Résumé de PHP

Cette page résume les principales caractéristiques du langage PHP (syntaxe, structures de contrôles...) et est illustrée par des exemples basiques.

Quelques-uns des exemples donné ci-dessous sont testables ici

Syntaxe et variables	1. La syntaxe 2. Les variables 3. Variables dynamiques 4. Types de variables 5. Le transtypage 6. Opérateurs et expressions
Structures de contrôle	1. if 2. switch 3. while 4. do / for 5. break et continue

Types de variables

Les variables n'ont pas à être explicitement déclarées.

Leur type est déduit des valeurs qui leurs sont affectées.

Cependant la notion de type existe fondamentalement dans PHP.

Nous ne présenterons pas ici la notion de classe qui apparaît progressivement jusqu'à la version 5 du langage PHP. En effet, ce concept sort de l'objectif général de ce cours.

Les entiers:

Les entiers sont de nombres dont la plage dépend des plate-formes, mais reste équivalente à la portée du type long en C. Sur une plate-forme 32 bits, les valeurs vont de -2 147 483 648 à +2 147 483 647.

En cas de dépassement de capacité, PHP convertit automatiquement l'entier incriminé en nombre décimal.

On pourra exprimer un entier en décimal, hexadécimal ou encore en octal.

```
$decimal = 10;
$hexa = 0x0F;
$octal = 020;
```

Les nombres décimaux:

Ce type de donnée a la même portée que le type C double. Elle varie selon les plates-formes, mais va en général de 1.7E-308 à 1.7E+308. On peut exprimer ce type sous forme de nombre normal avec un point décimal, ou en notation scientifique.

```
$normal = 0.017;
$scientifique = 17.0E-3;
```

Les chaînes:

Une chaîne est une suite de caractères. Elle peut être délimitée par des guillements simples ou doubles.

Attention, leur utilisation ne produira pas le même effet: les chaines entourées de guillemets doubles sont sujettes aux substitutions de variables et au traitement des séquences d'échappement, contrairement aux chaines entourées de guillemets simples.

```
<?
$chaine = "(mais pas à CS)";
echo "Je joue à Quake 3 $chaine\tQ3 rocks!";
echo 'Je joue à Quake 3 $chaine\tQ3 rocks!';
?>
```

Le premier appel à echo va afficher "Je joue à Quake 3 (mais pas à CS) Q3 rocks!",

tandis que le deuxième va afficher "Je joue à Quake 3 $chaine\tQ3 rocks!".

Les séquences d'échappement de PHP:

```
\n: Nouvelle ligne
\t: Tabulation
\r: Retour chariot
\\: Anti slash
\$: Signe dollar
```

Les tableaux:

Un tableau est un type de données pouvant contenir plusieurs valeurs, indexées numériquement ou à l'aide de chaines.

```
$tableau[0] = "ProgWorld";
$tableau[1] = "rulez!";
```

On remarquera que lorsqu'on veut ajouter un élément au tableau, il suffit de faire une affectation dans spécifier d'indice:

```
$tableau[]="pouet";
```

Cette technique est aussi valable pour les tableaux dont les valeurs ne sont pas indexées successivement (i.e si les postes 1, 10, et 3 d'un tableau sont affectées, PHP affectera dans ce cas le poste d'indice 11).

Précision sur les chaines de caractères: notez que celles-ci peuvent tout à fait être considérées comme des tableaux de caractères.

Une affectation de type $MaChaine[3]='a'; est tout à fait possible.

Les tableaux peuvent également, comme nous l'avons dit, utiliser des chaines pour leur indexation. On parle alors de tableaux associatifs. Il est même possible de mélanger indexation numérique et indexation par chaine dans le même tableau.

```
< // exemple de tableau associatif
$mois["Janvier"] = 1;
$mois["Février"] = 5;
$mois["Mars"] = 3;

foreach ($mois as $elt) {
echo "<p>".$elt;
}
?>
```

Pour les explications sur la structure de contrôle foreach, voir plus loin dans ce document.

Le transtypage

Le transtypage désigne l'action de convertir un type de données en un autre.

Ici, il peut servir en plus à définir explicitement le type d'une donnée. La façon d'opérer est exactement la même qu'en C:

```
$UnEntier = (int) "4320Pastèque";
```

Ici, au lieu de créer une chaine, PHP va, grâce au "(int)", créer un entier qui aura pour valeur 4320.

Voici les opérateurs de transtypage du PHP:

```
(int), (integer): Conversion en entier
(real), (double), (float): Conversion en nombre décimal
(string): Conversion en chaine
(array): Conversion en tableau
(object): Conversion en objet
```

Le PHP propose des fonctions qui permettent de vérifier le type d'une variable: is_int(), is_float(), is_double(), is_array(), is_string(), is_object(), etc...

Opérateurs et expressions

Opérateurs arithmétiques

$a + $b	Addition	Somme de $a et $b.
$a - $b	Soustraction	Reste de la différence de $b et $a.
$a * $b	Multiplication	Produit de $a par $b.
$a / $b	Division	Dividende de $a par $b.
$a % $b	Modulo	Reste de la division entiàre de $a par $b.

Opérateurs sur les chaînes

Il n'existe qu'un seul opérateur en PHP3 sur les chaînes, c'est l'opérateur de concaténation.

Cependant PHP3 dispose en standard de toutes les fonctions nécessaires au maniement des chînes de caractères. Pour avoir leur description, reportez-vous à la documentation en ligne sur PHP : http://www.nexen.net/docs/php/annotee/language.basic-syntax.phpmode.php.

```
$a = "Hello ";
$b = $a . "World!"; // maintenant $b contient "Hello World!";
```

Il est à noter que vous pouvez spécifier une chaîne en l'encadrant entre des simples ou des doubles quotes. Dans le second cas, PHP3 tentera d'interprêter le contenu de la chaîne. La différence au niveau performance existe, mais est mineure.

```
$val='Pengo';
echo 'Hello $val';   // Affiche  Hello $val
echo "Hello $val";   // Affiche  Hello Pengo
```

L'utilisation de l'opérateur arithmétique d'addition en vue de concaténer deux chaînes de caractères **est interdite**. Un paramètre de PHP3.INI permet de signaler ces mauvaises utilisations de l'opérateur +.

```
warn_plus_overloading = Off   ; warn if the + operator is used with strings
```

Opérateurs Binaires

Il s'agit des opérateurs binaires dits "bit à bit", et non pas des opérateurs logiques booléens.

$a & $b	Et	Les bits qui sont à 1 dans $a ET dans $b sont mis à 1.
$a | $b	Ou	Les bits qui sont à 1 dans $a OU dans $b sont mis à 1.
~ $a	Non	Les bits sont inversés.

Opérateurs Logiques

Chose originale avec PHP3, il existe deux versions des opérateurs logiques ET et OU, avec des précédences différentes.

$a and $b	Et	Résultat vrai si $a ET $b sont vrais
$a or $b	Ou	Résultat vrai si $a OU $b est vrai (ou les deux)
$a xor $b	Ou exclusif	Résultat vrai si $a OU $b est vrai, mais pas si les deux sont vrais
! $a	Non	Résultat vrai si $a est faux, et réciproquement
$a && $b	Et	Résultat vrai si $a ET $b sont vrais
$a || $b	Ou	Résultat vrai si $a OU $b est vrai (ou les deux)

Opérateurs non documentés

Pour les nostalgiques des notations binaires et langages machines, deux opérateurs de décalage existent en PHP3, mais ils sont bizarrement absents de la documentation de référence :

```
// Shift the bits of $a $b steps to the left  (each step means "multiply by two")
$a << $b
// Shift the bits of $a $b steps to the right (each step mean "divide by two")
$a >> $b
```

Opérateurs de Comparaison

Encore du classique. Sans surprise pour les habitués de la syntaxe C.

$a == $b	égal	Résultat vrai si $a est égal à $b
$a != $b	Différent	Résultat vrai si $a est différent de $b
$a < $b	Inférieur	Résultat vrai si $a est strictement inférieur à $b
$a > $b	Supérieur	Résultat vrai si $a est strictement supérieur à $b
$a <= $b	Inf ou égal	Résultat vrai si $a est inférieur ou égal à $b
$a >= $b	Sup ou égal	Résultat vrai si $a est supérieur ou égal à $b

Structures de contrôle

Les structures de contrôle du PHP sont très similaires à celles du langage C. Cependant, PHP dispose de deux syntaxes équivalentes.

L'une est dans un pur style C, avec des accolades pour englober les instructions, tandis que l'autre est un peu plus explicite, à la manière du pseudo-code.

L'utilisation de l'une ou de l'autre est une simple affaire de goût, sachant qu'il n'est jamais obligatoire d'en utiliser une en particulier.

if

L'instruction if est une des structures de contrôle de base que l'on retrouve dans la plupart des langages. Voici les deux syntaxes:

```
if(expression) {
    instructions
}
elseif(expression) {
    instructions
}
else {
    instructions
}
    if(expression):
    instructions
elseif(expression):
    instructions
else:
    instructions
endif;
```

Si expression est évaluée à vrai, la portion de code suivant le if sera exécutée.

Sinon, l'expression suivant le elseif sera évaluée, et le code suivant cette instruction exécuté si l'expression est évaluée à vrai.

Dans le cas contraire, les instructions suivant les else seront exécutées.

Le else et le elseif sont bien sûr optionnels.

Notez que dans la première syntaxe, on peut se passer des accolades s'il n'y a qu'une seule instruction à exécuter.

```
<?
$i = 3;

if( $i == 2 )
{
    echo "La variable i est égale à 2";
}
elseif( $i == 3 )
{
    echo "La variable i est égale à 3";
}
else
{
    echo "La variable i n'est égale ni à 2, ni à 3";
}
?>
```

Ce code affichera donc "La variable i est égale à 3"...

A vous de changer la valeur de $i au début du script pour bien comprendre le fonctionnement du if.

Pour tester cet exemple, voir essai-ifcascade.php

switch

L'instruction switch peut élégamment remplacer une longue série de if peu lisibles. Voici les deux syntaxes:

```
switch(expression) {
case expression:
    instructions
break;
default:
    instructions
break;
}
      switch(expression):
case expression:
    instructions
break;
default:
    instructions
break;
endswitch;
```

L'expression de chaque instruction case est comparée à l'expression du switch.

Si elles sont égales, le bloc d'instructions suivant le case est exécuté.

Le mot-clef break permet de sortir du switch. Sans lui, toutes les exécutions suivant le case concerné (y compris celles des case suivants) s'exécuteraient.

Si aucun des case ne correspond à l'expression switch, c'est le bloc d'instructions suivant default qui sera exécuté.

Si on reprend l'exemple précédent, le script devient:

```
<?
$i = 3;

switch($i)
{
    case 2:  echo "La variable i est égale à 2";  echo "<br>le php ressemble au C";  break;
    case 3:  echo "La variable i est égale à 3";  break;
    default:  echo "La variable i n'est égale ni à 2, ni à 3";  break;
}
?>
```

Remarquez que, malgré son omniprésence, break n'est pas requis par l'instruction switch. En effet, on peut très sciemment s'en passer: imaginons le cas suivant:

```
<?
$i = 3;

switch($i)
{
    case 2:  echo "La variable i est égale à 2";  break;
    case 3:  echo "La variable i est égale à 3";  break;
    default:  echo "<br>le php ressemble au C";  break;
}
?>
```

On pourrait très bien contracter ce code en:

```
<?
$i = 3;

switch($i)
{
    case 3:  echo "La variable i est égale à 3";  break;
    case 2:  echo "La variable i est égale à 2";
    default:  echo "<br>le php ressemble au C";  break;
}
?>
```

Ce code donnera exactement le même résultat que le précédent.

On notera aussi que l'on peut grouper les case, lorsqu'on teste, par exemple, l'appui sur les touches d'un clavier.

Supposons que l'on veuille réagir lorsque l'utilisateur enfonce la touche 'r', que ce soit en majuscules ou en minuscules. On écrira alors:

```
<?
// on suppose que la variable $touche
// contient la touche enfoncée

switch($touche)
{
    case 'r':
    case 'R':  echo "Vous avez enfoncé R ou r.";  break;
    default:  echo "Vous n'avez pas enfoncé R ou r.";  break;
}
?>
```

while

L'instruction while permet de répéter le bloc d'instructions qu'il contient aussi longtemps qu'une expression est vraie. Voici les deux syntaxes:

```
while(expression) {
    instructions
}
```

```
while(expression):
    instructions
endwhile;
```

De la même façon que pour le if, la première syntaxe peut se passer des accolades s'il n'y a qu'une instruction à exécuter.

L'expression while est évaluée avant chaque itération (une itération = un tour de boucle).

Ce qui veut dire que si votre expression est fausse, le programme ne passera même pas une fois dans votre boucle.

Si l'expression est vraie, le code sera exécuté, puis l'expression réévaluée.

Si elle se révèle vraie une fois de plus, le code sera exécuté une seconde fois, etc, jusqu'à ce que l'expression soit fausse.

Si l'expression est fausse, l'exécution du programme reprend immédiatement après la fermeture de la boucle.

```
<?
$i = 0;

while( $i < 10 )  // tant que $i est plus petit que 10
{
```

```
    echo "La variable i est égale à $i<br>";
    $i++;       // on incrémente $i
}
?>
```

Dans ce script on met une variable $i à 0, puis, dans un while, on l'incrémente après avoir affiché sa valeur.

La condition d'accomplissement de la boucle est que $i soit strictement inférieur à 10.

Les boucles while sont souvent sources d'erreurs aux conséquences assez catastrophiques. Afin de minimiser ce risque, prenez garde aux choses suivantes:

- N'oubliez pas d'initialiser toute variable faisant partie de l'expression while, sous peine de résultats aléatoires...
- De même, si vous faîtes une boucle telle que celle ci-dessus, n'oubliez pas de faire varier la variable qui fait partie de l'expression. Vous risquez sinon de créer une boucle infinie.

do / while

L'instruction do/while fonctionne comme l'instruction while, à une exception près: l'expression est évaluée après la première itération et non avant.

Ainsi, même si l'expression est fausse, le code contenu dans le do/while sera exécuté une fois.

```
do {
    instructions
} while(expression);
```

(nb: il n'existe qu'un seule syntaxe pour cette structure de contrôle.)

Les accolades peuvent être omises s'il n'y a qu'une seule instruction.

Le script précédent, avec un do/while:

```
<?
$i = 0;
do
{
    echo "La variable i est égale à $i<br>";
    $i++;
} while( $i < 10 );
?>
```

for

Une boucle for, c'est un peu comme un while, mais en automatique. Voici les deux syntaxes:

```
for(expr_de_départ; expr_conditionnelle; expr_itérative) {
    instructions
}
    for(expr_de_départ; expr_conditionnelle; expr_itérative):
    instructions
endfor;
```

Une boucle for contient trois expressions. La première est l'expression de démarrage: elle sera évaluée une et une seule fois au début de la boucle. On l'utilise en général pour initialiser un compteur.

La seconde expression est l'expression conditionnelle. L'expression est évaluée avant chaque itération, et, si elle est vraie, le code contenu dans la boucle est exécuté. Sinon l'exécution du programme se poursuit à partir de l'instruction suivant la fermeture de la boucle.

La troisième et dernière expression est l'expression itérative. Elle est évaluée à la fin de chaque itération. On l'utilise en général pour incrémenter le compteur qu'on a initialisé dans l'expression de démarrage.

Dans la première syntaxe, on peut se passer des accolades si la boucle for ne contient qu'une instruction.

Voici le même script que précédemment, cette fois avec un boucle for:

```
<?
for($i = 0; $i < 10; $i++)
{
    echo "La variable i est égale à $i<br>";
}
?>
```

Même si ça parait moins clair de prime abord, on s'aperçoit vite que c'est bien plus pratique et lisible qu'un while.

foreach

Lorsque des valeurs sont contenues dans un tableau associatif, on peut traiter une à une chacune de ces valeurs, sans pour autant connaître d'avance les noms des indexes de la table.

Dans l'exemple ci-dessous, le tableau `$mois` contient 3 valeurs, chacune accessible respectivement par les chaînes d'index : "`Janvier`", "`Février`" et "`Avril`".
On ne sait pas d'avance que le mois de "`Mars`" n'est pas concerné.

```
< // exemple de tableau associatif
$mois["Janvier"] = 1;
$mois["Février"] = 5;
$mois["Avril"] = 3;

foreach ($mois as $elt) {
echo "<p>".$elt;
}
?>
```

L'instruction `foreach` permet d'instancier la variable `$elt;` successivement avec chacune des valeurs trouvées dans le tableau `$mois`.

Pour tester cet exemple, voir essai-tab.php

break et continue

`break` et `continue` sont deux mots-clefs bien utiles avec les boucles `while`, `do/while` et `for`.

Ils permettent respectivement de sortir de la boucle en cours et de passer à l'itération suivante.

Si on utilise `break` dans des boucles imbriquées, on sortira de la boucle courante pour atterrir dans la boucle de niveau supérieur.

Cependant, on peut sortir de plusieurs niveaux de boucles en une seuls instruction en utilisant `break n`, où `n` est le nombre de boucles imbriquées à interrompre.

De même, en utilisant continue n, on peut sauter les itérations courantes de n boucles imbriquées.

Ces deux mot-clefs fonctionnent de la même façon avec `while`, `do/while` et `for`, sauf pour continue dans le cas d'une boucle `for`.

En effet, dans ce cas, on notera que l'utilisation de ce mot-clef force l'évaluation de l'expression itérative avant la vérification de l'expression conditionnelle de la boucle.

Plus simplement, on pourrait dire que le programme "n'oublie pas" de traiter votre compteur de boucle comme vous lui avez dit de le faire, même si vous utilisez un `continue`.

Supposons que l'on veuille reprendre l'exemple précédent, mais cette fois sans faire d'affichage lorsque `$i` vaut 5:

```
<?
for($i = 0; $i < 10; $i++)
{
    if( $i == 5 )  continue;
    echo "La variable i est égale à $i<br>";
}
?>
```

Avec break, nous pouvons arrêter l'affichage lorsqu'on atteint le nombre 5 en sortant de la boucle:

```
<?
for($i = 0; $i < 10; $i++)
{
    if( $i == 5 )  break;
    echo "La variable i est égale à $i<br>";
}
?>
```

Pour tester les exemples avec `break`, dont les exemples des `switch` (`case`), voir form-essai.php