

## Goal

*“Any collaborative project needs consistency and stability to stay strong.”*

Good coding standards are important in any development project, particularly when multiple developers are working on the same project. Having coding standards helps to ensure that the code is of high quality, has fewer bugs, and is easily maintained.

Abstract goals we strive for:

- extreme simplicity
- tool friendliness, such as use of method signatures, constants, and patterns that support IDE tools and auto-completion of method, class, and constant names.

When considering the goals above, each situation requires an examination of the circumstances and balancing of various trade-offs.

This coding guidelines aims to provide a goal to all PHPUGPH code to strive to. All code must adhere to these standard as closely as possible.

## Overview

### Scope

- PHP File Formatting
- Naming Conventions
- Coding Style
- Inline Documentation
- Errors and Exceptions

## PHP File Formatting

### General

- All PHP code file should use the .php extension
- All php tags must be 'full' tags like `<?php ?>` ... not 'short' tags like `<? ?>`.
- Indentation - Use an indent of 4 spaces with no tab characters. Editors should be configured to treat tabs as spaces in order to prevent injection of tab characters into the source code.
- Maximum line length - The target line length is 80 characters; i.e., developers should aim keep code as close to the 80-column boundary as is practical. However, longer lines are acceptable. The maximum length of any line of PHP code is 120 characters.

## Naming Convention

### Classes

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are

discouraged. Underscores are only permitted in place of the path separator. For example, the filename

Code:

```
<?php
class Home_Controller extends Controller
{
    public function foo($required, $optional = null)
    {
        parent::Controller
    }
}
?>
```

## Filenames

For all other files, only alphanumeric characters, underscores, and the dash character ("-") are permitted. Spaces are prohibited.

## Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in function names but are discouraged.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called the "camelCaps" method. Verbosity is encouraged. Function names should be as illustrative as is practical to enhance understanding.

These are examples of acceptable names for functions:

- filterInput()
- getElementById()
- widgetFactory

## Optional Parameters

Use "null" as the default value instead of "false", for situations like this:

Code:

```
<?php
public function foo($required, $optional = null)
?>
```

when \$optional does not have or need a particular default value.

However, if an optional parameter is boolean, and its logical default value should be true, or false, then using true or false is acceptable.

## Variables

Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable names but are discouraged.

For class member variables that are declared with the private or protected construct, the first character of the variable name must be a single underscore. This is the only acceptable usage of an underscore in a variable name. Member variables declared as "public" may never start with an underscore. For example:  
Code:

```
<?php
class Foo_Bar
{
    protected $_bar;
}
?>
```

## Constants

Constants may contain both alphanumeric characters and the underscore. Numbers are permitted in constant names.

Constant names must always have all letters capitalized.

To enhance readability, words in constant names must be separated by underscore characters. For example, "EMBED\_SUPPRESS\_EMBED\_EXCEPTION" is permitted but "EMBED\_SUPPRESSEMBEDEXCEPTION" is not.

Constants must be defined as class members by using the "const" construct. Defining constants in the global scope with "define" is permitted but discouraged.

## Booleans and the NULL Value

Use lowercase for both boolean values and the "null" value.

## Coding Style

### PHP Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags.

```
Quote
<?php
```

```
?>
```

Short tags are never allowed.

# Strings

## String Literals

When a string is literal (contains no variable substitutions), the apostrophe or "single quote" must always be used to demarcate the string:

Code:

```
<?php
$string = 'Example string';
?>
```

## String Literals Containing Apostrophes

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or "double quotes". This is especially encouraged for SQL statements:

Code:

```
<?php
$sql = "SELECT `id`, `name` from `people` WHERE `name`='Fred' OR `name`='Susan'";
?>
```

The above syntax is preferred over escaping apostrophes.

## Variable Substitution

Variable substitution is permitted using either of these two forms:

Code:

```
<?php
$greeting = "Hello $name, welcome back!";

$greeting = "Hello {$name}, welcome back!";
?>
```

For consistency, this form is not permitted:

Code:

```
<?php
$greeting = "Hello ${name}, welcome back!";
?>
```

## String Concatenation

Strings may be concatenated using the "." operator. A space must always be added before and after the "." operator to improve readability:

Code:

```
<?php
$company = 'PHP Philippine' . 'User Group';
?>
```

When concatenating strings with the "." operator, it is permitted to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with whitespace such that the "." operator is aligned under the "=" operator:

Code:

```
<?php
$sqlString = "
    SELECT
        myField1
        ,myField2
        ,myField3
        ,myField4
    FROM
        myTable
    WHERE
        myField1 = 'myFilter1'
";
$this->db->query($sqlString);
?>
```

## Arrays

### Numerically Indexed Arrays

Negative numbers are not permitted as array indices. An indexed array may be started with any non-negative number, however this is discouraged and it is recommended that all arrays have a base index of 0. When declaring indexed arrays with the array construct, a trailing space must be added after each comma delimiter to improve readability:

Code:

```
<?php
$myarrays = array('one_sentence', 'two_sentences', 'three_sentences', 'four_sentences',
'five_sentences', 'six_sentences', 'seven_sentences', 'eight_sentences', 'nine_sentences
');
?>
```

It is also permitted to declare multi-line indexed arrays using the array construct. In this case, each successive line must be padded with spaces such that beginning of each line aligns as shown below:

Code:

```
<?php
$myarrays = array(
    'one_sentence'
    , 'two_sentences'
    , 'three_sentences'
    , 'four_sentences'
    , 'five_sentences'
    , 'six_sentences'
```

```
        , 'seven_sentences'  
        , 'eight_sentences'  
        , 'nine_sentences'  
    );  
?>
```

### Associative Arrays

When declaring associative arrays with the array construct, it is encouraged to break the statement into multiple lines. In this case, each successive line must be padded with whitespace such that both the keys and the values are aligned:

Code:

```
<?php  
$sampleArray = array('firstKey' => 'firstValue',  
                    'secondKey' => 'secondValue');  
?>
```

## Classes

### Class Declarations

Classes must be named by following the naming conventions. The brace is always written on the line underneath the class name ("one true brace" form). Every class must have a documentation block that conforms to the [phpDocumentor](#) standard. Any code within a class must be indented the standard indent of four spaces. Only one class is permitted per PHP file.

Placing additional code in a class file is permitted but discouraged. In these files, two blank lines must separate the class from any additional PHP code in the file.

This is an example of an acceptable class declaration:

Code:

```
<?php  
/**  
 * Class Docblock Here  
 */  
class Class_Name  
{  
    // entire content of class  
    // must be indented four spaces  
}  
?>
```

### Class Member Variables

Member variables must be named by following the variable naming conventions. Any variables declared in a class must be listed at the top of the class, prior to declaring any functions.

The var construct is not permitted. Member variables always declare their visibility by using one of the private, protected, or public constructs. Accessing member variables directly by making them public is permitted but discouraged in favor of accessor methods having the set and get prefixes.

## Functions and Methods

### Function and Method Declaration

Functions and class methods must be named by following the naming conventions. Methods must always declare their visibility by using one of the private, protected, or public constructs.

Following the more common usage in the PHP developer community, static methods should declare their visibility first:

Code:

```
<?php
public    static foo() { ... }
private  static bar() { ... }
protected static goo() { ... }
?>
```

As for classes, the opening brace for a function or method is always written on the line underneath the function or method name ("one true brace" form). There is no space between the function or method name and the opening parenthesis for the arguments.

This is an example of acceptable class method declarations:

Code:

```
<?php
/**
 * Class Docblock Here
 */
class Foo_Bar
{
    /**
     * Method Docblock Here
     */
    public function sampleMethod($a)
    {
        // entire content of function
        // must be indented four spaces
    }

    /**
```

```

    * Method Docblock Here
    */
protected function _anotherMethod()
{
    // ...
}
}
?>

```

The return value must not be enclosed in parentheses. This can hinder readability and can also break code if a function or method is later changed to return by reference.

Code:

```

<?php
function foo()
{
    // WRONG
    return($this->bar);

    // RIGHT
    return $this->bar;
}
?>

```

The use of [type hinting](#) is encouraged where possible with respect to the component design. For example, Code:

```

<?php
class Component_Class
{
    public function foo(SomeInterface $object)
    {}

    public function bar(array $options)
    {}
}
?>

```

Where possible, try to keep your use of exceptions vs. type hinting consistent, and not mix both approaches at the same time in the same method for validating argument types. However, before PHP 5.2, "Failing to satisfy the type hint results in a fatal error," and might fail to satisfy other coding standards involving the use of throwing exceptions. Beginning with PHP 5.2, failing to satisfy the type hint results

in an `E_RECOVERABLE_ERROR`, requiring developers to deal with these from within a custom error handler, instead of using a `try..catch` block.

## Function and Method Usage

Function arguments are separated by a single trailing space after the comma delimiter. This is an example of an acceptable function call for a function that takes three arguments:

Code:

```
<?php
threeArguments(1, 2, 3);
?>
```

Call-time pass by-reference is prohibited. Arguments to be passed by reference must be defined in the function declaration.

For functions whose arguments permit arrays, the function call may include the "array" construct and can be split into multiple lines to improve readability. In these cases, the standards for writing arrays still apply:

Code:

```
<?php
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Zend', 'Studio',
                    $a, $b, $c,
                    56.44, $d, 500), 2, 3);
?>
```

## Control Statements

### If / Else / Elseif

Control statements based on the "if", "else", and "elseif" constructs must have a single space before the opening parenthesis of the conditional, and a single space between the closing parenthesis and opening brace.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping of larger conditionals.

The opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented four spaces.

Code:

```
<?php
if ($a != 2) {
    $a = 2;
}
?>
```

For "if" statements that include "elseif" or "else", the formatting must be as in these examples:  
Code:

```
<?php
if ($a != 2) {
    $a = 2;
} else {
    $a = 7;
}
```

```
if ($a != 2) {
    $a = 2;
} else if ($a == 3) {
    $a = 4;
} else {
    $a = 7;
}
?>
```

PHP allows for these statements to be written without braces in some circumstances. The coding standard makes no differentiation and all "if", "elseif", or "else" statements must use braces.

Use of the "elseif" construct is permitted but highly discouraged in favor of the "else if" combination.

## Switch

Control statements written with the "switch" construct must have a single space before the opening parenthesis of the conditional statement, and also a single space between the closing parenthesis and the opening brace.

All content within the "switch" statement must be indented four spaces. Content under each "case" statement must be indented an additional four spaces.

Code:

```
<?php
switch ($numPeople) {
    case 1:
        break;

    case 2:
        break;

    default:
```

```
        break;
    }
?>
```

The construct "default" may never be omitted from a "switch" statement.

Quote from: Note

Please note

It is sometimes useful to write a "case" statement which falls through to the next case by not including a "break" or "return". To distinguish these cases from bugs, such "case" statements must contain the comment "// break intentionally omitted".

## Inline Documentation

### Documentation Format

All documentation blocks ("docblocks") must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit <http://phpdoc.org>.

All source code file written for the PHPUGPH Project or that operates with the framework must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class.

Below are examples of such docblocks.

To avoid losing track of @todo}s in the source code, either use only an issue in our issue tracker, or include the issue identifier (e.g., ProjectCodeName-123) in the {{@todo. This allows {{@todo}}s to be tracked and monitored the same as any other issue. It also makes {{@todo}}s more visible to the community, and helps find volunteers.

The sharp, '#', character should not be used to start comments.

## Files

Every file that contains PHP code must have a header block at the top of the file that contains these phpDocumentor tags at a minimum:

Code:

```
<?php
/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * LICENSE: Some license information
 *
 * @copyright 2006 Zend Technologies
```

```
* @license      http://www.zend.com/license/3_0.txt    PHP License 3.0
* @version      $Id$
* @link         http://dev.zend.com/package/PackageName
* @since       File available since Release 1.2.0
*/
?>
```

## Classes

Every class must have a docblock that contains these phpDocumentor tags at a minimum:

Code:

```
<?php
/**
 * Short description for class
 *
 * Long description for class (if any)...
 *
 * @copyright    2006 Zend Technologies
 * @license      http://www.zend.com/license/3_0.txt    PHP License 3.0
 * @version      Release: @package_version@
 * @link         http://dev.zend.com/package/PackageName
 * @since       Class available since Release 1.2.0
 */
?>
```

## Functions

Every function, including object methods, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values
- If a function/method may throw an exception, use "@throws"

Quote from: Please Note

Please note

It is not necessary to use the "@access" tag because the access level is already known from the "public", "private", or "protected" construct used to declare the function.

Code:

```
<?php
```

```

/**
 * Does something interesting
 *
 * @param Place $where Where something interesting takes place
 * @param integer $repeat How many times something interesting should happen
 * @throws Some_Exception_Class If something interesting cannot happen
 * @return Status
 */
public function doSomethingInteresting(Place $where, $repeat = 1)
{
    // implementation...
}
?>

```

## Require / Include

If a component uses another component, then the using component is responsible for loading the other component. If the use is conditional, then the loading should also be conditional.

If the file(s) for the other component should always load successfully, regardless of input, then use PHP's `require_once` statement.

If the file(s) loaded are variable dependent (e.g., `require_once $userSelectedLogger`), then use `Zend::loadClass()`, or wrap the `require_once` statement to throw an helpful exception on failure.

The `include`, `include_once`, `require`, and `require_once` statements should not use parentheses.

## Creating Libraries

from [http://www.codeigniter.com/user\\_guide/general/creating\\_libraries.html](http://www.codeigniter.com/user_guide/general/creating_libraries.html)

Classes should have this basic prototype.

Code.

```

<?php if (!defined('BASEPATH')) exit('No direct script access allowed');

class Someclass {

    public function some_function()
    {
    }

}

?>

```