Coding standards

Note: The Drupal Coding Standards apply to code within Drupal and its contributed modules. This document is loosely based on the PEAR Coding standards.
One overall note: comments and names should use US English spelling (e.g., "color" not "colour").

Drupal coding standards are version-independent and "always-current". All new code should follow the current standards, regardless of (core) version. Existing code in older versions *may* be updated, but doesn't necessarily have to be. Especially for larger code-bases (like Drupal core), updating the code of a previous version for the current standards may be too huge of a task. However, code in current versions should follow the current standards.
Note: Do not squeeze coding standards updates/clean-ups into otherwise unrelated patches. Only touch code lines that are actually relevant. To update existing code for the current standards, always create separate and dedicated issues and patches.
See the Helper modules section at the bottom of this page for information on modules that can review code for coding standards problems (and in some cases, even fix the problems).
Proposed changes to Drupal coding standards are first discussed in the Coding Standards issue queue.
Here is a helpful overview video tutorial, Understanding the Drupal Coding Standards, about the Drupal coding standards, why they are important, and how to use them.

Indenting and Whitespace

Use an indent of 2 spaces, with no tabs.

Lines should have no trailing whitespace at the end.

Files should be formatted with \n as the line ending (Unix line endings), not \r\n (Windows line endings).

All text files should end in a single newline (\n). This avoids the verbose "\ No newline at end of file" patch warning and makes patches easier to read since it's clearer what is being changed when lines are added to the end of a file.

All blocks at the beginning of a PHP file should be separated by a blank line. This includes the `/** @file */` block, the namespace declaration and the `use` statements (if present) as well as the subsequent code in the file. So, for example, a file header might look as follows:

```
<?php

namespace This\Is\The\Namespace;

use Drupal\foo\Bar;

/**
 * Provides examples.
 */
class ExampleClassName {
```

Or, for a non-class file (e.g., `.module`):

```
<?php

/**
 * @file
 * Provides example functionality.
 */

use Drupal\foo\Bar;

/**
 * Implements hook_help().
 */
function example_help($route_name) {
```

Operators

All binary operators (operators that come between two values), such as `+`, `-`, `=`, `!=`, `==`, `>`, etc. should have a space before and after the operator, for readability. For example, an assignment should be formatted as `$foo = $bar;` rather than `$foo=$bar;`. Unary operators (operators that operate on only one value), such as `++`, should not have a space between the operator and the variable or number they are operating on.
Checks for weak-typed inequality MUST use the `!=` operator. The `<>` operator MUST NOT be used in PHP code.

Casting

Put a space between the (type) and the $variable in a cast: `(int) $mynumber`.

Control Structures

Control structures include if, for, while, switch, etc. Here is a sample if statement, since it is the most complicated of them:

```
if (condition1 || condition2) {
  action1;
}
elseif (condition3 && condition4) {
  action2;
}
else {
  defaultaction;
}
```

(Note: Don't use "else if" -- always use elseif.)

Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

Always use curly braces even in situations where they are technically optional. Having them increases readability and decreases the likelihood of logic errors being introduced when new lines are added. The opening curly should be on the same line as the opening statement, preceded by one space. The closing curly should be on a line by itself and indented to the same level as the opening statement.

For switch statements:

```
switch (condition) {
  case 1:
    action1;
    break;

  case 2:
    action2;
    break;

  default:
    defaultaction;
}
```

For do-while statements:

```
do {
  actions;
} while ($condition);
```

Alternate control statement syntax for templates

In templates, the alternate control statement syntax using : instead of brackets is allowed. Note that there should not be a space between the closing parenthesis after the control keyword, and the colon, and HTML/PHP inside the control structure should be indented. For example:

```
<?php if (!empty($item)): ?>
  <p><?php print $item; ?></p>
<?php endif; ?>

<?php foreach ($items as $item): ?>
  <p><?php print $item; ?></p>
<?php endforeach; ?>
```

Line length and wrapping

The following rules apply to code. See Doxygen and comment formatting conventions for rules pertaining to comments.

- In general, all lines of code should not be longer than 80 characters.
- Lines containing longer function names, function/class definitions, variable declarations, etc are allowed to exceed 80 characters.
- Control structure conditions may exceed 80 characters, if they are simple to read and understand:

```
  if ($something['with']['something']['else']['in']['here'] == mymodule_check_something($whatever['else'])) {
    ...
  }
  if (isset($something['what']['ever']) && $something['what']['ever'] > $infinite && user_access('galaxy')) {
    ...
  }
  // Non-obvious conditions of low complexity are also acceptable, but should
  // always be documented, explaining WHY a particular check is done.
  if (preg_match('@(/|\\)(\.\.|~)@', $target) && strpos($target_dir, $repository) !== 0) {
```

- `        return FALSE;`
- `    }`
- Conditions should not be wrapped into multiple lines.
- Control structure conditions should also NOT attempt to win the *Most Compact Condition In Least Lines Of Code Award™*:

- `    // DON'T DO THIS!`
- `    if ((isset($key) && !empty($user->uid) && $key == $user->uid) || (isset($user->cache) ? $user->cache : '') == ip_address() || isset($value) && $value >= time())) {`
- `        ...`
- `    }`

Instead, it is recommended practice to split out and prepare the conditions separately, which also permits documenting the underlying reasons for the conditions:

```
    // Key is only valid if it matches the current user's ID, as otherwise other
    // users could access any user's things.
    $is_valid_user = isset($key) && !empty($user->uid) && $key == $user->uid;

    // IP must match the cache to prevent session spoofing.
    $is_valid_cache = isset($user->cache) ? $user->cache == ip_address() : FALSE;

    // Alternatively, if the request query parameter is in the future, then it
    // is always valid, because the galaxy will implode and collapse anyway.
    $is_valid_query = $is_valid_cache || (isset($value) && $value >= time());

    if ($is_valid_user || $is_valid_query) {
        ...
    }
```

Note: This example is still a bit dense. Always consider and decide on your own whether people unfamiliar with your code will be able to make sense of the logic.

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:

```
$var = foo($bar, $baz, $quux);
```

Function Declarations

```
function funstuff_system($field) {
    $system["description"] = t("This module inserts funny text into posts randomly.");
    return $system[$field];
}
```

Arguments with default values go at the end of the argument list. Always attempt to return a meaningful value from a function if one is appropriate.

Anonymous functions should have a space between "function" and its parameters, as in the following example:

```
array_map(function ($item) use ($id) {
    return $item[$id];
}, $items);
```

Class Constructor Calls

When calling class constructors with no arguments, always include parentheses:

```
$foo = new MyClassName();
```

This is to maintain consistency with constructors that have arguments:

```
$foo = new MyClassName($arg1, $arg2);
```

Note that if the class name is a variable, the variable will be evaluated first to get the class name, and then the constructor will be called. Use the same syntax:

```
$bar = 'MyClassName';
$foo = new $bar();
$foo = new $bar($arg1, $arg2);
```

Arrays

Arrays should be formatted using short array syntax with a space separating each element (after the comma), and spaces around the => key association operator, if applicable:

```
$some_array = ['hello', 'world', 'foo' => 'bar'];
```

Note that if the line declaring an array spans longer than 80 characters (often the case with form and menu declarations), each element should be broken into its own line, and indented one level:

```
$form['title'] = [
  '#type' => 'textfield',
  '#title' => t('Title'),
  '#size' => 60,
  '#maxlength' => 128,
  '#description' => t('The title of your node.'),
];
```

Note the comma at the end of the last array element; This is not a typo! It helps prevent parsing errors if another element is placed at the end of the list later.

Please note, short array syntax is unsupported in versions of PHP prior to 5.4. This means that Drupal 7 core and Drupal 7 contributed projects without an explicit PHP 5.4+ requirement must use long array syntax.

Quotes

Drupal does not have a hard standard for the use of single quotes vs. double quotes. Where possible, keep consistency within each module, and respect the personal style of other developers.

With that caveat in mind, single quote strings should be used by default. Their use is recommended except in two cases:

1. Deliberate in-line variable interpolation, e.g. "<h2>$header</h2>".
2. Translated strings where one can avoid escaping single quotes by enclosing the string in double quotes. One such string would be "He's a good person." It would be 'He\'s a good person.' with single quotes. Such escaping may not be handled properly by .pot file generators for text translation, and it's also somewhat awkward to read.

String Concatenations

Always use a space between the dot and the concatenated parts to improve readability.

```
<?php
  $string = 'Foo' . $bar;
  $string = $bar . 'foo';
  $string = bar() . 'foo';
  $string = 'foo' . 'bar';
?>
```

When you concatenate simple variables, you can use double quotes and add the variable inside; otherwise, use single quotes.

```
<?php
  $string = "Foo $bar";
?>
```

When using the concatenating assignment operator ('.='), use a space on each side as with the assignment operator:

```
<?php
$string .= 'Foo';
$string .= $bar;
$string .= baz();
?>
```

Comments

Comment standards are discussed on the separate Doxygen and comment formatting conventions page.

I
ncluding Code

Anywhere you are unconditionally including a class file, use `require_once()`. Anywhere you are conditionally including a class file (for example, factory methods), use `include_once()`. Either of these will ensure that class files are included only once. They share the same file list, so you don't need to worry about mixing them - a file included with `require_once()` will not be included again by `include_once()`.

Note: `include_once()` and `require_once()` are statements, not functions. You don't need parentheses around the file name to be included.

When including code from the same directory or a sub-directory, start the file path with ".":
`include_once ./includes/mymodule_formatting.inc`

In Drupal 7.x and later versions, use DRUPAL_ROOT:

```
require_once DRUPAL_ROOT . '/' . variable_get('cache_inc', 'includes/cache.inc');
```

To include code in a module:

```
module_load_include('inc', 'node', 'node.admin');
```

PHP Code Tags

Always use `<?php ?>` to delimit PHP code, not the shorthand, `<? ?>`. This is required for Drupal compliance and is also the most portable way to include PHP code on differing operating systems and set-ups.

Note that as of Drupal 4.7, the `?>` at the end of code files is purposely omitted. This includes for module and include files. The reasons for this can be summarized as:

- Removing it eliminates the possibility for unwanted whitespace at the end of files which can cause "header already sent" errors, XHTML/XML validation issues, and other problems.
- The closing delimiter at the end of a file is optional.
- PHP.net itself removes the closing delimiter from the end of its files (example: prepend.inc), so this can be seen as a "best practice."

Semicolons

The PHP language requires semicolons at the end of most lines, but allows them to be omitted at the end of code blocks. Drupal coding standards require them, even at the end of code blocks. In particular, for one-line PHP blocks:

```
<?php print $tax; ?>  --  YES
<?php print $tax ?>  --  NO
```

Example URLs

Use "example.com" for all example URLs, per RFC 2606.

Naming Conventions

Functions and variables

Functions should be named using lowercase, and words should be separated with an underscore. Functions should in addition have the grouping/module name as a prefix, to avoid name collisions between modules.

Variables should be named using lowercase, and words should be separated either with uppercase characters (example: `$lowerCamelCase`) or with an underscore (example: `$snake_case`). Be consistent; do not mix camelCase and snake_case variable naming inside a file.

Persistent Variables

Persistent variables (variables/settings defined using Drupal's variable_get()/variable_set() functions) should be named using all lowercase letters, and words should be separated with an underscore. They should use the grouping/module name as a prefix, to avoid name collisions between modules.

Constants

- Constants should always be all-uppercase, with underscores to separate words. (This includes pre-defined PHP constants like `TRUE`, `FALSE`, and `NULL`.)
- Module-defined constant names should also be prefixed by an uppercase spelling of the module that defines them.
- In Drupal 8 and later, constants should be defined using the `const` PHP language keyword (instead of `define()`), because it is better for performance:

```
/**
 * Indicates that the item should be removed at the next general cache wipe.
 */
 const CACHE_TEMPORARY = -1;
```

Note that `const` does not work with PHP expressions. `define()` should be used when defining a constant conditionally or with a non-literal value:

```
if (!defined('MAINTENANCE_MODE')) {
  define('MAINTENANCE_MODE', 'error');
}
```

Global Variables

If you need to define global variables, their name should start with a single underscore followed by the module/theme name and another underscore.

Classes

All standards related to classes and interfaces, including naming, are covered on http://drupal.org/node/608152 instead of here.

File names

All documentation files should have the file name extension ".txt" to make viewing them on Windows systems easier. Also, the file names for such files should be all-caps (e.g. README.txt instead of readme.txt) while the extension itself is all-lowercase (i.e. txt instead of TXT).

Examples: README.txt, INSTALL.txt, TODO.txt, CHANGELOG.txt etc.

Helper Modules

There are several contributed modules/projects available to assist with review for coding standards compliance:

- Coder module, which includes both Coder Review (reviews) and Coder Upgrade (updates your code). To use it:

1. Install the module (like any other module)
2. Click on the "Code Review" link in your navigation menu.
3. Scroll down to "Select Specific Modules".

4. Select the module you wish to review, and click the "Submit" button.
   As an alternative to starting from the Code Review link in navigation, you can also review a particular module's code by clicking on the link on the Modules admin screen.

- Dreditor (a browser plug-in for reviewing patches and more)
- PAReview (a set of scripts for reviewing project applications, which runs some coding tests)
- Coder Sniffer (runs coding standards validation without loading drupal)
- The Grammar Parser module provides an automated way of rewriting code files in compliance with code standards. You'll probably also need the Grammar Parser UI module.

REFERENCES :

https://chromatichq.com/blog/drupal-code-standards-how-do-we-implement-them

http://italomairo.italomairo.com/it/content/drupal-coding-standards-mamp-code-sniffer-and-php-storm