Search Wiki Documents

# PHP Coding Standards for VG

New Document Edit Document View History

Document Index · **Awesome Documentation** » **Coding Standards** » **PHP Coding Standards for VG**

Last updated 49 days ago by andrer.

## Overview

### Scope

This document provides the coding standards and guidelines for developers and teams working for or with VG Multimedia. The subjects covered are:

- PHP file formatting
- Naming conventions
- Coding style
- Inline documentation
- Errors and exceptions

### Goals

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.

## PHP file formatting

### General

For files that contain only PHP code, the closing tag ("?>") is to be omitted. It is not required by PHP, and omitting it prevents trailing whitespace from being accidentally injected into the output.

### 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.

### Line termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character, no carriage returns (CR). Lines should not contain trailing spaces. In order to facilitate this convention, most editors can be configured to strip trailing spaces, such as upon a

Table of Contents

save operation.

## Line length

Try to keep line length below 100 characters.

## Naming Conventions

All files, classes, functions, variables, parameters, database names, table names etc should be English, and as generic terms as possible. Exceptions are allowed where the name is intended to reflect very specific Norwegian names or concepts, e.g. "skattelister", "rampelys" etc.

Use plural variants where applicable. Try to use verbs followed by the name of the entity.

Examples:

- files: editUser.php, listUsers.php
- DB Databases: tipAuthors
- DB Tables: authors, tips, tipDetails
- DB columns: authorId(not only id), authorName(not only name), createdBy, updated etc.

### Classes

VG Multimedia employs a class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory is the *VGF/* directory, under which all classes are stored hierarchically.

Class names may only contain alphanumeric characters. Underscores are only permitted in place of the path separator. For example, the filename *VGF/Article/List.php* must map to the class name *VG_Article_List*.

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. If a class name contains a known word with all capital letters, it's adviced to use all caps in the class name; e.g. *VGF_Cache_APC* instead of *VGF_Cache_Apc*.

### Interfaces

Interface classes must follow the same conventions as other classes (see above), but must end with *_Interface*. Because of this all files that contain interfaces will be named Interface.php.

### Abstract classes

Abstract classes must follow the same conventions as other classes (see above), but must end with *_Abstract*. Because of this all files that contain abstract classes will be named Abstract.php.

### Filenames

For all other files, only alphanumeric characters are permitted. Use camelCase. Spaces are prohibited.

Any file that contains any PHP code must end with the extension *.php* with the exception of views in Zend Framework, that defaults to *.phtml*. All other scripts must use *.php*.

## Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted.

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 "camelCase" 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()
- generateLink()

In object-oriented programming, access functions for object members should be prefixed with either *get* or *set*. When using design patterns, such as the Singleton or Factory patterns, the name of the method should contain the pattern name where practical to make the pattern more readily recognizable. With boolean attributes, one can create a method that both sets and gets the value. Example:

```php
<?php
class VGF_RemoteContent {
    /**
     * Timeout for the cache
     *
     * @var int
     */
    protected $timeout = 2;

    /**
     * Wether or not to use the cache
     *
     * @var boolean
     */
    protected $useCache = true;

    /**
     * Set the timeout attribute
     *
     * @param int $timeout Timeout in seconds
     * @return VGF_RemoteContent
     */
    public function setTimeout($timeout) {
        $this->timeout = (int) $timeout;

        return $this;
    }

    /**
     * Get the timeout
     *
     * @return int
     */
    public function getTimeout() {
        return $this->timeout;
    }

    /**
     * Set or get the useCache attribute
     *
     * @param boolean $flag Set this to true or false to set the value
     * @return VGF_RemoteContent|boolean When called with no argument the method will return the curren
     */
    public function useCache($flag = null) {
        if ($flag !== null) {
            $this->useCache = (bool) $flag;

            return $this;
        }

        return $this->useCache;
    }
}
```

Functions in the global scope, or "floating functions," are permitted but strongly discouraged. It is recommended that these functions be wrapped in a class and declared static. If the function has a generic appeal, the VG_Functions class in the VG Framework may be a suitable place.

Functions or variables declared with a *static* scope in a class generally should not be *private*, but protected instead. Use *final* if the function should not be extended. The usage of static method is not advised as it makes unit testing much more difficult. The usage of final is also not advised.

## Optional parameters

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

```
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.

Like function names, variable names must always start with a lowercase letter and follow the "camelCase" capitalization convention.

Verbosity is encouraged. Variable names should always be as verbose as practical. Terse variable names such as $i and $n are discouraged for anything other than the smallest loop contexts. If a loop contains more than 20 lines of code, variables for such indices or counters need to have more descriptive names.

### Constants

Constants may contain both alphanumeric characters and the underscore, but must always start with a letter. 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

Unlike PHP's documentation, VG Multimedia uses 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 (although you should see the note about the closing PHP tag above):

```
<?php
// code
?>
```

Short tags are never allowed.

### Type casting

Casting values should be done in this manner:

```
<?php
$userId = (int) $user['userId'];
```

There should never be any spaces inside the parentheses, and it should always be followed by a space.

### Strings

#### String literals

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

```
<?php
$a = 'This is a string';
```

### String literals containing apostophes

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:

```
<?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 this form:

```
<?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:

```
<?php
$name = 'First part' . 'Second part';
```

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:

```
<?php
$sql = "SELECT `id`, `name`, FROM `people` "
     . "WHERE `name` = 'susan' "
     . "ORDER BY `id` ASC";
```

## 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:

```
<?php
$sampleArray = array(1, 2, 3, 'foo', 'bar');
```

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:

```
<?php
$sampleArray = array(1, 2, 3,
                     $a, $b, $c,
                     'foo', 'bar');
```

### 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:

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

Single quotes should be used for array keys as shown in the example above.

## Classes

### Class declarations

Classes must be named by following the naming conventions.

- The brace is always written on the line behind class name.
- 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:

```
<?php
/**
 * Class docblock
 */
class VG_Class {
    // 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.

Member variables should always be initiated with a proper default value. Use "null" if a default value cannot be used/trusted, or where lazy-loading of attributes should be implemented.

Examples:

```
<?php
class SomeClass {
    public $articleAuthors = null;
    private $expireCache = 600;
    protected $active = false;
}
```

### Constructors

All constructor methods in a class must be called __construct(). PHP allows constructors to have the same name as the class, but this is not allowed.

```
<?php
// Correct
class Foo {
    public function __construct() {
        // Magic
    }
}

// Incorrect
class Bar {
    public function Bar() {
        // Magic
    }
}
```

## Functions, closures 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.

Static methods and variables must be declared as static prior to the visibility:

```
<?php
class SomeClass {
    static protected $foo = 'value';
    static public function foo() { ... }
    static private function bar() { ... }
    static protected function foobar() { ... }
}
```

As for classes, the opening brace for a function or method is always written on the line behind the function or method name. 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:

```
<?php
/**
 * Class docblock
 */
class VG_Class {
    /**
     * Function docblock
     */
    public function sampleFunction($a) {
        // entire content of function
        // must be indented four spaces
    }

    /**
     * Function docblock
     */
    protected function anotherFunction() {
        // ...
    }
}
```

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.

```
<?php
class SomeClass {
    function foo() {
        // Incorrect
        return ($this->bar);

        // Correct
        return $this->bar;
    }
}
```

The use of type hinting is encouraged where possible with respect to the component design. For example,

```
<?php
```

```
class VG_Component {
    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.

## Closures

As of php-5.3 we can use closures (anonymous functions). They are to use the same syntax as regular functions regarding spacing and the argument listing. No closures shall be written in one line no matter how short they are.

```
<?php
// Incorrect
$var = preg_replace_callback('/-([a-z])/', function ($match) { return strtoupper($match[1]); }, 'hello-w

// Correct
$var = preg_replace_callback('/-([a-z])/', function ($match) {
    return strtoupper($match[1]);
}, 'hello-world');
```

We use the same syntax when assigning a closure to a variable:

```
<?php
$func = function ($a, $b) {
    return $a + $b;
};
```

Remember to place a single space between the function keyword and the opening parenthesis.

## 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:

```
<?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:

```
<?php
threeArguments(array(1, 2, 3), 2, 3);
threeArguments(array(1, 2, 3, 'foo', 'bar',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);
```

## Control statements

### if / else / else if

Control statements based on the "if", "else", and "else if" 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.

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

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

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

if ($a < 2) {
    $a = 2;
} else if ($a < 10) {
    $a = 7;
} else {
    $a = 8;
}
```

PHP allows for these statements to be written without braces in some circumstances. The coding standard makes no differentiation and all "if", "else if", or "else" statements must use braces. PHP also allows the "elseif" construct but this should not be used. Use "else if" (with space) instead.

## 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.

```php
<?php
switch ($numPeople) {
    case 1:
        break;
    case 2:
        break;
    default:
        break;
}
```

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

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*".

## for

When using a for loop the three parts must be separated by spaces:

```php
<?php
for ($articleCounter = 0; $articleCounter < $numArticles; $articleCounter++) {
    // display article
}
```

There must also be a single whitespace before the opening parenthesis and one between the closing parenthesis and the opening brace.

## foreach

A foreach statement follows the same rules as all other control statments, a single whitespace must be placed before the opening parenthesis, and one between the closing parenthesis and the opening brace. Spaces should also be included between the "rocket" operator (=>) and neighboring variables.

```php
<?php
foreach ($var as $key => $value) {
    // do something
}
```

## try/catch

When wrapping code that can throw exceptions in try/catch blocks, use the following syntax:

```
<?php
try {
    someFunctionThatMightThrowAnException();
} catch (SomeException $e) {
    // Do something clever with $e
}
```

## 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.

One-line comments should use and *NOT the sharp ('#') character. All comments must start with a space for improved readability*.

```
<?php
//this is incorrect
doSomething();

// This is correct.
doSomething();
```

### 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:

```
<?php
/**
 * Short description for file
 *
 * Long description for file (if any)
 */
```

### Classes

Every class must have a docblock that contains the same phpDocumentor tags at a minimum as files (see above).

### 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"

```php
<?php
/**
 * Does something interesting
 *
 * @param Place $where Where something interesting takes place
 * @param int $repeat How many times something interesting should happen
 * @throws Some_Exception If something interesting cannot happen
 * @return boolean
 */
function doesSomethingInteresting(Place $where, $repeat = 1) {
    // Implementation
}
```

### require / include

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

```php
<?php
// Incorrect
require('/path/to/file.php');

// Correct
require '/path/to/file.php';
```

## Empty class and function bodies

Empty classes and functions/methods shall have a completely empty body.

### Classes

```php
<?php
// incorrect
class SomeException extends Exception {

}

// correct
class SomeException extends Exception {}
```

### Methods

```php
<?php
// incorrect
class MyClass extends OtherClass {
    /**
     * Override some method
     */
    public function doSomething() {

    }
}

// correct
class MyClass extends OtherClass {
    /**
     * Override some method
     */
    public function doSomething() {}
}
```