Coding Standards PHP

Table of Contents

1. NAMING CONVENTIONS 1

1.1 Class Names (Pascal Case) 1
1.2 Method and Function Names (Camel Case) 1
1.3 No All Upper Case Abbreviations 2
1.4 Class Library Names 2
1.5 Variable Names 2
1.6 Class Attribute Names 3
1.7 Method Argument Names 3
1.8 Single or Double Quotes 3
1.9 Global Variables 4
1.10 Define Names / Global Constants 4

2. GUIDELINES 4

2.1 Braces {} Guideline 4
2.2 Indentation/Tabs/Space Guideline 5
2.3 Parens () with Key Words and Functions Guideline 6

3. FORMATTING 6

3.1 If Then Else Formatting 6
3.2 Switch Formatting 7
3.3 Use of continue, break and ?: 7
3.4 Alignment of Declaration Blocks 9
3.5 Do Not Default If Test to Non-Zero 9

4. COMMENTS 9

5. OPEN/CLOSED PRINCIPLE 14

6. HTTP_*_VARS 15

7. PHP FILE EXTENSIONS 15

8. MISCELLANEOUS 15

9. USE IF (0) TO COMMENT OUT CODE BLOCKS (GUIDELINE) 16

10. SOURCE CODE CONTROL SYSTEM (SVN) COMMENTS 16

11. PHP CODE TAGS 17

12. NO MAGIC NUMBERS 17

Change History

Date	Ver	Change description	Prepared By	Reviewed By	Approved By
26-08-16	1.0	First Release	Subbiah	Kumaran	Harihara G
07-02-2017	1.1	Updated the document ID	Deepa	Deepa	Harihara.G
01-02-19	1.2	Updated the document ID	Document Controller	MR	CISO

1. Naming Conventions

1.1 Class Names (Pascal Case)

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 upper case letters as word separators, lower case for the rest of a word.
- First character in a name is upper case No underscores ('_').
- Name the class after what it is. If you can't think of what it is, that is a clue you have not thought through the design well enough.
- Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
- Suffixes are sometimes helpful. For example, if your user interface class of Candidates then naming something `CandidatesUI` conveys real information.

Example:

- class `Attachments`
- class `Candidates`

1.2 Method and Function Names (Camel Case)

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized. Most methods and functions perform actions, so the name should make clear what it does, as concisely as possible.

Like: `candidateAddImage()` instead of `candidate_addImage()`, `candidateDeletePermission()` instead of `candidate_deletePermission()`.

Suffixes are sometimes useful:

- Max - to mean the maximum value something can have.
- Cnt - the current count of a running count variable.
- Key - key value.

For example: retryMax to mean the maximum number of retries, retryCnt to mean the current retry count.

Prefixes are sometimes useful:

- Is - to ask a question about something. Whenever someone sees “Is” they will know it's a question.
- Get - get a value.
- Set - set a value.

For example: `isPostBack()`, `isGetBack()`, `getUserList()`, `getProfilePic()`.

1.3 No All Upper Case Abbreviations

- When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters. No matter what.

  Do use: GetHtmlStatistic.
  Do not use: GetHTMLStatistic.

Justification

- People seem to have very different intuitions when making names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable.

  Take for example *NetworkABCKey*. Notice how the C from ABC and K from key are confused. Some people don't mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

Example

```
class GetHtmlStatistic        // NOT GetHTMLStatistic
```

1.4 Class Library Names

- Now that name spaces are becoming more widely implemented, name spaces should be used to prevent class name conflicts among libraries from different vendors and groups.
- When not using name spaces, it's common to prevent class name clashes by prefixing class names with a unique string. Two characters is sufficient, but a longer length is fine.

Example

Database management library could use *Informm* as a prefix, so classes would be:

```
class InformmDatabaseConnection extends DatabaseConnection
{
}
```

1.5 Variable Names

Variable names should be all lowercase, with words separated by underscores. For example, $current_user is correct, but $currentuser, $currentUser or $CurrentUser are not. Variable names should be short yet meaningful. The choice of a variable name should indicate to the casual observer the intent of its use.

One-character variable names should be avoided except for temporary variables and loop indices. Common names for temporary variables are i, j, k, m, and n for integers; c, d, e for strings.

- use all lower case letters

- use '_' as the word separator
- do not use 'l' (lowercase 'L') as a temporary variable
- do not use '-' as the word separator

Justification

- This allows a variable name to have the same name as a database column name, which is a very common practice in PHP.
- 'l' (lowercase 'L') is easily confused with 1 (the number 'one')
- If '-' is used as a word separator, it will generate warnings used with magic quotes.

1.6 Class Attribute Names

- All private, static class member attribute/method names should be pre appended with the character '_'.
- After the '_' use the same rules as for variable/method names and for public members same rule as variable and method names.
- '_' always precedes other name modifiers like 'r' for reference.

Justification

- Pre appending '_' prevents any conflict with method names and it access

specifiers. Example: `private $_fields = array();public $data = array();`

1.7 Method Argument Names

Since function arguments are just variables used in a specific context, they should follow the same guidelines as variable names. It should be possible to tell the purpose of a method just by looking at the first line, e.g. getUserData($user_name).

By examination, you can make a good guess that this function gets the user data of a user with the username passed in the $user_name argument. Method arguments should be separated by spaces, both when the function is defined and when it is called. However, there should not be any spaces between the arguments and the opening/closing parentheses.

1.8 Single or Double Quotes

- Use single quote when you don"t want to substitute any variable value within the string.
- Use doublr quote when you want to replace a variable within a string with value.

Example:

$element_name = 'foo_bar';

$count = 3;

$message = “$count records found.”;

1.9 Global Variables

Global variables should be prepended with a 'g'. It's important to know the scope of a variable.

Example: global $g_country_time_zone_lookup instead of global $countryTimezoneLookup

1.10 Define Names / Global Constants

Global constants should be all caps with '_' separators. It's tradition for global constants to name this way. You must be careful to not conflict with other predefined global.

Example: define('INFORMM_VERSION', '2.1 Beta');

2. Guidelines

2.1 Braces {} Guideline

Of the three major brace placement strategies two are acceptable, with the first one listed being preferable:

- Place brace under and inline with keywords:

```
if ($condition)        while ($condition)
{                      {
    ...                    ...
}                      }
```

- Traditional Unix policy of placing the initial brace on the same line as the keyword and the trailing brace inline on its own line with the keyword:

```
if ($condition) { ...      while ($condition) {
                               ...
}                      }
```

Justification

- Another religious issue of great debate solved by compromise. Either form is acceptable, many people, however, find the first form more pleasant. Why is the topic of many psychological studies.

  There are more reasons than psychological for preferring the first style. If you use an editor (such as vi) that supports brace matching, the first is a much better style. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace. Example:

```
if ($very_long_condition && $second_very_long_condition)
{
```

```
  ...
}
else if (...)
{
...
}
```

To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

2.2 Indentation/Tabs/Space Guideline

- Do not use Spaces, use Tab. Most editors can substitute spaces for tabs.
- Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level. If the indenting level is more than 4 or 5 levels you may think about factoring out code.

Justification

- When people using different tab settings the code is impossible to read or print, which is why spaces are preferable to tabs.
- Nobody can ever agree on the correct number of spaces, just be consistent. In general people have found 3 or 4 spaces per indentation level workable.
- As much as people would like to limit the maximum indentation levels it never seems to work in general. We'll trust that programmers will choose wisely how deep to nest code.

Example

```
function func()
{
   if (something bad)
   {
      if (another thing bad)
      {
         while (more input)
         {
         }
      }
   }
}
```

SQL Query Indentation:

```
SELECT

    u1.employee_name,

    u2.employee_name

FROM
```

```
    users AS u1

    JOIN users AS u2 ON (u2.employee_id = u1.manager_id)

WHERE

    Employee_id = 10
```

2.3 Parens () with Key Words and Functions Guideline

- Do not put parens next to keywords. Put a space between.
- Do put parens next to function names.
- Do not use parens in return statements when it's not necessary.

Justification

- Keywords are not functions. By putting parens next to keywords and function names are made to look alike.

Example

```
if (condition)
{
}

while (condition)
{
}

strcmp($s, $s1);

return 1;
```

3. Formatting

3.1 If Then Else Formatting

Layout

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

```
if (condition)                    // Comment
{
}
```

```
else if (condition)            // Comment
{
}
else                           // Comment
{
}
```

If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

Condition Format

Always put the constant on the left hand side of an equality/inequality comparison. For example:

```
if (6 == $errorNum) ...
```

One reason is that if you leave out one of the = signs, the pharser will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

3.2 Switch Formatting

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
- The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

Example

```
switch (...)
{
   case 1:
      ...
   // FALL THROUGH

   case 2:
   {
      $v = get_week_number();
      ...
   }
   break;

   default:
}
```

3.3 Use of continue, break and ?:

Continue and Break

Continue and break are really disguised gotos so they are covered here.

Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.

The two main problems with continue are:

- It may bypass the test condition
- It may bypass the increment/decrement expression

Consider the following example where both problems occur:

```
while (TRUE)
{
   ...
   // A lot of code
   ...
   if (/* some condition */) {
      continue;
   }
   ...
   // A lot of code
   ...
   if ( $i++ > STOP_VALUE) break;
}
```

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.

From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

?:

The trouble is people usually try and stuff too much code in between the *?* and *:*. Here are a couple of clarity rules to follow:

- Put the condition in parens so as to set it off from other code
- If possible, the actions for the test should be simple functions.
- Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

Example

```
(condition) ? funct1() : func2();

or

(condition)
   ? long statement
   : another long statement;
```

3.4 Alignment of Declaration Blocks

- Block of declarations should be aligned.

Justification

- Clarity.
- Similarly blocks of initialization of variables should be tabulated.
- The „&" token should be adjacent to the type, not the name.

Example

```
var         $mDate
var         $mrDate
&           $mrName
var         $mName
&
var

$mDate      = 0;
$mrDat      = NULL;
e           = 0;
$mrNam      = NULL;
e
$mName
```

3.5 Do Not Default If Test to Non-Zero

Do not default the test for non-zero, i.e.

```
if (FAIL != f()) is better than     if (f())
```

even though FAIL may have the value 0 which PHP considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., **if (!($bufsize % strlen($str)))** should be written instead as **if (0 == ($bufsize % strlen($str)))** to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should **never ever** be defaulted.

The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

- Returns 0 for false, nothing else.
- Is named so that the meaning of (say) a **true** return is absolutely obvious. Call a predicate IsValid(), not CheckValid().

4. Comments

phpDocumentor style of documentation to be used.

Two audiences tend to be opposed to each other in their needs.

An end-user generally wants:

- Instruction-style writing, that explains and describes general concepts more than how a particular variable is used

- Interface information only, no low-level details
- Examples of how to use, and tutorials

Whereas a programmer may want in addition:

- Details on how program elements interact, which elements use others
- Where in the source code an action or series of actions occurs
- How to extend the code to add new functionality

phpDocumentor can be used to create this option using a few things. First, using the command-line file-selection options, one can write two sets of documentation, one for end-users, and the other for programmers, and put them in different subdirectories.

In addition, using the @access tag, one can mark programmer-level elements with @access private, and they will be ignored by default.

The @internal tag and inline {@internal}} inline tag construct can be used to mark documentation that is low-level or internal to a smaller audience of developers. When creating programmer documentation, turn on the parse private option (see -pp, --parseprivate), and the low-level detail will be generated.

More important, the in-code documents in your DocBlocks must be written for both end-users and programmers wherever possible.

DocBlock comment is a C-style comment with two leading asterisks (*), like so:

```
/**
*
*/
```

Caution

- All other comments are ignored by the documentation parser.
- Any line within a DocBlock that doesn't begin with a * will be ignored.

A DocBlock contains three basic segments in this order:

- Short Description
- Long Description
- Tags

DocBlock Templates

The purpose of a DocBlock template is to reduce redundant typing.

A DocBlock template is distinguished from a normal DocBlock by its header. Here is the most basic DocBlock template:

```
/**#@+
 *
*/
```

Example:

```
class Bob
 {
        /**#@+
       * @access private
```

```
        * @var string
        */
        var $_var1 = 'hello';
        var $_var2 = 'my';
       /**
        * Two words
        */
        var $_var3 = 'like
        strings'; /**#@-*/
        var $publicvar = 'Lookee me!';
}
```

This example will parse as if it were:

```
class Bob
{
        /**
        * @access private
        * @var string
        */
        var $_var1 = 'hello';
        /**
        * @access private
        * @var string
        */
        var $_var2 = 'my';
        /**
        * Two words
        * @access private
        * @var string
        */
        var $_var3 = 'like strings';
        var $publicvar = 'Lookee me!';
}
```

Tags

Tags are single words prefixed by a "@" symbol. Tags inform phpDocumentor how to present information and modify display of documentation.

Here are some of the phpDocumentor tags:

@abstract
@access *private*
@author *Gregory Beaver <cellog@php.net>*
@copyright *Copyright (c) 2002, Gregory Beaver*
@final *[document a class method that should never be overridden in the child class]*
@global *bool $var_name*
@ignore *[Prevents documenting of an element]*
@internal *the class uses the private methods*
@license *http://opensource.org/licenses/gpl-license.php GNU Public License*
@link *http://www.valuepitch.com*
@name *[Specifies an aliasto use for linking]*
@param *bool $flag*
@return *[Specify the return type of a function or method]*

```
@see [Display a link to the documentation for an
element] @static [Document a static property and
method] @staticvar integer used to calculate the division
tables @package pagelevel_package
@subpackage data
@todo make it do something
@tutorial phpDocumentor.pkg
@var  [Document the data type of the class variable ]
{@internal }}
{@inheritdoc }
{@source 3 1}
@tutorial phpDocumentor/phpDocumentor.pkg
```

Example:

```
<?php
/**
* Sample File
*
* This file demonstrates the rich information that can be included in
* in-code documentation through DocBlocks and tags.
* @author Greg Beaver <cellog@php.net>
* @version 1.0
* @package sample
*
*  Updated By: Developer Name
*  Updated Date: 22-Dec-2010
*  Description: Bug Id# 12 fixed, Updated function 1, function 2….
*
*/

/**
* Special global variable declaration DocBlock
* @global integer $GLOBALS['_myvar']
* @name $_myvar
*/
$GLOBALS[ '_myvar'] = 6;

/**#@+
* Constants
*/
define('testing', 6);

define('anotherconstant', strlen('hello'));
/**#@-*/

/**
* A sample function docblock
* @global string document the fact that this function uses $_myvar
* @staticvar integer $staticvarth is is actually what is returned
* @param string $param1 name to declare
* @param string $param2 value of the name
* @return integer
*/
function firstFunc($param1, $param2 = 'optional')
{
```

```
        static $staticvar = 7;
         global $_myvar;
         return $staticvar;
}

/**
* The first example class, this is in the same package as the
* procedural stuff in the start of the file
* @package sample
* @subpackage classes
*/
class myclass {
        /**
        * A sample private variable, this can be hidden with the --parseprivate
        * option
        * @access private
        * @var integer|string
        */
        var $firstvar = 6;
        /**
        * @link http://www.example.com Example link
        * @see myclass()
        * @uses testing, anotherconstant
        * @var array
        */
        var $secondvar =
        array(
                'stuff' =>
                array(
                6,
                17,
                'armadillo'
                ),
                testing => anotherconstant
        );

        /**
        * Constructor sets up {@link $firstvar}
        */
        function myclass()
        {
                $this-> firstvar = 7;
        }

        /**
        * Return a thingie based on $paramie
        * @param boolean $paramie
        * @return integer|babyclass
        */
        function parentfunc($paramie)
        {
                if ($paramie) {
                return 6;
                } else {
                return new babyclass;
               }
```

```
        }
}


/**
* @package
sample1 */
class babyclass extends myclass
        { /**
        * The answer to Life, the Universe and Everything
        * @var integer
        */
        var $secondvar = 42;
        /**
        * Configuration values
        * @var array
        */
        var $thirdvar;

        /**
        * Calls parent constructor, then increments {@link $firstvar}
        */
        function babyclass()
        {
                parent::myclass();
                $this-> firstvar++;
        }

        /**
        * This always returns a myclass
        * @param ignored $paramie
        * @return myclass
        */
        function parentfunc($paramie)
        {
                return new myclass;
        }
}
?>
```

5. Open/Closed Principle

The Open/Closed principle states a class must be open and closed where:

- open means a class has the ability to be extended.
- closed means a class is closed for modifications other than extension. The idea is once a class has been approved for use having gone through code reviews, unit tests, and other qualifying procedures, you don't want to change the class very much, just extend it.

The Open/Closed principle is a pitch for stability. A system is extended by adding new code not by changing already working code. Programmers often don't feel comfortable changing old code because it works! This principle just gives you an academic sounding justification for your fears :-)

In practice the Open/Closed principle simply means making good use of our old friends abstraction and polymorphism. Abstraction to factor out common processes and ideas. Inheritance to create an interface that must be adhered to by derived classes.

6. HTTP_*_VARS

HTTP_*_VARS are either enabled or disabled. When enabled all variables must be accessed through $HTTP_*_VARS[key]. When disabled all variables can be accessed by the key name.

- use HTTP_*_VARS when accessing variables.
- use enabled HTTP_*_VARS in PHP configuration.

Justification

- HTTP_*_VARS is available in any configuration.
- HTTP_*_VARS will not conflict with existing variables.
- Users can't change variables by passing values.

7. PHP File Extensions

There is lots of different extension variants on PHP files (.html, .php, .php3, .php4, .phtml, .inc, .class...).

- Use extension .html or .php.
- Always use the extension .php for your class and function libraries.
- Enable .html and .php files to be pharsed by PHP in your webserver configuration.

When you choose to use the .html extension on all your web documents, you should put all your libraries in files with the extension .php. When compared with the c language, the .c becomes .html and .h becomes .php.

Justification

- The extension describes what data the user will receive. Pharsed PHP becomes HTML. (Example: If you made a software who encoded mp3 files, you wouldn't use the extension .mysoft for the encoded files)
- The use of .inc or .class can be a security problem. On most servers these extensions aren't set to be run by a pharser. If these are accessed they will be displayed in clear text.

8. Miscellaneous

This section contains some miscellaneous do's and don'ts.

- Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).
- Avoid Using Variables When Accessing Files

Consider the following code:

```
// $lib_dir is an optional configuration
variable include($lib_dir . "functions.inc");
```

or worse still:

```
// $page is a variable from the
URL include($page);
```

The user could set the $lib_dir or $page variables and include files such as **/etc/passwd** or remote files such as **http://www.some-site.com/whatever.php** with malicious code. This malicious code could potentially delete files, corrupt databases, or change the values of variables used to track authentication status.

See Search code for the following functions: **readfile, fopen, file, include, require**

Avoid using variables as file names or if need to use variable as an file names then validate varibable with define file names using **define() or array().**

- Do Not Trust Global Variables for further steps on ensuring variables cannot be maliciously set.

9. Use if (0) to Comment Out Code Blocks (Guideline)

Sometimes large blocks of code need to be commented out for testing or while debugging a code. The easiest way to do this is with an if (0) block:

```
function example()
{
   great looking code

   if (0) {
   lots of code
   }

   more code
 }
```

You can't use **/**/** style comments because comments can't contain comments and surely a large block of your code will contain a comment, won't it?

10. Source Code Control System (SVN) Comments

Some issues to keep in mind:

- On checking in the code to repository comments should be written describing for what the change has been made and format of log comments should be like as following:

- Bug Fix: bug-id, brief description
- Change Request: SRS Number, Brief description

11. PHP Code Tags

PHP Tags are used for delimit PHP from html in a file. There are several ways to do this. <?php ?>, <? ?>, <script language="php"> </script>, <% %>, and <?=$name?>. Some of these may be turned off in your PHP settings.

- Use <?php ?> and PHP short tag <? ?> and <?=$variable?>

Justification

- <?php ?> is always available in any system and setup.

Example

```
<?php print "Hello world"; ?> // Will print "Hello world"
<? print "Hello world"; ?> // Will print "Hello world"
<script language="php"> print "Hello world"; </script> // Will print
"Hello world"

<% print "Hello world"; %> // Will print "Hello world"
<?=$street?> // Will print the value of the variable $street
```

12. No Magic Numbers

A magic number is a bare-naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

```
if       (22 == $foo) { start_thermo_nuclear_war(); }
else if (19 == $foo) { refund_lotso_money(); }
else if (16 == $foo) { infinite_loop(); }
else                 { cry_cause_im_lost(); }
```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Heavy use of magic numbers marks a programmer as an amateur more than anything else. Such a programmer has never worked in a team environment or has had to maintain code or they would never do such a thing.

Instead of magic numbers use a real name that means something. You should use define(). For example:

```
define("PRESIDENT_WENT_CRAZY", "22");
define("WE_GOOFED", "19");
define("THEY_DIDNT_PAY", "16");

if      (PRESIDENT_WENT_CRAZY == $foo) { start_thermo_nuclear_war(); }
else if (WE_GOOFED            == $foo)  { refund_lotso_money(); }
else if (THEY_DIDNT_PAY       == $foo)  { infinite_loop(); }
```

```
else                                    { happy_days_i_know_why_im_here(); }
```

Now isn't that better?

---

i