Pre-1.6.1.0 PHP Coding Standards

Starting with version 1.6.1.0, the PrestaShop Core codebase has switched to the PSR-1 coding standard and PSR-2 coding style guide. See the reasons why on the announcement article on the Build PrestaShop devblog.

This page is kept up for reference's sake.

If you are developping new modules for PrestaShop 1.6.1.0+, or contributing to the PrestaShop project, **you are not supposed to use these guidelines anymore**. Please follow the current coding standards. Thank you.

Table of contents

PHP

Variable names

Just like class, method and function names, variable names should be written in English so as to be readable to as many people as possible.

Use lowercase letters, and separate words using underscores. Do not ever use CamelCase for variable names, only for method/function and object/class names.

1. Corresponding to data from databases: `$my_var`.
2. Corresponding to algorithm: `$my_var`.
3. The visibility of a member variable does not affect its name: `private $my_var`.

Assignments

1. There should be a space between variable and operators:

```
$my_var = 17;
$a = $b;
```

Operators

1. "+", "-", "*", "/", "=" and any combination of them (e.g. "/=") need a space between their left and right members.

   ```
   $a + 17;
   $result = $b / 2;
   $i += 34;
   ```

2. "." does not have a space between its left and right members.

   ```
   echo $a.$b;
   $c = $d.$this->foo();
   ```

   Recommendation
   ⚠
   For performance reasons, please do not overuse concatenation.
3. ".=" needs a space between its left and right members.

   ```
   $a .= 'Debug';
   ```

4. When testing a boolean variable, do not use a comparison operator, but directly use the value itself, or the value prefixed with an exclamation mark:

   ```
   // do not use this
   if ($var == true)
   // ...nor this
   if ($var == false)

   // use this
   if ($var)
   // ...or this
   if (!$var)
   ```

Statements

1. `if`, `elseif`, `while`, `for`: need a space between the `if` keyword and the parentheses `()`.

   ```
   if (<condition>)

   while (<condition>)
   ```

2. When a combination of `if` and `else` is used and both can return a value, the `else` statement has to be omitted.

   ```
   if (<condition>)
           return false;
   return true;
   ```

   Recommendation
   ⚠
   We recommend you to use only one `return` statement per method/function.

3. When a method/function returns a boolean and the current method/function's returned value depends on it, the `if` statement has to be avoided.

```
public aFirstMethod()
{
        return $this->aSecondMethod();
}
```

4. Tests must be grouped by entity.

```
if ($price && !empty($price))
        ...
if (!Validate::$myObject || $myObject->id === NULL)
        ...
```

Visibility

1. The visibility must be defined every time, even when it is a public method.
2. The order of the method properties should be: `visibility static function functionName()`.

```
private static function foo()
```

Method / Function names

1. Method and function names always use CamelCase: begin with a lowercase character and each following words must begin with an uppercase character.

```
public function myExampleMethodWithALotOfWordsInItsName()
```

2. Braces introducing method code have to be proceeded by a carriage return.

```
public function myMethod($arg1, $arg2)
{
        ...
}
```

3. Method and function names must be explicit, so function names such as `b()` or `ef()`are completely forbidden.

   Exceptions

   The only exceptions are the translation function (called `l()`) and the debug functions (named `p()` and `d()`).

Enumeration

Commas have to be followed (and not preceded) by a space.

```
protected function myProtectedMethod($arg1, $arg2, $arg3 = null)
```

Objects / Classes

1. Object name must be singular.

```
class Customer
```

2. Class name must follow the CamelCase practice, except that the first letter is uppercase.

```
class MyBeautifulClass
```

Constants

1. Constant names must be written in uppercase, except for "true", "false" and "null" which must be lowercase: `ENT_NOQUOTE`, `true`.
2. Constant names have to be prefixed with "`PS_`" inside the core and module.

```
define('PS_DEBUG', 1);
define('PS_MODULE_NAME_DEBUG', 1);
```

3. Constant names should only use alphabetical characters and "_".

Keywords

All keywords have to be lowercase: `as, case, if, echo, null`.

Configuration variables

Configuration variables follow the same rules as defined above.

Strings

Strings have to be surrounded by simple quotes, never double ones.

```
echo 'Debug';
$myObj->name = 'Hello '.$name;
```

Comments

1. Inside functions and methods, only the "`//`" comment tag is allowed.
2. After the "`//`" comment marker, a space is required:

```
// My great comment
```

3. The "`//`" comment marker is tolerated at the end of a code line.

```
$a = 17 + 23; // A comment inside my example function
```

4. Outside of functions and methods, only the "`/*`" and "`*/`" comment markers are allowed.

```
/* This method is required for compatibility issues */
public function foo()
{
        // Some code explanation right here
        ...
}
```

5. A phpDoc comment block is required before the declaration of the method.

```
/**
 * Return field value if possible (both classical and multilingual fields)
 *
 * Case 1: Return value if present in $_POST / $_GET
 * Case 2: Return object value
 *
 * @param object $obj Object
 * @param string $key Field name
 * @param integer $id_lang Language id (optional)
 * @return string
 */
protected function getFieldValue($obj, $key, $id_lang = NULL)
```

For more informations

For more information about the PHP Doc syntax: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.pkg.html.

Return values

1. The `return` statement does not need brackets, except when it deals with a composed expression.

```
return $result;
return ($a + $b);
return (a() - b());
return true;
```

2. The `return` statement can be used to break out of a function.

```
return;
```

Call

Performing a function call preceded by a "@" is forbidden, but beware of function/method call with login/password or path arguments.

```
myfunction();

// In the following example, we put a @ for security reasons
@mysql_connect(...);
```

Tags

1. There must be an empty line after the PHP opening tag.

```
<?php

require_once('my_file.inc.php');
```

2. The PHP closing tag is forbidden at the end of a file.

Indentation

1. The tabulation character ("`\t`") is the only indentation character allowed.
2. Each indentation level must be represented by a single tabulation character.

```
function foo($a)
{
        if ($a == null)
                return false;
        ...
}
```

Array

1. The `array` keyword must not be followed by a space.

```
array(17, 23, 42);
```

2. When too much data is inside an array, the indentation has to be as follows:

```
$a = array(
        36 => $b,
        $c => 'foo',
        $d => array(17, 23, 42),
        $e => array(
                0 => 'zero',
                1 => $one
        )
);
```

Block

Braces are prohibited when they only define one instruction or a combination of statements.

```
if (!$result)
    return false;

for ($i = 0; $i < 17; $i++)
    if ($myArray[$i] == $value)
    {
        $result[] = $myArray[$i];
        return $result;
    }
    else
        $failed++;
```

Security

1. All users' data (data entered by users) has to be cast.

```
$data = Tools::getValue('name');

$myObject->street_number = (int)Tools::getValue('street_number');
```

`getValue()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

2. All method/function's parameters must be typed (when `Array` or `Object`) when received.

```
public myMethod(Array $var1, $var2, Object $var3)
```

3. For all other parameters, they have to be cast each time they are used, except when they are sent to other methods/functions.

```
protected myProtectedMethod($id, $text, $price)
{
        $this->id = (int)$id;
        $this->price = (float)$price;
        $this->callMethod($id, $price);
}
```

Limitations

1. Source code lines are limited to 150 characters wide.
2. Functions and methods lines are limited to 80 characters. Functions must have a good reason to have an overly long name: keep it to the essential!

Other

1. It is forbidden to use a ternary into another ternary, such as `echo ((true ? 'true' : false) ? 't' : 'f');`.
2. We recommend the use of `&&` and `||` into your conditions instead of `AND` and `OR`: `echo ('X' == 0 && 'X' == true)`.
3. Please refrain from using reference parameters, such as:

```
function is_ref_to(&$a, &$b) { ... }
```

SQL

Table names

1. Table names must begin with the PrestaShop "`_DB_PREFIX_`" prefix.

```
... FROM `'. _DB_PREFIX_.'customer` ...
```

2. Table names must have the same name as the object they reflect: "`ps_cart`".
3. Table names have to stay singular: "`ps_order`".
4. Language data have to be stored in a table named exactly like the object's table, and with the "`_lang`" suffix: "`ps_product_lang`".

SQL query

1. Keywords must be written in uppercase.

```
SELECT `firstname`
FROM `'._DB_PREFIX_.'customer`
```

2. Back quotes ("`") must be used around SQL field names and table names.

```
SELECT p.`foo`, c.`bar`
FROM `'._DB_PREFIX_.'product` p, `'._DB_PREFIX_.'customer` c
```

3. Table aliases have to be named by taking the first letter of each word, and must be lowercase.

```
SELECT p.`id_product`, pl.`name`
FROM `'._DB_PREFIX_.'product` p
NATURAL JOIN `'._DB_PREFIX_.'product_lang` pl
```

4. When conflicts between table aliases occur, the second character has to be also used in the name.

```
SELECT ca.`id_product`, cu.`firstname`
FROM `'._DB_PREFIX_.'cart` ca, `'._DB_PREFIX_.'customer` cu
```

5. A new line has to be created for each clause.

```
$query = 'SELECT pl.`name`
FROM `'._DB_PREFIX_.'product_lang` pl
WHERE pl.`id_product` = 17';
```

6. It is forbidden to make a `JOIN` in a `WHERE` clause.

Installing the code validator (PHP CodeSniffer)

This is a brief tutorial on how to install a code validator on your PC and use it to validate your files. The code validator uses PHP CodeSniffer, which is a PEAR package (http://pear.php.net/package/PHP_CodeSniffer/). The PrestaShop code standard was created specifically for CodeSniffer, using many rules taken from existing standards, with added customized rules in order to better fit our project.

You can download the PrestaShop code standard using Git: https://github.com/PrestaShop/PrestaShop-norm-validator (you must perform this step before going any further with this tutorial).

In order for it to be recognized as a basic standard, it must be placed in the CodeSniffer's `/Standards` folder

PhpStorm integration

If you use PhpStorm (http://www.jetbrains.com/phpstorm/), follow these steps:

1. Go to Settings -> Inspection -> PHP -> PHP Code Sniffer.
2. Set the path to the `phpcs` executable.
3. Set the coding standard as "PrestaShop" (which is only available if you did put in CodeSniffer's `/Standards` folder).

Integration to vim

Several plugins are available online. For instance, you can use this one: https://github.com/bpearson/vim-phpcs/blob/master/plugin/phpcs.vim
Put in your `~/.vim/plugin` folder.

You can add two shortcuts (for instance, F9 to display everything and Ctrl+F9 to hide warnings) in your `.vimrc` file in normal and insert mode:

```
nmap <C-F9>:CodeSniffErrorOnly<CR>
imap <C-F9> <Esc>:CodeSniffErrorOnly<CR>
nmap <F9>:CodeSniff<CR>
imap <F9> <Esc>:CodeSniff<CR>a
```

Command line (Linux)

You do not have to use PhpStorm to use PHP CodeSniffer, you can also install it so that it can be called from the command line.

1. Install PEAR: http://pear.php.net/
   `$> apt-get install php-pear`
2. Install PHP CodeSniffer in PEAR: http://pear.php.net/package/PHP_CodeSniffer
   `$> pear install PHP_CodeSniffer`

3. Add the PrestaShop standard that you downloaded from SVN earlier, and place it in PHP CodeSniffer's "Standards" folder.
   `$> git clone https://github.com/PrestaShop/PrestaShop-norm-validator /usr/share/php/PHP/CodeSniffer/Standards/Prestashop`
4. Set the Prestashop standard as the default one
   `$> phpcs --config-set default_standard Prestashop`

The various options for this command are well explained in its documentation. For now, here is the easy way to launch it:

```
$> phpcs --standard=/path/to/norm/Prestashop /folder/or/fileToCheck
```

In order to only display errors, not warnings:

```
$> phpcs --standard=/path/to/norm/Prestashop --warning-severity=99 /folder/or/fileToCheck
```

If you have already manually installed PHP CodeSniffer, the program should be in PEAR's `/scripts` folder.

Windows users: although the `phpcs.bat` file should be in that `/scripts` folder, you might have to edit it in order for it to work properly (replace the paths with yours):

```
path/to/php.exe -d auto_apprend_file="" -d auto_prepend_file -d include_path="path/to/PEAR/" path/to/pear/scripts/phpcs
```