de l'exécution du code précédent est le suivant :

Résultat de l'exécution du programme

```
Le portrait robot du suspect a 7 critères :
```

```
homme
yeux bleux
brun
cheveux coupés courts
1m85
nez crochu
balafré à la joue gauche
```

Conclusion

Nous avons maintenant fait le tour de ce que l'on peut dire à propos des fonctions utilisateurs. Nous avons tout d'abord défini ce qu'est une fonction et une procédure. Puis nous avons montré comment la déclarer et lui faire retourner une ou plusieurs valeurs. Nous nous sommes ensuite attardés sur le principe d'appel de procédure avant d'expliquer les principes de visibilité et de portée d'une variable. Enfin nous avons étudié les types de passages de paramètres et le retour de valeur (par copie ou par référence) avant de terminer par les fonctions à arguments infinis.

Traitement des formulaires avec $_GET et $_POST

Qui dit « site web dynamique » dit généralement « formulaires » et donc traitement de ces derniers. PHP a notamment été inventé pour ce type de tâche et c'est ce que nous allons étudier dans ce nouveau tutoriel. Nous apprendrons à exploiter les formulaires par le biais des tableaux superglobaux $_GET et $_POST. Nous déterminerons aussi la différence qui existe dans l'utilisation de chacun d'eux.

Remarques :

[1] Pour la suite du tutoriel, nous considèrerons que les bases concernant les formulaires HTML sont acquises. Nous n'aborderons que synthétiquement les points essentiels de ces derniers à savoir les*noms, valeurs, enctypes* et *methodes*.
[2] Par ailleurs nous considèrerons que tous les tutoriels précédents des bases de PHP sont compris et acquis.

Traitement PHP ou traitement Javascript ?

Il est bien évident que **TOUS** les formulaires doivent être traités en priorité avec PHP. Toutefois, rien n'empêche l'utilisation de Javascript en tant que surcouche au travail de PHP. Pourquoi PHP en priorité alors ? Il y'a plusieurs raisons à cela :

- PHP est executé sur le serveur alors que Javascript est exécuté sur le client (navigateur). De ce fait, il peut-être désactivé ou non fonctionnel rendant les contrôles impossibles.
- Etant exécuté sur le serveur, seul le programme peut agir sur les informations transmises. La sécurité est alors accrue par rapport aux contrôles côtés client.
- PHP dispose d'une série de fonctions natives capables de manipuler les variables et de les contrôler.
- La puissance des expressions régulières permet aussi de vérifier des formats de données personnalisés.

Remarque : dans la mesure où les informations proviennent de personnes anonymes, nous ne pouvons garantir la véracité et la dangerosité de ces dernières. C'est pourquoi tout doit être vérifié. **Le principe numéro 1 lorsque l'on interagit avec un utilisateur est de ne jamais lui faire confiance**.

Les parties essentielles d'un formulaire

Introduction

Afin de pouvoir faire dialoguer correctement un formulaire HTML avec un script PHP, il faut s'assurer que les points suivants soient présents :

- L'attribut **action** de la balise **<form>** est renseigné par l'url du fichier PHP qui va recevoir les informations. Cela peut-être un fichier différent de la page courante mais il est conseillé de traiter les formulaires dans la même page.
- La méthode HTTP (attribut **method**) du formulaire est renseignée par l'une de ces deux valeurs : **get**ou **post**.

- Si l'on a affaire à un formulaire d'upload de fichiers, la balise <form> doit comporter l'attribut *enctype*et la valeur *multipart/form-data*.
- Tous les éléments d'un formulaire doivent posséder un attribut *name* rempli par une valeur. Le nom du champ sera ensuite considéré par le script PHP comme une variable contenant la valeur saisie.

A titre d'information, voici un exemple tout simple de formulaire d'upload qui présente tous les éléments obligatoires évoqués ici.

Exemple de formulaire d'upload

```
<!-- Debut du formulaire -->

<form enctype="multipart/form-data" action="./upload.php"
method="post">
    <fieldset>
        <legend>Formulaire</legend>
            <p>
                <label>Envoyer le fichier :</label>
                <input name="fichier" type="file" />
                <input type="submit" name="submit"
value="Uploader" />
            </p>
    </fieldset>
</form>
<!-- Fin du formulaire -->
```

Détaillons ici les méthodes GET et POST. Quelle différence les sépare ? Pour laquelle opter ?

La méthode HTTP GET

La méthode GET (celle qui est utilisée par défaut si rien n'est renseigné) fait circuler les informations du formulaire en clair dans la barre d'adresse en suivant le format ci-après :

Exemple d'url créée à partir de la méthode GET d'un formulaire

```
http://www.unsite.com/chemin/scriptphp.php?var1=valeur1&var2=v
aleur2
```

Cette adresse signifie que nous allons transmettre à la page *scriptphp.php* les couples variable / valeur transmis en paramètre. La première variable d'une url est toujours précédée du symbôle ? alors que les autres seront précédées du symbôle &. Les noms des variables correspondent aux attributs *name* des éléments du formulaire et les valeurs aux attributs *value*.

Note : contrairement à ce que l'on peut lire fréquemment sur la toile, la limite maximale d'une URL n'est pas de 255 caractères. Il n'existe en réalité aucune limite standard. En effet, la taille maximale d'une URL peut-être configurée à la fois côté serveur ou côté client. Un administrateur de serveur web peut à sa guise augmenter ou diminuer la longueur maximale des URLs. Quant aux navigateurs, eux aussi fixent par défaut une taille maximale. Il est donc recommandé de ne pas abuser de la longueur d'une URL lorsque l'on ne maîtrise pas l'intégralité de son environnement de production (serveur Web et clients).

La méthode HTTP POST

La méthode POST, quant à elle, transmet les informations du formulaire de manière masquée **mais non cryptée**. Le fait de ne pas afficher les données ne signifie en rien qu'elles sont cryptées. Rappelons nous d'ailleurs que ces informations utilisent le protocole HTTP et non HTTPS qui lui crypte les données.
Quelle est la meilleure méthode à adopter alors ? Et bien la réponse est : « ça dépend ». Le choix de l'une ou de l'autre se fera en fonction du contexte. Si par exemple, nous souhaitons mettre en place un moteur de recherches alors nous pourrons nous contenter de la méthode GET qui transmettra les mots-clés dans l'url. Cela nous permettra aussi de fournir l'url de recherches à d'autres personnes. C'est typiquement le cas des URLs de Google :

Exemple d'une URL du moteur de recherches Google

```
http://www.google.fr/search?q=php
```

La méthode POST est préférée lorsqu'il y'a un nombre important de données à transmettre ou bien lorsqu'il faut envoyer des données sensibles comme des mots de passe. Dans certains cas, seule la méthode POST est requise : un upload de fichier par exemple.

Les tableaux superglobaux $_POST et $_GET

$_GET et $_POST sont des tableaux de données associatifs et superglobaux. Voici leurs principales caractéristiques :

- Ils sont générés à la volée par PHP avant même que la première ligne du script ne soit exécuté.
- Ce sont des tableaux associatifs comme ceux que l'on déclare traditionnellement. Leur manipulation est exactement semblable à ces derniers. Les clés correspondent aux noms des variables transmises et les valeurs à celles associées à ces variables.
- Ils sont **superglobaux**, c'est à dire visibles de partout dans le programme (même à l'intérieur d'une fonction utilisateur).
- Ils sont **accessibles en lecture et en écriture**. Il est donc possible de les modifier.

Le tableau $_GET contient tous les couples variable / valeur transmis dans l'url. Pour accéder à la valeur d'une variable dont le nom est *prenom*, on l'appelle ainsi :

Lecture d'une variable appartenant au tableau $_GET

```
<?php
  echo $_GET['prenom'];
?>
```

Le tableau $_POST contient tous les couples variable / valeur transmis en POST, c'est à dire les informations qui ne proviennent ni de l'url, ni des cookies et ni des sessions. Pour accéder à la valeur d'une variables dont le nom est *prenom*, on l'appelle ainsi :

Lecture d'une variable appartenant au tableau $_POST

```php
<?php
  echo $_POST['prenom'];
?>
```

La casse des variables est importante. Il faut bien penser à mettre $_GET et $_POST en majuscules. Dans le cas contraire, il sera impossible d'obtenir une valeur et une erreur de type undefined variable sera retournée.

***Note** : il existe aussi le tableau associatif superglobal $_REQUEST qui regroupe les 3 tableaux $_GET, $_POST et $_COOKIE que nous verrons au prochain cours. Il fonctionne exactement comme tous les autres tableaux.*

Exemple simple et concret de traitement de formulaire

Dans cette partie, nous allons utiliser un exemple simple de traitement de formulaire. Nous récupèrerons et vérifierons les données en provenance d'un formulaire d'authentification. Les principes de session seront évincés car ils ne constituent pas l'objet majeur du cours. Nous nous intéresserons uniquement à la saisie et la réception des données du formulaire.

Voici les pré-requis nécessaire à l'élaboration de notre exemple :

- Un formulaire d'identification a besoin de deux éléments : un champ texte qui reçoit le login et un champ password qui reçoit le mot de passe du visiteur.
- Nous traiterons les données dans la même page avant la première balise HTML du document. Notre formulaire devra donc s'appeller lui même.
- Les identifiants de comparaison seront stockés dans deux constantes.
- Les erreurs seront signalées à l'utilisateur.
- Le login sera réaffiché dans le champ si l'utilisateur s'est trompé.
- Si toutes les informations sont vérifiées, un message de réussite simulera l'ouverture de la session du membre.

Passons au script complet. Il est entièrement commenté donc seules quelques petites explications seront apportées ensuite :

Script complet d'identification : exemple de traitement de formulaire

```php
<?php
  /*****************************************
  *  Constantes et variables
  *****************************************/
  define('LOGIN','Rasmus');  // Login correct
  define('PASSWORD','lerdorf');  // Mot de passe correct
  $message = '';      // Message à afficher à l'utilisateur

  /*****************************************
  *  Vérification du formulaire
  *****************************************/
  // Si le tableau $_POST existe alors le formulaire a été
envoyé
  if(!empty($_POST))
```

```
  {
    // Le login est-il rempli ?
    if(empty($_POST['login']))
    {
      $message = 'Veuillez indiquer votre login svp !';
    }
      // Le mot de passe est-il rempli ?
      elseif(empty($_POST['motDePasse']))
    {
      $message = 'Veuillez indiquer votre mot de passe svp !';
    }
      // Le login est-il correct ?
      elseif($_POST['login'] !== LOGIN)
    {
      $message = 'Votre login est faux !';
    }
      // Le mot de passe est-il correct ?
      elseif($_POST['motDePasse'] !== PASSWORD)
    {
      $message = 'Votre mot de passe est faux !';
    }
      else
    {
      // L'identification a réussi
      $message = 'Bienvenue '. LOGIN .' !';
    }
  }
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
  <head>
    <title>Formulaire d'identification</title>
  </head>
  <body>
    <?php if(!empty($message)) : ?>
      <p><?php echo $message; ?></p>
    <?php endif; ?>
    <form action="<?php echo
htmlspecialchars($_SERVER['REQUEST_URI'], ENT_QUOTES); ?>"
method="post">
      <fieldset>
        <legend>Identifiant</legend>
          <p>
             <label for="login">Login :</label>
            <input type="text" name="login" id="login"
value="<?php if(!empty($_POST['login'])) { echo
htmlspecialchars($_POST['login'], ENT_QUOTES); } ?>" />
          </p>
          <p>
            <label for="password">Mot de passe :</label>
```

```
            <input type="password" name="motDePasse"
id="password" value="" />
            <input type="submit" name="submit"
value="Identification" />
          </p>
      </fieldset>
    </form>
  </body>
</html>
```

D'un point de vue général, nous avons structuré la page de cette façon :

1. Déclaration des constantes et variables.
2. Traitement des données.
3. Présentation des données

Ce schéma de conception fait partie des bonnes pratiques de développement à adopter. On traite toujours les informations avant de les présenter. Avec cette méthode, nous pourrons par exemple effectuer des redirections sans risque d'erreur d'entêtes déjà envoyés.

Quelques lignes du code méritent tout de même des explications :

La première condition vérifie que le tableau $_POST existe et n'est pas vide. Si c'est le cas, alors elle renverra vrai (TRUE) et sera franchie pour accéder aux tests suivants.

On effectue ensuite une série de tests pour contrôler que les champs du formulaire ont bien été remplis et que les valeurs transmises correspondent aux constantes définies en tête du fichier. La fonction empty()vérifie que la variable passée en paramètre existe et qu'elle est vide ou nulle. Si l'on détecte une erreur, on l'enregistre dans la variable *$message*.
Si aucune erreur n'est détectée, cela signifie que les idenfiants sont corrects. On entre alors dans la clause *else*. Pour information, nous aurions du, en réalité, ouvrir une session et rediriger l'utilisateur vers la page protégée.
Dans le corps de la page, nous ajoutons une condition qui vérifie si la variable *$message* existe et qu'elle est bien remplie. Si c'est le cas, on franchit la condition et l'on affiche le message dans un bloc de paragraphe HTML.
La fonction htmlspecialchars() protège les variables en transformant les chevrons (< et >) et certains caractères HTML en entités HTML équivalentes. On utilise cette fonction pour se protéger d'éventuels actes de piratage par injection de code Javascript ou HTML (attaques de Cross Site Scripting ou XSS).
Nous appelons la variable d'environnement *$_SERVER['REQUEST_URI']* qui contient l'url qui mène jusqu'à la page courante. Grâce à elle, nous précisons que le fichier qui va recevoir les données est lui même. Il est nécessaire de protéger cette variable car elle est sensible au piratage par injection de code HTML.
Enfin, dans l'attribut *value* du champ texte *login*, nous replaçons la dernière valeur postée par l'utilisateur. On la protège aussi car si le visiteur poste du code html, celui-ci sera directement interprêté. Il faut donc s'assurer de la sécurité de la variable que l'on souhaite réafficher.

Un mot sur les apostrophes magiques

Si la directive **magic_quotes_gpc** (GPC pour GET / POST et COOKIE) du php.ini est activée sur le serveur web, alors PHP va automatiquement protéger les chaînes en échappant par un antislash certains caractères spéciaux (apostrophes, guillemets...). Si par exemple nous postons la chaîne *L'école est finie*alors nous obtiendrons le résultat suivant :

Résultat d'une chaine de caractères échappée automatiquement

```
L\'école est finie
```

Cette protection agit donc comme la fonction addslashes(). L'antislash protège ici l'apostrophe. Ce caractère peut-être dangereux dans une requête SQL s'il n'est pas échappé. On parle de risque de piratage par injection de code SQL. Il faut alors faire attention à ne pas échapper une nouvelle fois les données avec cette fonction si les apostrophes magiques sont activées car les antislashes seraient alors eux aussi échapper par d'autres antislashes. Notre exemple précédent pourrait alors ressembler à cela si l'on échappe les données plusieurs fois :

Résultat d'une chaine de caractères échappée automatiquement

```
L\\\\\\\\\'école est finie
```

Cette initiative de protection a suscité de nombreux débats dans la communauté PHP car elle était en effet prévue pour combler les trous de sécurité créés par des programmeurs débutants et peu soucieux de la sécurité. Le passage à PHP 6 entraînera la désactivation de cette directive, ce qui forcera les développeurs à protéger eux mêmes leurs applications.
Nous recommandons de désactiver cette directive soit directement dans le php.ini si l'on a la main mise sur le serveur web ou bien en ajoutant cette ligne en haut de chacune des pages PHP :

Configuration temporaire du serveur web

```
<?php
  // Désactivation des magic_quotes_gpc
  ini_set('magic_quotes_gpc', 0);
?>
```

Conclusion

Nous sommes arrivés au terme de ce tutoriel. Nous avons tout d'abord démontré en quoi PHP devait être utilisé en priorité par rapport à Javascript pour traiter les formulaires. Puis, nous avons déterminé les éléments essentiels d'un formulaire HTML qui permettent un traitement par un script PHP. Dans cette lancée, nous avons expliqué les caractéristiques de chacune des deux méthodes HTTP GET et POST. Nous nous sommes ensuite arrêtés sur les tableaux superglobaux $_GET, $_POST et $_REQUEST qui permettent de récupérer les informations issues d'un formulaire ou d'une URL. Nous avons

d'ailleurs illustré leur emploi au moyen d'un exemple de script réel modélisant un système d'identification, avant de conclure sur des principes de sécurité tels que les failles par injection de code HTML ou Javascript (XSS) et l'injection de commandes SQL.

Les cookies

Ce nouveau tutoriel introduit le mécanisme des cookies. Nous définirons ensemble ce qu'est un cookie et à quoi il sert. Puis nous aborderons les principes de sécurité relatifs aux cookies. Ensuite nous apprendrons à générer et lire le contenu d'un cookie. Nous terminerons ce tutoriel sur la suppression du cookie, le stockage des valeurs d'un type complexe par sérialisation / désérialisation avant de conclure sur les cas d'utilisation les plus fréquents.

Qu'est-ce qu'un cookie ?

Le mécanisme des cookies a été inventé par la société Netscape dans le but de palier à certaines faiblesses du protocole HTTP mais aussi d'étendre les possibilités de relation entre le client et le site web. Leur fonction est le stockage, pendant une période donnée, d'une information relative à l'utilisateur (son pseudo, sa date de dernière connexion, son âge, ses préfèrences...).

En pratique, les cookies sont de simples fichiers textes ne pouvant excéder 4Ko. Ils sont stockés sur le disque dur de l'internaute et gérés par les navigateurs (Firefox, Internet Explorer, Safari, Opéra, AvantBrowser...). Leur acceptation est soumise aux filtres des navigateurs. En effet, ces derniers sont capables de refuser des cookies. Leur utilisation doit donc être scrupulesement réfléchie.

Pour des raisons de sécurité, des "normes" ont fixé à 20 le nombre maximal de cookies envoyés pour un même domaine.

Qu'en est-il de la sécurité ?

Il y'a encore quelques années les cookies faisaient peur. Certains profanes se persuadaient qu'ils étaient dangereux, qu'ils pouvaient exécuter des programmes malicieux sur leur ordinateur ou bien récupérer des informations personnelles ou confidentielles. Or ce n'est pas le cas. Un cookie n'est ni plus ni moins qu'un fichier texte de très petite taille. Il ne peut donc ni être exécuté ni même lancer des programmes à lui seul. Il ne sert uniquement à stocker une information en vue d'une réutilisation ultérieure. En revanche, rien n'empêche un programme pirate d'être exécuté par l'utilisateur (mais à son insu) qui va récupérer des informations confidentielles ou personnelles, puis les stocker dans un cookie qu'il crée et envoie au serveur Web. Ce dernier se chargera de récupérer les informations transmises et de les utiliser contre l'utilisateur.

La création d'un cookie demande néanmoins le respect de quelques règles de sécurité et de bon sens. Le cookie étant stocké sur le disque dur du client, son accès n'y est donc pas sécurisé. Un cookie pourra être lu, modifié ou supprimé sans difficulté par un utilisateur malintentionné. Il est donc fortement déconseillé de stocker des informations sensibles comme :

- des informations confidentielles à l'intérieur (mot de passe par exemple).
- des identifiants de connexion facilement identifiables comme un login.

Un cookie devrait généralement ne servir qu'à des fins statistiques, de personnalisation d'affichage ou bien à la multi pagination de formulaires. Et encore dans ce dernier cas, la meilleure solution serait d'avoir recours au mécanisme des sessions.

Génération d'un cookie avec setcookie()

La création d'un cookie repose sur l'envoi d'entêtes HTTP au navigateur du client au moyen de la fonctionsetcookie(). Cela sous-entend donc qu'il faudra l'appeller avant tout envoi de données au navigateur (print(), echo(), tag html, espace blanc...) sous peine de générer une erreur de type *« Warning : Cannot send session cookie - headers already sent by... »*.
La fonction setcookie() peut prendre jusqu'à 7 paramètres. Seul le premier est obligatoire car il définit le nom du cookie. Elle renvoie un booléen : true en cas de succès et false si échec.

Signature de la fonction setcookie()

```
bool setcookie ( string name [, string value [, int expire [,
string path [, string domain [, bool secure [, bool
httponly]]]]]] )
```

Le tableau ci-dessous récapitule les valeurs que peuvent prendre chacun de ces paramètres.

Paramètre	Explications

Paramètre	Explications
name	Nom du cookie
value	Valeur du cookie
expire	Date d'expiration du cookie au format timestamp UNIX, c'est à dire un nombre de secondes écoulées depuis le 01 janvier 1970
path	Chemin sur le serveur dans lequel le cookie sera valide. Si la valeur '/' est spécifiée alors le cookie sera visible sur tout le domaine. Si '/examples/' est précisé alors le cookie sera visible dans le répertoire /examples/ et tous ses sous-dossiers
domain	Domaine sur lequel le cookie sera valable. Si la valeur est '.domaine.com' alors le cookie sera disponible sur tout le domaine (sous-domaines compris). Si la valeur est 'www.domaine.com' alors le cookie ne sera valable que dans le sous-domaine 'www'
secure	Ce paramètre prend pour valeur un booléen qui définit si le cookie doit être utilisé dans un contexte de connexion sécurisée par protocole HTTPS
httponly	Ce paramètre prend pour valeur un booléen qui définit si le cookie sera accessible uniquement par le protocole HTTP. Cela signifie que le cookie ne sera pas accessible via des langages de scripts, comme Javascript. Cette configuration permet de limiter les attaques via XSS (bien qu'elle ne soit pas supportée par tous les navigateurs). Ajouté en PHP 5.2.0

Le code suivant illustre la création d'un cookie caractérisé par un nom, une valeur et une date de validité d'un mois. Il simule l'enregistrement du design d'un site que l'utilisateur préfère. Grâce à ce cookie, le site sera affiché avec ce thème graphique automatiquement à la prochaine visite de l'internaute.

Création d'un cookie

```php
<?php
  // Création du cookie
  setcookie('designPrefere','prairie',time()+3600*24*31);
?>
```

Remarques :

[1] Lorsqu'un cookie est créé depuis une page, il ne devient disponible qu'à partir de la page suivante car il faut que le navigateur envoit le cookie au serveur.
[2] Un cookie dont la date d'expiration n'est pas précisée est enregistré dans la

mémoire vive de l'ordinateur et non sur le disque dur. Il sera effacé à la fermeture du navigateur.

Lecture d'un cookie

Lorsqu'un internaute interroge un site web repéré par un nom de domaine, son navigateur envoie au serveur la liste des cookies disponibles pour ce domaine. PHP les réceptionne puis construit un tableau associatif superglobal nommé $_COOKIE. Les clés correspondent aux noms des cookies et les valeurs aux valeurs inscrites dans ces derniers. Il devient alors très simple d'accèder à la valeur d'un cookie en appellant le tableau $_COOKIE avec la clé qui convient. L'exemple ci-dessous nous permet de récupérer la valeur du cookie que nous avions créé précédemment.

Lecture d'un cookie

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<html>
  <head>
    <title>Lecture d'un cookie !</title>
  </head>
  <body>
    <p>
      Votre thème favoris est :
      <?php
        // Lecture de la valeur du cookie designPrefere
        echo $_COOKIE['designPrefere'];    // Affiche
'prairie'
      ?>
    </p>
  </body>
</html>
```

Remarques :

[1] Ajouter un couple clé / valeur au tableau $_COOKIE ne crée pas de nouveau cookie.
[2] Pour modifier la valeur d'un cookie, il faut réutiliser la fonction setcookie().
[3] Pour connaître tous les cookies utilisés, il faut lister le tableau $_COOKIE au moyen d'une boucle foreach() ou d'un appel à print_r().

Suppression d'un cookie

Pour supprimer un cookie, il faut de nouveau appeller la fonction setcookie() en lui passant en paramètre le nom du cookie uniquement. Dans notre exemple, nous utiliserons le code suivant pour supprimer notre cookie *designPrefere*.

Script de suppression d'un cookie

```php
<?php
  // Suppression du cookie designPrefere
  setcookie('designPrefere');
?>
```

Le code précédent demande au navigateur d'effacer le cookie mais ne supprime pas la valeur présente dans le tableau $_COOKIE. Il faut donc penser à supprimer les deux. Pour cela, nous utilisons le script ci-après.

Script de suppression complète d'un cookie

```php
<?php
  // Suppression du cookie designPrefere
  setcookie('designPrefere');
  // Suppression de la valeur du tableau $_COOKIE
  unset($_COOKIE['designPrefere']);
?>
```

Stocker des types complexes dans un cookie

Jusqu'à maintenant nous avons stocké dans notre cookie une valeur de type chaine de caractères. C'est donc un type primitif. Mais il est aussi possible de stocker un tableau entier dans un cookie. Pour réaliser cette opération, il faut obligatoirement transformer ce dernier en chaine de caractères puis le reconstruire lors de sa réception. C'est ce que l'on appelle **la sérialisation** (ou encore pliage, marshalling) et la**désérialisation** (ou encore dépliage ou unmarshalling). Ces deux opérations sont réalisées au moyen des fonctions serialize() et unserialize().
Illustrons ces principes à partir d'un exemple simulant le lancé d'un dé. Notre cookie s'appellera*"lancesDe"*. Nous simulerons également les lancés du dé avec la fonction rand(). Tous les lancés seront stockés dans un vecteur (tableau à une dimension indexé numériquement) que nous sérialiserons / désérialiserons pour les besoins du cookie.

Simulation de lancés de dé et enregistrement des scores dans un cookie

```
<?php

  // Définition des structures de données

  $nombreLancesDe = 0;      // Nombre de lances de dé

  $listeSerialisee = '';      // Chaine serialisee du cookie

  $listeLancesDe = array();    // Tableau des valeurs de
lancés de dé



  // Test de l'existence du cookie

  if(!empty($_COOKIE['lancesDe']))
```

```
  {
    // On récupère les valeurs dans $listeLancesDe par
désérialisation
    $listeSerialisee = $_COOKIE['lancesDe'];
    $listeLancesDe = unserialize($listeSerialisee);
  }


  // On lance une nouvelle fois le dé
  $listeLancesDe[] = rand(1,6);
  // On sérialise le tableau et on crée le cookie
  $listeSerialisee = serialize($listeLancesDe);
  setcookie('lancesDe', $listeSerialisee, time()+3600*24);
  // On calcule le nombre de lancés de dé
  $nombreLancesDe = count($listeLancesDe);
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
 <head>
   <title>Simulation de lancés d'un dé !</title>
 </head>
 <body>
    <p>
      Vous avez lancé <?php echo $nombreLancesDe; ?> fois le
dé :
    </p>
    <?php
      if($nombreLancesDe > 0)
      {
```

```
        echo '<ul>';

        // On affiche le contenu du tableau désérialisé

        foreach($listeLancesDe as $numeroLance =>
$valeurLance)

        {

          echo '<li>Lancé n#', ($numeroLance+1) ,' : ',
$valeurLance ,'</li>';

        }

        echo '</ul>';

      }

    ?>

  </body>

</html>
```

En appellant cinq fois le script, nous simulons cinq lancés de dé. Le résultat produit est similaire à celui-ci.

Résultat de l'exécution du script

```
Vous avez lancé 5 fois le dé :


 * Lancé #1 : 4

 * Lancé #2 : 6

 * Lancé #3 : 3

 * Lancé #4 : 3

 * Lancé #5 : 3

 * Lancé #6 : 2
```

Les principaux cas d'utilisation des cookies

Les cookies se révèlent donc très utiles dans des cas de figure particuliers. Parmi eux, nous pouvons recenser :

- la sauvegarde du design préféré d'un site pour un internaute
- l'affichage les nouveaux messages depuis la dernière visite de l'internaute
- l'affichage les messages non-lus par le visiteur dans un forum

- la fragmentation d'un formulaire sur plusieurs pages. Les valeurs envoyées sur la première page sont sérialisées et envoyées par cookie à la seconde qui les désérialisera
- les compteurs de visites
- la reconnaissance des visiteurs ayant déjà voté à un sondage
- ...

Conclusion

Nous retiendrons que les cookies sont un moyen efficace de retenir des informations de faible importance concernant un utilisateur. Néanmoins, pour des raions de sécurité, leur taille est limitée à 4Ko et leur nombre à 20 pour un même domaine. Il ne faut donc pas les utiliser pour transférer des données nombreuses mais uniquement pour stocker des informations à but statistique ou de personnalisation d'affichage.

Les sessions

Depuis PHP4, on entend beaucoup parler de sessions. De nombreuses personnes utilisant PHP ignorent encore ce qu'elles sont et à quoi elles servent. D'autres, en revanche, ne savent pas les utiliser à bon escient. Ce tutoriel est une approche à la fois théorique et pratique des sessions. Elles seront présentées au moyen d'un exemple simple tout au long de ce cours. Il s'agit d'un espace de site sécurisé par authentification.

Une session c'est quoi ?

Une session est un mécanisme technique permettant de sauvegarder **temporairement sur le serveur**des informations relatives à un

internaute. Ce système a notamment été inventé pour palier au fait que le protocole HTTP agit en mode **non connecté**. A chaque ouverture de nouvelle session, l'internaute se voit attribué un identifiant de session. Cet identifiant peut-être transmis soit en GET (PHPSESSID ajouté à la fin de l'url), POST ou Cookie (cookie sur poste client) selon la configuration du serveur. Les informations seront quant à elles transférées de page en page par le serveur et non par le client. Ainsi, la sécurité et l'intégrité des données s'en voient améliorées ainsi que leur disponibilité tout au long de la session. Une session peut contenir tout type de données : nombre, chaîne de caractères et même un tableau.
Contrairement à une base de données ou un système de fichiers, la session conserve les informations pendant quelques minutes. Cette durée dépend de la configuration du serveur mais est généralement fixée à 24 minutes par défaut. Le serveur crée des fichiers stockés dans un repertoire particulier.

Les sessions sont particulièrement utilisées pour ce type d'applications :

- Les espaces membres et accès sécurisés avec authentification.
- Gestion d'un caddie sur un site de vente en ligne.
- Formulaires éclatés sur plusieurs pages.
- Stockage d'informations relatives à la navigation de l'utilisateur (thème préféré, langues...).

La théorie, c'est bien beau mais en pratique comment ça se passe ? La partie suivante explique l'initialisation et la restauration d'une session ouverte.

Initialisation (et restauration) d'une session

PHP introduit nativement une unique fonction permettant de démarrer ou de continuer une session. Il s'agit de session_start(). Cette fonction ne prend pas de paramètre et renvoit toujours **true**. Elle vérifie l'état de la session courante. Si elle est inexistante, alors le serveur la crée sinon il la poursuit.

Initialisation et restauration d'une session

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

Dans le cas d'une utilisation des sessions avec les cookies, la fonction session_start() doit obligatoirement être appellée avant tout envoi au navigateur sous peine de voir afficher les fameuses erreurs :

"Cannot modify header information - headers already sent by ..." ou *"Cannot send session cookie - headers already sent by ..."*. Cela est du au fait que PHP ne peut plus envoyer de cookie à l'utilisateur car il y'a déjà eu une sortie au navigateur (echo(), print(), espace blanc, tag html...).
Note : *il faut appeller session_start() sur chaque page utilisant le système de session.*

Lecture et ecriture d'une session

Le tableau $_SESSION

Lorsqu'une session est créée, elle est par défaut vide. Elle n'a donc aucun intérêt. Il faut donc lui attribuer des valeurs à sauvegarder temporairement. Pour cela, le langage PHP met en place le tableau superglobal**$_SESSION**. Le terme *superglobal* signifie que le tabeau a une visibilité maximale dans les scripts. C'est à dire que l'on peut y faire référence de manière globale comme locale dans une fonction utilisateur sans avoir à le passer en paramètre. Le tableau **$_SESSION** peut-être indexé numériquement mais aussi associativement. En règle générale, on préfère la seconde afin de pouvoir donner des noms de variables de session clairs et porteurs de sens.

L'écriture de session

Pour enregistrer une nouvelle variable de session, c'est tout simple. Il suffit juste d'ajouter un couple clé / valeur au tableau $_SESSION comme l'illustre l'exemple suivant.

Déclaration et initialisation d'une variable une session

```php
<?php
  // Démarrage ou restauration de la session
  session_start();
  // Ecriture d'une nouvelle valeur dans le tableau de session
  $_SESSION['login'] = 'Dupond';
?>
```

Le tableau $_SESSION, qui était vide jusqu'à présent, s'est agrandit dynamiquement et contient maintenant une valeur (**Dupond**) à la clé associative **login**. Une variable de session est alors créée.
***Note de rappel** : à place de la chaîne de caractères « Dupond », nous aurions pu mettre un nombre, un booléen ou encore un tableau par exemple.*

Lecture d'une variable de session

Après l'écriture, c'est au tour de la lecture. Il n'y a rien de plus simple. Pour lire la valeur d'une variable de session, il faut tout simplement appeler le tableau de session avec la clé concernée. L'exemple ci-dessous illustre tout ça.

Lecture d'une variable de session

```php
<?php
  // Démarrage ou restauration de la session
  session_start();
  // Lecture d'une valeur du tableau de session
  echo $_SESSION['login'];
?>
```

Cette instruction aura pour effet d'afficher à l'écran la chaîne de caractères **Dupond**.

Destruction d'une session

Comme cela a été évoqué plus haut, le serveur détruit lui même la session au bout d'un certain temps si la session n'a pas été renouvellée. En revanche, il est possible de forcer sa destruction au moyen de la fonction session_destroy(). Cela permet par exemple aux webmasters de proposer une page de déconnexion aux membres loggués à leur espace personnel. Cependant, l'utilisation de session_destroy() seule n'est pas très "propre". Le code suivant présente une manière plus correcte de mettre fin à une session.

Destruction propre et complète d'une session

```
<?php
  // Démarrage ou restauration de la session
  session_start();
  // Réinitialisation du tableau de session
  // On le vide intégralement
  $_SESSION = array();
  // Destruction de la session
  session_destroy();
  // Destruction du tableau de session
  unset($_SESSION);
?>
```

Pour être convaincu de la destruction de la session, il suffit juste d'essayer d'afficher le contenu du tableau de session au moyen de la fonction print_r().

Configuration des sessions sur le serveur

Une session ne reste ouverte que pendant un certain temps. Tout au plus ce sera celle indiquée par la directive *session.gc_maxlifetime* du php.ini, entre deux clics consécutifs du client. Il est recommandé de ne pas augmenter la valeur inscrite par défaut. Mais pourquoi ? Tout simplement parceque si la session à une durée de vie plus importante, on s'expose à des risques de piratage par *vol de session* notamment (cf: voir les liens annexes en fin de tutoriel pour plus d'informations).
Pour les mêmes raisons de sécurité, il est conseillé de configurer le serveur de la façon suivante :

Configuration de PHP recommandée pour les sessions

```
session.use_cookies 1
session.use_only_cookies 1
session.use_trans_sid 0
```

Cette configuration implique néanmoins une restriction totale pour les personnes n'acceptant pas les cookies. Ci-dessous, la signification dans le même ordre des 3 lignes de configuration précédentes.

- L'identifiant de session est transmis par un cookie.

- Seul le cookie peut transmettre l'identifiant de session.
- Le PHPSESSID transmis dans l'url est strictement refusé.

Approche pratique : concevoir un accès restreint

Présentation du cas pratique

Le présent chapitre introduit un cas concret d'utilisation des sessions. Il s'agit d'un accès restreint basique. Seul un utilisateur n'est autorisé à être loggué mais cet exemple est à la base de la création d'un espace membre. C'est exactement la même chose. Pour réaliser tout ça, nous aurons besoin de 2 fichiers : le formulaire accompagné de son script de login, et la page protégée. Commençons par le formulaire de login. Le code étant commenté, il n'y aura pas plus d'explications.

Formulaire d'authentification : authentification.php

Cas pratique : formulaire d'identification à un espace membre

```php
<?php
  // Definition des constantes et variables
  define('LOGIN','toto');
  define('PASSWORD','tata');
  $errorMessage = '';

  // Test de l'envoi du formulaire
  if(!empty($_POST))
  {
    // Les identifiants sont transmis ?
    if(!empty($_POST['login']) && !empty($_POST['password']))
    {
      // Sont-ils les mêmes que les constantes ?
      if($_POST['login'] !== LOGIN)
      {
        $errorMessage = 'Mauvais login !';
      }
        elseif($_POST['password'] !== PASSWORD)
      {
        $errorMessage = 'Mauvais password !';
      }
        else
      {
        // On ouvre la session
        session_start();
        // On enregistre le login en session
        $_SESSION['login'] = LOGIN;
        // On redirige vers le fichier admin.php
        header('Location: http://www.monsite.com/admin.php');
        exit();
      }
```

```
    }
      else
    {
      $errorMessage = 'Veuillez inscrire vos identifiants svp
!';
    }
  }
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Formulaire d'authentification</title>
  </head>
  <body>
    <form action="<?php echo
htmlspecialchars($_SERVER['PHP_SELF']); ?>" method="post">
      <fieldset>
        <legend>Identifiez-vous</legend>
        <?php
          // Rencontre-t-on une erreur ?
          if(!empty($errorMessage))
          {
            echo '<p>', htmlspecialchars($errorMessage)
,'</p>';
          }
        ?>
       <p>
          <label for="login">Login :</label>
          <input type="text" name="login" id="login" value=""
/>
        </p>
        <p>
          <label for="password">Password :</label>
          <input type="password" name="password" id="password"
value="" />
          <input type="submit" name="submit" value="Se
logguer" />
        </p>
      </fieldset>
    </form>
  </body>
</html>
```

Exemple de page protégée : admin.php

Exemple de page sécurisée avec les sessions

```
<?php
// On prolonge la session
```

```
session_start();
// On teste si la variable de session existe et contient une
valeur
if(empty($_SESSION['login']))
{
  // Si inexistante ou nulle, on redirige vers le formulaire
de login
  header('Location:
http://www.monsite.com/authentification.php');
  exit();
}
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Administration</title>
  </head>
  <body>
 <?php
    // Ici on est bien loggué, on affiche un message
    echo 'Bienvenue ', $_SESSION['login'];
  ?>
  </body>
</html>
```

Petite explication : pour protéger chacune des pages de l'administration, il faut absolument ajouter en tête de fichier le premier script entièrement. Celui-ci redirige instantanément l'utilisateur s'il n'est pas convenablement loggué. Sinon il affiche la page Web.

Liens annexes

Voici une serie de liens pour obtenir un complément d'information sur l'utilisation des sessions.

- Documentation officielle des sessions - PHP.net
- Les sessions - tutoriel de PHPDebutant.org
- Les sessions - tutoriel de PHPFrance.com
- Créer un espace sécurisé - PHPTeam.net
- La sécurité des sessions PHP
- Ouverture d'une session PHP - Nikrou.net
- Simple comme les sessions - Les mises au point de Frédéric Bouchery

Conclusion

Le tutoriel s'achève sur ces références. Nous vous recommandons vivement de jeter un oeil à la documentation de PHP au sujet des sessions pour regarder du côté des autres fonctions existantes non évoquées ici. Ceci n'était que le fondement théorique des sessions basé sur un exemple concret et pratique. Vous

serez à présent en mesure de construire vos propres applications web à partir de sessions.

Les importations de fichiers avec require() et include()

La grande majorité des sites web dynamiques ou des applications ont besoin de réutiliser des parties de code identique à plusieurs endroits d'une même page, ou bien dans plusieurs pages différentes. C'est le cas par exemple des librairies de fonctions utilisateurs ou bien des fichiers de configuration. Plutôt que de réécrire à chaque fois le code, il existe des fonctions (structures de langage en réalité) capables d'importer et exécuter le code à réutiliser dans la page. Il s'agit des fonctions include() et require()

Les fonctions include() et require()

Tout d'abord pourquoi existe-t-il deux fonctions différentes qui remplissent la même fonction ? Leur fonctionnement est strictement le même mais la différence qui les sépare réside dans la gestion des erreurs.

La fonction include() renverra une erreur de type WARNING si elle n'arrive pas à ouvrir le fichier en question. De ce fait l'exécution du code qui suit dans la page sera exécuté. En revanche, la fonction require() affichera une erreur de type FATAL qui interrompt l'exécution du script.

Laquelle préférer alors ? Tout dépend du contexte en fait. Si le fichier doit obligatoirement être présent pour le reste du programme, alors require() est à préférer sinon un include() fera l'affaire.

Comment utiliser include() et require()

Ces deux fonctions prennent un seul paramètre de type chaîne de caractères. C'est le chemin qui mène au fichier à importer. Le chemin est par défaut relatif par rapport au répertoire dans lequel se trouve le script. Illustrons cela avec des exemples :

```php
<?php

  // Importations avec require()
  require('../dossier/fichier.php');
  require 'fichier2.php';

  // Importations avec include()
  include('../dossier/fichier.php');
  include 'fichier2.php';
?>
```

Nous remarquons ici deux syntaxes possibles pour chacune d'elles. Les parenthèses sont facultatives (comme pour echo() par exemple).

Explications : dans la première importation de chaque fonction, nous demandons d'inclure le fichier "fichier.php" se trouvant dans le répertoire "dossier" situé au même niveau que le script "../". Grâce à ../ nous remontons l'arborescence de fichier d'un niveau (retour dans le dossier parent). Dans la seconde importation, nous souhaitons importer le fichier "fichier2.php" se trouvant au même niveau que ce script.
Lorsqu'un fichier est importé, le code se trouvant à l'intérieur est exécuté. Les variables, constantes, objets, tableaux... du fichier importé peuvent donc être réutilisés dans la suite du programme. Prenons un exemple simple. Nous avons un fichier "config.php" contenant le code suivant :

Listing du fichier "config.php"

```
<?php

  // Définition des variables
  $a = 15;
  $b = 5;
  // Affichage d'un texte
  echo 'Un peu de mathématiques...';
?>
```

Puis un programme principal nommé "programme.php", figurant au même niveau que le premier, qui contient les lignes ci-dessous :

Listing du fichier "programme.php"

```
<?php

  // Importation et exécution du fichier
  require('config.php');
  // Calcul de la somme
  $somme = $a + $b;
  // Affichage de la somme
  echo 'Somme de $a + $b = ', $somme;
?>
```

Le résultat de l'exécution du code est le suivant :

Résultat de l'exécution du fichier "programme.php"

```
Un peu de mathématiques...



Somme de $a + $b = 20
```

Le texte introductif et la somme de $a et $b s'affichent sur la sortie standard. Nous en déduisons que les variables ont bien été créées à l'importation du fichier puis utilisées ensuite dans le programme principal.

Note : ces fonctions ne sont à utiliser uniquement pour des pages intégrant du code PHP à exécuter. Pour l'importation de pages purement statiques (html ou

texte par exemple), il faut utiliser file_get_contents()précédée d'une instruction echo().

Require_once() et include_once(), c'est quoi ?

Les fonctions require_once() et require_once() permettent d'importer **une fois seulement** un fichier même s'il y'a plusieurs tentatives d'importation du fichier dans la page.
Néanmoins l'utilisation de ces deux fonctions est dépréciée pour des raisons d'optimisation. Elles sont en effet plus lentes que leur petite soeur respective car elles doivent vérifier en plus que le fichier n'a été importé qu'une fois.

Include() et require(), sensibles au piratage !!!

Il existe une faille de sécurité très dangereuse lorsque ces fonctions ne sont pas utilisées correctement. C'est une faille liée à la négligence et à la méconnaissance des programmeurs débutants.

Dans nos exemples précédents, il n'y avait aucun risque de piratage car nous avions entré les chemins des fichiers en dur. Mais que se passe-t-il si nous écrivons une horreur de ce type ?

Exemple d'import dynamique de page non sécurisé

```
<?php

  include($_GET['page']);
?>
```

Rappelons-nous le tutoriel sur le traitement des formulaires. Le tableau $_GET contient toutes les valeurs passées dans l'url. Dans cet exemple, nous faisons passer une variable nommée *page* dans l'url qui contient une valeur. Trois cas de figure du comportement d'include() s'offrent à nous :

- La valeur de $_GET['page'] est vide ou mauvaise donc include() ne peut rien importer et renvoie une erreur.
- La valeur de $_GET['page'] est remplie et représente un fichier existant du serveur. L'importation se fera sans problème.
- La valeur de $_GET['page'] est remplie mais contient l'adresse menant à un fichier pirate dangereux présent sur un serveur différent. Par exemple $_GET['page'] vaut*http://www.unsitepirate.com/hacker.php*. Le fichier *hacker.php* sera donc importé et le site piraté !!!

Nous l'aurons bien compris, il ne FAUT JAMAIS utiliser le code précédent pour importer dynamiquement des fichiers.
Quelles solutions pouvons-nous mettre en place dans ce cas ? La première est tout d'abord de préfixer et suffixer la variable afin de rendre faux les liens vers des fichiers pirates et ainsi empêcher leur importation. Ce qui donne par exemple :

```
<?php
```

```
require('/path/to/something/'. $_GET['page'] .'/.php');
```

Ce code nous protège des importations malicieuses mais ne nous met pas à l'abri d'une erreur disgracieuse à l'écran si le fichier n'existe pas et ne peut-être importé.
Une solution intéressante est d'utiliser un tableau associatif dont la clé correspond à la valeur passée dans la variable *$page* de l'url, et la valeur le nom du fichier à importer.
Si le code suivant est écrit dans le fichier *index.php*, alors les pages devront être appellées ainsi :

- http://www.unsite.com/index.php?page=tutoriels
- http://www.unsite.com/index.php?page=downloads
- http://www.unsite.com/index.php?page=contacts

Note *: les pages sont situées dans un répertoire nommé "pages" situé au même niveau que "index.php"*

Exemple d'import dynamique de pages sécurisé

```
<?php

  // Tableau des fichiers à importer
  $arrayPages = array(

    'home' => 'homepage.php',
    'tutoriels' => 'tutoriels.php',
    'downloads' => 'downloads.php',
    'contacts' => 'contacts.php'
  );

  // La variable $page existe-elle dans l'url ?
  if(!empty($_GET['page']))
  {
    // Vérification de la valeur passée dans l'url : est-elle
une clé du tableau ?
    if(array_key_exists(strtolower($_GET['page']),
$arrayPages))
    {
      // Oui, alors on l'importe
      include('pages/'. $arrayPages[ strtolower($_GET['page'])
] );
    }
      else
    {
      // Non, alors on importe un fichier par défaut
      include('pages/erreur-404.php');
    }
  }
    else
  {
    // Non, on affiche la page d'accueil par défaut
    include('pages/'. $arrayPages['home']);
  }
```

```
?>
```

Dans cet exemple, nous vérifions que la page demandée dans l'url correspond à une clé du tableau (cf: fonction array_key_exists()). Si oui, alors on l'importe, sinon on inclut une page d'erreur 404 (fichier non trouvé). Notons la présence de la fonction strtolower() qui force la valeur de $_GET['page'] en minuscules.
Pour un complément d'informations sur la faille include() / require(), nous vous invitons à consulter le billet de Frédéric Bouchery intitulé include(), gouffre ou fêlure ?.

Conclusion

Les fonctions *include()* et *require()* se révèlent très pratiques pour fragmenter du code utilisé dans plusieurs fichiers à la fois. Néanmoins il faut les utiliser avec prudence pour éviter d'ouvrir des failles aux pirates amateurs.

Imagefilter() : les effets spéciaux

Le langage PHP permet de manipuler les images depuis de nombreuses années et pour appliquer des effets spéciaux sur celle-ci, nous étions souvent obligés d'écrire de nombreuses lignes de programmation. Depuis la version PHP 5, une nouvelle

fonction est apparue : « IMAGEFILTER » permettant d'obtenir des effets avec la même qualité que des logiciels de dessins.
Cette fonction permet de personnaliser l'aspect visuel de votre site ou aussi de réaliser une galerie photos différentes des autres.

Nous verrons donc :

- Les bases
- La fonction ImageFilter()
- Comment jouer avec quelques filtres ?

Les bases

Avant d'utiliser ces nouveaux effets, il est important d'avoir la librairie GD activée. Si celle-ci n'est pas active, sachez qu'elle se trouve dans le fichier de php.ini. Pour vérifier que la librairie GD fonctionne correctement, nous allons tenter de charger une image, puis l'afficher :

Chargement et affichage d'une image JPG

```
<?php


  $image = @imagecreatefromjpeg('paysage.jpg'); // Charge
l'image JPG
  imagejpeg($image); // Affiche l'image
  imagedestroy($image); // libère l'image
?>
```

Voici le résultat :

La fonction ImageFilter

Cette fonction permet d'appliquer un filtre sur une image. Elle se présente comme ceci :

Prototype de la fonction imagefilter()

```
imagefilter ( resource $image , int $filtertype [, int $arg1 [, int $arg2 [, int $arg3 ]]] )
```

Imagefilter() se décompose en 3 critères :

- *image* : nom de l'image
- *filtertype* : le type de filtre de son choix
- *arg* : les critères, si nécessaire, par rapport au type de filtre choisi. La tranche de nuance permet d'effectuer de -255 à +255

Pour illustrer nos différents exemples, nous partirons sur le paysage que nous avons affiché ci-dessus.

Le filtre IMG_FILTER_BRIGHTNESS

Ce filtre permet de modifier la luminosité de l'image. Nous n'utiliserons que l'argument 1 (arg1). La valeur possible sera comprise entre -255 et 255 qui représente :

- 255 : Eclaircir l'image avec un maximum vers le blanc (effet de brillance)
- 0 : Valeur par défaut. Couleur inchangée
- -255 : Assombrir l'image au maximum vers le noir (effet sombre)

Exemple d'utilisation du filtre IMG_FILTER_BRIGHTNESS

```
<?php


  $nom_fichier='paysage.jpg';
  $valeur=0;
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_BRIGHTNESS, $valeur);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Voici quelques resultats avec différentes valeurs :

$valeur=-100	$valeur=-5
	

$valeur=100	$valeur=50
	

Le filtre IMG_FILTER_COLORIZE

Ce filtre permet de modifier les tendances des couleurs, c'est à dire de soustraire une couleur par rapport aux autres. Nous utiliserons les 3 arguments, qui représentent les nuances de *Rouge*, *Vert*, *Bleu.*Chacune des valeurs de ses nuances elles se trouvant un intervalle de –255 à 255.

Exemple d'utilisation du filtre IMG_FILTER_COLORIZE

```
<?php


  $nom_fichier='paysage.jpg';
  $r=100;
  $v=0;
  $b=-50;
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image,IMG_FILTER_COLORIZE,$r,$v,$b);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Voici quelques résultats avec quelques valeurs différentes :

R : 0 V :-100 B : 0	R : 0 V :100 B : 5

R : 100 V : 0 B : -50

Le filtre IMG_FILTER_CONTRAST

Ce filtre modifie le contraste de l'image. Pour déterminer ce contraste, nous utiliserons l'argument 1 (arg1). Sa valeur est comprise dans l'intervalle -255 et 255 où :

- 255 : Eclaircir l'image avec un maximum vers le blanc (effet de brillance)
- 0 : Valeur par défaut. Couleur inchangée
- -255 : Assombrir l'image au maximum vers le noir (effet sombre)

Exemple d'utilisation du filtre IMG_FILTER_CONTRAST

```
<?php


  $nom_fichier='paysage.jpg';
  $valeur=-50;
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image,IMG_FILTER_CONTRAST,$valeur);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Voici quelques résultats :

-100

-50

15

Le filtre IMG_FILTER_EDGEDETECT

Ce filtre utilise la détection des bords pour les mettre en évidence dans l'image. Aucun argument sera utilisé car ce filtre est appliqué sur l'ensemble de l'image

Exemple d'utilisation du filtre IMG_FILTER_EDGEDETECT

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_EDGEDETECT);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Le filtre IMG_FILTER_EMBOSS

Ce filtre permet de graver l'image en relief. Aucun argument sera utilisé car ce filtre est appliqué sur l'ensemble de l'imag.

Exemple d'utilisation du filtre IMG_FILTER_EMBOSS

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier); /* Tentative
d'ouverture */
  imagefilter($image, IMG_FILTER_EMBOSS);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Le filtre IMG_FILTER_GAUSSIAN_BLUR

Ce filtre permet de brouiller l'image en utilisant la méthode gaussienne. Aucun argument n'est utilisé car ce filtre est appliqué sur l'ensemble de l'image. Il peut être associé au filtre*IMG_FILTER_SELECTIVE_BLUR* (voir plus loin).

Exemple d'utilisation du filtre IMG_FILTER_GAUSSIAN_BLUR

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_GAUSSIAN_BLUR);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Le filtre IMG_FILTER_SELECTIVE_BLUR

Ce filtre permet de rendre flou une image. Aucun argument n'est utilisé car ce filtre est appliqué sur l'ensemble de l'image. Il peut être associé au filtre *IMG_FILTER_GAUSSIAN_BLUR* (voir plus loin).

Exemple d'utilisation du filtre IMG_FILTER_SELECTIVE_BLUR

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_SELECTIVE_BLUR);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Le filtre IMG_FILTER_SMOOTH

Ce filtre permet de lisser l'image. Nous utiliserons l'argument 1 (arg1) pour déterminer le degré de lissage. La valeur possible sera comprise entre -255 et 255 qui représente :

- 255 : Eclaircir l'image avec un maximum vers le blanc (effet de brillance)
- 0 : Valeur par défaut. Couleur inchangée
- -255 : Assombrir l'image au maximum vers le noir (effet sombre)

Exemple d'utilisation du filtre IMG_FILTER_SMOOTH

```
<?php


  $nom_fichier='paysage.jpg';
  $valeur=50;
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_SMOOTH, 10);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Voici quelques résultats avec des valeurs différentes :

100	50
	
-100	-50

Le filtre IMG_FILTER_GRAYSCALE

Ce filtre convertit l'image en noir et blanc, c'est à dire que nous retrouvons les mêmes possibilités que le filtre *IMG_FILTER_COLORIZE*, sauf qu'ici, nous ne pouvons pas choisir la couleur.

Exemple d'utilisation du filtre IMG_FILTER_GRAYSCALE

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_GRAYSCALE);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Le filtre IMG_FILTER_NEGATE

Ce filtre permet de renverser toutes les couleurs de l'image pour la rendre en négative.

Exemple d'utilisation du filtre IMG_FILTER_NEGATE

```
<?php
```

```
  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_NEGATE);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Le filtre IMG_FILTER_MEAN_REMOVAL

Ce filtre permet d'appliquer un effet de bruit à l'image.

Exemple d'utilisation du filtre IMG_FILTER_MEAN_REMOVAL

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_MEAN_REMOVAL);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Maintenant que nous avons vu l'ensemble des filtres possibles pour cette fonction, nous pouvons appliquer plusieurs filtres sur une même image.

Jouons avec quelques types de filtres

Nous allons, pour chaque effet présenté, partir sur une image d'origine différente. Le premier effet consistera en la dissociation des teintes RVB de l'image. Le second script que nous découvrirons applitira notre image. Notre troisième essai aura pour objectif d'appliquer un flou, puis nous clôturerons ce tutoriel par la création d'un d'effet monochrome. Etudions le premier effet :

Effet 1 : Dissocier les couleurs

Pour réaliser cet effet très répandu dans les logiciels de dessins, nous devrons appliquer deux opérations à l'image. Tout d'abord, il faut convertir l'image en noir et blanc avec le filtre *IMG_FILTER_GRAYSCALE*, puis recoloriser la teinte de son choix avec le filtre *IMG_FILTER_COLORIZE*.

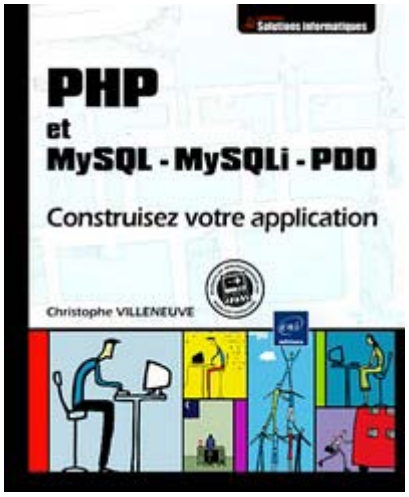
Script de dissociation des couleurs RVB d'une image

```php
<?php



  // Teinte rouge
  $r=255; $v=0; $b= 0;

  // Teinte verte
  $r=0;$v=255;$b= 0;

  // Teinte bleue
  $r=0;$v=0;$b= 255;


  $nom_fichier='livreeni.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  imagefilter($image, IMG_FILTER_GRAYSCALE);
  imagefilter($image, IMG_FILTER_COLORIZE, $r,$v,$b);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Vous l'aurez compris,vous devrez exécuter le script à trois reprise en commantant à chaque fois deux des trois lignes de code numéro 4, 7 et 10.

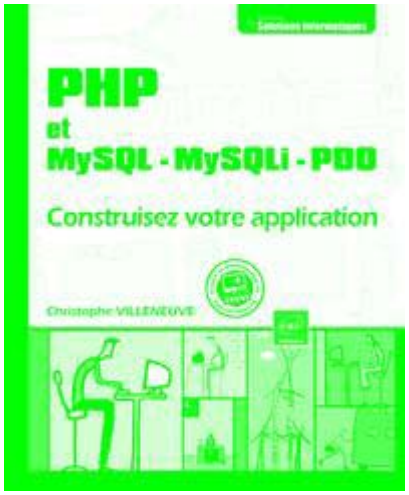
Nous obtenons comme résultat final :

Image originale	$r=255; $v=0; $b=0



$r=0; $v=255; $b=0

$r=0; $v=0; $b=255



Effet 2 : Effet aplatir

Pour réaliser cet effet d'aplatissement qui peut se rapprocher d'un effet de gravure, nous devrons décomposer l'algorithme en deux opérations élémentaires.

- Détecter les différences de l'image avec le filtre IMG_FILTER_EDGEDETECT
- Transformer l'image en relief avec le filtre IMG_FILTER_EMBOSS

Exemple de script d'aplatissement d'une image

```
<?php


  $nom_fichier='afup.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
```

```
  imagefilter($image, IMG_FILTER_EDGEDETECT);
  imagefilter($image, IMG_FILTER_EMBOSS);
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Image originale	Image avec effet applâti
	

Effet 3 : Effet de flou

Cet effet montre comment créer un effet de flou. Nous utiliserons 2 filtres pour arriver à nos fins :

- Brouiller l'image avec le filtre *IMG_FILTER_GAUSSIAN_BLUR*
- Brouiller un peu plus avec le filtre *IMG_FILTER_SELECTIVE_BLUR*

Script de réalisation d'un flou sur une image

```
<?php


  $nom_fichier='php_logo.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  Imagefilter ($image, IMG_FILTER_GAUSSIAN_BLUR);
  imagefilter ($image , IMG_FILTER_SELECTIVE_BLUR );
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

Image originale	Image avec effet de flou
	

Effet 4 : Effet monochrome

Nous allons effectuer l'opération en 2 temps :

- Transformer une image en gris avec le filtre *IMG_FILTER_GRAYSCALE*
- Ressortir les niveaux de couleurs avec le filtre *IMG_FILTER_NEGATE*

Script de création de flou sur une image

```
<?php


  $nom_fichier='paysage.jpg';
  $image = @imagecreatefromjpeg($nom_fichier);
  Imagefilter ($image, IMG_FILTER_GRAYSCALE);
  imagefilter ( $image , IMG_FILTER_NEGATE );
  imagejpeg($image);
  imagedestroy($image);
?>
```

Pour obtenir le résultat suivant :

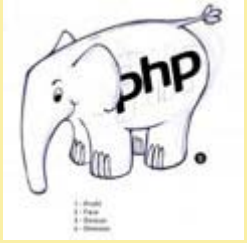
Image originale	Image avec filtre monochrome
	

Conclusion

Il est possible de réaliser tous sortes d'effets supplémentaires avec l'ensemble des filtres qui nous sont proposés. Il ne tient plus qu'à vous d'effectuer des tests par vous même.

Les classes et objets

- *Par Emacs*
- *26 commentaires*
- *40 017 lectures*
- *Format PDF*
- *RSS* - *Atom*



La programmation par objet (POO) a été intégrée au langage PHP dans sa version 4. Mais à cette époque, le modèle objet de PHP était beaucoup trop sommaire. Nous ne pouvions réellement parler de programmation orientée objet. Les développeurs de PHP se sont alors penchés sur la question et ont amélioré ce modèle objet qui, depuis la version 5, n'a plus rien à envier aux autres langages objets comme Java ou C++.
Ce premier cours sur la programmation orientée objet sera une première découverte et une présentation de la syntaxe d'écriture de classes. Nous aborderons dans un premier temps les avantages relatifs à une approche par objet. Puis nous définirons les notions de classes, d'objets, d'instance et de type avant de nous lancer pleinement dans l'écriture de nos premières classes.

Quels sont les avantages et inconvénients d'une approche objet ?

La programmation orientée objet offre de nombreux avantages. Parmi eux :

- La possibilité de réutiliser le code dans différents projets. Les classes ainsi créées pourront avoir une nouvelle vie dans une application tierce.
- Une conception de l'algorithme plus claire et organisée. Le programmeur identifie chaque élément de son programme comme un objet ayant son contexte, ses propriétés et des actions qui lui sont propres.
- Un code modulaire. Chaque type d'objet possède son propre contexte et ne peut agir avec d'autres suivant des interfaces bien précises. Cela permet d'isoler chaque module et d'en créer séparemment de nouveaux qui viendront s'ajouter à l'application.Cette approche est particulièrement employée dans le cas de projets répartis entre pusieurs développeurs.
- Possibilité de s'adapter aux *design patterns (motifs de conception)* pour une meilleure structuration du code.

Il existe cependant quelques inconvénients à l'utilisation de la programmation par objet :

- Une application orientée objet mal conceptualisée sera difficilement maintenable et modulaire.
- La programmation par objet nécessite généralement plus de ressources et de temps d'exécution qu'un code procédural.

Remarque : *la programmation par objet n'est pas synonyme de bonne programmation. Il existe d'excellents développeurs capables de produire du bon code procédural, maintenable, structuré et modulaire. Mais il existe aussi des développeurs en programmation objet qui produisent du mauvais code suite à un manque de conceptualisation.*

Qu'est-ce qu'un objet ?

Définition

Un « objet » est une représentation d'une chose matérielle ou immatérielle du réel à laquelle on associe des propriétés et des actions.

Par exemple : une voiture, une personne, un animal, un nombre ou bien un compte bancaire peuvent être vus comme des objets.

Attributs

Les « attributs » (aussi appelés « données membres ») sont les caractères propres à un objet.

Une personne, par exemple, possèdent différents attributs qui lui sont propres comme le nom, le prénom, la couleur des yeux, le sexe, la couleur des cheveux, la taille...

Méthodes

Les « méthodes » sont les actions applicables à un objet.

Un objet personne, par exemple, dispose des actions suivantes : manger, dormir, boire, marcher, courir...

Qu'est-ce-qu'une classe ?

Une « classe » est un modèle de données définissant la structure commune à tous les objets qui seront créés à partir d'elle. Plus concrètement, nous pouvons percevoir une classe comme un moule grâce auquel nous allons créer autant d'objets de même ***type*** et de même structure qu'on le désire.
Par exemple, pour modéliser n'importe quelle personne, nous pourrions écrire une classe Personne dans laquelle nous définissons les attributs (couleurs des yeux, couleurs des cheveux, taille, sexe...) et méthodes (marcher, courir, manger, boire...) communs à tout être humain.

Qu'est-ce-qu'une instance ?

Une instance est une représentation particulière d'une classe.

Lorsque l'on crée un objet, on réalise ce que l'on appelle une « instance de la classe ». C'est à dire que du moule, on en extrait un nouvel objet qui dispose de ses attributs et de ses méthodes. L'objet ainsi créé aura pour type le nom de la classe.

Par exemple, les objets Hugo, Romain, Nicolas, Daniel sont des instances (objets) de la classe Personne.
***Remarque** : une classe n'est pas un objet. C'est un abus de langage de dire qu'une classe et un objet sont identiques.*

Déclaration d'une classe

Nous venons de définir le vocabulaire propre à la programmation orientée objet. Entrons à présent dans le vif du sujet, c'est à dire la déclaration et l'instanciation d'une classe. Nous allons déclarer une classe*Personne* qui nous permettra ensuite de créer autant d'instances (objets) de cette classe que nous le souhaitons.

Syntaxe de déclaration d'une classe

Le code suivant présente une manière de déclarer et de structurer correctement une classe. Nous vous conseillons vivement de suivre ces conventions d'écriture.

Déclaration d'une classe PHP 5

```php
<?php

  class NomDeMaClasse
  {
    // Attributs

    // Constantes

    // Méthodes
  }

?>
```

Voici par exemple ce que cela donne avec notre exemple :

Exemple de la classe Personne

```php
<?php

  class Personne
  {
    // Attributs
    public $nom;
    public $prenom;
    public $dateDeNaissance;
```

```
    public $taille;
    public $sexe;

    // Constantes
    const NOMBRE_DE_BRAS = 2;
    const NOMBRE_DE_JAMBES = 2;
    const NOMBRE_DE_YEUX = 2;
    const NOMBRE_DE_PIEDS = 2;
    const NOMBRE_DE_MAINS = 2;

    // Méthodes
    public function __construct() { }

    public function boire()
    {
      echo 'La personne boit<br/>';
    }

    public function manger()
    {
      echo 'La personne mange<br/>';
    }
}
```

***Remarque** : par convention, on écrit le nom d'une classe en « Upper Camel Case »*, c'est à dire que tous les mots sont accrochés et chaque première lettre de chaque mot est écrit en capital.
Comme vous pouvez le constater, nous avons déclaré 5 attributs publics, 5 constantes, 1 méthode constructeur et 2 méthodes classiques. Détaillons chaque élément.

Les attributs

Comme nous l'avons expliqué au début de ce cours, les attributs sont les caractéristiques propres d'un objet. Toute personne possède un nom, un prenom, une date de naissance, une taille, un sexe... Tous ces éléments caractérisent un être humain.

Par cet exemple, nous déclarons les attributs de notre classe *public.* Nous expliquerons dans un tutoriel suivant les trois niveaux de visibilité (*public, private et protected)* qui peuvent être appliqués à un attribut.
Le mot-clé *public* permet de rendre l'attribut accessible depuis l'extérieur de la classe. Ce n'est pas une bonne pratique à adopter mais pour ce premier tutoriel nous l'utiliserons tel quel pour faciliter la compréhension.
En programmation orientée objet, un attribut n'est ni plus ni moins qu'une variable.

Notez également que nous avons juste déclaré les attributs. En revanche, aucun type ni aucune valeur ne leur ont été attribués. Ils sont donc par défaut initialisés à la valeur NULL.

***Remarque** : deux classes différentes peuvent avoir les même attibuts sans risque de conflit.*

Les constantes

Il est aussi possible de déclarer des constantes propres à la classe. Contrairement au mode procédural de programmation, une constante est déclarée avec le mot-clé *const.*
***Remarque** : une constante doit être déclarée et initialisée avec sa valeur en même temps.*

Le constructeur

Le constructeur est une méthode particulière. C'est elle qui est appelée implicitement à la création de l'objet (instanciation).

Dans notre exemple, le constructeur n'a ni paramètre ni instruction. Le programmeur est libre de définir des paramètres obligatoires à passer au constructeur ainsi qu'un groupe d'instructions à exécuter à l'instanciation de la classe. Nous nous en passerons pour simplifier notre exemple.

***Remarque** : en PHP, la surcharge de constructeur et de méthodes n'est pas possible. On ne peut définir qu'une seule et unique fois la même méthode.*

Les méthodes

Les méthodes sont les actions que l'on peut appliquer à un objet. Il s'agit en fait de fonctions qui peuvent prendre ou non des paramètres et retourner ou non des valeurs / objets. S'agissant d'actions, nous vous conseillons de les nommer avec un verbe à l'infinitif.

Elles se déclarent de la même manière que des fonctions traditionnelles.
Au même titre que les attributs, on déclare une méthode avec un niveau de visibilité. Le mot-clé *public* indique que l'on pourra appliquer la méthode en dehors de la classe, c'est à dire sur l'objet lui même.
***Remarque** : deux classes différentes peuvent avoir les mêmes méthodes sans risque de conflit.*

Utilisation des classes et des objets

Notre classe est désormais prête mais ne sert à rien tout seule. Comme nous l'avons expliqué plus haut, une classe est perçue comme un moule capable de réaliser autant d'objets de même type et de même structure qu'on le souhaite. Nous allons donc présenter maintenant la phase de concrètisation d'une classe.

Instanciation d'une classe

L'instanciation d'une classe est la phase de création des objets issus de cette classe. Lorsque l'on instancie une classe, on utilise le mot-clé *new* suivant du nom de la classe. Cette instruction appelle la méthode constructeur (__construct()) qui construit l'objet et le place en mémoire. Voici un exemple qui illustre 4 instances différentes de la classes *Personne.*

Création d'objets de type Personne

```
<?php

  $personne1 = new Personne();
  $personne2 = new Personne();
  $personne3 = new Personne();
  $personne4 = new Personne();
```

Un objet est en fait une variable dont le type est celui de la classe qui est instanciée.

***Remarque** : si nous avions défini des paramètres dans la méthode constructeur de notre classe, nous aurions du les indiquer entre les parenthèses au moment de l'instance. Par exemple : $personne1 = new Personne('Hamon','Hugo');*

Accès aux attributs

Accès en écriture

Nous venons de créer 4 objets de même type et de même structure. Dans l'état actuel, ce sont des clones car leurs attributs respectifs sont tous déclarés mais ne sont pas initialisés. Nous affectons à présent des valeurs à chacun des attributs de chaque objet.

Utilisation des attributs d'un objet

```
<?php

  // Définition des attributs de la personne 1
  $personne1->nom = 'Hamon';
  $personne1->prenom = 'Hugo';
  $personne1->dateDeNaissance = '02-07-1987';
  $personne1->taille = '180';
  $personne1->sexe = 'M';

  // Définition des attributs de la personne 2
  $personne2->nom = 'Dubois';
  $personne2->prenom = 'Michelle';
  $personne2->dateDeNaissance = '18-11-1968';
  $personne2->taille = '166';
  $personne2->sexe = 'F';

  // Définition des attributs de la personne 3
  $personne3->nom = 'Durand';
  $personne3->prenom = 'Béatrice';
  $personne3->dateDeNaissance = '02-08-1975';
  $personne3->taille = '160';
  $personne3->sexe = 'F';

  // Définition des attributs de la personne 4
```

```
  $personne4->nom = 'Martin';
  $personne4->prenom = 'Pierre';
  $personne4->dateDeNaissance = '23-05-1993';
  $personne4->taille = '155';
  $personne4->sexe = 'M';
```

Nous avons maintenant des objets ayant chacun des caractèristiques différentes.

Dans notre exemple, nous accèdons directement à la valeur de l'attribut. Cela est possible car nous avons défini l'attribut comme étant *public.* Si nous avions déclaré l'attribut avec les mots-clés *private* ou*protected*, nous aurions du utiliser un autre mécanisme pour accéder à sa valeur ou bien la mettre à jour. Nous verrons cela dans un prochain cours.

Accès en lecture

La lecture de la valeur d'un attribut d'un objet se fait exactement de la même manière que pour une variable traditionnelle. Le code suivant présente comment afficher le *nom* et le *prénom* de la personne 1.

Affichage du nom et du prénom de la personne 1

```
<?php

  echo 'Personne 1 :<br/><br/>';
  echo 'Nom : ', $personne1->nom ,'<br/>';
  echo 'Prénom : ', $personne1->prenom;
```

L'exécution de ce programme produit le résultat suivant sur la sortie standard :

Résultat d'exécution du code

```
Personne 1 :


Nom : Hamon

Prénom : Hugo
```

Accès aux constantes

L'accès aux constantes ne peut se faire qu'en lecture via l'opérateur **::** . L'exemple suivant illustre la lecture d'une constante de la classe *Personne.*

Accès à une constante

```
<?php

  echo 'Chaque personne a ', Personne::NOMBRE_DE_YEUX ,'
yeux.';
```

```
?>
```

L'exécution de ce code affiche la chaine suivante sur la sortie standard :

Résultat de l'exécution du code

```
Chaque personne a 2 yeux.
```

***Remarque** : si l'on tente de redéfinir la valeur d'une constante, PHP génèrera une erreur de ce type :*

Erreur générée en cas de redéfinition de constante

```
Parse error: syntax error, unexpected '=' in
/Users/Emacs/Sites/Demo/Autres/Personne.php on line 36
```

Accès aux méthodes

Avant de clore ce tutoriel, il ne nous reste plus qu'à présenter l'utilisation des méthodes. Leur utilisation est exactement la même que pour les attributs. Rappellez-vous, nous avions défini deux méthodes pour notre classe *Personne.* Il s'agit des méthodes *boire()* et *manger().* Toutes deux affichent une chaine de caractères sur la sortie standard lorsqu'elles sont exécutées.
Reprenons nos 4 objets précédents. Nous simulons que ces 4 personnes sont à table et qu'elles dînent. Simultanément les deux premières personnes boivent le contenu d'un verre pendant que les deux autres mangent. Cela se représente donc par l'appel de la méthode *boire()* sur les objets *personne1* et*personne2*, puis par l'appel de la méthode *manger()* sur les objets *personne3* et *personne4*.

Appel de méthode sur des objets

```
<?php

  $personne1->boire();
  $personne2->boire();
  $personne3->manger();
  $personne4->manger();

?>
```

Après exécution, le résultat est le suivant :

Résulat de l'exécution du code

```
La personne boit

La personne boit

La personne mange

La personne mange
```

Remarque : *comme pour le constructeur, si nos méthodes avaient eu besoin de paramètres en entrée, nous les aurions indiqués au moment de l'appel entre les parenthèses de la fonction.*

Conclusion

A l'issue de ce premier tutoriel d'initiation à la programmation orientée objet en PHP5, nous avons expliqué les notions d'objet, de classe et d'instance. Puis nous avons créé notre première classe contenant attributs, constantes et méthodes. Ensuite, nous avons instancié notre classe pour fabriquer 4 objets auxquels nous avons mis à jour les valeurs des attributs. Enfin, nous nous sommes intéressés à l'appel de méthodes pour chacun de nos objets.

Les prochains tutoriels de ce chapitre sur la programmation orientée objet présentera les notions de visibilité, des attributs et méthodes, la référencement à l'objet en cours, les méthodes et constantes magiques, l'héritage, les interfaces, les exceptions...

Visibilité des propriétés et des méthodes

La visibilité des propriétés et des méthodes d'un objet constitue une des particularités élémentaires de la programmation orientée objet. Ce tutoriel a pour objectif de présenter les différents niveaux de visibilité que propose le modèle objet de PHP 5. Nous les passerons en revue un par un au travers d'exemples pratiques et nous apporterons quelques bonnes pratiques à adopter lorsqu'on les utilise.

Qu'est-ce que la visibilité des propriétés et méthodes d'un objet ?

La visibilité permet de définir de quelle manière un attribut ou une méthode d'un objet sera accessible dans le programme. Comme Java, C++ ou bien ActionScript 3, PHP introduit 3 niveaux différents de visibilité applicables aux propriétés ou méthodes de l'objet.
Il s'agit des visibilités ***publiques, privées*** ou ***protégées*** qui sont respectivement définies dans la classe au moyen des mots-clés ***public, private*** et ***protected***.
L'exemple suivant illustre la syntaxe de déclaration de données membres (attributs) et de méthodes ayant des visibilités différentes. Nous expliquerons juste après chaque particularité des niveaux de visibilité.

Utilisation des mots-clés public, private et protected

```php
<?php

class Vehicule
{
  // Attributs
  public $marque;
  private $_volumeCarburant;
  protected $_estRepare;

  // Méthodes
```

```
  public function __construct()
  {
    $this->_volumeCarburant = 40;
    $this->_estRepare = false;
  }

  // Démarre la voiture si le réservoir
  // n'est pas vide
  public function demarrer()
  {
    if ($this->_controlerVolumeCarburant())
    {
      echo 'Le véhicule démarre';
      return true;
    }

    echo 'Le réservoir est vide...';
    return false;
  }

  // Vérifie s'il y'a du carburant dans le réservoir
  private function _controlerVolumeCarburant()
  {
    return ($this->_volumeCarburant > 0); // renvoi true ou
false
  }

  // Met le véhicule en maintenance
  protected function reparer()
  {
    $this->_estRepare = true;
    echo 'Le véhicule est en réparation';
  }
}
```

Nous remarquons ici que nous utilisions déjà les attributs et méthodes publiques dans les exemples du premier tutoriel sur la programmation orientée objet. Présentons à présent chacun des niveaux de visibilité.

L'accès public

C'est l'accès par défaut de PHP si l'on ne précise aucune visibilité. Tous les attributs et méthodes qui sont déclarés sans l'un de ces trois mots-clés sera considéré automatiquement par l'interprêteur comme*publique*.
Le mot-clé ***public*** indique que les propriétés et méthodes d'un objet seront accessibles depuis n'importe où dans le programme principal ou bien dans les classes mères héritées ou classes filles dérivées. Retenez ces termes dans un coin de votre mémoire car nous présenterons les notions d'héritage au prochain tutoriel.
Concrètement, c'est ce que nous faisions dans le tutoriel précédent. Ce qui donne avec notre exemple :

Utilisation de la visibilité publique

```php
<?php

  // Instanciation de l'objet : appel implicite à la méthode
__construct()
  $monVehicule = new Vehicule();

  // Mise à jour de la marque du véhicule
  $monVehicule->marque = 'Peugeot';

  // Affichage de la marque du véhicule
  echo $monVehicule->marque;
?>
```

Dans cet exemple, nous remarquons que nous pouvons lire et modifier directement la valeur de l'attribut publique *marque* en l'appellant directement de cette manière : *$monVehicule->marque* (sans le dollars).
Nous verrons juste après que cette même utilisation sera impossible avec les attributs privés et protégés.

Note : *lorsque l'on instancie la classe pour créer un nouvel objet, le mot-clé "new" se charge d'appeler la méthode constructeur de la classe. Cette dernière doit donc obligatoirement être déclarée publique car elle est appellée depuis l'extérieur de la classe.*

L'accès private

Le mot-clé ***private*** permet de déclarer des attributs et des méthodes qui ne seront visibles et accessibles directement que depuis l'intérieur même de la classe. C'est à dire qu'il devient impossible de lire ou d'écrire la valeur d'un attribut privé directement en faisant *$monVehicule->_volumeCarburant.* De même pour utiliser la méthode *_controlerVolumeCarburant()* via *$monVehicule->_controlerVolumeCarburant().* Vous remarquerez en testant, qu'une erreur fatale est générée à l'exécution.
La valeur de l'attribut privé *$_volumeCarburant* est initialisée dans le constructeur de la classe. C'est à dire quand on construit l'objet.
Par exemple, lorsque notre véhicule est enfin assemblé et prêt à être utilisé, il sera vendu au client avec le réservoir d'essence rempli,soit 40L.

Une méthode est déclarée *private* lorsqu'elle n'a pas vocation à être utilisée en dehors de la classe. Les méthodes privées servent notamment à valider des données dans un objet ou bien à effectuer différents traitements internes. C'est par exemple la tâche que remplit la méthode privée*_controlerVolumeCarburant()* lorsqu'elle est appellée par la méthode publique *demarrer().* Cette méthode privée vérifie qu'il y'a bien du carburant dans le véhicule en testant la valeur de l'attribut privé*$_volumeCarburant.* Si tel est le cas, la méthode retourne *TRUE* et donc la voiture démarre. Dans le cas contraire, la valeur retournée est *FALSE* et la voiture ne peut démarrer.

Appel de la méthode demarrer() sur l'objet $monVehicule

```php
<?php

  // Affiche : "Le véhicule démarre"
  $monVehicule->demarrer();

?>
```

Note : *par convention, on déclare les attributs privés et protégés avec un underscore afin de les identifier plus facilement lorsque l'on relit le code. Cette règle d'usage n'est pas suivie par tous les développeurs mais nous vous recommandons vivement de l'adopter dans vos développements.*

L'accès protected

L'accès protégé (*protected*) est un intermédiaire entre l'accès publique et l'accès privé. Il permet d'utiliser des attributs et méthodes communs dans une classe parente et ses classes dérivées (héritantes).
Le chapitre concernant l'héritage n'est pas au programme de ce tutoriel. Nous l'étudierons dans le prochain cours. Toutefois, nous allons devoir l'anticiper afin de comprendre comment fonctionne l'accès protégé. Pour cela, nous allons ajouter une classe *Voiture* qui héritera de notre classe actuelle *Véhicule*. Comme une voiture est une sorte de véhicule, il nous est donc possible de dériver la classe *Véhicule* avec une classe *Voiture*.
La classe *Vehicule* contiendra les attributs et méthodes communs à tout type de véhicule tandis que la classe *Voiture* encapsulera les propriétés et méthodes propres aux voitures. Par exemple, la *marque* ou le booléen *estRepare* peuvent très bien être des attributs partagés. Par contre le volume de carburant dans le réservoir n'est pas commun à tous les véhicules.
En effet, un vélo, une trottinette ou bien encore un aviron peuvent être considérés comme des véhicules non motorisés. Ils n'ont donc pas de réservoir et ne fonctionnent pas avec du carburant.

Utilisation des accès protégés

```php
<?php

class Vehicule
{
  // Attributs
  protected $_marque;
  protected $_estRepare;

  // Méthodes
  public function __construct($marque)
  {
    $this->_marque = $marque;
    $this->_estRepare = false;
  }

  // Met le véhicule en maintenance
  public function reparer()
```

```
  {
    $this->_estRepare = true;
    echo 'Le véhicule est en réparation';
  }
}


class Voiture extends Vehicule
{
  // Attributs
  private $_volumeCarburant;

  // Constructeur
  public function __construct($marque)
  {
    // Appel du constructeur de la classe parente
    parent::__construct($marque);
    $this->_volumeCarburant = 40;
  }

  // Démarre la voiture si le réservoir
  // n'est pas vide
  public function demarrer()
  {
    if ($this->_controlerVolumeCarburant())
    {
      echo 'Le véhicule démarre';
      return true;
    }

    echo 'Le réservoir est vide...';
    return false;
  }

  // Vérifie qu'il y'a du carburant dans le réservoir
  private function _controlerVolumeCarburant()
  {
    return ($this->_volumeCarburant > 0);
  }
}
```

Nous venons de déclarer deux attributs protégés dans la classe parente *Vehicule* et nous avons redescendu l'attribut *_volumeCarburant* dans la classe *Voiture*. De même nous avons redescendu les méthodes *demarrer()* et *_controlerVolumeCarburant()* dans la classe Voiture. Ainsi, nous pouvons désormais instancier un nouvel objet de type Voiture et profiter des attributs et des méthodes de la classe*Vehicule* en plus. Enfin, notez que la méthode *reparer()* est passée du mode *protected* à *public* afin de pouvoir l'appeler depuis l'extérieur de la classe. Ce qui donne :

Utilisation des attributs et méthodes protégés

```php
<?php

  $monVehicule = new Voiture('Peugeot');
  $monVehicule->demarrer();
  $monVehicule->reparer();

?>
```

Mise à jour d'un attribut privé ou protégé : rôle du mutator

Nous avons évoqué plus haut qu'il était impossible d'accéder à la valeur d'un attribut privé ou protégé en utilisant la syntaxe suivante : *$objet->attribut*. Une erreur fatale est générée. Comment faire pour mettre à jour la valeur de l'attribut dans ce cas ? C'est là qu'intervient le ***mutator.***
Le mutator n'est ni plus ni mois qu'une méthode publique à laquelle on passe en paramètre la nouvelle valeur à affecter à l'attribut. Par convention, on nomme ces méthodes avec le préfixe ***set.*** Par exemple :*setMarque(), setVolumeCarburant()...* C'est pourquoi on entendra plus souvent parler de ***setter*** que de***mutator.***
Ajoutons un mutator à notre classe *Voiture* qui permet de modifier la valeur du volume de carburant. Cela donne :

Ajout du mutator (setter) setVolumeCarburant à la classe Voiture

```php
<?php

class Voiture extends Vehicule
{

  // ...

  public function setVolumeCarburant($dVolume)
  {
    $this->_volumeCarburant = $dVolume;
  }
}
```

Pour respecter les conventions de développement informatique, nous utilisons dans notre exemple l'écriture au moyen du préfixe *set.* Mais ce nom est complètement arbitraire et notre méthode aurait très bien pu s'appeller *faireLePlein()* ou *remplirLeReservoir().* Il ne tient qu' à vous de donner des noms explicites à vos méthodes.
Quant à l'utilisation de cette méthode, il s'agit tout simplement d'un appel de méthode traditionnel.

Mise à jour de la valeur de l'attribut privé $_volumeCarburant

```php
<?php

  // Je remplis mon réservoir de 50 L d'essence
  $monVehicule->setVolumeCarburant(50);
```

```
?>
```

Obtenir la valeur d'un attribut privé ou protégé : rôle de l'accessor

Nous venons d'étudier comment mettre à jour (écriture) la valeur d'un attribut privé ou protégé. Il ne nous reste plus qu' à aborder le cas de la lecture de cette valeur. Comment restitue-t-on cette valeur dans un programme sachant que l'accès direct à l'attribut est interdit ? De la même manière que pour le mutator, nous devons déclarer une méthode qui va se charger de **retourner** la valeur de l'attribut. Ce n'est donc ni plus ni moins qu'une fonction avec une instruction ***return.***
On appelle ce type de méthode un ***accessor*** ou bien plus communément un ***getter.*** En effet, par convention, ces méthodes sont préfixées du mot ***get*** qui se traduit en français par « *obtenir, récupérer* ».Au même titre que les mutators, il est possible d'appliquer n'importe quel nom à un accessor.
Reprenons notre exemple précédent. Le getter ci-après explique le principe de retour de la valeur d'un attribut privé. Nous renvoyons ici la valeur de la variable *$_volumeCarburant.*

Déclaration d'un accessor pour l'attribut privé $_volumeCarburant de la classe Voiture

```
<?php

class Voiture extends Vehicule
{
  // ...

  public function getVolumeCarburant()
  {
    return $this->_volumeCarburant;
  }
}
```

Voici un exemple d'utilisation de la méthode *getVolumeCarburant :*

Utilisation de l'accessor getVolumeCarburant() sur l'objet $monVehicule

```
<?php

echo sprintf("Il me reste %u L d'essence", $monVehicule-
>getVolumeCarburant());

?>
```

C'est tout ce qu'il faut savoir d'essentiel sur la visibilité des méthodes et des attributs ainsi que sur les accès aux valeurs de ces derniers. Avant de conclure ce tutoriel, faisons un court rappel de bonnes pratiques de développement orienté objet histoire que vous puissiez démarrer immédiatement avec des bases solides.

Quelques bonnes pratiques...

Voici une petite série récapitulative de conseils et de réflexes à mettre en oeuvre en développement orienté objet afin d'uniformiser et de standardiser le code. Ces astuces syntaxiques ne sont pas des techniques personnelles puisque ce sont des conventions éprouvées par des spécialistes des langages orientés objet. Malgré tout, ne le prenez pas comme des paroles d'évangile. Vous êtes libres d'adopter les conventions et règles syntaxiques de votre choix.

- Préfixer les noms de méthodes et d'attributs privés avec un underscore afin de les distinguer plus rapidement à la relectue du code.
- Placer les attributs et méthodes en accès privé ou bien en accès protégé si l'on souhaite dériver la classe dans le futur.
- Utiliser autant que possible les conventions de nommage pour les accessors et les mutators (*getNomAttribut()* et *setNomAttribut()).*

Conclusion

Nous voilà arrivés à l'issue de ce tutoriel sur la programmation orientée objet. Nous avons étudié chacune des visibilités relatives aux attributs et aux méthodes d'un objet. Puis nous nous sommes intéressés aux concepts de *mutators (setters)* et *accessors (getters)* qui permettent respectivement d'affecter une nouvelle valeur à un attribut ou bien d'en récupérer cette dernière via l'utilisation de méthodes. Enfin, nous avons donné une petite série de bonnes pratiques à adopter pour produire du code standard et maintenable.
Les tutoriels suivants de ce chapitre s'intéresseront particulièrement aux concepts d'héritage, de classes abstraites, d'interfaces, de propriétés ou méthodes statiques, ou bien encore aux fameuses méthodes magiques dont on entend souvent parler.

Méthodes magiques : __set() et __get()

- *Par Palleas*

PHP a fait un grand pas en avant en matière de programmation orientée objet avec sa version 5. Depuis cette version, il permet d'implémenter des méthodes au comportement prédéfini par PHP. Ces méthodes sont appelées « méthodes magiques », les méthodes __set() et __get() en font partie.

Rappels concernant la visibilité des propriétés et des méthodes

Accès public des propriétés

La programmation orientée objet permet de spécifier la visibilité des méthodes et propriétés d'une classe. Dans le cas des propriétés, il est très peu recommandable de leur donner une visiblité publique car de cette manière il devient possible de les modifier sans qu'aucun contrôle ne soit effectué sur les valeurs.

```php
<?php

class Kid {

  /**
   * Age du kid
   *
   * @var int
   * @access public
   */
  public $age;

  /**
   * Retourne l'âge du Kid sous forme d'une chaîne
   *
   * @param void
   * @return string
   */
  public function showAge() {
    return 'Son âge est : '. $this->age;
  }
}

$billy = new Kid();
$billy->age = 'rouge';
echo $billy->showAge();
?>
```

Cette portion de code vous retournera donc « Son âge est rouge » ce qui, vous en conviendrez, ne veut strictement rien dire.

Accès privé et protégé des propriétés

Il est donc préférable de leur accorder une visiblité limitée : private ou protected.

```php
<?php

class Kid {

  /**
   * Age du kid
   *
```

```
    * @var int
    * @access private
    */
   private $age;
}

$billy = new Kid();
$billy->age = "encore plus rouge qu'avant";
?>
```

Ce bout de code vous retournera une erreur fatale :

```
Fatal error: Cannot access private property Kid::$age in
/path/to/Apprendre-php/magic_methods.php on line 6.
```

Une solution serait de créer un accesseur public setAge() qui permettrait de spécifier l'age en s'assurant que vous lui avez bien passé un chiffre, et getAge() pour pouvoir l'afficher.

Cette méthode est tout à fait envisageable, à condition de ne pas avoir un grand nombre de propriétés, le nombre d'accesseurs devenant bien trop important : vous pouvez utiliser implicitement les méthodes magiques __set() et __get().

De plus, la syntaxe suivante est tout à fait valable avec PHP :

```
<?php

class Kid {

}

$billy = new Kid();
$billy->age = 14;
$billy->cheveux = 'noirs';
// etc...

echo 'Billy est agé de ', $billy->age,' ans et ses cheveux
sont de couleur ', $billy->cheveux;
?>
```

Et oui, vous pouvez renseigner et ensuite récuperer des propriétés à un objet php, sans que celles-ci aient été déclarées dans votre classe, plus sale non ? Il est donc possible de « boucher » cette petite bévue, toujours en utilisant implicitement les méthodes magiques __set() et __get() qui seront respectivement appelées lorsque l'on renseigne la propriété et lorsqu'on essaye d'en lire la valeur.

La méthode magique __set()

La méthode magique __set() permet de faire ce que l'on appelle de la surchage de propriétés d'une classe. En effet, lorsque l'on essaie de fixer la valeur d'une propriété inexistante de la classe, PHP appelle automatiquement cette méthode de manière implicite. Voici sa structure générale.

```
<?php
```

```php
class MyObject
{
  /**
   * Methode magique __set()
   *
   * @param string $property Nom de la propriété
   * @param mixed $value Valeur à affecter à la propriété
   * @return void
   */
  public function __set($property, $value)
  {
    // Code personnalisé à exécuter
  }
}

?>
```

En redéfinissant explicitement cette méthode dans le corps de la classe, cela permet au développeur de réaliser des contrôles d'accès et de s'assurer que seules quelques propriétés peuvent être mises à jour. C'est ce que nous introduirons dans notre cas d'application plus loin dans ce cours.

La méthode magique __get()

La méthode magique __get() permet, quant à elle, de lire la valeur d'une propriété inexistante de la classe. Au même titre que la méthode magique __set(), la méthode magique __get() doit être redéfinie dans la classe pour exécuter du code personnalisé lorsque PHP appelle implicitement cette méthode. Là encore, cela permet de réaliser un contrôle d'accès sur les propriétés dont on essaie de lire les valeurs. Le prototype de cette méthode est présenté ci-dessous et sera développé davantage dans le cas d'application pratique de ce tutoriel.

```php
<?php

class MyObject
{
  /**
   * Methode magique __get()
   *
   * @param string $property Nom de la propriété à atteindre
   * @return mixed|null
   */
  public function __get($property)
  {
    // Code personnalisé à exécuter
  }
}

?>
```

Cas d'application pratique

Nous allons reprendre notre exemple précédent de notre classe Kid et nous allons lui intégrer ces deux méthodes magiques _get() et __set(). L'objectif est ainsi de réaliser un contrôle d'accès lorsque l'on essaie de fixer ou de lire la valeur de la propriété privée $age.

```php
<?php

class Kid {

  /**
   * Age du kid
   *
   * @var int
   * @access private
   */
  private $age;

  /**
   * Methode magique __get()
   *
   * Retourne la valeur de la propriété appelée
   *
   * @param string $property
   * @return int $age
   * @throws Exception
   */
  public function __get($property) {

    if('age' === $property) {
      return $this->age;
    } else {
      throw new Exception('Propriété invalide !');
    }
  }

  /**
   * Methode magique __set()
   *
   * Fixe la valeur de la propriété appelée
   *
   * @param string $property
   * @param mixed $value
   * @return void
   * @throws Exception
   */
  public function __set($property,$value) {

    if('age' === $property && ctype_digit($value)) {
      $this->age = (int) $value;
```

```
    } else {
      throw new Exception('Propriété ou valeur invalide !');
    }
  }
}
?>
```

De cette manière, on s'assure de ne pouvoir spécifier que l'âge de Billy, et rien de plus. Vous remarquerez que l'on effectue des tests « en dur » c'est à dire que l'on ne vérifie pas simplement que la propriété $age existe, mais seulement que l'utilisateur a cherché à spécifier « $age ». De plus, on contrôle le type de la valeur au moyen de la fonction *ctype_digit()* qui s'assure que le format de la valeur correspond bien à un nombre entier.
Il est tout à fait possible de récupérer les propriétés d'un objet, mais pour faire cela proprement, il est préférable d'utiliser les classes d'introspection (« reflection » en anglais), un tutoriel est actuellement en cours d'écriture concernant cet sujet de POO.

Inconvénients des méthodes magiques __get() et __set()

Bien que ces deux méthodes magiques soient très pratiques à utiliser, elles posent tout de même deux désagréments non négligeables lorsque l'on développe en environnemet professionnel. En effet, l'utilisation de __get() et __set() empêche tout d'abord la génération automatique de documentation de code au moyen des APIs (PHPDocumentor par exemple) utilisant les objets d'introspection (Reflection). D'autre part, cela empêche également les IDE tels qu'Eclipse d'introspecter le code de la classe et ainsi proposer l'auto-complétion du code.

Méthodes magiques : __call()

PHP a fait un grand pas en avant en matière de programmation orientée objet avec sa version 5. Depuis cette version, il permet d'implémenter des méthodes au comportement prédéfini par PHP. Ces méthodes sont nommées « méthodes magiques », __call() est l'une d'entre elles.

Appeler une méthode qui n'existe pas

Prenons l'exemple d'une classe qui modélise un Manchot, que l'on instancie pour ensuite appeler sa méthode « voler ».

```
<?php

class Manchot
{
```

```
}

$georges = new Manchot();
$georges->voler('Afrique');
?>
```

Ce morceau de code vous lèvera une erreur. Vous ne le saviez peut-être pas mais les manchots ne peuvent pas voler :

```
Fatal error: Call to undefined method Manchot::voler() in
/path/to/Apprendre-php/magic_methods.php on line 4.
```

Ce petit rappel morphologique vous permet surtout de voir la chose suivante : on ne peut pas appeler une méthode qui n'existe pas. Cependant PHP, grâce à la méthode magique __call(), va vous permettre de violer une loi élémentaire de la nature, à savoir faire voler un manchot ou plus généralement appeler une méthode qui n'a pas été déclarée dans votre classe.

Implémenter la méthode __call()

La méthode __call() prend deux paramètres. Le premier contient le nom de la méthode que vous avez cherché à appeler, le seconde contient les arguments que vous lui avez passés. Le listing ci-après présente le structure globale de cette méthode.

```
<?php

class MyObject
{
  /**
   * Methode magique __call()
   *
   * @param string $method Nom de la méthode à appeler
   * @param array $arguments Tableau de paramètres
   * @return void
   */
  public function __call($method, $arguments)
  {
    // Code personnalisé à exécuter
  }
}

?>
```

Maintenant reprenons l'exemple du Manchot.

```
<?php
class Manchot
{
  /**
   * Methode magique __call()
   *
   * @param string $method Nom de la méthode à appeler
   * @param array $arguments Tableau de paramètres
```

```
    * @return void
    * @access private
    */
   private function __call($method,$arguments)
   {
       echo 'Vous avez appelé la méthode ', $method, 'avec les
arguments : ', implode(', ',$arguments);
   }
}

$george = new Manchot();
$george->voler('Afrique');
?>
```

Quelques remarques :

Si vous avez rendu votre méthode *__call()* publique, vous aurez aussi la possibilité de l'appeler directement en faisant : *$georges->__call('voler','Afrique');* mais il y aura une petite différence. En appelant directement la méthode *voler()*, la variable *$arguments* sera un array stockant les différents arguments. A contrario, si vous passez par la méthode *__call()*, le second argument sera du type que vous voudrez.
A l'heure actuelle, il est impossible d'en faire de même avec des méthodes statiques, c'est quelque chose qui est désormais corrigée dans la version 5.3 de PHP qui vient tout juste de sortir en version alpha 1. Une méthode magique nommée « *__callStatic()* » permet, en PHP 5.3, d'appeler des méthodes statiques qui ne sont pas déclarées dans la classe.

Exemple concret : création d'un moteur de recherche

Vous vous dites que cela n'a pas grand intérêt, et pourtant avec l'exemple suivant vous devriez y voir un peu plus clair.

Nous allons tenter de recréer un moteur de recherche. Vous remarquerez que nous utilisons la classe SPDO présentée dans un précédent tutoriel et qui permet d'accéder à la base de données via l'extension native PDO.

```
<?php

class SearchEngine
{
  /**
   * Effectue une recherche dans la base de données à
   * partir des critères fournis en argument
   *
   * @param array $conditions Tableau de critères de recherche
   * @return array $return Tableau des résultats
   * @see SPDO
   */
  public function search($conditions = array())
  {
      $query = 'SELECT id FROM table';
```

```
      if(sizeof($conditions) > 0) {
        $query.=' WHERE '.implode(' AND ',$conditions);
      }

      // Exécution de la requête SQL avec une classe PDO
      $result = SPDO::getInstance()->query($query);
      $return = $result->fetchAll(PDO::FETCH_ASSOC);
      return $return;
    }
}
?>
```

Comme vous pouvez le constater, ce moteur de recherche possède une méthode search() qui prend en paramètre un tableau des différentes conditions à appliquer à la requête effectuant la recherche. Ces conditions étant de la forme suivante : nomDuChamp= "valeur".

Vous admettrez comme moi (j'espère !) que cette syntaxe n'est pas des plus pratiques, je ne me vois pas utiliser la requête de cette manière :

```
<?php

$mySearchEngine = new SearchEngine();
$mySearchEngine->search(array(
  'champ1' => 'apprendre-php',
  'champ2' => 'palleas'
));
```

Ce serait vraiment sympa de pouvoir faire *$mySearchEngine->searchByName('palleas');* par exemple, ou encore *$mySearchEngine->searchByNameAndDate('palleas','25/07/1987');* pas vrai ?
Et c'est là que l'on va pouvoir mettre en application la méthode *__call().*

```
<?php

class SearchEngine
{
  /**
   * Effectue une recherche dans la base de données à
   * partir des critères fournis en argument
   *
   * @param array $conditions Tableau de critères de recherche
   * @return array $return Tableau des résultats
   * @see SPDO
   */
  public function search($conditions = array())
  {
      $query = 'SELECT id FROM table';

      if(sizeof($conditions) > 0) {
        $query.=' WHERE '.implode(' AND ',$conditions);
      }

      // Exécution de la requête SQL avec une classe PDO
```

```
      $result = SPDO::getInstance()->query($query);
      $return = $result->fetchAll(PDO::FETCH_ASSOC);
      return $return;
    }

  /**
   * Méthode magique __call() permettant d'appeller une
méthode virtuelle
   * du type searchByName(), searchByAge() ou
searchByNameAndAge()...
   *
   * @param string $method Nom de la méthode virtuelle appelée
   * @param array $args Tableau des critères de recherche
   * @return array|null $return Tableau des résultats ou NULL
   * @see SearchEngine::search()
   */
  public function __call($method, $args)
  {
    if(preg_match('#^searchBy#i',$method))
    {
      $searchConditions = str_replace('searchBy','',$method);
      $searchCriterias = explode('and',$searchConditions);
      $conditions = array();
      $nbCriterias = sizeof($searchCriterias);

      for($i=0; $i < $nbCriterias; $i++)
      {
        $conditions[] =
strtolower($searchCriterias[$i]).'="'.$args[$i] .'"';
      }
      return $this->search($conditions);
    }
    return null;
  }
}
?>
```

Voilà un morceau de code assez conséquent à digérer, nous allons donc le décortiquer étape par étape :

- Pour commencer, on vérifie que la méthode que l'on a cherché à appeler est une méthode dont le nom commence par « searchBy ». Cette étape n'est pas indispensable, nous appellerons ça une précaution : nous nous assurons ainsi de l'intuitivité du code.
- On récupère ce qu'il y a après searchBy, dans cet exemple : name
- Au cas ou nous aurions plusieurs Conditions, par exemple searchByNameAndDate, on récupère chacun des champs à tester, ici NameAndDate.
- Pour chacun des paramètres, on crée la condition, dans le cas de : $google->searchByNameAndDate('palleas','25/07/1987'); on obtient un tableau avec

name=«palleas » et date=«25/07/1987» que l'on va pouvoir passer en paramètre à la méthode search(), comme on en parlait plus haut.

Inconvénients de l'utilisation de la méthode magique __call()

Au même titre que les méthodes magiques __get() et __set(), la méthode magique __call() possède deux inconvénients non négligeables lorsque l'on développe en environnemet professionnel. En effet, l'utilisation de __call() empêche tout d'abord la génération automatique de documentation de code au moyen des APIs (PHPDocumentor par exemple) utilisant les objets d'introspection (Reflection). D'autre part, cela empêche également les IDE tels qu'Eclipse d'introspecter le code de la classe et ainsi proposer l'auto-complétion du code. A utiliser donc avec parcimonie !

Méthodes magiques : __clone

PHP depuis sa version 5 implémente des méthodes magiques, que vous pouvez implémenter dans vos classes, et qui seront automatiquement appelées par votre script. La méthode magique __clone() est l'une de ces méthodes. Le tutoriel qui suit introduit le fonctionnement de la méthode magique __clone() en se basant sur des exemples simples et concrets.

Rappels sur la programmation orientée objet

Pour bien comprendre le principe de cette méthode et son champ d'application, voici comment marche (dans les grandes lignes), la programmation orientée objet.

Prenons un objet « Point » qui dispose des propriétés protégées « _x » et « _y », qui ici correspondront aux coordonnées du point. On y ajoute une méthode setCoords() qui permet de spécifier ces valeurs :

```php
<?php

class Point {
 /**
  * Abscisse du point
  *
  * @var integer
  */
  protected $_x = 0;

 /**
  * Ordonnée du point
  *
  * @var integer
  */
  protected $_y = 0;


 /**
  * Fixe les coordonnées du point
  *
  * @param integer $x Abscisse du point
  * @param integer $y Ordonnée du point
  * @return void
  */
  public function setCoords($x,$y) {
    $this->_x = (int)$x;
    $this->_y = (int)$y;
  }
}
```

Lorsque l'on souhaite utiliser cet objet, on appelle son constructeur (ici, il est déclaré implicitement) et nous le stockons dans une variable le résultat de cet appel. C'est ce que l'on appelle la phase d'instanciation de la classe. La variable est une instance de notre objet.

```php
<?php

// instanciation de l'objet Point

// et stockage de l'instance dans la variable "oDot"

$oDot = new Point();
?>
```

PHP 5 et le passage des objets par référence

Voilà pour les grandes lignes. Maintenant, parlons (désolé) PHP 4. Avec la version 4 de PHP, les objets étaient passés par « valeur » (ou bien par « copie »), ce qui signifie que lorsqu'une variable « $a » contenant l'instance d'un objet Point était copiée dans une variable $b, alors les deux variables $a et $b contenaient chacune une instance (objet) unique. Depuis PHP 5, les objets sont passés et copiés par référence. Cela implique donc qu'en cas de copie d'une variable $a vers $b, cette

dernière contiendra la référence référence vers l'objet de la variable $a. Prenons un exemple simple pour illustrer concrètement la théorie.

```
<?php

$oDot = new Point;
$oDot->setCoords(10,10);
var_dump($oDot);
$oNewDot = $oDot;
$oNewDot->setCoords(20,20);
var_dump($oDot);
?>
```

Ici, nous instancions notre objet Point et nous stockons l'instance dans la variable $oDot. Puis nous copions le contenu de la variable $oDot dans la variable $oNewDot. Nous modifions les propriétés de l'instance de l'objet Point stocké dans $oNewDot et enfin nous affichons les informations de l'instance de notre objet.

Le premier var_dump() affiche alors :

```
object(Point)#1 (2) { ["_x:protected"]=> int(10)
["_y:protected"]=> int(10) }
```

Le second retourne :

```
object(Point)#1 (2) { ["_x:protected"]=> int(20)
["_y:protected"]=> int(20) }
```

Nous constatons ici que nous avons affaire à une seule et unique instance bien que l'on ait essayé de copier la variable.

Seulement voilà, dans certains cas il peut être utile de dupliquer une instance de classe, ce qui est impossible en passant par la méthode « PHP 4 », comme nous venons de le voir. La solution est donc d'avoir recours au mot clé « clone ».

Le mot-clé clone

L'utilisation est simple, au lieu de faire :

```
<?php

$oNewDot = $oDot;
?>
```

Nous allons faire :

```
<?php

$oNewDot = clone $oDot;
?>
```

En modifiant et en exécutant le code précédent :

```
<?php

$oDot = new Point;
```

```
$oDot->setCoords(10,10);
var_dump($oDot);
$oNewDot = clone $oDot;
$oNewDot->setCoords(20,20);
var_dump($oNewDot);
var_dump($oDot);
?>
```

Voici l'affichage du premier var_dump() :

```
object(Point)#1 (2) { ["_x:protected"]=> int(10)
["_y:protected"]=> int(10) }
```

Et voici l'affichage du deuxième, nous pouvons constater que les propriétés _x et _y sont bien modifiées :

```
object(Point)#1 (2) { ["_x:protected"]=> int(20)
["_y:protected"]=> int(20) }
```

Voici l'affichage du troisième et dernier var_dump :

```
object(Point)#1 (2) { ["_x:protected"]=> int(10)
["_y:protected"]=> int(10) }
```

Vous le constatez vous-même, nous avons bien deux instances distinctes de la classe Point.

Implémentation de la méthode magique __clone

Maintenant que nous avons bien saisi le concept de clonage d'objets en PHP, nous allons enfin pouvoir parler de la méthode magique « __clone() ». Le principe de cette méthode est simple, elle sera automatiquement appelée lorsque l'on utilisera le mot clé *clone* sur un objet. Prenons un exemple.
Voici une classe Sheep (mouton, pour les anglophobes) :

```
<?php

class Sheep {
 /**
  * Nom du mouton
  *
  * @var String
  */
  protected $_name;

 /**
  * Contructeur de la classe Sheep
  *
  * @param String $name nom du mouton
  */
  public function __construct($name) {
    $this->_name = (string)$name;
  }
```

```
  /**
   * Methode magique clone
   *
   * @return void
   */
  public function __clone() {
    $this->_name = 'Copie de '.$this->_name;
  }
}
```

Instancions notre classe, puis dupliquons notre objet :

```
<?php

$oSheep = new Sheep('Dolly');
$oNewSheep = clone $oSheep;
var_dump($oSheep);
var_dump($oNewSheep);
?>
```

Voici ce qu'affiche le premier var_dump() :

```
object(Sheep)#3 (1) { ["_name:protected"]=> string(5) "Dolly"
}
```

Et voici ce qu'affiche le deuxième var_dump() :

```
object(Sheep)#4 (1) { ["_name:protected"]=> string(14) "Copie
de Dolly" }
```

En dupliquant l'instance « $oSheep , ma méthode __clone a été automatiquement appelée et a modifié le nom en ajoutant « Copie de » en préfix du nom du mouton. Vous remarquerez que nous avons bien utilisé la référence sur l'objet lui même $this pour modifier les informations de la nouvelle instance.

Implémentation dans le cas d'un Singleton

Le Singleton est un design pattern permettant de s'assurer de n'avoir qu'une seule instance d'un objet dans un script. Pour plus d'informations, rendez-vous sur ce billet.
Seulement, même si votre constructeur est déclaré en accès privé ou protégé, il sera toujours possible de cloner votre objet. Vous perdrez alors le principe de singleton. Pour pallier à ce problème, il suffit de lever une exception lorsque le développeur utilisant le Singleton décide de le cloner :

```
<?php

class Singleton {
  /**
   * Instance de la classe Singleton
   *
   * @var Singleton
   */
  protected static $_instance = null;
```

```
  /**
   * Constructeur de la classe
   *
   * @access protected
   */
  protected function __construct() {}

  /**
   * getInstance() : recuperation de l'instance de la classe
   *
   * @return Singleton
   */
  public static function getInstance() {
    if(null === self::$_instance) self::$_instance = new
Singleton();
    return self::$_instance;
  }

  /**
   * Methode magique clone
   *
   * @return void
   */
  public function __clone() {
    throw new Exception('Are you Trying to clone me ? I\'m a
Singleton dude !');
  }
}

try {
  $oSingleton = Singleton::getInstance();
  clone $oSingleton;
}catch(Exception $e) {
  echo 'Oops, exception : ', $e->getMessage();
}
?>
```

Et voici ce qui s'affichera à l'exécution du code :

```
Oops, exception : Are you Trying to clone me ? I'm a Singleton
dude !
```

L'exception a bien été levée et a donc empêché la tentative de clônage de l'objet. Le singleton conservera ainsi son unicité dans l'application développée.

Conclusion

Ce tutoriel vous a présenté le fonctionnement global de la méthode magique __clone() et du clônage d'objet en PHP 5. Vous êtes à présent à même d'utiliser cette méthode magique dans vos applications pour faciliter les clônages de vos objets ou bien pour en modifier les comportements d'origine.

Méthodes magiques : __sleep() et __wakeup()

Nous avons étudié dans les précédents articles les méthodes magiques__clone(), __set(), __get() et __call(). PHP ajoute à ces dernières deux méthodes magiques supplémentaires __sleep() et __wakeup() qui permettent de surcharger le processus natif de sérialisation et de désérialisation des données de PHP. C'est ce que nous allons expliquer au cours de tutoriel avec quelques exemples concrets et faciles à comprendre.

Qu'est ce que la sérialisation de données ?

Pour résumer très simplement, l'action de sérialiser une variable consiste à convertir celle-ci en une chaine de caractère pour, par exemple, la stocker dans un fichier texte. A contrario, l'action de désérialiser une chaine de caractères consiste à appliquer le procédé inverse afin d'en récupérer la variable d'origine. Ce procédé ne se limite pas à PHP, vous le retrouverez dans beaucoup de langages tels que Java, Action Script, C, C#... pour ne citer qu'eux.

Les mots sérialisation et désérialisation sont sans doute les mots les plus courants mais vous entendrez / lirez peut-être quelque part les mots linéarisation / délinéarisation ou bien encore marshalling / unmarshalling. Sérialiser, linéariser, marshaller (respectivement désérialiser, délinéariser et unmarshaller) définissent les notions de sérialisation et désérialisation.

Syntaxiquement parlant, la serialisation de données se traduit par l'utilisation de la fonction serialize(), l'action inverse se traduit par unserialize(). Voyons quelques exemples.

Serialisation / Désérialisation d'une variable de type integer

```php
<?php

$iVar = 1;
$sSerialized = serialize($iVar);
$iUnserialized = unserialize($sSerialized);

echo '<pre>';
var_dump($sSerialized,$iUnserialized);
echo '</pre>';
?>
```

Le premier var_dump() affiche la chaine suivante sur la sortie standard :

```
string(4) "i:1;"
```

Et le second affiche la suivante :

```
int(1)
```

Vous déduirez bien sûr que ce n'est pas intéressant de serializer un integer (ou bien une chaîne de caractères évidemment) étant donné que vous pouvez le stocker directement. Néanmoins, cet exemple reste utile à l'illustration d'un principe simple : **la serialisation d'une variable préserve le type de celle-ci**. Ainsi on aura bien au retour un integer, au même titre que l'on conservera un Array ou encore un objet.

Sérialisation / désérialisation d'un tableau (Array)

Prenons à présent un tableau comme exemple. La sérialisation et la désérialisation deviennent ici beaucoup plus intéressantes dans la mesure où elles vont nous permettre de transformer (et de récupérer) l'intégralité d'un tableau.

```
<?php

$aTableau = array('Riri','Fifi','Loulou','Donald','Picsou');
$sSerialized = serialize($aTableau);
$aUnserialized = unserialize($sSerialized);

echo '<pre>';
var_dump($sSerialized,$aUnserialized);
echo '</pre>';
?>
```

Le premier var_dump affiche alors :

```
string(87)
"a:5:{i:0;s:4:"Riri";i:1;s:4:"Fifi";i:2;s:6:"Loulou";i:3;s:6:"
Donald";i:4;s:6:"Picsou";}"
```

La sérialisation du tableau finalement construit une chaîne de caractères d'une longueur de 87 caractères. Etudions rapidement la structure de cette chaine.

- Le premier caractères de la chaine indique le type de la variable sérialisée. Ici, la lettre "a" indique donc que nous venons de sérialiser un tableau (array).
- Puis, nous lisons le chiffre 5 qui indique le nombre d'éléments dans le tableau. Entre les deux parenthèses, nous trouvons tous les éléments du tableau.
- La première le lettre d'un groupe élément indique le type de l'index dans le tableau suivi de sa valeur. Ici nous avons sérialisé un tableau indexé numériquement, c'est pourquoi chaque index est un entier ("i") dont la valeur est comprise entre 0 et 4 compris. Si nous avions utilisé un tableau associatif, les indexes seraient devenus des chaines de caractères ("s").
- Nous lisons ensuite pour chaque valeur stockée, son type ("s"), sa longueur (4) et sa valeur stockée dans le tableau ("Riri").

Grâce à toutes ces informations, la fonction unserialize() est capable de reconstruire entièrement le tableau en analysant la chaine formattée.

C'est pourquoi le second var_dump() affiche le résultat suivant dans lequel nous retrouvons toutes les informations de la chaîne sérialisée :

```
array(5) {

[0]=>

string(4) "Riri"

[1]=>

string(4) "Fifi"
```

```
[2]=>

string(6) "Loulou"

[3]=>

string(6) "Donald"

[4]=>

string(6) "Picsou"

}
```

Tentons maintenant de sérialiser / désérialiser un objet.

Sérialisation et désérialisation d'un objet

Voici maintenant l'exemple le plus intéressant, la sérialisation d'un objet. Prenons une classe basique avec différentes propriétés. Nous utilisons volontairement des propriétés publiques, privées et protégées afin de visualiser comment se comporte la sérialisation d'un objet avec des propriétés différentes.

```php
<?php

class Dormeur {


  /**
   * Age du nain Dormeur
   *
   * @var integer
   * @access protected
   */
  protected $_age;

  /**
   * Le nom porte-t-il un bonnet ?
   *
   * @var boolean
   * @access protected
   */
  protected $_aSonBonnet;


  /**
   * Sa couleur préférée
   *
   * @var string
   * @access private
   */
  private $_couleurPreferee;
```

```
  /**
   * Ses hobbies
   *
   * @var array
   * @access public
   */
  public $_gouts;

  /**
   * Constructeur de la classe
   */
  public function __construct() {
    $this->_age = 19;
    $this->_aSonBonnet = true;
    $this->_couleurPreferee = 'rouge';
    $this->_gouts = array('musique','cinéma','curling');
  }
}
?>
```

Le nom de la classe vous semblera sans doute débile, mais nous trouvions plutôt de circonstance d'avoir une classe nommée « Dormeur » pour introduire une méthode nommée « __sleep() ».

Passons maintenant successivement à son instanciation, sa serialisation et sa deserialisation :

```
<?php

$oDormeur = new Dormeur();
$sSerialized = serialize($oDormeur);
$oUnserialized = unserialize($sSerialized);

echo '<pre>';
var_dump($sSerialized,$oUnserialized);
echo '</pre>';
?>
```

Nous obtenons alors respectivement les résultats suivants :

```
string(181) "O:7:"Dormeur":4:{s:7:" *_age";i:19;s:14:"
*_aSonBonnet";b:1;s:25:"
Dormeur_couleurPreferee";s:5:"rouge";s:6:"_gouts";a:3:{i:0;s:7
:"musique";i:1;s:7:"cinéma";i:2;s:7:"curling";}}

object(Dormeur)#2 (4) {

["_age:protected"]=>

int(19)

["_aSonBonnet:protected"]=>
```

```
bool(true)

["_couleurPreferee:private"]=>

string(5) “rouge”

["_gouts"]=>

array(3) {

[0]=>

string(7) “musique”

[1]=>

string(7) “cinéma”

[2]=>

string(7) “curling”

}

}
```

Vous saviez déjà que le type des variables était préservé, il en va de même pour la visiblité (private, protected et public), cela ne change en aucun cas la structure de la classe. Nous vous parlions de stockage d’instance, il faut bien faire attention à ce que votre **classe soit déclarée avant la désérialisation**, sans quoi vous aurez une erreur :

```
Fatal error: Class ‘Dormeur’ not found in
/home/path/to/your/script.php on line xx
```

Vous remarquerez le « o » comme premier caractère de la chaîne sérialisée qui indique que nous venons bien de sérialiser un objet.

Méthodes magiques __sleep() et __wakeup()

Maintenant que nous avons bien saisi le principe de la serialisation, nous allons pouvoir entrer dans le vif du sujet et parler des méthodes magiques __sleep() et __wakeup().

Ces méthodes seront respectivement appelée par votre script lors de l’utilisation de serialize() et de unserialize(). Voici une démonstration, nous allons ajouter les méthodes __sleep() et _wakeup() à notre Dormeur.

La méthode magique __sleep() doit cependant effectuer une action pour que la serialisation se passe bien. C’est ici que l’on peut voir le premier intérêt de la serialization d’une instance de classe : vous allez selectionner quelles propriétés de votre instance vous souhaitez stocker. Pour ce faire, la méthode __sleep() doit retourner un tableau contenant les noms des propriétés à conserver :

```
<?php
```

```php
class Dormeur {

  /**
   * Méthode magique __sleep() Appelée lors d'un serialize()
   *
   * @return Array la liste des paramètres à conserver
   */
  public function __sleep() {
    echo 'Bon ben moi, je vais dormir.';
    return array('_age','_aSonBonnet','_couleurPreferee');
  }

  /**
   * Méthode magique __wakeup() Appelée lors d'un
unserialize()
   *
   * @return void
   */
  public function __wakeup() {
    echo 'bon ben moi, je vais me faire un café.';
  }
}
?>
```

Ici on ne choisit de garder que l'âge de Dormeur, sa couleur préférée et si oui, ou non, il a son bonnet sur la tête.

En exécutant de nouveau le code précédent, vous allez voir les deux echo s'afficher à la suite («Bon ben moi, je vais dormir. » puis « bon ben moi, je vais me faire un café. ») confirmant que les méthodes __sleep() et __wakeup() ont bien été successivement appelées.

Le résultat de la sérialisation est le suivant :

```
string(108)
"O:7:"Dormeur":3:{s:7:"*_age";i:19;s:14:"*_aSonBonnet";b:1;s:2
5:"Dormeur_couleurPreferee";s:5:"rouge";}"
```

On constate que l'on ne conserve pas les goûts de Dormeur. Et voici ce que vous affichera le deuxième var_dump :

```
object(Dormeur)#2 (4) {

["_age:protected"]=>

int(19)

["_aSonBonnet:protected"]=>

bool(true)

["_couleurPreferee:private"]=>
```

```
string(5) “rouge”

["_gouts"]=>

NULL

}
```

Vous remarquerez que seules les propriétés spécifiées dans le __sleep() ont été mémorisées, la propriété « _gouts », quant à elle, est égale à NULL, son contenu n'a pas été conservé. Bien entendu, l'intérêt ne s'arrête pas là, vous allez pouvoir effectuer les actions que vous désirez dans ces méthodes : synchronisation avec une base de données, fermer une connexion à une base de données lors de la sérialisation, la réouvrir lors de la deserialisation, etc.

Pour les amoureux du bricolage, il est également possible d'implémenter la notion de « transtypage » d'objet, totalement absente de PHP à l'heure actuelle.

Conclusion

Nous venons de découvrir les notions de sérialisation et de désérialisation d'une variable, d'un tableau ou bien d'un objet. Puis, nous avons montré comment il était possible de surcharger le comportement original de sérialisation / désérialisation d'un objet grâce aux méthodes magiques __sleep() et __wakeup().

Les classes abstraites et finales

PHP intègre un autre concept de la programmation orientée objet : les classes abstraites. Ce cours définit et introduit la notion de classes abstraites. Nous présenterons ce que sont les classes abstraites, à quoi elles servent au développement et comment les déclarer et les utiliser. Nous étudierons enfin le cas particulier des classes et méthodes finales qui participent à la sécurité du code en programmation orientée objet.

Présentation et déclaration des classes abstraites

Définition des classes abstraites

Les classes abstraites s'inscrivent davantage dans la sûreté de la programmation orientée objet. La première particularité d'une classe abstraite, c'est qu'elle ne peut être instanciée (et donc créer un objet). De cette affirmation, on en déduit logiquement qu'une classe abstraite est déclarée afin d'être dérivée par des classes concrètes.

Une classe abstraite se comporte comme une classe concrète typique. C'est-à-dire qu'elle peut déclarer des attributs et des méthodes traditionnels qui seront accessibles dans les classes dérivées. En fonction bien sûr de la visibilité choisie (*public*, *private* et *protected*) pour chacun des attributs et méthodes.
Jusque là, il n'y a aucun changement par rapport aux classes concrètes si ce n'est le fait que l'on ne puisse pas instancier les classes abstraites. C'est là qu'interviennent les méthodes abstraites. Nous verrons par la suite qu'une classe déclarée abstraire peut aussi définir des méthodes abstraites. Ces dernières devront obligatoirement être redéfinies dans les classes dérivées. C'est un moyen de s'assurer que la classe dérivée adoptera le comportement désiré.

Déclaration d'une classe abstraite

La déclaration d'une classe abstraite se réalise au moyen du mot-clé « ***abstract*** ». Prenons l'exemple d'une classe abstraite « ***EtreHumain*** » qui sera, par exemple, dérivée par deux classes concrètes «***Homme*** » et « ***Femme*** ».

Déclaration d'une classe abstraite

```php
<?php

abstract class EtreHumain
{
  /**
   * Sexe de la personne
   *
   * @var string
   */
  protected $sexe;

  /**
   * Nom de la personne
   *
```

```
    * @var string
    */
   protected $nom;

   /**
    * Met à jour le nom
    *
    * @param string $nom
    * @return void
    */
   public function setNom($nom)
   {
     $this->nom = $nom;
   }

   /**
    * Retourne le nom de la personne
    *
    * @param void
    * @return string $nom
    */
   public function getNom()
   {
     return $this->nom;
   }

   /**
    * Retourne le sexe de la personne
    *
    * @param void
    * @return string $sexe
    */
   public function getSexe()
   {
     return $this->sexe;
   }
}
?>
```

Vous remarquez que nous n'avons volontairement pas implémenté de constructeur dans cette classe puisque les classes abstraites ne peuvent être instanciée. Si nous essayons d'instancier cette classe pour créer un objet de type *EtreHumain*, nous obtiendrons le message d'erreur ci-après :

```
Fatal error: Cannot instantiate abstract class EtreHumain in
/Users/Emacs/Sites/Demo/Tutoriels/abstract.php on line 35
```

Intéressons nous à présent à l'autre subtilité des classes abstraites ***: les méthodes abstraites.***

Déclaration et redéfinition des méthodes abstraites

Une méthode abstraite est aussi déclarée au moyen du mot-clé « ***abstract*** ». C'est une méthode partiellement définie dans la classe. En effet, lorsque l'on déclare une méthode abstraite, on ne définit que son prototype (sa signature). Les classes dérivées devront obligatoirement redéfinir entièrement (prototype + corps) toutes les méthodes abstraites de la classe parente.
Reprenons notre exemple précédent et ajoutons quelques méthodes abstraites à notre classe abstraite «***EtreHumain*** ».

Déclaration de méthodes abstraites

```php
<?php

abstract class EtreHumain
{
  /**
   * Sexe de la personne
   *
   * @var string
   */
  protected $sexe;

  /**
   * Nom de la personne
   *
   * @var string
   */
  protected $nom;

  /**
   * La personne fait du sport
   *
   * @abstract
   */
  abstract function faireDuSport();

  /**
   * Divertit la personne
   *
   * @abstract
   */
  abstract function seDivertir();

  /**
   * Met à jour le nom
   *
   * @param string $nom
   * @return void
   */
```

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

  /**
   * Retourne le nom de la personne
   *
   * @param void
   * @return string $nom
   */
  public function getNom()
  {
    return $this->nom;
  }

  /**
   * Retourne le sexe de la personne
   *
   * @param void
   * @return string $sexe
   */
  public function getSexe()
  {
    return $this->sexe;
  }
}
?>
```

Notre classe dispose à présent de deux nouvelles méthodes abstraites *faireDuSport()* et *seDivertir()*.Comme nous pouvons le constater, le corps de ces deux méthodes n'est pas défini. Nous devrons les définir dans les classes dérivées.

Remarque : *toute classe qui définit une ou plusieurs méthodes abstraites doit obligatoirement être déclarées abstraite elle aussi.*

C'est pourquoi nous allons à présent définir deux classes « ***Homme*** » et « ***Femme*** » qui hériteront toutes les deux des propriétés et méthodes de notre classe abstraite. Commençons simplement par les déclarer et tenter de les instancier.

Déclaration des classes dérivées

Déclaration des classes dérivées Homme et Femme

```
<?php

class Homme extends EtreHumain
{
  /**
   * Construit l'objet Homme
   *
```

```
    * @param string $nom Nom de l'homme
    * @return void
    */
   public function __construct($nom)
   {
     $this->nom = $nom;
     $this->sexe = 'M';
   }
}

class Femme extends EtreHumain
{
   /**
    * Construit l'objet Femme
    *
    * @param string $nom Nom de la femme
    * @return void
    */
   public function __construct($nom)
   {
     $this->nom = $nom;
     $this->sexe = 'F';
   }
}
?>
```

Tentons à présent d'instancier une des deux classes.

```
<?php

$oBob = new Homme('Bobby');
echo $oBob->getNom();

?>
```

Bien entendu, nous obtenons l'erreur suivante car nous n'avons pas redéfini les méthodes abstraites de la superclasse.

```
Fatal error: Class Homme contains 2 abstract methods and must
therefore be declared abstract or implement the remaining
methods (EtreHumain::faireDuSport, EtreHumain::seDivertir) in
/Users/Emacs/Sites/Demo/Tutoriels/abstract.php on line 74
```

Ce message nous indique que nous devons soit redéfinir explicitement les méthodes abstraites de la superclasse ou bien rendre notre classe Homme abstraite. Comme nous souhaitons pouvoir instancier la classe dérivée, il ne nous reste que la première solution. Redéfinissons donc ces deux classes abstraites dans chacune des classes dérivées.

Redéfinition des méthodes abstraites dans les classes dérivées

```
<?php
```

```php
class Homme extends EtreHumain
{
  /**
   * Construit l'objet Homme
   *
   * @param string $nom Nom de l'homme
   * @return void
   */
  public function __construct($nom)
  {
    $this->nom = $nom;
    $this->sexe = 'M';
  }

  /**
   * Affiche le sport de l'homme
   *
   * @param void
   * @return void
   */
  public function faireDuSport()
  {
    echo $this->nom .' fait de la boxe';
  }

  /**
   * Affiche la distraction de l'homme
   *
   * @param void
   * @return void
   */
  public function seDivertir()
  {
    echo 'Soirée foot et bières';
  }
}

class Femme extends EtreHumain
{
  /**
   * Construit l'objet Femme
   *
   * @param string $nom Nom de la femme
   * @return void
   */
  public function __construct($nom)
  {
    $this->nom = $nom;
    $this->sexe = 'F';
  }
```

```php
  /**
   * Affiche le sport de la femme
   *
   * @param void
   * @return void
   */
  public function faireDuSport()
  {
    echo $this->nom .' fait du fitness';
  }

  /**
   * Affiche la distraction de la femme
   *
   * @param void
   * @return void
   */
  public function seDivertir()
  {
    echo 'Shopping entre filles';
  }
}
?>
```

Instancions maintenant chacune des deux classes.

Instances des classes Homme et Femme

```php
<?php

$oAlice = new Femme('Alice');
$oAlice->faireDuSport();
echo '<br/>';
$oAlice->seDivertir();
echo '<br/>';
echo 'Sexe : ', $oAlice->getSexe();

echo '<br/><br/>';

$oBob = new Homme('Bobby');
$oBob->faireDuSport();
echo '<br/>';
$oBob->seDivertir();
echo '<br/>';
echo 'Sexe : ', $oBob->getSexe();

?>
```

Le résultat obtenu est bien celui attendu :

```
Alice fait du fitness

Shopping entre filles
```

```
Sexe : F


Bobby fait de la boxe

Soirée foot et bières

Sexe : M
```

Pour les instances de la classe ***Femme***, nous retrouvons bien la valeur ***F*** pour l'attribut ***$sexe*** ainsi que les méthodes abstraites redéfinies qui affiche correctement le fitness et le shopping. Quant aux instance de la classe ***Homme***, le résultat est celui attendu. L'attribut ***$sexe*** prend bien la valeur ***M*** et les deux méthodes abstraites affichent bien le sport Boxe et le divertissement Soirée football.

Cas particuliers des classes et méthodes finales

Présentation des classes et méthodes finales

Jusqu'à maintenant nous avons présenté les classes et méthodes abstraites. PHP introduit un mécanisme supplémentaire pour assurer la surêté de la programmation. Il s'agit du mot-clé « ***final*** » qui peut-être appliqué soit à une classe ou bien à une méthode.
Lorsque l'on définit une classe comme « finale », cela signifie qu'elle ne pourra plus être dérivée par une sous-classe. Cela implique également que ses attributs et méthodes ne pourront plus être redéfinis. En revanche, si l'on applique le mot-clé « ***final*** » à une méthode d'une classe, alors c'est uniquement cette méthode qui ne pourra plus être redéfinie dans les classes dérivées.
En interdisant la dérivation d'une classe ou la redéfinition (surchage) des méthodes d'une classe, cela vous permet de vous assurer que le développeur ne contournera pas directement la logique que vous avez mise en place.

Déclaration de classes finales

Reprenons nos 3 classes précédentes. Nous décidons maintenant que les classes « ***Homme*** » et «***Femme*** » ne pourront être dérivées. Nous les déclarerons donc comme étant finales. Ce qui nous donne :

Déclaration de classes finales

```php
<?php

final class Homme extends EtreHumain
{
  // Suite du code de la classe
}

final class Femme extends EtreHumain
```

```
{
  // Suite du code de la classe
}
?>
```

Tentons à présent de dériver l'une des deux classes et de l'instancier. Pour cela, il nous suffit d'écrire une classe « ***JeuneGarcon*** » qui hérite des propriétés et méthodes de la classe « ***Homme*** ».

```
<?php

class JeuneGarcon extends Homme
{
  /**
   * Construit l'objet JeuneGarcon
   *
   * @param string $nom Nom du jeune garcon
   * @return void
   */
  public function __construct($nom)
  {
    parent::__construct($nom);
  }
}

?>
```

Bien entendu une erreur est générée puisque la classe Homme n'est plus dérivable du fait de sa déclaration "finale".

```
Fatal error: Class JeuneGarcon may not inherit from final
class (Homme) in
/Users/Emacs/Sites/Demo/Tutoriels/abstract.php on line 153
```

Remarque : *une classe abstraite ne peut-être déclarée "finale". Une erreur de syntaxe sera générée. De plus, déclarer une classe abstraite et finale est synonyme d'un manque de logique puisque le but d'une classe abstraite est d'être dérivée par des classes filles.*

Déclaration de méthodes finales

Etudions maintenant le cas des méthodes finales. Reprenons la classe abstraite « ***EtreHumain*** » et déclarons les méthodes « ***getNom()*** » et « ***getSexe()*** » comme finales.

Déclaration de méthodes finales

```
<?php

abstract class EtreHumain
{
  // Attributs et autres méthodes de la classe
  // ...
```

```
  /**
   * Retourne le nom de la personne
   *
   * @param void
   * @return string $nom
   * @final
   */
  public final function getNom()
  {
    return $this->nom;
  }

  /**
   * Retourne le sexe de la personne
   *
   * @param void
   * @return string $sexe
   * @final
   */
  public final function getSexe()
  {
    return $this->sexe;
  }
}

?>
```

A présent les classes dérivées « ***Homme*** » et « ***Femme*** » ne peuvent plus redéfinir ces deux méthodes. Essayons de surcharger l'une de ces méthodes dans la classe « ***Homme*** ».

Redéfinition de méthodes abstraites

```
<?php

final class Homme extends EtreHumain
{
  // Autres méthodes de la classe
  // ...

  /**
   * Retourne le sexe
   *
   * @param void
   * @return string sexe de l'homme
   */
  public function getSexe()
  {
    return 'Masculin';
  }
}
```

```
?>
```

Ici, la méthode « ***getSexe()*** » ne retourne plus la valeur de l'attribut protégé ***$sexe*** mais tente de renvoyer la chaine de caractères « ***masculin*** ». On s'en doute, la redéfinition de la méthode génère une erreur fatale par l'interprêteur PHP.

```
Fatal error: Cannot override final method
EtreHumain::getSexe() in
/Users/Emacs/Sites/Demo/Tutoriels/abstract.php on line 115
```

Conclusion

Tout au long de ce cours, nous avons découvert le mécanisme des classes et méthodes abstraites qui permettent de bénéficier des avantages de l'héritage et de s'assurer que les classes dérivées implémentent bien certaines actions.

Enfin nous avons étudié le cas particulier des classes et méthodes finales qui empêchent toute surcharge dans les classes dérivées et assurent une sécurité plus importante du code.

Le mécanisme des exceptions a été introduit à PHP dans sa version 5 en complément de son nouveau modèle orienté objet. Au même titre qu'en Java, C++, Action Script 3 ou bien Visual Basic (pour ne citer que ces langages de programmation), les exceptions permettent de simplifier, personnaliser et d'organiser la gestion des « erreurs » dans un programme informatique. Ici le mot « erreurs » ne signifie pas « bug », qui est un comportement anormal de l'application développée, mais plutôt « cas exceptionnel » à traiter différemment dans le déroulement du programme. Etudions donc comment fonctionnent les exceptions.
Nous introduirons dans un premier temps la classe native Exception de PHP 5. Puis nous étudierons comment lancer et attraper des exceptions dans un programme. A partir de là, nous serons capables d'étendre le modèle Exception pour développer des exceptions dérivées et de types différents. Enfin, nous aborderons la notion de gestion évènementielle des exceptions au moyen du handler d'exception natif de php.

La classe native Exception

Comme nous l'avons déjà expliqué dans les précédents tutoriels du chapitre de programmation orientée objet, PHP dispose depuis sa version 5 d'un modèle objet semblable à celui de Java. Pour agrémenter ce nouveau moteur, l'équipe de développement de PHP a intégré en natif une classe Exception. Cette classe très particulière permet au développeur de simplifier le traitement des cas exceptionnels susceptibles d'apparaître pendant l'exécution d'un programme en ***générant des objets*** de ***type Exception.*** En fin de compte, retenez qu'une exception n'est rien de plus qu'un objet (instance d'une classe). Le listing ci-après présente le code de la classe Exception.

Code source de la classe native Exception (extrait de la documentation officielle de PHP)

```
<?php

class Exception
{
  protected $message = 'exception inconnu'; // message de
l'exception
  protected $code = 0;                      // code de
l'exception défini par l'utilisateur
  protected $file;                          // nom du fichier
source de l'exception
  protected $line;                          // ligne de la
source de l'exception

  function __construct(string $message=NULL, int code=0);

  final function getMessage();              // message de
l'exception
  final function getCode();                 // code de
l'exception
```

```
  final function getFile();                  // nom du fichier
source
  final function getLine();                  // ligne du
fichier source
  final function getTrace();                 // un tableau de
backtrace()
  final function getTraceAsString();         // chaîne
formattée de trace

  /* Remplacable */
  function __toString();                     // chaîne formatée
pour l'affichage
}
?>
```

Nous identifions ici très clairement 4 parties distinctes qui composent cette classe Exception. Le premier bloc de code déclare les 4 données membres (attributs) de la classe en visibilité protégée. Respectivement, ces attributs enregistrent le message d'erreur, son code, le fichier source concerné et la ligne à laquelle l'erreur a été générée dans le programme.

Puis vient la déclaration du constructeur de la classe qui prend 2 paramètres facultatifs. Le premier est le message d'erreur et le second le code d'erreur.

La troisième partie du corps de la classe concerne la déclaration de 6 méthodes **accesseur** qu'il est impossible de redéfinir par héritage. Cette restriction est représentée par le mot-clé ***final*.** Ces méthodes permettent respectivement de récupérer le message d'erreur, son code d'erreur, le fichier source concerné, la ligne de l'erreur dans le fichier, et la pile des appels.
Enfin, nous découvrons la déclaration de la méthode magique ***__toString()*** qui pourra être rédéfinie ensuite par héritage. Cette méthode particulière, lorsqu'elle est surchargée, permet de spécifier de quelle manière on décide de représenter l'état en cours de l'objet sous forme de chaine de caractères.

Générer, lancer et attraper des exceptions à travers le programme

Générer une exception

La création d'une exception est réalisée par l'appel au constructeur de la classe native Exception. Le code ci-dessous illustre cette étape.

```
<?php

  // Création de l'objet Exception
  $e = new Exception('Une erreur s\'est produite');

  // Affiche le message d'erreur
  echo $e->getMessage();
?>
```

Remarquez la simplicité. La première ligne créer l'objet de type Exception ($e) et assigne automatiquement le message d'erreur dans le constructeur. La seconde ligne de code affiche le message d'erreur enregistré sur la sortie standard.

***Note** [1] : en développement informatique, les programmeurs ont pris l'habitude de nommer une exception uniquement avec la lettre miniscule "e". C'est à la fois une convention de nommage et une bonne pratique très largement répandue. Toutefois, aucune règle n'oblige les développeurs à l'adopter.*
***Note** [2] : la classe native Exception est chargée automatiquement par PHP, c'est pourquoi il n'est pas nécessaire d'avoir recours à un quelconque import avant de pouvoir l'utiliser.*

Lancer une exception à travers le programme

Notre code précédent nous a permis de générer des exceptions. En l'état, ce script ne nous sert strictement à rien puisqu'une exception n'est utile que si elle est créée lorsqu'un évènement exceptionnel se déroule pendant l'exécution du programme. Par exemple : une requête SQL qui échoue, un fichier impossible à ouvrir, une valeur d'un formulaire inattendue pour un champ...

Lorsqu'un tel évènement se produit, c'est que quelque chose d'inhabituel s'est passé. Par conséquent, la poursuite de l'exécution du programme doit être interrompue et signaler l'incident. C'est là qu'interviennent réllement les exceptions. Pour réaliser cette opération, le programme doit automatiquement « ***lancer*** » (retenez le vocabulaire de la POO) une exception. Le lancement d'une exception provoque immédiatement l'interruption du déroulement normal du programme
Le lancement d'une exception à travers le programme est réalisée grâce au mot-clé « ***throw*** ».

Exemple de lancement d'une Exception à travers le programme

```
<?php

  $password = 'Toto';
  if('Emacs' !== $password) {
    throw new Exception('Votre password est incorrect !');
  }

  // Cette ligne ne sera jamais exécutée
  // car une exception est lancée pour interrompre
  // l'exécution normale du programme
  echo 'Bonjour Emacs';
?>
```

L'utilisation du mot-clé *throw* permet de stopper l'exécution du programme et rediriger l'exception à travers ce dernier. La commande echo() quant à elle ne sera jamais exécutée. Le mot de passe $password n'étant pas égal à la chaine 'Emacs', l'exception est automatiquement générée puis lancée.
***Note** [1] : remarquez que l'exception généréese fait à la volée. Du fait qu'elle est automatiquement renvoyée auprogramme, nul besoin de stocker l'objet créé dans une variable.*
***Note** [2] : les exceptions peuvent également être lancées depuis l'intérieur d'une classe.*
Cet exemple est encore loin d'être exploitable en l'état. En effet, l'exception qui est lancée au programme est ici perdue pour toujours. Comme si l'on avait oublié de

stocker dans une variable le résultat retourné par une fonction. Nous avons déclaré au tout début de ce cours que les exceptions devaient permettre de traiter différemment les cas exceptionnels survenant au cours d'un programme. Pour cela, il est nécessaire de pouvoir « ***intercepter / attraper*** » l'exception générée pour appliquer le traitement adéquat. C'est là qu'intervient le bloc ***try / catch.***

Intercepter / attraper une exception générée

Comme en Java, AS 3, VB, C++... PHP dispose d'une structure conditionnelle capable d'intercepter les exceptions en plein vol afin de permettre d'appliquer des traitements particuliers. Il s'agit donc des blocs try { } catch() { }. Cette structure particulière se présente de la manière suivante :

Structure conditionnelle try { } catch() { }

```php
<?php

  try {
    // Liste d'actions à appeller
    // Ces actions peuvent potentiellement
    // lancer des exceptions à travers le programme
  }
  catch(Exception $e)
  {
    // Bloc des actions spéciales lorsqu'une
    // exception $e de type Exception est levée
  }
?>
```

Le bloc ***try*** « *essaie* » d'exécuter le script entre les deux premières accollades. Si une exception est lancée dans ce bloc, elle est immédiatement « *attraper* » dans le bloc ***catch()*** et les traitements particuliers sont exécutés à la place.
Concrètement cela donne le listing ci-après avec notre exemple précédent :

Exemple d'interception d'une exception

```php
<?php

  try {

    $password = 'Toto';

    if('Emacs' !== $password) {
      throw new Exception('Votre password est incorrect !');
    }

    echo 'Bonjour Emacs';
  }
  catch(Exception $e)
  {
    echo 'L\'erreur suivante a été générée : '."\n";
```

```
    echo $e->getMessage();
  }
?>
```

En exécutant ce code, nous découvrons que c'est le code du bloc catch() qui a été exécuté car l'exception lancée dans le bloc try { } a été interceptée en plein vol.

***Note** [1] : lorsqu'une exception est levée, elle remonte dans le programme et est interceptée par le premier bloc try {} catch() {} englobant qu'elle rencontre. Si aucun bloc try {} catch() {} n'entoure l'exception, alors l'objet sera perdu et l'exécution du programme interrompue.*
***Note** [2] : nous verrons plus loin dans ce tutoriel que nous pouvons appliquer plusieurs bloc catch() sous le bloc try {} afin d'identifier chaque type d'exception potentiellement générée.*
Bien sûr, il est possible d'imbriquer les blocs try { } catch() { } comme il est naturellement possible d'imbriquer les blocs if(). Notre exemple peut-être amélioré de la manière suivante :

Exemple d'utilisation de bloc try { } catch() { } imbriqués

```
<?php

  $login = 'Titi';
  $password = 'Toto';

  try {

    if('Hello' !== $login) {
      throw new Exception('Votre login est incorrect !');
    }

    try {

      if('Emacs' !== $password) {
        throw new Exception('Votre password est incorrect !');
      }

      echo 'Bonjour Emacs';
    }
    catch(Exception $e)
    {
      echo 'L\'erreur suivante a été générée : '."\n";
      echo $e->getMessage();
    }
  }
  catch(Exception $e)
  {
    echo 'L\'erreur suivante a été générée : '."\n";
    echo $e->getMessage();
  }
?>
```

Dans cet exemple, la valeur du login est d'abord testée. Comme la valeur du login est incorrecte, une exception est lancée à travers le programme, et c'est le bloc catch() le plus bas qui l'intercepte et exécute le traitement associé. Si maintenant nous remplissons correctement la variable $login, alors le premier bloc try est passé puis l'exception du second bloc try est lancée. Cette dernière est alors interceptée dans le bloc catch() associé pour exécuter le traitement adéquat de l'erreur.

Cet exemple n'est cependant pas très pertinent car il ne profite pas véritablement de toute la puissance des exceptions. La seconde partie de ce cours montre pas à pas comment concevoir des exceptions typées et personnalisées, et comment profiter de celles-ci pour simplifier et optimiser le code.

Introduction à la seconde partie de ce tutoriel

Dans la suite de ce cours, nous étudierons comment rendre l'utilisation des exceptions pertinentes en créant nos propres classes d'exception personnalisées. Nous aurons recours au concept de l'héritage, pilier majeur dans la méthodologie de l'approche orientée objet. Enfin, nous présenterons un aspect pratique de PHP qui permet d'intercepter toutes les exceptions à volée et d'exécuter automatiquement une fonction de *callback* (de rappel) pour simplifier et centraliser leur traitement.

Les exceptions - 2ème partie

La première partie de ce tutoriel a été l'occasion de présenter le mécanisme des exceptions de manière très théorique. Au travers d'exemples simples et concrets, nous avons découvert comment générer, lancer et attraper des exceptions en plein vol. A ce stade, nous sommes encore loin de pouvoir profiter pleinement des exceptions dans des applications plus conséquente. C'est pourquoi cette seconde et dernière partie s'intéressera à la manière de dériver la classe Exception pour créer des exceptions personnalisées. Enfin, nous étudierons un mécanisme natif de PHP qui permet de centraliser et d'unifier le traitement des exceptions non capturées dans une fonction de callback appelée automatiquement par l'exception handler.

Dériver la classe Exception et créer des exceptions personnalisées

Principe de création de classes personnalisées

Cette partie du tutoriel est sans aucun doute la plus triviale. Elleexplique comment dériver la classe Exception (via le concept del'héritage de programmation orientée objet) afin de générer ensuite desexceptions personnalisées. En effet, comme une ***classe = un type***,il est alors possible de créer de nouvelles classes (donc denouveaux types) qui héritent des propriétés et méthodes de la classeException.
Quel sont les avantages à créer des classes dérivées ? La raison estsimple. Avoir plusieurs types d'exceptions permet tout d'abord de lesdistinguer plus facilement. En distinguant chaque type d'erreur, on peut appliquer des traitements appropriés différents.Enfin, la dérivation de la classe Exception permet au développeur desurchager l'objet en lui ajoutant des propriétés et méthodessupplémentaires. Le code ci-dessous présente la manière la plus simplede développer une nouvelle classe d'exception personnalisée.

Principe de création d'une classe dérivée de Exception

```
<?php


  /**
   * Fichier MyChildException.class.php
   */
  class MyChildException extends Exception
```

```
  {
    public function __construct($message=NULL, $code=0)
    {
      parent::__construct($message, $code);
    }
  }
?>
```

Tout d'abord le nom de la nouvelle classe est déclarée (*MyChildException)* ainsi que la classe mère (*Exception*) qu'elle dérive et dont elle hérite les propriétés et méthodes. C'est grâce au mot-clé ***extends***que l'héritage a lieu.
Puis nous redéfinissons la méthode constructeur de la classe parente *Exception*. Lorsque le constructeur de la classe *MySuperException* sera appelé, il appellera automatiquement le constructeur de la classe parente. Cela aura pour effet de créer un objet de type *MySuperException* mais aussi de type *Exception*par héritage.
Lorsque l'on redéfinit le constructeur parent, il faut donc bienévidemment penser à récupérer au moins ses paramètres pour lesréinjecter dans le constructeur fils.

Exemples de classes d'exceptions personnalisées

Développons maintenant à titre d'exemple deux classes personnaliséespour gérer les exceptions liées aux fichiers. La première classe permetde contrôler l'inexistence d'un fichier sur un serveur tandis que laseconde permettra de générer des erreurs indiquant que le fichier n'estpas accessible en écriture. Ajoutons en plus à chacune de ces deuxclasses, un nouvel attribut stockant la date à laquelle s'est produitel'erreur et un accesseur pour récupérer cette valeur. Nous obtenonsdonc :

Classes FileNotFoundException et FileNotWriteableException

```
<?php


  /**
   * Fichier FileNotFoundException.class.php
   */
  class FileNotFoundException extends Exception
  {
    protected $timestamp;

    public function __construct($message=NULL, $code=0)
    {
      parent::__construct($message, $code);
      $this->timestamp = time();
    }

    public function getTimestamp() {
      return $this->timestamp;
    }
```

```
  }

  /**
   * Fichier FileNotWriteableException.class.php
   */
  class FileNotWriteableException extends Exception
  {
    protected $timestamp;

    public function __construct($message=NULL, $code=0)
    {
      parent::__construct($message, $code);
      $this->timestamp = time();
    }

    public function getTimestamp() {
      return $this->timestamp;
    }
  }
?>
```

Profitons tout de suite de l'héritage pour faire dériver ces deux classes de la même classe *FileException*et qui contiendra l'attribut et la méthode commune. Cette nouvelleclasse, quant à elle, dérivera bien entendu la classe native *Exception.*

Classes de gestion des erreurs liées au fichiers

```
<?php



  /**
   * Fichier FileException.class.php
   */
  class FileException extends Exception
  {
    protected $timestamp;

    public function __construct($message=NULL, $code=0)
    {
      parent::__construct($message, $code);
      $this->timestamp = time();
    }

    public function getTimestamp() {
      return $this->timestamp;
    }
  }

  /**
   * Fichier FileNotFoundException.class.php
   */
```

```
  class FileNotFoundException extends FileException
  {
    public function __construct($message=NULL, $code=0)
    {
      parent::__construct($message, $code);
      $this->timestamp = time();
    }
  }

  /**
   * Fichier FileNotWriteableException.class.php
   */
  class FileNotWriteableException extends FileException
  {
    public function __construct($message=NULL, $code=0)
    {
      parent::__construct($message, $code);
      $this->timestamp = time();
    }
  }
?>
```

L'exemple qui suit montre comment récupérer chaque type d'exceptionavec plusieurs blocs catch() associés au même bloc try {}. Grâce à cesblocs catch() en ligne et les différents types d'exceptions, il estalors très simple de reconnaître les erreurs levées et donc exécuterles traitements les plus adaptés.

Exemple d'utilisation des 3 classes de gestion des erreurs de fichier

```
<?php



  // Import des 3 classes précédentes
  require_once(dirname(__FILE__).'/FileException.class.php');

require_once(dirname(__FILE__).'/FileNotFoundException.class.p
hp');

require_once(dirname(__FILE__).'/FileNotWriteableException.cla
ss.php');

  // Variables
  $fichier = '/var/www/projet/toto.txt';

  try
  {
    // Le fichier existe-t-il ?
    if(!file_exists($fichier)) {
      throw new FileNotFoundException('Le fichier '. $fichier
.' est inexistant');
```

```
    }

    // Le fichier est-il inscriptible ?
    if(!is_writeable($fichier)) {
      throw new FileNotWriteableException('Le fichier '.
$fichier .' n\'a pas les droits d\'écriture');
    }

    // A-t-on ouvert le fichier en mode écriture ?
    if(!($fp = @fopen($fichier,'w'))) {
      throw new FileException('L\'ouverture du fichier '.
$fichier .' a échoué');
    }

    // J'écris dans mon fichier
    fwrite($fp, "Coucou Emacs\n");

    // Puis je ferme mon fichier
    fclose($fp);
  }
  catch(FileNotFoundException $e)
  {
    // Je crée le fichier
  }
  catch(FileNotWriteableException $e)
  {
    // Je change les droits du fichier
  }
  catch(FileException $e)
  {
    // Je stoppe tout
    exit($e->getMessage());
  }
  catch(Exception $e)
  {
    // Je stoppe tout
    exit($e->getMessage());
  }
?>
```

Vous remarquez ici les quatre blocs catch() en ligne permettantd'intercepter les trois types d'exceptions potentiellement jetablesdepuis le bloc try(). En précisant le type de l'exception interceptéedans le bloc catch(), nous savons à quel genre d'erreur nous avonsaffaire. N'est-ce pas plus clair et maintenable à présent ?

Note : *il est important de conserverl'ordre des blocs catch(), c'est-à-dire de mettre en premier le bloccatch() indiquant l'erreur la plus précise (donc la fille la plus bassepar héritage). De ce fait, la classe Exception, la plus générale, doitarriver dans le dernier bloc catch(). Si le bloc catch() du typeException avait été placé avant les autres, alors toutes les exceptions(tout type confondu) auraient toujours été levées dans celui-ci. Eneffet, une exception de type FileNotFoundException est*

aussi parhéritage une exception de type FileException et donc aussi uneexception de type Exception.

Centraliser le traitement des erreurs non capturées

Présentation du mécanisme d'interception automatique des exceptions

Jusqu'à maintenant nous savons que le moyen le plus simple de capturer une exception lancée est d'avoir recours à des blocs try { } catch() { }. Rappelez-vous le tout premier exemple de la 1ère partie du tutoriel, l'exception générée n'était pas capturable donc elle était forcément perdue à jamais.

Exemple de lancement d'une Exception à travers le programme

```
<?php

  $password = 'Toto';
  if('Emacs' !== $password) {
    throw new Exception('Votre password est incorrect !');
  }

  // Cette ligne ne sera jamais exécutée
  // car une exception est lancée pour interrompre
  // l'exécution normale du programme
  echo 'Bonjour Emacs';
?>
```

Heureusement PHP dispose d'un mécanisme qui permet de capturer automatiquement toutes les exceptions qui sont lancées mais qui ne sont pas entourées de blocs try {} catch() {} comme c'est le cas dans le listing ci-dessus. Il s'agit de l'*exception handler*.
L'exception handler, lorsqu'il intercepte une exception, interrompt complètement l'exécution du programme et appelle une fonction personnalisée de callback qui se chargera du traitement adéquat de ces exceptions perdues. Le code ci-après illustre sa mise en place.

Mise en place du mécanisme d'interception automatique des exceptions

```
<?php

  /**
   * Fonction de rappel appellée automatiquement par
l'exception handler
   *
   * @param Exception $e une exception lancée et perdue dans
le programme
   * @return void
   */
  function traitementExceptionPerdue(Exception $e) {

    echo 'Une exception orpheline a été attrapée : ';
```

```
    echo $e->getMessage(), "\n";
    exit;
  }

  /**
   * Enregistrement de la fonction de rappel dans l'exception
handler de PHP
   */
  set_exception_handler('traitementExceptionPerdue');

  // Exemple de génération d'exception perdu
  $password = 'Toto';
  if('Emacs' !== $password) {
    throw new Exception('Votre password est incorrect !');
  }

  // Cette ligne ne sera jamais exécutée
  // car une exception est lancée pour interrompre
  // l'exécution normale du programme
  echo 'Bonjour Emacs';

?>
```

Quelques implications s'imposent car le principe n'est pas si simple à assimiler. Dans un premier temps, nous déclarons la fonction de rappel qui personnalisera le traitement des exceptions orphelines interceptées. C'est une fonction utilisateur qui prend un seul et unique paramètre. Ce paramètre n'est autre qu'un objet de type Exception (ou type dérivé d'Exception). Le corps de la fonction effectue les traitement particuliers pour les exceptions orphelines. Ici nous affichons simplement un message d'erreur, suivi du message de l'exception. Puis nous stoppons strictement toute la suite du programme PHP. Nous aurions pu par exemple redirigé automatiquement l'utilisateur vers une page d'erreur de type 500 en utilisant la fonction header().

***Note** : en PHP 5, tous les objets sont automatiquement passés par référence (pointeur) en paramètre de fonction. Il n'est donc pas nécessaire de les préfixer du symbole &* dans la liste des paramètres de la signature.
La seconde partie de ce listing concerne l'enregistrement du nom de cette fonction comme gestionnaire d'exception par défaut à appeller automatiquement à la capture d'une exception orpheline. Il suffit simplement d'appeller la fonction set_exception_handler() de PHP, et de lui indiquer en paramètre le nom de la fonction *traitementExceptionPerdue*(). PHP prend ensuite le relais à la place du développeur.
***Note** : PHP interprête le code PHP en le lisant de haut en bas, c'est pourquoi il faut toujours déclarer la fonction de rappel avant d'appeller la fonction set_exception_handler().*
Enfin la dernière partie reprend l'exemple de code qui génère une exception non capturable. Au moment où l'exception est levée, PHP appelle automatiquement la fonction de rappel en lui passant en paramètre l'exception orpheline qui vient d'être générée. Le résultat ci-après est obtenu sur la sortie standard :

```
Une exception orpheline a été attrapée : Votre password est
incorrect !
```

Le mécanisme, bien que peu facile à assimiler de prime abord, se révèle très simple à mettre en place et efficace pour gérer les cas exceptionnels inattendus. Grâce à cet outil, les développeurs peuvent écrire une fonction complète pour faciliter le debugging en affichant par exemple tout le contexte d'éxécution du programme : variables globales, fichiers concernés, pile des appels de fonctions...

Effets de bord néfastes avec set_exception_handler

Attention, l'utilisation de set_exception_handler() doit être utilisé avec prudence car elle peut entraîner des effets de bords particulièrement gênants si elle est mal maîtrisée. En effet, si le corps de la fonction de rappel fait appel à d'autres fonctions ou méthodes susceptibles de lancer des exceptions, le gestionnaire d'exception sera solllicité en boucle. C'est la boucle infinie !!! Il faut donc s'assurer que le corps de la fonction de callback n'exécute pas de traitements susceptibles de générer des exceptions non maîtrisées.

Inconvénients et limitation des exceptions en PHP

L'utilisation des exceptions apporte un intérêt non négligeable au développeur lorsqu'il développe son application. Nous avons compris tout au long de ce tutoriel que ce mécanisme permettait d'identifier clairement les types d'erreurs générées dans le but de les traiter spécifiquement et efficacement. Néanmoins, nous pouvons relever quelques limitations au sujet des exceptions :

- PHP 5 n'exploite pas en natif les exceptions. C'est-à-dire que les fonctions PHP continuent de générer des erreurs et non des exceptions. De ce fait, le développeur est obligé de gérer à la fois les exceptions et les erreurs PHP dans ses procédures de debugging.
- Lorsqu'une exception est soulevée, le contexte local contenu dans un bloc try est sauvegardé, ceci peut être amplifié par l'imbrication de bloc try. L'abus de l'utilisation des exceptions peut donc diminuer considérablement les performances. (extrait du tutoriel des exceptions sur le site Developpez.com)
- Par choix. La plupart des langages choisissent de ne pas supprimer la gestion des erreurs au profit des exceptions pour des raisons de performances et de style de programmation (les exceptions sont équivalentes pour certains programmateurs à l'affreux goto, de plus les exceptions augmentent considérablement le nombre de lignes de code). D'autres langages, comme l'ADA, n'utilisent que les exceptions. (extrait du tutoriel des exceptions sur le site Developpez.com)

Conclusion

Nous arrivons à l'issue de ce tutoriel concernant la manipulation des exceptions. Depuis le début, nous avons appris à générer une exception, la lancer à travers le programme puis l'intercepter dans un bloc try {} catch() {} en vue d'un traitement spécifique. Puis nous nous sommes intéressés à la création d'exceptions personnalisées dans le but de faciliter la compréhension du code et l'identification

des types d'erreurs générées dans le programme. Enfin nous nous sommes arrêtés sur le mécanisme d'interception automatique d'exception orpheline intégré nativement dans le langage PHP. Nous sommes donc parés pour développer des applications orientées objets capables de générer et de traiter avec adéquation les cas exceptionnels d'erreur.

Utiliser l'interface Iterator avec PHP 5

PHP 5 a apporté son lot de nouveautés en matière de programmation orientée objet, notamment la possibilité d'utiliser des Interfaces. Iterator en est une et nous allons voir comment l'utiliser grâce à ce tutoriel. La structure conditionnelle " foreach " vous permettait jusqu'à lors de parcourir le contenu d'un tableau. Avec la version 5 de PHP, vous pouvez désormais parcourir tout un objet pour récupérer ses propriétés.

Rappel sur les Interfaces

Une interface est une manière de passer un contrat avec une classe, afin de s'assurer que celle-ci disposera bien de certaines méthodes/propriétés.

Contrairement à l'héritage, il est possible d'implémenter plusieurs interfaces[1], ce qui permet de pallier à l'impossibilité d'utiliser l'héritage multiple en PHP.

L'interface Iterator

L'interface native Iterator existe depuis déjà quelques temps en java et apporte un atout non négligeable en matière de programmation orientée objet. Voici ce à quoi ressemble le code de l'interface Iterator :

```
<?php
Interface Iterator {
    public function rewind();
    public function key();
    public function current();
    public function next();
    public function valid();
}
?>
```

Parcourir un objet avec l'instruction foreach()

```
<?php
class MaClasse {
    protected $arg1 = 'plop' ;
    protected $arg2 = 'plip';
    //...
}
$c = new MaClasse();
foreach($c as $key=>$value) {
    echo $key, ' : ', $value, '<br/>';
}
?>
```

Ce qui affichera :

```
arg1 : plop

arg2 : plip
```

C'est là qu'intervient l'interface Iterator. Elle va en effet vous permettre de personnaliser le comportement de foreach. Implémenter cette interface dans votre classe vous oblige de surcharger (ou redéfinir) les 5 methodes suivantes : rewind(), next(), key(), current() et valid().

```
<?php
class MaClasse implements Iterator {
    protected $n;
    const MAX = 5;

    public function rewind() {
        $this->n = 0;
    }
```

```php
    public function next() {
        $this->n++;
    }

    public function key() {
        return 'increment '.$this->n+1;
    }

    public function current() {
        return $this->n;
    }

    public function valid() {
        return $this->n<=self::MAX;
    }
}
$c = new MaClasse();
foreach($c as $key => $val) {
    echo $key,' : ',$val, '<br/>';
}
?>
```

Ce qui affichera :

```
increment 1 : 0

increment 2 : 1

increment 3 : 2

increment 4 : 3

increment 5 : 4
```

Ces méthodes seront appelées par foreach, dans cet ordre :

1. rewind lors de la première iteration, qui vous permet de rembobiner votre objet (ici on remet $n à 0)
2. valid qui vérifie que l'on n'est pas arrivé à la fin de notre itération (ici que $n est inferieur au nombre maximum d'itération fixé par la constante MAX). Si valid retourne TRUE, on poursuit sinon on s'arrête, c'est la fin de l'itération.
3. current qui retourne la valeur de l'itération courante (ici $n)
4. key qui retourne la clé de l'itération courante (ici key.$n)
5. next qui lance l'itération suivante (ici, on incrémente $n de 1)
6. Est en suite rappelée la méthode valid() qui vérifie une nouvelle fois que l'on n'est pas arrivé aux termes de l'Itération.

Note [1] : *Une classe ne peut cependant pas implémenter deux interfaces partageant des noms de fonctions.*

En savoir plus sur l'interface Iterator

- http://fr.php.net/interface
- http://fr.php.net/iterator
- http://fr.php.net/manual/fr/language.oop5.php
- http://expreg.com/fred_article.php?art=iterateurphp5
- http://alain-sahli.developpez.com/tutoriels/php/les-interfaces/
- http://julien-pauli.developpez.com/tutoriels/php/spl/#LII-A

Singleton : instance unique d'une classe

Dans la plupart des développements professionnels ou de grande envergure, il est nécessaire de savoir structurer correctement son application dans le but de faciliter sa conception et sa maintenance. Les « design patterns », où « patrons de conception » en français, constitue l'une des meilleures solutions à cette problématique. Avec le nouveau modèle orienté objet de PHP 5 proche de celui de Java, l'implémentation des design patterns est facilitée. Ce tutoriel s'intéresse à la présentation et à l'implémentation du motif Singleton, particulièrement employé au sein des applications web.

Qu'est-ce qu'un patron de conception ?

Définition extraite de Wikipedia :
En génie logiciel, un patron de conception (design pattern en anglais) est un concept destiné à résoudre les problèmes récurrents suivant le paradigme objet. En français on utilise aussi le terme motif de conception qui est une traduction alternative de «design pattern», perçue comme incorrecte par certains.
Les patrons de conception sont des solutions qui répondent à des problèmes récurrents d'architecture et de conception logiciels. Ce ne sont ni des algorithmes ni des fragments de code. Les desing patterns décrivent des procédés de conception généraux indépendemment du langage de programmation employé. C'est d'ailleurs pour cette raison que l'on présente les motifs de conception sous forme de diagramme de classes UML.

En clair, les patrons de conception doivent être perçus comme un outil de conception et de structuration formelle des applications informatiques. Ce sont des solutions éprouvées et figurant parmi les bonnes pratiques de programmation à adopter.

Introduction au patron de conception Singleton

Présentation du design pattern Singleton

Le Singleton, en programmation orientée objet, répond à la problématique de n'avoir qu'une seule et unique instance d'une même classe dans un programme. Par exemple, dans le cadre d'une application web dynamique, la connexion au serveur de bases de données est unique. Afin de préserver cette unicité, il est judicieux d'avoir recours à un objet qui adopte la forme d'un singleton. Il suffit donc par exemple de créer l'unique objet représentant l'accès à la base de données, et

de stocker la référence à cet objet dans une variable globale du programme afin que l'on puisse y accéder de n'importe où dans le script.

Structure d'une classe Singleton

Concrètement, un singleton est très simple à mettre en place. Il est composé de 3 caractéristiques :

- Un attribut privé et statique qui conservera l'instance unique de la classe.
- Un constructeur privé afin d'empêcher la création d'objet depuis l'extérieur de la classe
- Une méthode statique qui permet soit d'instancier la classe soit de retourner l'unique instance créée.

Le code ci-dessous présente une classe minimaliste qui intègre le motif de conception Singleton. Nous y retrouvons le strict minimum, à savoir les trois caractéristiques présentées juste au dessus.

Structure minimale du Singleton

```php
<?php

class Singleton {

  /**
   * @var Singleton
   * @access private
   * @static
   */
   private static $_instance = null;

   /**
    * Constructeur de la classe
    *
    * @param void
    * @return void
    */
   private function __construct() {
   }

   /**
    * Méthode qui crée l'unique instance de la classe
    * si elle n'existe pas encore puis la retourne.
    *
    * @param void
    * @return Singleton
    */
   public static function getInstance() {

     if(is_null(self::$_instance)) {
       self::$_instance = new Singleton();
```

```
        }

        return self::$_instance;
    }
}

?>
```

La variable $_instance est déclarée comme statique. C'est une variable de classe, c'est-à-dire que sa valeur ne dépend pas de l'objet créé à partir de cette classe. Ainsi, si deux objets sont issus de la même classe, ils peuvent partager cette même variable et lire la même valeur quel que soit leur état à l'instant T. Cette variable $_instance se charge de conserver l'objet de type Singleton qui sera créé. Grâce au constructeur privé et à la méthode getInstance(), il est impossible de créer plusieurs objets de type Singleton.

Le constructeur est déclaré privé afin de ne pas pouvoir être appelé depuis l'extérieur de la classe. Plus concrètement, il est ainsi impossible de faire un "new" en dehors de la classe. Le seul moyen d'obtenir une instance de la classe est de passer par la méthode publique et statique getInstance().

La méthode getInstance() doit obligatoirement être publique et statique. La déclaration avec le mot-clé "public" lui permet d'être appelée en dehors de la classe. Le mot-clé "static", quant à lui, lui permet d'être appelée sans passer par l'objet (puisque nous n'avons pas d'objet lorsqu'on l'appel). Le but de cette méthode est de faire du contrôle d'accès. Elle vérifie tout d'abord que la propriété statique $_instance est NULL ou non. Si elle est NULL alors elle crée dans cette variable une instance de la même classe. Ensuite elle retourne cette instance. C'est LA seule et unique instance possible que l'on peut faire : c'est le singleton.

Test de la classe Singleton

Pour contrôler que notre classe fait bien son travail, nous allons tout d'abord essayer d'instancier la classe depuis l'extérieur puis nous essaierons d'obtenir deux instances différentes de la classe.

Tentative d'instanciation depuis l'extérieur de la classe

```
<?php

  // Import de la classe
  require(dirname(__FILE__).'/Singleton.class.php');

  // Tentative d'instanciation de la classe
  $oSingleton = new Singleton();
?>
```

Du fait de la déclaration "private" du constructeur, nous ne pouvons instancier la classe directement. PHP génère donc une erreur non surprenante.

```
Fatal error: Call to private Singleton::__construct() from
invalid context in /Users/Emacs/Sites/Demo/MySingleton.php on
line 7
```

Tentative de construction de deux objets de type Singleton

```php
<?php

  // Import de la classe
  require(dirname(__FILE__).'/Singleton.class.php');

  // Tentative d'instanciation de la classe
  $oSingletonA = Singleton::getInstance();
  $oSingletonB = Singleton::getInstance();

  echo '<pre>';
  var_dump($oSingletonA);
  echo '</pre>';

  echo '<pre>';
  var_dump($oSingletonB);
  echo '</pre>';
?>
```

Nous obtenons le résultat suivant :

```
object(Singleton)#1 (0) {

}



object(Singleton)#1 (0) {

}
```

Le #1 correspond à la référence de l'instance. On constate ici que pour les deux objets $oSingletonA et $oSingletonB, cette référence est strictement identique. On en déduit donc que ces deux variables référencent (pointent) le même objet en mémoire. Nous avons donc réussi à créer une instance unique de la classe, le fameux Singleton.

Exemple d'implémentation de Singleton

En l'état, notre classe de Singleton ne nous permet pas d'être utilisable dans un projet concret. Elle constitue uniquement l'ossature minimaliste d'un objet adoptant la forme d'un Singleton. Nous allons donc nous appuyer sur un exemple concret tiré de la réalité pour illustrer davantage ce patron.

Comme nous le savons tous, une seule et unique personne se trouve à la tête de notre pays. Il s'agit du Président de la République. Il ne peut y en avoir plus d'un. Nous pouvons donc très facilement représenter cette contrainte sous forme d'une classe intégrant les spécificités du singleton.

```php
<?php

class PresidentDeLaRepublique {

  /**
   * L'objet unique PresidentDeLaRepublique
   *
   * @var PresidentDeLaRepublique
   * @access private
   * @static
   */
   private static $_instance = null;

   /**
    * Le nom du Président
    *
    * @var string
    * @access private
    */
   private $_nom='';

    /**
    * Le prénom du Président
    *
    * @var string
    * @access private
    */
   private $_prenom='';

   /**
    * Représentation chainée de l'objet
    *
    * @param void
    * @return string
    */
   public function __toString() {

     return $this->getPrenom() .' '. strtoupper($this-
>getNom());
   }

   /**
    * Constructeur de la classe
    *
    * @param string $nom Nom du Président
    * @param string $prenom Prénom du Président
    * @return void
    * @access private
    */
   private function __construct($nom, $prenom) {
```

```
      $this->_nom = $nom;
      $this->_prenom = $prenom;
    }

    /**
     * Méthode qui crée l'unique instance de la classe
     * si elle n'existe pas encore puis la retourne.
     *
     * @param string $nom Nom du Président
     * @param string $prenom Prénom du Président
     * @return PresidentDeLaRepublique
     */
    public static function getInstance($nom, $prenom) {

      if(is_null(self::$_instance)) {
        self::$_instance = new PresidentDeLaRepublique($nom,
$prenom);
      }

      return self::$_instance;
    }

   /**
     * Retourne le nom du Président
     *
     * @return string
     */
   public function getNom() {
     return $this->_nom;
   }

   /**
     * Retourne le prénom du Président
     *
     * @return string
     */
   public function getPrenom() {
     return $this->_prenom;
   }
}

?>
```

Création de l'objet :

```
<?php

  // Import de la classe

require(dirname(__FILE__).'/PresidentDeLaRepublique.class.php'
);
```

```
  // Instanciation de l'objet
  $oPresident =
PresidentDeLaRepublique::getInstance('Sarkozy','Nicolas');

  // Appel implicite à la méthode __toString()
  echo $oPresident;
?>
```

Nous obtenons bien le résultat ci-après sur la sortie standard :

```
Nicolas SARKOZY
```

Conclusion

Nous avons présenté ici le motif de conception Singleton qui permet de répondre à la problématique de n'avoir qu'une seule et même instance d'une classe. L'implémentation d'une telle structure est très rapide et simple à mettre en place. Il est possible par exemple de l'implémenter pour gérer la connexion au serveur de bases de données, le contexte de votre application, l'aire de jeux d'un jeu vidéo...

Introduction aux Cross Site Request Forgeries ou Sea Surf

Vous connaissez peut-être les attaques XSS qui consistent à injecter du code malveillant, et où l'utilisateur est directement victime de l'action du code (boîtes de dialogue, redirections, vols de cookies, etc.) ? Eh bien, ce tutoriel va vous apprendre un autre type de faille radicalement opposé à celui-ci. Il s'agit bien entendu des attaques CSRF.

Présentation générale des CSRF

Beaucoup moins répandues que les injections SQL,les attaques CSRF (autrement prononcées « *Sea Surf* ») exploitent les utilisateurs et les rendent complices de l'attaque. Notez que ce genre d'attaqueest spécifique aux applications web puisqu'elles exploitent essentiellement le navigateur de l'utilisateur.
Le principe est enfantin : on incite un internaute à se rendre sur une page afin qu'une requête spécifique soit déclenchée à son insu. Ainsi, par le fait qu'il se soit rendu sur la page,l'internaute devient complice de l'attaque.
Bien entendu, l'internaute en question est connecté (par exemple sur son espace membre, un backoffice d'administration, son interface de gestion de son compte bancaire...), et c'est bien là tout le danger de la faille...

Un exemple concret

Prenons un exemple concret afin que vous compreniez mieux le principe de ces failles. Certains scripts d'annuaires permettent l'affichage des sites les mieux notés en fonction du nombre de votes. A partir de celà, il est possible de décortiquer le

code de l'annuaire, de trouver la fonction qui incrémente la variable du nombre de votes et de s'en servir. On peut, par exemple, l'inclure en tant qu'image avec la balise <img /> et lui appliquer la propriété CSS display:none;. Cela nous donnerait quelque chose comme :

```
<div style="display:none;">
    <img src="http://www.example.com/vote.php?id=45" />
</div>
```

Au chargement de la page contenant ce code HTML, le navigateur exécutera une requête GET sur la ressource distante dont l'url est spécifiée par l'attribut "src" de la balise <img/>. Vous le savez peut-être, mais il est possible de renvoyer une image à partir d'un script PHP. C'est pourquoi il n'est pas étonnant de voir apparaître une url avec une page PHP et des paramètres, à la place d'un chemin absolu vers une véritable image.

Le navigateur, ne se souciant pas du type de fichier distant, va tenter de charger et d'afficher la fausse image. Si cela échoue, tant pis... La requête sur le fichier PHP a tout de même été exécutée ! Le fait d'utiliser display:none; rendra l'image invisible aux visiteurs. Ainsi, vous pourrez gonflez votre nombre de votes sans que personnes ne s'en rende compte. L'exemple ci-dessus est quelque peu simplet, mais la portée d'action de la faille n'est pas des moindres. En se basant sur le même principe, on aurait pu vous faire transférer une somme d'argent de votre compte, vous faire acheter un produit, etc.

Quels sont les moyens pour se protéger des CSRF ?

La seule chose à savoir c'est qu'il n'existe pas de technique ou de fonction 100% fiable et ultime pour vous protéger des CSRF, contrairement aux XSS pour lesquelles il suffit d'échapper les données de sortie. Néanmoins, vous pouvez rendre la tâche un peu plus ardue à ceux qui essaieraient ce genre d'attaque.

Vérifiez vos referers

Envoyé par votre navigateur, le referer permet de déterminer la provenance des requêtes. La vérification du referer n'est pas une fin en soi car on peut bloquer son envoi ou encore le modifier. Néanmoins, vous pouvez vous en servir pour diminuer le nombre d'attaques par CSRF.

Les tokens, pensons-y !

Cette technique permet de spécifier une durée de vie au formulaire. L'astuce consiste donc à vérifier que le formulaire n'a pas été saisi dans un délai trop court (robot) ou bien trop long. Voici un exemple :

```
<?php

$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;
$_SESSION['token_time'] = time();
```

```
?>
```

A partir de cela, il faudrait un champ caché dans notre formulaire (type="hidden") qui aurait pour valeur $token. Avec ces informations, il suffit alors de faire une petite condition comme celle-ci :

Vérification du jeton appliqué à un formulaire

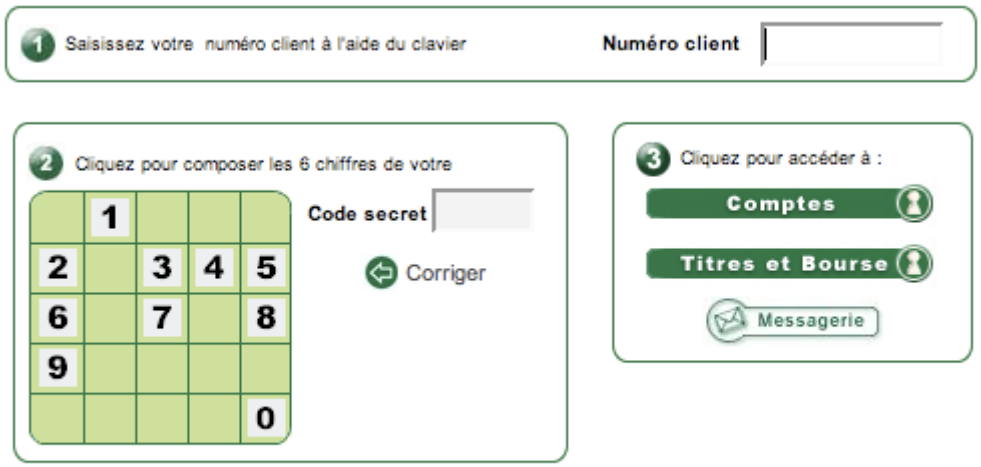
```php
<?php

  // Définition de la durée de vie du jeton
  // Ici il est de 180 secondes soit 3 minutes
  define('DELAI_MAXIMUM', 180);

  if($_POST['token'] == $_SESSION['token']
    && (time() - $_SESSION['token_time']) <= DELAI_MAXIMUM)
  {
    actions();
  }
  else
  {
    die('Jeton expiré !');
  }
?>
```

Vous pouvez également générer une suite de nombres aléatoires que vous mettrez dans chaque formulaire en champ caché (cf. paragraphe ci-dessus) A partir de cela, il faudra obligatoirement que cette suite de nombres soient contenue dans l'URL sans quoi votre code n'interprétera pas la requête. Vous devrez néanmoins choisir une suite de nombres assez longue afin que la chance qu'un pirate tombe sur un couple nombre / utilisateur valide soit infime.

Faites valider les actions critiques

Avant chaque action critique (ajout, suppression, édition, virement bancaire...), n'hésitez pas à soumettre un captcha à l'utilisateur afin d'être certains qu'il a bel et bien demandé l'exécution de la requête. Pour ne pas trop perdre en ergonomie, vous pouvez utiliser une vérification JavaScript (boîte de dialogue), mais cette pratique est très fortement déconseillée dans la mesure où l'exécution du script Javascript dépendra de la configuration du navigateur client.

L'un des plus bel exemple pour illustrer ces validations d'actions sont les formulaires dynamiques que l'on retrouve sur les sites bancaires. Il s'agit de formulaires qui vous demandent de saisir votre code avec la souris en pointant les chiffres adéquats (voir la capture ci-dessous).



Bien sûr, l'ordre et la disposition des chiffres dans la grille sont générés aléatoirement à chaque rechargement de page. Quel en est l'intérêt ? La disposition aléatoire permet tout d'abord de limiter les CSRF mais également de ralentir les attaques avec les nouveaux systèmes capables de détecter et d'enregister les positions du curseur sur l'écran (via un logiciel ou un javascript intégré avec une XSS).

Configuration de PHP et bonnes habitudes de programmation

Limitez l'utiliation de $_GET et desactivez les *register_globals* sur votre serveur. Préférez également toujours le traitement des actions importantes avec la méthode POST plutôt que la méthode GET. Vous limiterez ainsi fortement les risques d'attaques CSRF puisqu'elles se propagent généralement via des injections XSS et des URL.
Utilisez également des captchas, demandez une seconde saisie du mot de passe ou effectuez des vérifications javascript pour valider les actions importantes (achat, vente, transfert) sur votre site. L'ergonomie en souffrira quelque peu, mais c'est le prix à payer pour limiter les risques.

***Remarque** : notez au passage que ces conseils sont en contradiction avec la philosophie du « Web 2.0 » où les développeurs s'efforcent de limiter le nombre d'écrans pour obtenir des interfaces ergonomiques et « user friendly » (ami avec l'utilisateur). L'utilisation massive d'Ajax peut conduire à de nombreuses failles XSS et CSRF, et c'est pour cette raison que des sites comme Facebook, Twitter... sont la cible quotidienne de pirates malintentionnés.*
Comme il l'a été évoqué au début du tutoriel, il faut que l'utilisateur soit authentifié pour que l'attaque représente unintérêt pour le pirate. Parconséquent,vous pouvez commencer par limiter la durée de vie des sessions sur votre serveur en modifiantla directive *session.gc_maxlifetime* de votre php.ini.

Conclusion

Vous savez donc désormais ce qu'est une attaque par CSRF ainsi que différents moyens techniques pour vous en protéger. N'hésitez donc pas à les appliquer.Pour

ceux qui se le demandent encore, je n'ai volontairement pas donné d'exemples véritablementnuisibles pour ne pas vous tenter de les mettre en oeuvre ;)