|  |
| --- |
| Contexte GSB  Normes de développement  Applications web écrites en PHP |
| Référence : GSB-STDWEBPHP  Version : 1.0 |
|  |

Table des matières

[Introduction 3](#_Toc476818789)

[1 Fichiers 4](#_Toc476818790)

[1.1 Extension des fichiers 4](#_Toc476818791)

[1.2 Nom des fichiers 4](#_Toc476818792)

[1.3 Format des fichiers 4](#_Toc476818793)

[2 Préambule XML 4](#_Toc476818794)

[2.1 Inclusion de scripts 5](#_Toc476818795)

[1 Présentation du code 6](#_Toc476818796)

[1.1 Tag PHP 6](#_Toc476818797)

[1.2 Séparation PHP/ HTML 6](#_Toc476818798)

[1.3 Caractères et lignes 7](#_Toc476818799)

[1.4 Indentation et longueur des lignes 7](#_Toc476818800)

[1.5 Espacement dans les instructions 8](#_Toc476818801)

[1.6 Présentation des blocs logiques 9](#_Toc476818802)

[1.7 Appels de fonctions / méthodes 10](#_Toc476818803)

[1.8 Définition de fonctions 10](#_Toc476818804)

[2 Nommage des identificateurs 11](#_Toc476818805)

[2.1 Nommage des fonctions 11](#_Toc476818806)

[2.2 Nommage des constantes 11](#_Toc476818807)

[2.3 Nommage des variables et paramètres 12](#_Toc476818808)

[3 Algorithmique 13](#_Toc476818809)

[3.1 Fonctions/méthodes 13](#_Toc476818810)

[3.1.1 Modularité 13](#_Toc476818811)

[3.1.2 Nombre de paramètres des fonctions 13](#_Toc476818812)

[3.2 Instructions 13](#_Toc476818813)

[3.2.1 Écriture des instructions d’affectation 13](#_Toc476818814)

[3.2.2 Parenthésage des expressions 14](#_Toc476818815)

[3.2.3 Interdiction des instructions imbriquées 14](#_Toc476818816)

[3.2.4 Limitation de l’utilisation des *break* et *continue* dans les itératives 15](#_Toc476818817)

[*3.2.5* Écriture des *switch* 15](#_Toc476818818)

[3.2.6 Utilisation de l’opérateur ternaire conditionnel 16](#_Toc476818819)

[4 Gestion des formulaires HTML 17](#_Toc476818820)

[4.1 Nommage des formulaires et des champs de formulaires 17](#_Toc476818821)

[4.2 Méthodes de soumission des formulaires 17](#_Toc476818822)

[4.3 Variables superglobales $\_GET, $\_POST 18](#_Toc476818823)

# Introduction

Ce document s'appuie sur différentes sources de règles de codage, en particulier du projet communautaire PEAR - "PHP Extension and Application Repository"[[1]](#footnote-1) et du cadre Zend Framework[[2]](#footnote-2) qui fournissent entre autres des règles de codage pour les scripts PHP. Le document s'inspire aussi des règles de codage issues d'autres langages tels que Java.

Les règles énoncées par le présent document comportent au minimum :

* une **description** concise de la règle,

Si nécessaire :

* des **compléments** par rapport à la description,
* des **exemples** illustrant la règle, et éventuellement des **exceptions,**
* une partie **intérêts** en regard des critères qualité**.**

*NB : La version actuelle des règles de codage ne comporte pas de règles sur les notions de POO (classe, niveau d'accès, membres d'instance ou de classe, etc.).*

# Fichiers

Ce paragraphe a pour but de décrire l’organisation et la présentation des fichiers mis en jeu dans un site Web dynamique écrit en PHP.

## Extension des fichiers

Description

Les fichiers PHP doivent obligatoirement se terminer par l'extension **.php** pour une question de sécurité. En procédant ainsi, il n'est pas possible de visualiser le source des fichiers PHP (qui contiennent peut-être des mots de passe), le serveur web les fait interpréter par PHP.

Les fichiers qui ne constituent pas des pages autonomes (des fichiers destinés à être inclus dans d'autres pages web) se terminent par l'extension **.inc.php**.

Les fichiers contenant uniquement des définitions de fonctions se terminent par l'extension **.lib.php**.

Un fichier contenant une classe se nommera **class.<nom de la classe>.inc.php**

Les fichiers contenant des pages statiques (sans code PHP) doivent porter l'extension **.html**.

## Nom des fichiers

Description

Seuls les caractères alphanumériques, tirets bas et tirets demi-cadratin ("-") sont autorisés. Les espaces et les caractères spéciaux sont interdits.

## Format des fichiers

Description

Tout fichier .php ou page .html doit :

Etre stocké comme du texte ASCII

Utiliser le jeu de caractères UTF-8

Etre formaté Dos

Compléments

Le << formatage Dos >> signifie que les lignes doivent finir par les combinaisons de retour chariot / retour à la ligne (CRLF), contrairement au << formatage Unix >> qui attend uniquement un retour à la ligne (LF). Un retour à la ligne est représenté par l'ordinal 10, l'octal 012 et l'hexa 0A. Un retour chariot est représenté par l'ordinal 13, l'octal 015 et l'hexa 0D.

## Préambule XML

Description

Les pages Web doivent se conformer à une des normes HTML ouXHTML.

Toute page Web devra donc débuter par la directive <!DOCTYPE précisant quelle norme est suivie.

Elles seront validées à l'aide du validateur en ligne http://validator.w3.org

Exemple

Pour exemple, voici l'en-tête d'un fichier XHTML 1.0 :

|  |
| --- |
| <?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> |

## Inclusion de scripts

L'inclusion de scripts peut être réalisée par plusieurs instructions prédéfinies en PHP : include, require, include\_once, require\_once. Toutes ont pour objectif de provoquer leur propre remplacement par le fichier spécifié, un peu comme les commandes de préprocesseur C #include.

Les instructions include et require sont identiques, hormis dans leur gestion des erreurs. include produit une alerte ( warning ) tandis que require génère une erreur fatale . En d'autres termes, lorsque le fichier à inclure n'existe pas, le script est interrompu. include ne se comporte pas de cette façon, et le script continuera son exécution.

La différence entre require et require\_once (idem entre include et include\_once) est qu'avec **require\_once()**, on est assuré que le code du fichier inclus ne sera ajouté qu'une seule fois, évitant de ce fait les redéfinitions de variables ou de fonctions, génératrices d'alertes.

Description

L'inclusion de scripts sera réalisée à l'aide de l'instruction require\_once lorsque les fichiers inclus contiennent des bibliothèques, require sinon.

Exemple

Script prem.inc.php

<?php

function uneFonction () {

echo "Fonction définie dans prem.inc.php <br />";

}

?>

Script second.inc.php

<?php

require\_once("prem.inc.php");

uneFonction();

echo "Pas de problème : uneFonction n'a pas été redéfinie 2 fois. Merci require\_once ! <br />";

?>

Script monscript.php

<?php

require\_once(prem.inc.php);

require\_once(second.inc.php);

echo "Tout va bien ! <br />";

?>

# Présentation du code

## Tag PHP

Description

*Toujours* utiliser <?php ?> pour délimiter du code PHP, et non la version abrégée <? ?>. C'est la méthode la plus portable pour inclure du code PHP sur les différents systèmes d'exploitation et les différentes configurations.

Intérêts

Portabilité

## Séparation PHP/ HTML

Description

Les balises HTML doivent se situer au maximum dans les sections HTML et non incluses à l'intérieur du texte des messages de l'instruction d'affichage echo.

Exemples

*Ne pas écrire*

<?php

echo "<select id=\"lstAnnee\" name=\"lstAnnee\">";

$anCours = date("Y");

for ( $an = $anCours – 5 ; $an <= $anCours + 5 ; $an++ ) {

echo "<option value=\"" . $an ."\">" . $an . "</option>";

}

echo "</select>";

?>

</select>

*Mais écrire*

<select id="lstAnnee" name="lstAnnee">

<?php

$anCours = date("Y");

for ( $an = $anCours – 5 ; $an <= $anCours + 5 ; $an++ ) {

?>

<option value="<?php echo $an ; ?>">echo $an; ?></option>

<?php

}

?>

</select>

Intérêts :

Dans les sections HTML, l'éditeur de l'outil de développement applique la coloration syntaxique sur les balises et attributs HTML. Ceci accroît donc la lisibilité et la localisation des erreurs de syntaxe au niveau du langage HTML. Il est aussi plus aisé d'intervenir uniquement sur la présentation, sans effet de bord sur la partie dynamique.

Maintenabilité : lisibilité

Portabilité : indépendance

## Caractères et lignes

Description

Chaque ligne doit comporter au plus une instruction.

Les caractères accentués ne doivent pas être utilisés dans le code source, excepté dans les commentaires et les messages texte.

Un fichier source ne devrait pas dépasser plus de **500 lignes**.

Exemples

*Ne pas écrire*

$i-- ; $j++ ;

*Mais écrire*

$i-- ;  
$j++ ;

Intérêts

Maintenabilité : lisibilité et clarté du code.

## Indentation et longueur des lignes

Description

Le pas d’indentation doit être **fixe** et correspondre à **4 caractères.** Ce pas d'indentation doit être paramétré dans l'éditeur de l'environnement de développement. L’indentation est stockée dans le fichier sous forme de 4 caractèresespace(sans tabulation réelle).

Il est recommandé que la longueur des lignes ne dépasse pas 75 à 85 caractères.

Lorsqu’une ligne d’instruction est trop longue, elle doit être coupée après une virgule ou avant un opérateur. On alignera la nouvelle ligne avec le début de l’expression de même niveau de la ligne précédente.

Exemples

***Exemple 1 : découpage d’un appel de fonction***

*On découpe la ligne après une virgule :*

$maVar = fonctionA(expressionLongue1, expressionLongue2,  
 fonctionB(expressionLongue3,  
 expressionLongue4));

***Exemple 2 : découpage d’une expression arithmétique***

*La ligne est découpée avant un opérateur :*

$maVar = expressionLongue1 \* expressionLongue2

+ expressionLongue3 - expressionLongue4

***Exemple 3 : découpage d'une expression conditionnelle***

*La ligne est découpée avant un opérateur :*

if(((condition1 && condition2) || (condition3 && condition4))

&& !(condition5 && condition6)) {

doSomething();

}

Intérêts

Maintenabilité : lisibilité.

## Espacement dans les instructions

Description

1. Un mot-clé suivi d'une parenthèse ouvrante doit être séparé de celle-ci par un espace. Ce n'est pas le cas entre un identificateur de fonction et la parenthèse ouvrante.
2. Tous les opérateurs binaires, sauf l'opérateur « -> » doivent être séparés de leurs opérandes par un espace.
3. Les opérateurs unaires doivent être accolés à leur opérande.

Exemples :

1. while (true) {  
    ...  
   }
2. $totalTTC **=** $totalHT + ($totalHT \* ($tauxTVA / 100));  
   $totalTTC = round($totalTTC, 2);  
   $existe = $unElt**->hasAttribute**();
3. **$nb = 0**;  
   $fin = false;  
   while (**!$fin**) {  
      
   ... $nb++;  
   }
4. for ($nbLignes = 1**;** $nbLignes < 4**;** $nbLignes++) {  
      
   }

Intérêts

Maintenabilité : lisibilité

## Présentation des blocs logiques

Description

1. Chaque bloc logique doit être délimité par des accolades, même s’il ne comporte qu’ une seule instruction (cf. exemple 1),
2. Dans une instruction avec bloc, l’accolade ouvrante doit se trouver sur la fin de la ligne de l’instruction ; l’accolade fermante doit débuter une ligne, et se situer au même niveau d’indentation que l’instruction dont elle ferme le bloc (cf. exemple 2),
3. Les instructions contenues dans un bloc ont un niveau supérieur d'indentation.

Exemples

***Exemple 1 :***

*Ne pas écrire Mais écrire*

if ($prime > 2000) if ($prime > 2000) **{**

$prime = 2000; $prime = 2000;

**}**

***Exemple 2 : écriture des instructions avec blocs :***

*Structures de contrôle conditionnelles*

if (*...*) **{**

...

**}** elseif (*...*) **{**

...

**}** else **{**

...

**}**

switch (...) **{**

case ... :

...

case ... :

...

default :

...

**}**

*Définition de fonction*

function uneFonction() **{**

...

**}**

*Structures de contrôle itératives*

for (...; ...; ...) **{**

...

**}**

while (...) **{**

...

**}**

do **{**

…  
 **}**while (...);

Intérêts

La présence d’accolades ainsi que l'indentation facilitent la localisation des débuts et fins de blocs et réduit le risque d'erreur logique lors de l'ajout de nouvelles lignes de code.

Maintenabilité : lisibilité.

## Appels de fonctions / méthodes

Description

Les fonctions doivent être appelées sans espace entre le nom de la fonction, la parenthèse ouvrante, et le premier paramètre ; avec un espace entre la virgule et chaque paramètre et aucun espace entre le dernier paramètre, la parenthèse fermante et le point virgule.

Il doit y avoir un espace de chaque côté du signe égal utilisé pour affecter la valeur de retour de la fonction à une variable. Dans le cas d'un bloc d'instructions similaires, des espaces supplémentaires peuvent être ajoutés pour améliorer la lisibilité.

Exemples

<?php

$total = round($total, 2);

?>

<?php

$courte         = abs($courte);

$longueVariable = abs($longueVariable);

?>

Intérêts

Maintenabilité : lisibilité

## Définition de fonctions

Description

Les fonctions définies à usage exclusif d'un script seront définies en début du script.

La déclaration des fonctions respecte l'indentation classique des accolades.

Les arguments possédant des valeurs par défaut vont à la fin de la liste des arguments.

Exemples

<?php

function maFonction($arg1, $arg2 = '') {

    if (condition) {

        statement;

    }

    return $val;

}

?>

# Nommage des identificateurs

Cette convention concerne les éléments suivants du langage :

* les fonctions,
* les paramètres formels de fonctions,
* les constantes,
* les variables globales à un script,
* les variables locales,
* les variables de session.

**Pour l'ensemble de ces éléments, la clarté des identificateurs est conseillée. Le nom attribué aux différents éléments doit être aussi explicite que possible, c'est un gage de compréhension du code.**

## Nommage des fonctions

Description

L'identificateur d’une fonction est un verbe, ou groupe verbal.

Les noms de fonctions ne peuvent contenir que des caractères alphanumériques. Les tirets bas ("\_") ne sont pas permis. Les nombres sont autorisés mais déconseillés.

Les noms de fonctions doivent toujours commencer avec une lettre en minuscule. Quand un nom de fonction est composé de plus d'un seul mot, la première lettre de chaque mot doit être mise en majuscule. C'est ce que l'on appelle communément la "notationCamel".

*Exemples :*

filtrerChaineBD(), verifierInfosConnexion(),estEntier()

Intérêts

Maintenabilité : lisibilité.

## Nommage des constantes

Description

Les constantes doivent être déclarées grâce à la commande **define()** en utilisant un nom réellement significatif. Les constantes peuvent contenir des caractères alphanumériques et des tirets bas. Les nombres sont autorisés.

Les constantes doivent toujours être en majuscules, les mots séparés par des '\_'.

On limitera l'utilisation des constantes littérales (nombre ou chaîne de caractères) dans les traitements.

Exemples

define("TYPE\_USER\_ADMIN", "ADM")

// définit la constante de nom TYPE\_USER\_ADMIN et de valeur ADM

Exceptions

Les constantes numériques -1, 0, 1 peuvent toutefois être utilisées dans le code.

Intérêts

Maintenabilité : lisibilité.

## Nommage des variables et paramètres

Description

L'identificateur d’une variable ou paramètre indique le rôle joué dans le code ; c’est en général un nom, ou groupe nominal. Il faut éviter de faire jouer à une variable plusieurs rôles.

Les noms de variables et paramètres ne peuvent contenir que des caractères alphanumériques. Les tirets bas sont autorisés uniquement pour les membres privés d'une classe. Les nombres sont autorisés mais déconseillés.

Comme les identificateurs de fonctions, les noms de variables et paramètres adoptent la notation Camel.

*Exemples :*

$nomEtud, $login

Intérêts

Maintenabilité : lisibilité.

# Algorithmique

## Fonctions/méthodes

### Modularité

Description

Le codage doit être réalisé en recherchant le plus possible la modularité :

- chaque fonction doit réaliser un et un seul traitement,

- chaque fonction doit être construite de manière à posséder la plus forte cohésion et la plus grande indépendance possible par rapport à son environnement.

Intérêts

Maintenabilité et fiabilité : modularité.

### Nombre de paramètres des fonctions

Description

Les fonctions ne doivent pas comporter un trop grand nombre de paramètres. La limite de 5 à 6 paramètres est recommandée. Tout dépassement de cette limite doit être justifié.

Compléments

Cette règle s'applique, tout spécialement, dans le cadre de la programmation par objets qui permet justement de réduire le nombre de paramètres des fonctions.

Intérêts

Maintenabilité : lisibilité.

## Instructions

### Écriture des instructions d’affectation

Description

Il faut utiliser dès que possible les formes abrégées des instructions d’affectation.

Compléments

*Les instructions d’affectation du type :*

A = A <op> <exp> ;

*peuvent être notées sous leur forme abrégée :*

A <op>= <exp> ;

Exemples

*Écrire au lieu de*

$total \*= 0.90; $total = $total \* 0.90;

### Parenthésage des expressions

Description

Il est recommandé d’utiliser les parenthèses à chaque fois qu’une expression peut prêter à confusion.

Exemples

*Il ne faut pas écrire ...*

if ($nbLignes == 0 && $nbMots == 0)

*...mais plutôt écrire*

if (($nbLignes == 0) && ($nbMots == 0))

Intérêts

L'ajout de parenthèses dans les expressions comportant plusieurs opérateurs permet d’éviter des confusions sur leur priorité.

Maintenabilité : lisibilité.

### Interdiction des instructions imbriquées

Description

Les instructions imbriquées doivent être évitées quand cela est possible. En particulier, les types d’instructions imbriquées suivantes sont à bannir :

* affectations dans les conditions, dans les appels de fonctions et dans les expressions ;
* affectations multiples.

Compléments

Une expression ne doit donc contenir que :

* des variables,
* des constantes,
* des appels de fonctions dont les arguments ne sont pas eux-mêmes des éléments variables.

Exemples

*Éviter les affectations dans les conditions :*

while ($ligne = mysql\_fetch\_assoc($idJeu))  
if ($nb++!= 20)

*Éviter les affectations dans les appels de fonctions :*

uneFonction($nb = rand(10,20), $qte);

*Éviter les affectations dans les expressions :*

$a = ($b **=** **$c--**) + $d;

*Éviter les affectations multiples :*

$a = $b = $c = $i++;

Intérêts

La complexité des expressions peut donner lieu à des erreurs d’interprétation. Par exemple, l'affectation dans une condition peut être lue comme un test d'égalité.

Maintenabilité : lisibilité.

### Limitation de l’utilisation des *break* et *continue* dans les itératives

Description

Utilisation modérée

Les ruptures de séquence break et continue doivent être utilisées avec modération dans les itératives.

Compléments

L’abus de ce type d’instructions peut rendre le code difficile à comprendre. Elles pourront toutefois être utilisées ponctuellement. Dans ce cas, un commentaire devra le signaler.

Intérêts

Limiter les instructions break et continue améliore la structuration du code. Ces instructions (qui sont des "goto" déguisés), lorsqu’elles sont utilisées fréquemment, peuvent en effet dénoter une mauvaise analyse des conditions d’itérations dans certains cas.

### Écriture des *switch*

Description

1. Tout le contenu à l'intérieur de l'instruction "switch" doit être indenté avec 4 espaces. Le contenu sous chaque "case" doit être indenté avec encore 4 espaces supplémentaires.
2. Les structures switch doivent obligatoirement comporter une clause default.
3. Le niveau d’imbrication des switch ne doit pas dépasser 2.
4. Chaque cas ou groupe de cas doit se terminer normalement par une instruction break. Les cas ne se terminant par un saut **break** doivent spécifier un commentaire rappelant que l’exécution se poursuit.
5. L’instruction break est obligatoire à la fin du cas par défaut. Cela est redondant mais protège d’une erreur en cas de rajout d’autres cas par la suite.

Compléments

switch (*choix*) {  
 case *expression1* :  
 i*nstructions*  
 /\* pas de break \*/  
 case *expression2* :  
 case *expression3* :  
 *instructions*  
 break;  
 default :  
 *instructions*;  
 break;  
}

Intérêts

Fiabilité : robustesse, clarté.

### Utilisation de l’opérateur ternaire conditionnel

Description

1. Il faut éviter d’utiliser l’abréviation « ? : » du « if ... else*»*, sauf si les conditions suivantes sont réunies (cf. exemple) :

 la valeur de l'expression conditionnelle est effectivement utilisée (dans un retour ou un appel de  
 fonction, une affectation, etc.),

 les 3 opérandes ne sont pas trop complexes.

1. Si toutefois on utilise cet opérateur, il faut mettre la condition (placée avant le « ? ») entre parenthèses.

Compléments

***Exemple***

*Le cas suivant...*

if ($a > $b) {  
 $maxi = $a;  
} else {  
 $maxi = $b;  
}

*...se prête bien à l’utilisation de l’opérateur conditionnel ternaire*

$maxi = ($a > $b) ? $a : $b;

*En effet, on utilise la valeur de l’expression conditionnelle et les opérandes ne sont pas trop complexes.*

Intérêts

Maintenabilité : lisibilité.

# Gestion des formulaires HTML

## Nommage des formulaires et des champs de formulaires

Description

Les noms des éléments HTML débuteront par un préfixe rappelant leur type.

Les préfixes retenus concernent les formulaires et les champs contenus dans les formulaires.

|  |  |
| --- | --- |
| **Type d'élément** | **Préfixe** |
| Formulaire | frm |
| Zone de texte mono\_ligne (text, password) / multi\_lignes | txt |
| Champ caché | hd |
| Bouton d'option (bouton radio) | opt |
| Case à cocher | chk |
| Zone de liste | lst |
| Bouton de type reset | br |
| Bouton de type button | bt |
| Bouton de type submit | cmd |

## Méthodes de soumission des formulaires

Les méthodes de soumission d'un formulaire sont au nombre de 2 : GET et POST. La première véhicule les noms et valeurs des champs dans l'URL de la requête HTTP, la seconde dans le corps de la requête HTTP.

Description

La méthode POST est à préférer pour des raisons de taille de données et de confidentialité. A noter que la confidentialité se résume ici à ne pas voir apparaître les noms et valeurs de champs dans la zone d'adresse du navigateur : les données sont, dans les 2 cas, transmises en clair sur le réseau dans le cas où le protocole applicatif utilisé reste HTTP.

Le choix de la méthode GET peut cependant se justifier s'il est souhaitable de pouvoir conserver les différentes soumissions d'un formulaire en favoris.

## Variables superglobales $\_GET, $\_POST

Les valeurs saisies dans un formulaire sont mises à disposition des scripts PHP dans les tableaux associatifs superglobaux $\_GET et $\_POST. Le tableau $\_GET contient également les valeurs transmises via la constitution d'un lien.

Description

Au cours de la mise au point des scripts, il est recommandé d'appeler la fonction var\_dump sur ces tableaux $\_GET et $\_POST afin d'apprécier réellement les données (nom et valeur) reçues.

De plus, pour éviter de se référer dans tout le script soit au tableau $\_GET, soit au tableau $\_POST, tout script PHP affectera initialement les éléments des deux tableaux dans des variables. Ceci permettra également de migrer facilement d'une méthode de soumission POST vers GET ou vice-versa.

On pourra définir des fonctions spécialisées afin de récupérer les valeurs des éléments à partir d'un tableau ou d'un autre, en prévoyant des valeurs par défaut en cas d'inexistence d'un élément.

Exemples

// acquisition des données reçues par la méthode post

$mois = $\_POST["lstMois"];

$etape = $\_POST["etape"];

// à commenter après débogage

var\_dump($\_GET, $\_POST);

1. <http://pear.php.net/manual/fr/standards.php> [↑](#footnote-ref-1)
2. <http://framework.zend.com/manual/fr/coding-standard.html> [↑](#footnote-ref-2)