IT314 Software Engineering Team 7

Coding Conventions

Version 1.0

31 March, 2013

Winter 2012-13
DA-IICT, Gandhinagar

Document Revision History:

Version	Primary Author(s)	Description	Reviewed By	Date
1.0	Sonu, Surabhi, Nitish	Coding Conventions	Swena	31 March, 2013

Table of Contents

1. Introduction

1.1 Purpose

The goal of these guidelines is to create uniform coding habits among software personnel in the project teams so that reading, checking, and maintaining code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency, yet leave to the authors of the project source code, the freedom to practice their craft without unnecessary burden.

When a project adheres to common standards many good things happen:

- Programmers can go into any code and figure out what’s going on, so maintainability, readability, and reusability are increased.
- New people can get up to speed quickly.
- People new to a language are spared the need to develop a personal style and defend it to death.
- People new to a language are spared making the same mistakes over and over again, so reliability is increased.
- Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns – high productivity, maintainability, shared authorship, etc.
- People make fewer mistakes in consistent environments.

1.2 Scope

This document describes general software coding standards for code written in any text based programming language. Each language specific coding standard will be written to expand on these concepts with specific examples, and define additional guidelines unique to that language.

1.3 Coding Standard Documents

Each project shall adopt a set of coding standards consisting of three parts:

- General Coding Standard, described in this document
- Language specific coding standards for each language used. These language

standards shall supplement, rather than override, the General Coding standards as much as possible.

- Project Coding Standards. These standards shall be based on the coding standards in this document and on the coding standards for the given language(s). The project coding standards should supplement, rather than override, the General Coding standards and the language coding standards. Where conflicts between documents exist, the project standard shall be considered correct.

1.4 Other Related Project Documents

The 'Software Life Cycle' and 'Configuration Management' policy standards define a set of documents required for each project, along with a process for coordinating and maintaining them. Documents referred to in this report include:

- High level Design Document(HLDD)
- Low Level Design Document(LLDD)
- Configuration Management (CM)

1.4 Terms Used In Document

- The term 'program unit' means a single function, procedure, subroutine or, in the case of various languages, an include file, a package, a task, a Pascal unit, etc.
- A 'function' is a program unit whose primary purpose is to return a value.
- A 'procedure' is a program unit which does not return a value (except via output parameters).
- A 'subroutine' is any function or procedure.
- An 'identifier' is the generic term referring to a name for any constant, variable, or program unit.
- A 'module' is a collection of 'program units' that work on a common domain.

2. FILE and MODULE GUIDELINES

This section lists commonly used file suffixes and names.

2.1 File Suffixes

We will use the following file suffixes:

File Type	Suffix
PHP files	.php
HTML files	.html
CSS files	.css
Javascript files	.js
TTF/OTF files	.ttf/.otf
Sql Files	.sql

2.2 File Organization

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

2.3 Source Files

Wordpress has its own coding conventions which we have followed during coding.

2.3.1 PHP Source files

Single and Double Quotes

Use single and double quotes when appropriate. When not evaluating anything in the string, use single quotes. Like so:

```
echo '<a href="/static/link" title="Yeah yeah!">Link name</a>';

echo "<a href='$link' title='$linktitle'>$linkname</a>";
```

Indentation

Indentation should always reflect logical structure. Use **real tabs** and **not spaces.** The only exception is for mid-line indenting which should be done using spaces.

```
$foo   = 'somevalue';
```

```
$foo2  = 'somevalue2';

$foo34 = 'somevalue3';

$foo5  = 'somevalue4';
```

No Shorthand PHP tags

Many servers have shorthand tags disabled for PHP thus using full PHP tags is safer and best practice.

```
Correct:

        <?php ... ?>

Incorrect:

        <? ----- ?>

        <?=$var?>
```

Remove Trailing Spaces

Remove trailing spaces at the end of each line of code.

Naming Conventions

Use lowercase letters in variable, action and function names (never camelCase). Separate words via underscores. Don't abbreviate variable names un-necessarily; let the code be unambiguous and self-documenting.

```
function some_name ( $some_variable ) { [ … ] }
```

Class names should use capitalized words separated by underscores. Any acronyms should be all upper case.

```
class Walker_Category extends Walker { [ … ] }
class WP_HTTP { […] }
```

Files should be named descriptively using lowercase letters. Hyphens should separate words.

```
My_plugin_name.php
```

Class file names should be based on the class name with class- prepended and the underscores in the class name replaced with hyphens, for example WP_Error becomes:

```
Class-wp-error.php
```

Files containing template tags in wp-includes should have -template appended to the end of the name so that they are obvious.

```
General-template.php
```

Self-Explanatory Flag Values for Function Arguments

Prefer string values to just true and false when calling functions.

```
// Incorrect
function eat( $what, $slowly = true ) {
...
}
eat( 'mushrooms' );
eat( 'mushrooms', true ); // what does true mean?
eat( 'dogfood', false ); // what does false mean? The opposite of true?
```

Since PHP doesn't support named arguments, the values of the flags are meaningless and each time we come across a function call like these above we have to search for the function definition. The code can be made more readable by using descriptive string values, instead of booleans.

```
//correct
function eat( $what, $speed = 'slowly' ) {
. . .
}
eat( 'mushrooms' );
eat( 'mushrooms', 'slowly' );
eat( 'dogfood', 'fast' );
```

2.3.2 HTML Source Files

Self-Closing Elements

All tags must be properly closed. For tags that can wrap nodes such as text or other elements, termination is a trivial task. For tags that are self-closing, the forward slash should have exactly one space preceding it <br />. The **W3C** specifies that a single space should precede the self-closing slash.

Attributes and Tags

All tags and attributes must be written in lowercase. Additionally, attribute values should be lowercase when the purpose of the text therein is only to be interpreted by machines. For instances in which the data needs to be human readable, proper title capitalization should be followed, such as:

For machines:

```
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
```

For humans:

```
<a href="http://example.com/" title="Description Goes Here">Example.com</a>
```

Quotes

According to the W3C specifications for XHTML, all attributes must have a value, and must use double or single-quotes. The following are examples of proper and improper usage of quotes and attribute/value pairs.

Correct:

```
<input type="text" name="email" disabled="disabled" />

<input type='text' name='email' disabled='disabled' />
```

Incorrect:

```
<input type=text name=email disabled>
```

In HTML, attributes do not all have to have values, and attribute values do not always have to be quoted. All of the examples above are valid HTML.
However, WordPress uses XHTML (and where HTML5 is used, we use the XHTML serialisation rather than the HTML serialisation). Therefore, the HTML style should never be used.

Indentation
As with PHP, HTML indentation should always reflect logical structure. Use tabs and not spaces.

When mixing PHP and HTML together, indent PHP blocks to match the surrounding HTML code. Closing PHP blocks should match the same indentation level as the opening block.

Correct:

```
<?php if ( ! have_posts() ) : ?>
 <div id="post-1" class="post">
     <h1 class="entry-title">Not Found</h1>
     <div class="entry-content">
        <p>Apologies, but no results were found.</p>
        <?php get_search_form(); ?>
     </div>
   </div>
<?php endif; ?>
```

Incorrect:

```
                <?php if ( ! have_posts() ) : ?>
        <div id="post-0" class="post error404 not-found">
            <h1 class="entry-title">Not Found</h1>
            <div class="entry-content">
                <p>Apologies, but no results were found.</p>
<?php get_search_form(); ?
          </div>
        </div>
<?php endif; ?>
```

2.3.3 JavaScript Source Files

Braces
Braces should be used for multiline blocks in the style shown here:

```
var a, b;
if ( a && b ) {
   action1();
   action2();
} else if ( a ) {
   action3();
   action4();
} else {
   defaultAction();
}
```

Rule of thumb: Opening brace on the same line as the function definition, the conditional, or the loop. Closing brace on the line directly following the last statement of the block.

Quotations
Single quotes are preferred over double quotes (for the sake of simplicity); however, double quotes should be used when appropriate.

```
'Just your everyday string.';

// Use double quotes when a string contains single quotes
"Note the capital 'P' in WordPress";
```

Whitespace
Use spaces liberally throughout your code. Proper spacing drastically improves the legibility of any line of code.

Naming Conventions
Name functions and variables using camelCase, not underscores. This is one area where we differ from the WordPress PHP coding standards.

Constructors
Name constructors with TitleCase.

```
var myVariable = 5,
   MyConstructor = function() {
       this.attr = 'example';
   };
```

Filenames
Separate words in files using hyphens. Include both an uncompressed version with the format file-name.js, and a compressed version using the format file-name.min.js.

The var Keyword
Each function should begin with a single comma-delimited var statement that declares any local variables necessary. If a function does not declare a variable using var, it will be assigned to the current context (which is frequently the global scope, and worst scenario) and can unwittingly refer to and modify that data.

Assignments within the var statement should be listed on individual lines, while declarations can be grouped on a single line. Any additional lines should be indented with an additional tab.

```
var i, j, length,
   value = 'Entelechy';
```

Objects and functions that occupy more than a handful of lines should be assigned after the var statement.

Operators
Surround operators with spaces in order to properly show the order of operations:

```
var i;
i = ( 20 + 30 ) - 17;
```

Arrays
In JavaScript, associative arrays are defined as objects. Creating arrays in JavaScript should be done using the shorthand [] constructor rather than the new Array() notation.

```
var myArray = [];
```

One can initialize an array during construction:

```
var myArray = [ 1, 'Entelechy', 2, 'Edition1' ];
```

Space should be placed between after the opening bracket and before the closing bracket. Each index should be separated by a comma and a bracket.

Conditionals

Follow same rules as for PHP.

Loops

There are a variety of loop constructors offered by JavaScript each of which are similar to those offered by PHP. Though there's no preferred loop to use, there are standards by which you use each of them.

2.3.3 CSS Source Files

Terminology

```
selector {
    property: value;
}
```

The whole thing above is called "a style" or "a style definition". "Style name" may be used as a synonymous for "selector".
"Property-value pair" needs a better name.

Spacing
Generally spacing promotes readability, so let's take advantage of it.

Empty lines between style definitions
Allow the separate style definitions to breathe, by adding one (and only one) empty line between them, like:

```
.some-class {
    color: red;
}


.another-class {
   color: blue;
}
```

Curly Brackets
Curly brackets that embrace all property-value pairs in a style definition follow your favourite if-then-else style you'd use in other programming languages. The opening bracket may be on the same line as the selector identifier or it can go alone on the next line

```
.class-name {
    color: green;
}
```

Or

```
.class-name
```

```
{
    color: green;
}
```

End of Line
Good practice is to always end the lines (the property-value pairs) with semi-colon, even when not necessary (last pair in a selector). This way it's easier to add more properties later on, without even bothering to check if the previous property-value pair was properly terminated.

Naming Selectors

- The most important thing to have in mind is the content nature of the HTML document, not its presentation. Selector names should describe the content.
- Avoid presentation-specific words in the name, like "blue", "text-gray", or "light-box". What happens when you decide to change the color? The class "blue" is actually red?
- Cascade the name, using a dash (-) as a separator. When cascading, have in mind what the content is. What do I mean by cascading? For example: "header" and "header-logo" (you have a page header that contains a logo at, say, the top-left corner) or "footer" and "footer-copyright"
- Use full descriptive words. Abbreviating a word may save you a few milliseconds to type initially, but may make your code harder to read, costing you more time down the road. Why crypting "product" to "prod" or the so common "txt" (just spell it out as it is, "text")? Being consistent in this will save you the time spent thinking (or trial-error-ing) how exactly you abbreviated a word five days ago.
- Names are lowercase. There are browser problems with case sensitivity, so you're safe always lowercasing.
- Distinct words in the class name are separated by a dash. It's clear, readable and gives the underscore a well-deserved break. Use footer-copyright, not footer copyright, footerCopyright, FooterCopyRight, fOOtercopyRighT (god forbid!) or footer copyright. Well, if you're attached (married) to the underscore, I guess it's OK to keep using it, but consider leaving it for your Javascript code. The dash is not allowed in other languages' variable names, so use it when you can (in CSS), plus it's somewhat consistent with the property names (think font-size, background-color, etc.)

- Once again - use names that describe the content ("footer", "navigation", etc), rather than the presentation ("blue", "left", "big"...)

Comments
We all love to read other people's code comments. Writing our own comments is probably not as fun, but is highly encouraged in the name of maintainability. When you comment, use the /* comment here */ style. Line comments (the ones with // at the beginning) are not part of the CSS2 standard and may cause issues in some browser (or alternatively used as a workaround hack for some browsers).