

# **PHP Coding Conventions**

By Janne Ohtonen, 2005

Contents

1	Introduction.....	1
1.1	Copyright Notice.....	1
1.2	Purpose.....	1
1.3	Why Have Coding Conventions.....	1
2	File Names.....	1
2.1	File Suffixes.....	1
2.2	File Formats.....	2
3	File Organization.....	2
3.1	Source Files.....	2
	3.1.1 Beginning Comments.....	2
	3.1.2 Import Statements.....	3
	3.1.3 Class and Interface Declarations.....	3
4	Indentation.....	3
4.1	Line Length.....	4
4.2	Wrapping Lines.....	4
5	Comments.....	5
5.1	Implementation Comment Formats.....	6
	5.1.1 Block Comments.....	6
	5.1.2 Single-Line Comments.....	6
	5.1.3 Trailing Comments.....	6
	5.1.4 End-Of-Line Comments.....	6
5.2	Documentation Comments.....	7
6	Declarations.....	7
6.1	Number Per Line.....	7
6.2	Initialization.....	7
6.3	Placement.....	8
6.4	Class and Interface Declarations.....	8
	6.4.1 Class abstraction.....	9
7	Statements.....	10
7.1	Simple Statements.....	10
7.2	Compound Statements.....	10
7.3	return Statements.....	10
7.4	Constructor.....	11
7.5	Destructor.....	11
7.6	if, if-else, if else-if else Statements.....	11
7.7	for Statements.....	12
7.8	while Statements.....	12
7.9	do-while Statements.....	13
7.10	switch Statements.....	13
7.11	try-catch Statements.....	14
7.12	Function calls.....	14
8	White Space.....	14
8.1	Blank Lines.....	14
8.2	Blank Spaces.....	15
9	Naming Conventions.....	15
10	Programming Practices.....	15
10.1	Providing Access to Instance and Class Variables.....	15
10.2	Referring to Class Variables and Functions.....	15
10.3	Constants.....	15
10.4	Variable Assignments.....	17
10.5	Miscellaneous Practices.....	17
	10.5.1 Parentheses.....	17
	10.5.2 Returning Values.....	18
	10.5.3 Expressions before `?' in the Conditional Operator.....	18
	10.5.4 Special Comments.....	18
	10.5.5 Global variables.....	18
11	Code Examples.....	18

## 1 Introduction

---

### 1.1 Copyright Notice

PHP is copyrighted by 2001-2004 The PHP Group. See the PHP Licence from here <http://www.php.net/license/>

### 1.2 Purpose

Coding conventions are not just about obsession; they are about productivity, professionalism and presentation. This document outlines conventions to aid developers to write code in a productive manner. Coding standards are important because they lead to a greater consistency within your code and the code of your team-mates. Greater consistency leads to code that is easier to understand, which in turn means it is easier to develop and to maintain. This reduces the overall cost of the applications that you create.

You have to remember that your PHP code will exist for a long time, long after you have moved on to other projects. An important goal during development is to ensure that you can transition your work to another developer, or to another team of developers, so that they can continue to maintain and enhance your work without having to invest an unreasonable effort to understand your code. Code that is difficult to understand runs the risk of being scrapped and rewritten. If everyone is doing their own thing then it makes it very difficult to share code between developers, raising the cost of development and maintenance. Inexperienced developers will often fight having to follow standards. They claim they can code faster if they do it their own way. They might be able to get code out the door faster, but they will get hung up during testing when several difficult-to-find bugs crop up, and when their code needs to be enhanced it often leads to a major rewrite by them because they're the only ones who understand their code.

These coding conventions are to ensure that our products are more readable, maintainable, robust and easy to enhance and understand.

### 1.3 Why Have Coding Conventions

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

## 2 File Names

---

This section lists commonly used file suffixes and names.

### 2.1 File Suffixes

PHP uses the following file suffixes:

File Type	Suffix
Source	.php
Includes	.inc

# PHP CODING CONVENTIONS

1.11.2005

Templates	.tpl
-----------	------

## 2.2 File Formats

All files should be formatted the same way:

File Type	Suffix
Character set	UTF-8
Format	Ascii text
Line ends	LF
File ends	One line feed after closing PHP tag.

## 3 File Organization

---

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

For an example of a properly formatted file, see "[PHP Source File Example](#)".

### 3.1 Source Files

Each source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Source files have the following ordering:

- Beginning comments (see "[Beginning Comments](#)")
- Import statements
- Class and interface declarations (see "[Class and Interface Declarations](#)")

#### 3.1.1 Beginning Comments

All source files should begin with comment that lists the class name, version information, date, and copyright notice:

```
/**
 *
 * Description of the file.
 *
 * @category CategoryName
 * @package PackageName
 * @author Your Name <firstname.lastname@domain.com>
 * @author More Auhors in same format as above
 * @version CVS: $Id$
 * @link http://<some> link to documentation
 * @copyright Some Company. All Rights Reserved.
 * @since File available since Release x.y.z
 * @deprecated File deprecated in Release x.y.z
 * @see Classes and functions that are related to this
 */
```

## PHP CODING CONVENTIONS

1.11.2005

Note that you have to use only the tags that apply your project. More information about documentation comment blocks (docblocks) can be found from <http://www.phpdoc.org/>.

### 3.1.2 Import Statements

PHP `include` statement should not be used because if file cannot be included, it is ignored then and this may cause bugs into your programs.

If you are including files unconditionally use always `require_once` and if you are including code conditionally then use `require`. Forget the `include` and `include_once` statements!

Notice that `require` and `require_once` are statements, not functions. Therefore you should not put any parentheses surrounding them. Here is an example:

```
require_once 'file.php';
if($somethingIsTrue) {
    require 'anotherFile.php';
}
```

Well, of course there may be justified usage of `include` and `include_once` statements in some cases. Main thing is that you know the consequences.

### 3.1.3 Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear in a file. See "[PHP Source File Example](#)" for an example that includes comments.

#	Part of Class/Interface Declaration	Notes
1	Description for the file	See " <a href="#">Beginning comments</a> " for information on what should be in this comment.
2	Requirements and globals	<code>require_once</code> statements, <code>define()</code> functions and globals.
3	Class/interface implementation comment ( <code>/*...*/</code> ), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	<code>class</code> or <code>interface</code> statement	
5	Class ( <code>static</code> ) variables	First the <code>public</code> class variables, then the <code>protected</code> , then package level (no access modifier), and then the <code>private</code> .
6	Instance variables	First <code>public</code> , then <code>protected</code> , then package level (no access modifier), and then <code>private</code> .
7	Constructors and destructor	
8	Functions	These functions should be grouped by functionality rather than by scope or accessibility. For example, a <code>private</code> class function can be in between two <code>public</code> instance functions. The goal is to make reading and understanding the code easier.

## 4 Indentation

---

Tab characters should be used as indentation.

## PHP CODING CONVENTIONS

1.11.2005

### 4.1 Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

**Note:** Examples for use in documentation should have a shorter line length; generally no more than 70 characters.

### 4.2 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 2 tabs instead.

Here are some examples of breaking function calls:

```
someFunction($longExpression1, $longExpression2, $longExpression3,  
            $longExpression4, $longExpression5);
```

```
$var = someFunction1($longExpression1,  
                   someFunction2($longExpression2,  
                                $longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
$longName1 = $longName2 * ($longName3 + $longName4 - $longName5)  
            + 4 * $longname6; // PREFER
```

```
$longName1 = $longName2 * ($longName3 + $longName4  
                          - $longName5) + 4 * $longname6; // AVOID
```

Following are two examples of indenting function declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation.

```
//CONVENTIONAL INDENTATION  
function someFunction($anArg, $anotherArg, $yetAnotherArg,  
                    $andStillAnother) {  
    ...  
}
```

```
//INDENT 2 TABS TO AVOID VERY DEEP INDENTS  
private static function horkingLongFunctionName($anArg,  
        $anotherArg, $yetAnotherArg,  
        $andStillAnother) {  
    ...  
}
```

Line wrapping for `if` statements should generally use the 2-tab rule, since conventional (1 tab) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
```

## PHP CODING CONVENTIONS

### 1.11.2005

```
if (($condition1 && $condition2)
    || ($condition3 && $condition4)
    || !($condition5 && $condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if (($condition1 && $condition2)
    || ($condition3 && $condition4)
    || !($condition5 && $condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if (($condition1 && $condition2) || ($condition3 && $condition4)
    || !($condition5 && $condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
$alpha = ($aLongBooleanExpression) ? $beta : $gamma;

$aalpha = ($aLongBooleanExpression) ? $beta
        : $gamma;

$aalpha = ($aLongBooleanExpression)
        ? beta
        : gamma;
```

## 5 Comments

---

PHP programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found from several programming languages, which are delimited by `/*...*/`, and `//` (do not use Perl style comments `#`). Documentation comment blocks (known as "docblocks") are delimited by `/**...*/`. Docblocks can be extracted to HTML files using the appropriate tool.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or non-obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

**Note:** The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

## PHP CODING CONVENTIONS

1.11.2005

### 5.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

#### 5.1.1 Block Comments

Block comments are used to provide descriptions of files, functions, data structures and algorithms. Block comments may be used at the beginning of each file and before each function. They can also be used in other places, such as within functions. Block comments inside a function or function should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/**
 * Here is a block comment.
 */
```

#### 5.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 5.1.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in code:

```
if ($condition) {

    /* Handle the condition. */
    ...
}
```

#### 5.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in code:

```
if (a == 2) {
    return TRUE; /* special case */
} else {
    return isPrime(a); /* works only for odd a */
}
```

#### 5.1.4 End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if ($foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false; // Explain why here.
}
//if ($bar > 1) {
//
//    // Do a triple-flip.
```

## PHP CODING CONVENTIONS

### 1.11.2005

```
//    ...  
//}  
//else {  
//    return false;  
//}
```

## 5.2 Documentation Comments

**Note:** See "[PHP Source File Example](#)" for examples of the comment formats described here.

Comments describe classes, interfaces, constructors, functions, and fields. Each comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**  
 * The Example class provides ...  
 */  
public class Example { ...
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (`/**`) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or function that isn't appropriate for documentation, use an implementation block comment (see section 5.1.1) or single-line (see section 5.1.2) comment immediately *after* the declaration. For example, details about the implementation of a class should go in such an implementation block comment *following* the class statement, not in the class doc comment.

Comments should not be positioned inside a function or constructor definition block, because PHP documentor associates documentation comments with the first declaration *after* the comment.

## 6 Declarations

---

### 6.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
<visibility> $level; // indentation level  
<visibility> $size; // size of table
```

is preferred over

```
<visibility> $level, $size;
```

Do not put different types on the same line. Example:

```
<visibility> $foo, $fooarray[]; //WRONG!
```

Notice that `<visibility>` (public, protected, private) is needed only for class variables.

### 6.2 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Initialise PHP files with `<?php` and with `?>`. Do not use the shorthand notation (`<?` and `?>`). This is for compliance reasons with different operating systems.

## PHP CODING CONVENTIONS

1.11.2005

### 6.3 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}"). Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
<visibility> function myFunction() {
    $int1 = 0; // beginning of function block
    $condition = true;
    if ($condition) {
        $int2 = 0; // beginning of "if" block
        ...
    }
}
```

One exception to the rule is indexes of `for` loops, which can be declared in the `for` statement:

```
for ($i = 0; i < $maxLoops; $i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
$count;
...
<visibility> function myFunction() {
    if ($condition) {
        $count = 0; // AVOID!
        ...
    }
    ...
}
```

### 6.4 Class and Interface Declarations

When coding classes and interfaces, the following formatting rules should be followed:

- No space between a function name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
    <visibility> $iVar1;
    <visibility> $ivar2;

    <visibility> function sample($i, $j) {
        $this->iVar1 = $i;
        $this->iVar2 = $j;
    }

    <visibility> abstract function emptyFunction() {}

    ...
}
```

- Functions are separated by a blank line
- Interfaces are declared with interface

```
interface iSomeInterface {
    <visibility> function someFunction($var1, $var2);
}
```

## PHP CODING CONVENTIONS

1.11.2005

```
}
```

Information on class declaration with PHP can be found from <http://fi.php.net/manual/en/language.oop5.basic.php> .

Information on interface declaration with PHP can be found from <http://fi.php.net/manual/en/language.oop5.interfaces.php> .

### 6.4.1 Class abstraction

It is not allowed to create an instance of a class that has been defined as abstract. Any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature they cannot define the implementation.

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same (or weaker) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public.

Example:

```
<?php
abstract class AbstractClass
{
    // Force Extending class to define this method
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);
    // Common method
    public function printOut() {
        print $this->getValue() . "\n";
    }
}
class ConcreteClass1 extends AbstractClass
{
    protected function getValue() {
        return "ConcreteClass1";
    }
    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}
class ConcreteClass2 extends AbstractClass
{
    public function getValue() {
        return "ConcreteClass2";
    }
}
```

## PHP CODING CONVENTIONS

### 1.11.2005

```
    }
    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass2";
    }
}
$class1 = new ConcreteClass1;
$class1->printOut();
echo $class1->prefixValue('FOO_') ."\n";
$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') ."\n";
?>
```

## 7 Statements

---

### 7.1 Simple Statements

Each line should contain at most one statement. Example:

```
$argv++;          // Correct
$argc--;         // Correct
$argv++; $argc--; // AVOID!
```

### 7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

### 7.3 return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return $myDisk->size();

return ($size ? $size : $defaultSize);
```

## PHP CODING CONVENTIONS

1.11.2005

### 7.4 Constructor

Constructor is used to make a new instance of a class. Below is code example of the use of constructor:

```
<?php
class BaseClass {
    function __construct() {
        print "In BaseClass constructor\n";
    }
}

class SubClass extends BaseClass {
    function __construct() {
        parent::__construct();
        print "In SubClass constructor\n";
    }
}

$obj = new BaseClass();
$obj = new SubClass();
?>
```

### 7.5 Destructure

Destructure is called after object is no more useful. Below is an example of destructure:

```
<?php
class MyDestructableClass {
    function __destruct() {
        print "Destroying " . $this->name . "\n";
    }
}
?>
```

### 7.6 if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition1 || condition2) {
    statements;
} else {
    statements;
```

## PHP CODING CONVENTIONS

1.11.2005

```
}

if (condition1 && condition2) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

**Note:** `if` statement always uses braces `{}`. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITTS THE BRACES {}!
    statement;
```

### 7.7 for Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}
```

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

When looping arrays, get first size of array. Do not get the size of array in loop. See the example below:

```
for($i = 0; $i < count($tmpArr); $i++) {           //Not like this!!!
    //do something
}

for($i = 0, $j = count($tmpArr); $i < $j; $i++) { //Correct!
    //Do something
}
```

### 7.8 while Statements

A `while` statement should have the following form:

```
while (condition) {
    statements;
}
```

An empty `while` statement should have the following form:

```
while ($condition);
```

When looping arrays, remember to use boolean value to evaluate arrays. See the example below:

```
while(false != list($k,$v) = each($array)) {
    //Do something
}
```

## PHP CODING CONVENTIONS

1.11.2005

```
}
```

### 7.9 do-while Statements

A do-while statement should have the following form:

```
do {
    statements;
} while (condition);
```

When looping arrays with do-while, don't use `count` or `sizeof` in while statement. See the examples below:

```
$i = 0;
do {
    //Do something
    $i++;
} while ($i < count($array)); //Wrong!
```

```
$i = 0;
$j = count($array);
do {
    //Do something
    $i++;
} while ($i < $j); //Correct!
```

### 7.10 switch Statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

## PHP CODING CONVENTIONS

1.11.2005

### 7.11 try-catch Statements

PHP 5 has an exception model similar to that of other programming languages. An exception can be thrown, try and caught within PHP. A Try block must include at least one catch block. Multiple catch blocks can be used to catch different classtypes; execution will continue after that last catch block defined in sequence. Exceptions can be thrown within catch blocks.

When an exception is thrown, code following the statement will not be executed and PHP will attempt to find the first matching catch block. If an exception is not caught a PHP Fatal Error will be issued with an Uncaught Exception message, unless there has been a handler defined with `set_exception_handler()`.

A try-catch statement should have the following format:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

A try-catch statement may also be followed by `finally`, which executes regardless of whether or not the try block has completed successfully.

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

More information on exceptions in PHP can be found from <http://fi.php.net/exceptions> .

### 7.12 Function calls

Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon. Here's an example:

```
<?php
$var = foo($bar, $baz, $quux);
?>
```

As displayed above, there should be one space on either side of an equals sign used to assign the return value of a function to a variable. In the case of a block of related assignments, more space may be inserted to promote readability:

```
<?php
$short           = foo($bar);
$long_variable = foo($baz);
?>
```

## 8 White Space

---

### 8.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

## PHP CODING CONVENTIONS

### 1.11.2005

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between functions
- Between the local variables in a function and its first statement
- Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment
- Between logical sections inside a function to improve readability

## 8.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {  
    ...  
}
```

Note that a blank space should not be used between a function name and its opening parenthesis. This helps to distinguish keywords from function calls.

- A blank space should appear after commas in argument lists.
- Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
$tmp++; //correct  
$tmp ++; //wrong  
$a += $c + $d; //correct  
$a = ($a + $b) / ($c * $d); //correct
```

- Use one blank space between concat operator (.)  
printSize("size is " . \$foo . "\n");
- The expressions in a for statement should be separated by blank spaces. Example:

```
for ($expr1; $expr2; $expr3)
```

## 9 Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it's a constant, package, or class—which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	<pre>class UpperCase; class Raster; class ImageSprite;</pre>
Interfaces	Interface names should have small I as prefix and rest of the name capitalized like class names.	<pre>interface iUpperCase; interface iRaster; interface iStoring;</pre>
Functions	Functions should be verbs, in mixed case with the first	<pre>run();</pre>

## PHP CODING CONVENTIONS

1.11.2005

	letter lowercase, with the first letter of each internal word capitalized. For private functions use underscore (_) at first character of the name.	<pre>lowerCase(); _someFunction();//private</pre>
Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters.  Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are \$i, \$j, \$k, \$m, and \$n for integers; \$c, \$d, and \$e for characters.  For private variables use underscore in name.	<pre>\$lower; \$lowerCase; \$myWidth; \$i;  private \$_var;</pre>
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	<pre>static final MIN_WIDTH = 4;  static final MAX_WIDTH = 999;  static final GET_THE_CPU = 1;</pre>

## 10 Programming Practices

---

### 10.1 Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Sometimes, instance variables don't need to be explicit setters or getters; often that happens as a side effect of function calls.

### 10.2 Referring to Class Variables and Functions

Avoid using an object to access a class (static) variable or function. Use a class name instead. For example:

```
classFunction();           //OK  
$AClass::classFunction(); //OK  
$anObject->classFunction(); //AVOID!
```

### 10.3 Constants

It is possible to define constant values on a per-class basis remaining the same and unchangeable. Constants differ from normal variables in that you don't use the \$ symbol to declare or use them. Like static members, constant values cannot be accessed from an instance of the object (using \$object::constant).

The value must be a constant expression, not (for example) a variable, a class member, result of a mathematical operation or a function call.

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

Example:

## PHP CODING CONVENTIONS

### 1.11.2005

```
<?php
class MyClass
{
    const constant = 'constant value';

    function showConstant() {
        echo self::constant . "\n";
    }
}

echo MyClass::constant . "\n";

$class = new MyClass();
$class->showConstant();
// echo $class::constant; is not allowed
?>
```

## 10.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
$fooBar->fChar = $barFoo->lChar = 'c'; // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if ($c++ == $d++) {           // AVOID!
    ...
}
```

Should be written as

```
if (($c++ = $d++) != 0) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
$d = ($a = $b + $c) + $r;      // AVOID!
```

Should be written as

```
$a = $b + $c;
$d = $a + $r;
```

## 10.5 Miscellaneous Practices

### 10.5.1 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if ($a == $b && $c == $d)      // AVOID!
if (($a == $b) && ($c == $d)) // RIGHT
```

## PHP CODING CONVENTIONS

1.11.2005

### 10.5.2 Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
    return $x;
}
```

return \$y;

should be written as

```
return ($condition ? $x : $y);
```

### 10.5.3 Expressions before '?' in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
($x >= 0) ? $x : -$x;
```

### 10.5.4 Special Comments

Use `TODO` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.

### 10.5.5 Global variables

Usage of globals should be avoided. This is because server administrator may decide to disable register globals and even though it is possible to enable them with `htaccess` it is strongly discouraged.

## 11 Code Examples

---

The following example shows how to format a source file containing a single public class. Interfaces are formatted similarly. For more information, see "[Class and Interface Declarations](#)" and "[Documentation Comments](#)".

This sample code is adapted from PEAR site. For more information see <http://pear.php.net/manual/en/standards.sample.php>.

```
<?php
/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * PHP version 5
 *
 * @category   CategoryName
 * @package   PackageName
 * @author    Original Author <author@example.com>
```

## PHP CODING CONVENTIONS

### 1.11.2005

```
* @author      Another Author <another@example.com>
* @copyright   1997-2005 The PHP Group
* @license     http://www.php.net/license/3_0.txt  PHP License 3.0
* @version     CVS: $Id:$
* @link        http://pear.php.net/package/PackageName
* @see         NetOther, Net_Sample::Net_Sample()
* @since       File available since Release 1.2.0
* @deprecated  File deprecated in Release 2.0.0
*/

/**
 * This is a "Docblock Comment," also known as a "docblock."  The class'
 * docblock, below, contains a complete description of how to write these.
 */
require_once 'PHP.php';

/**
 * Methods return this if they succeed
 */
define('NET_SAMPLE_OK', 1);

/**
 * The number of objects created
 * @global int $GLOBALS['NET_SAMPLE_Count']
 */
$GLOBALS['NET_SAMPLE_Count'] = 0;

/**
 * An example of how to write code to PEAR's standards
 *
 * Docblock comments start with "/*" at the top.  Notice how the "/"
 * lines up with the normal indenting and the asterisks on subsequent rows
 * are in line with the first asterisk.  The last line of comment text
 * should be immediately followed on the next line by the closing asterisk
 * and slash and then the item you are commenting on should be on the next
 * line below that.  Don't add extra lines.  Please put a blank line
 * between paragraphs as well as between the end of the description and
 * the start of the @tags.  Wrap comments before 80 columns in order to
 * ease readability for a wide variety of users.
 *
 * Docblocks can only be used for programming constructs which allow them
 * (classes, properties, methods, defines, includes, globals).  See the
 * phpDocumentor documentation for more information.
 * The Javadoc Style Guide is an excellent resource for figuring out
 * how to say what needs to be said in docblock comments.  Much of what is
 * written here is a summary of what is found there, though there are some
 * cases where what's said here overrides what is said there.
 * http://java.sun.com/j2se/javadoc/writingdoccomments/index.html#styleguide
 *
 * The first line of any docblock is the summary.  Make them one short
 * sentence, without a period at the end.  Summaries for classes, properties
 * and constants should omit the subject and simply state the object,
 * because they are describing things rather than actions or behaviors.
 *
 * Below are the tags commonly used for classes.  @category through @access
 * are required.  The remainder should only be used when necessary.
 * Please use them in the order they appear here.  phpDocumentor has
 * several other tags available, feel free to use them.
 *
```

## PHP CODING CONVENTIONS

### 1.11.2005

```
* @category    CategoryName
* @package     PackageName
* @author      Original Author <author@example.com>
* @author      Another Author <another@example.com>
* @copyright   1997-2005 The PHP Group
* @license     http://www.php.net/license/3_0.txt  PHP License 3.0
* @version     Release: @package_version@
* @link        http://pear.php.net/package/PackageName
* @see         NetOther, Net_Sample::Net_Sample()
* @since       Class available since Release 1.2.0
* @deprecated  Class deprecated in Release 2.0.0
*/
class Net_Sample
{

    /**
     * The status of foo's universe
     *
     * Potential values are 'good', 'fair', 'poor' and 'unknown'.
     *
     * @var string
     */
    private $_foo = 'unknown';

    /**
     * The status of life
     *
     * Note that names of private properties or methods must be
     * preceeded by an underscore.
     *
     * @var bool
     */
    private $_good = true;

    /**
     * Registers the status of foo's universe
     *
     * Summaries for methods should use 3rd person declarative rather
     * than 2nd person imperative, beginning with a verb phrase.
     *
     * Summaries should add description beyond the method's name. The
     * best method names are "self-documenting", meaning they tell you
     * basically what the method does. If the summary merely repeats
     * the method name in sentence form, it is not providing more
     * information.
     *
     * Summary Examples:
     * + Sets the label                (preferred)
     * + Set the label                  (avoid)
     * + This method sets the label    (avoid)
     *
     * Below are the tags commonly used for methods. A @param tag is
     * required for each parameter the method has. The @return and
     * @access tags are mandatory. The @throws tag is required if the
     * method uses exceptions. @static is required if the method can
     * be called statically. The remainder should only be used when
     * necessary. Please use them in the order they appear here.
     * phpDocumentor has several other tags available, feel free to use

```

## PHP CODING CONVENTIONS

### 1.11.2005

```
* them.
*
* The @param tag contains the data type, then the parameter's
* name, followed by a description. By convention, the first noun in
* the description is the data type of the parameter. Articles like
* "a", "an", and "the" can precede the noun. The descriptions
* should start with a phrase. If further description is necessary,
* follow with sentences. Having two spaces between the name and the
* description aids readability.
*
* When writing a phrase, do not capitalize and do not end with a
* period:
*   + the string to be tested
*
* When writing a phrase followed by a sentence, do not capitalize the
* phrase, but end it with a period to distinguish it from the start
* of the next sentence:
*   + the string to be tested. Must use UTF-8 encoding.
*
* Return tags should contain the data type then a description of
* the data returned. The data type can be any of PHP's data types
* (int, float, bool, string, array, object, resource, mixed)
* and should contain the type primarily returned. For example, if
* a method returns an object when things work correctly but false
* when an error happens, say 'object' rather than 'mixed.' Use
* 'void' if nothing is returned.
*
* Here's an example of how to format examples:
* <code>
* require_once 'Net/Sample.php';
*
* $s = new Net_Sample();
* if (PEAR::isError($s)) {
*     echo $s->getMessage() . "\n";
* }
* </code>
*
* Here is an example for non-php example or sample:
* <samp>
* pear install net_sample
* </samp>
*
* @param string $arg1 the string to quote
* @param int    $arg2  an integer of how many problems happened.
*                   Indent to the description's starting point
*                   for long ones.
*
* @return int the integer of the set mode used. FALSE if foo
*            foo could not be set.
* @throws exceptionclass [description]
*
* @access public
* @static
* @see Net_Sample::$foo, Net_Other::someMethod()
* @since Method available since Release 1.2.0
* @deprecated Method deprecated in Release 2.0.0
*/
public function setFoo($arg1, $arg2 = 0)
{
```

## PHP CODING CONVENTIONS

1.11.2005

```
/*
 * This is a "Block Comment." The format is the same as
 * Docblock Comments except there is only one asterisk at the
 * top. phpDocumentor doesn't parse these.
 */
if ($arg1 == 'good' || $arg1 == 'fair') {
    $this->foo = $arg1;
    return 1;
} elseif ($arg1 == 'poor' && $arg2 > 1) {
    $this->foo = 'poor';
    return 2;
} else {
    return false;
}
}

?>
```