



PHP Coding Conventions

Janne Ohtonen / Satama



This presentation is for International PHP Conference 2007 - Spring Edition. It talks about PHP Coding Conventions.

The documentation related to this presentation can be downloaded from:
http://janne.ohtonen.fi/uploads/File/PHP_Coding_Conventions.pdf

Author

- Educational background on information systems
 - Bachelor of Engineering, 2003; Bachelor of Science, 2007; Master of Science, 2008
- Occupational background
 - Software developer, systems specialist, software analyst. Employer: Satama.
- Main focus
 - PHP and web applications
 - Quality practices



1

Founded in 1997 Satama is already a leading digital services company. Our aim is to combine talent and technology to make work and client dialogue simple, fun and profitable. We help change the way you see your customers, how your customers interact with you, and how people work.

It's in this context that we help companies find and talk to their customers. We do this by making daily life and work simpler – and more fun. We envision a future in which everyday tasks will be as entertaining as games.

At Satama, we believe in focusing completely on our customers – and their customers. Sales and Marketing led companies are future winners. We know how to identify your critical opportunities, and help you turn them into good business. And we know how to help you keep the business you have.

Satama has helped hundreds of its clients to execute pivotal change. To mention few: Nokia, TUI Nordic, TeliaSonera and Vodafone have placed significant trust in us.

There are approximately 350 experts in the Satama Group. Satama's crew is ready to meet new challenges in the ideation, planning and execution phases of projects for lifting marketing and productivity – regardless of the media.

More information about Satama from: <http://www.satama.com>

More information about Janne Ohtonen from: <http://janne.ohtonen.fi>

Introduction

- What are coding conventions about?
 - Productivity, professionalism, presentation
- Why are they needed?
 - Help to write code in a productive manner
 - Better consistency
 - Code gets easier to understand
 - Decreases the cost of maintenance
- Team work, legacy projects, quality

2

Coding conventions are about productivity, professionalism and presentation. They are important because they lead to a greater consistency within your code and the code of your team-mates. Better consistency leads to code that is easier to understand, which in turn means it is easier to develop and to maintain.

Coding conventions help to ensure that you can transition your work to enhance your work without having to invest an unreasonable effort to understand your code.

Helps to share code between developers and to reduce the cost of development and maintenance. Thus, it makes the code of better quality.

Here are some reasons, why coding conventions do matter:

- 80% of the software lifetime cost goes to maintenance
- Hardly any software is maintained for its whole life by the original author
- Improve readability of software, allowing understanding more quickly
- If you ship your code to customers, they need to be clean

Files

- Names
 - Suffixes: php, inc (phtml is deprecated)
 - Formats: UTF-8, Ascii text, line end: LF
 - File name and class name should match
- Organization
 - Source files: beginning comments -> import statements -> class and interface declarations
 - Import statements: use *require* and *require_once* (*include* and *include_once* only if you really have to)
 - Class and interface declarations

A file consists of sections that should be separated by blank lines and optionally containing comments identifying each section. Files longer than 2000 lines are cumbersome and should be avoided.

Each source file contains a single public class or interface. Private classes and interfaces are associated with public classes and the can be in same source file.

Basic structure for source file is:

1. Beginning comments
2. Import statements
3. Class and interface declarations

Indentation

- Use tabs as indentation (spaces are deprecated)
 - Width of tab can be adjusted in IDEs
- Line length
 - avoid more than 80 characters
- Wrapping lines
 - Break after comma
 - Break before an operator
 - Prefer high-level breaks to low-level breaks
 - Align the new line with the previous line (if on same level)

4

Tab characters should be used as indentation (even though PEAR disagrees). You can adjust the width of indentation in IDEs (Integrated Development Environments, for example Zend Studio), but you cannot adjust spaces.

Avoid longer lines than 80 characters (easier to read and they fit better in the screen).

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.

Comments

- Implementation comments
 - Code commenting
 - Clears out complicated sections in code
 - Avoid duplicating the information that code gives
 - Formats: block, single-line, trailing and end-of-line
- Documentation comments
 - Describes the specification of code (classes, interfaces, constructors, functions and fields)
 - Implementation free perspective
- General
 - If you feel compelled to add a comment, consider rewriting the code to make it clearer

5

Implementation comments are meant for commenting out code or for comments about the particular implementation. Documentation 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.

Discussion of nontrivial or non-obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. 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.

Block comments are used to provide descriptions of files, functions, data structures and algorithms.

Single-line comments can appear on a single line indented to the level of the code that follows.

Trailing comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements.

The end-of-line `//` comment delimiter can comment out a complete line or only a partial line.

Declarations

- One declaration per line
 - `<visibility> $variable //Possible comment`
- Do not declare different types of variables on same line
 - Avoid: `<visibility> $fooInt, $barArray[];`
- Initialize local variables where they are declared if possible
- Put declarations only at the beginning of blocks (except for loops)
- Avoid local declarations that hide declarations at higher levels (do not declare same variable name in inner block, that outer block has)

6

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.

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.

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.

Statements

- Simple statements (only one at a line)
- Compound statements: braces {}, indentation
- Return statements: use parenthesis only if needed [*return (\$foo ? true : false)]*
- Constructor, Destructor: *__construct, __destruct*
- Conditional statements
- Iterating statements
- Switch: mark fall through cases with comments
- Try, catch, finally
- Function calls

Each line should contain at most one statement. Example:

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

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }".

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

```
return;  
return $myDisk->size();  
return ($size ? $size : $defaultSize);
```

Constructor is used to make a new instance of a class. Destructor is called after object is no more useful.

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

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

White spaces

- Use blank lines to make sections of code more clear
- Two blank lines between classes and sections of source file
- One blank line in following cases:
 - Between functions
 - Between the local variables in a function
 - Before a block or single-line comment
 - Between logical sections inside a function (to improve readability)
- Blank spaces on following cases
 - Keyword followed by parenthesis
 - After commas in argument lists
 - Expressions in a for statement should be separated [*for (\$expr1; \$expr2, \$expr3)]*

8

Blank lines improve readability by setting off sections of code that are logically related. 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 or single-line comment
- Between logical sections inside a function to improve readability

Blank spaces should be used in the following circumstances:

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

Naming conventions

- Make programs more understandable by making them easier to read
- Give information about the named thing
- Class' names should be nouns, in mixed case with the first letter of each internal word capitalized (example: *UpperCase*, *SomeWordsInClassName*)
- Interface names should have small i as prefix and rest like classes (*iInterfaceName*)
- Functions should be verbs, in mixed case, first letter lowercase. Private functions use underscore as first letter (*lowerCase()*, *_someFunction()*)
- Variable names should be meaningful and first letter is lower and then all internal words are in uppercase (*\$someVariable*, *\$meaningfulName*)
- Constants are written in uppercase, words are separated with underscore (example: *static final CONSTANT_VARIABLE = true;*)

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.

Programming practices

- Don't make instance or class variables public without a good reason
- Avoid using object to access a class static variable or function
- Avoid assigning several variables to the same value in a single statement
 - `$foo = $bar = $foobar //Avoid, difficult to read`
- Use parentheses to make expressions more clear
 - `if($foo == $bar && $foobar == $barfoo) -> if(($foo == $bar) && ($foobar == $barfoo))`
- Expressions containing binary operator before ternary operator should have parentheses
 - `($foo >= 0) ? $foo : $bar;`
- Use `//@TODO <comment>` to comment things that need to be done and `//@FIXME <what to fix>` for things that need to be fixed

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.

Referring to Class Variables and Functions: Avoid using an object to access a class (static) variable or function. Use a class name instead.

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`).

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
```

How to take in use?

It may be difficult to create new quality practices into company. If developers do not understand why they should use coding conventions, they will slip out of the practice eventually.

Education, training, code evaluations, quality practices...

11

Here is one example of successful adaption of new coding convention:

- The legacy projects are not easy to get under coding conventions, thus it is easier to let them be as they are and to try to imitate the style that has been used in them. If new coding convention is joined with old cumbersome way, then developers need to use two styles to understand the code.
- Adapt the coding convention to your needs. Not all rules are good for everyone, so they need to be made good ones for you.
- Present the coding convention to your developers. Help them to understand, why it is important. Listen carefully for all the feedback and if possible even take developers in the process of making coding conventions (early adaption helps to reduce change resistance).
- Give developers some time to put the coding convention into use. Do code reviews and discuss about feelings and thoughts that new procedure has risen.
- When coding convention is adapted, make it part of the whole quality practise of the company. Embed it into quality handbook and make it clear to everyone that this is the way we go now.
- Embed the coding conventions in marketing and sales. They can start selling extra value for your customers, since you have better quality practices now.

More information

From PHP site <http://www.php.net>
Each subject have guidelines how to use them.

PEAR
<http://pear.php.net/manual/en/standards.php>

You can find several kinds of coding conventions
for PHP and open source programs done with
PHP from Internet (for example with Google).

Thank you!

Any questions or comments?

Please, send some feedback to:
janne.ohtonen@satama.com

Contact details for Satama can be found from: <http://www.satama.com>

Contact information for Janne Ohtonen can be found from: <http://janne.ohtonen.fi>