Cuprins

1.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.

2.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.

3.Casting

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

4.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;
}
```

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

You are strongly encouraged to 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.

For switch statements:

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

```
  case 2:
    action2;
    break;

  default:
    defaultaction;
}
```

For do-while statements:

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

5.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 chars.
- Lines containing longer function names, function/class definitions, variable declarations, etc are allowed to exceed 80 chars.
- Control structure conditions may exceed 80 chars, 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.
  $is_valid_query = (isset($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.

6.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);
```

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:

```
$short         = foo($bar);
$long_variable = foo($baz);
```

7.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.

8.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);
```

9.Arrays

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

```
$some_array = 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'] = array(
  '#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.

10.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 are known to be faster because the parser doesn't have to look for in-line variables. Their use is recommended except in two cases:

1. In-line variable usage, 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.

11.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();
?>
```

12.Comments

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

13.Including 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');
```

14.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."

15.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
```

16.Example URLs

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

17.Naming Conventions

Functions and variables

Functions and variables 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.

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:

  ```
  <?php
  /**
  * 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:

```
<?php
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

Classes should be named using "CamelCase." For example:

```
<?php
abstract class DatabaseConnection extends PDO {
?>
```

Class methods and properties should use "lowerCamelCase":

```
<?php
public $lastStatement;
?>
```

The use of *private* class methods and properties should be avoided -- use *protected* instead, so that another class could extend your class and change the method if necessary. Protected (and public) methods and properties should *not* use an underscore prefix, as was common in PHP 4-era code.

For more information on class and OO standards, see the more detailed coverage.

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.

18.Helper Module

There is a contributed module for assisting with code review. To use this module you must complete the following steps:

- Install the Coder module.
- Click on the "Code Review" link in your navigation menu.
- Scroll down to "Select Specific Modules".
- 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.

The Coder module also comes with a command-line script called "Coder Format", which will not only check your files for standards compliance, but fix them. Use with care!

- Doxygen and comment formatting conventions
- Namespaces
- Object-oriented code
- PHP Exceptions
- SQL coding conventions
- Temporary placeholders and delimiters
- Use Drupal Unicode functions for strings
- Write E_ALL compliant code
- Drupal SimpleTest coding standards
- Drupal Markup Style Guide
- CSS coding standards
- JavaScript coding standards

‹ Standards, security and best practices up Doxygen and comment formatting conventions ›

Comments

Naming Conventions for files missing

Posted by Jax on *February 25, 2010 at 10:23am*

In the Naming Conventions, in the File names section there is mentioned how documentation file names should be formatted but no word on the naming conventions for php, css, js files.

- Login or register to post comments

also conventions for .tpl.php

Posted by xibun on *July 27, 2010 at 10:47am*

also conventions for .tpl.php should be listed there in case there are any (see this related post)

- Login or register to post comments

and class files

Posted by danielb on *December 25, 2011 at 8:12am*

and class files

- Login or register to post comments

Add "Module documentation guidelines"

Posted by drupalshrek on *November 20, 2010 at 3:33pm*

I think this page (under "Comments") should have a sentence such as:

Drupal modules should follow the "Module documentation guidelines".

- Login or register to post comments

Line wraps are not mentioned

Posted by erikwebb on *December 6, 2010 at 4:48pm*

Line wraps are not mentioned anywhere. Even if they are never to be used (as I assume), they need to be mentioned as such.

Erik Webb
Technical Consultant, Acquia

- Login or register to post comments

line wraps

Posted by cheekdotcom on *June 2, 2011 at 3:27pm*

Agree they need to be mentioned. My 2 cents are hard-wrap at 80 characters, but that's my preference. I did notice a mention of this in docs at http://drupal.org/node/161085, but I assume that's just for docs, not code.

Joseph Cheek
Drupal Architect, US Dept. of Education

- Login or register to post comments

Video on Why Coding Standards Are Important

Posted by chrisshattuck on *January 28, 2011 at 6:45pm*

If you need more convincing, check out this illustrated video about why coding standards are so important on Build a Module.com.

I've recorded over 500 focused Drupal video tutorials at Build a Module.com.

- Login or register to post comments

String concatenation

Posted by IRuslan on *August 17, 2011 at 3:49am*

I'm confused about this examples:

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

In this example we should put space between ' and dot

But in most parts of drupal core i see, that space is not putted.
So it should be:

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

Where is the truth?

- Login or register to post comments

policy change

Posted by cburschka on *September 15, 2011 at 3:22pm*

This reflects a change in the coding standards. At an earlier time, no space was placed between literal strings and concatenation periods. Spaces should now be inserted everywhere.

This has been fixed in nearly all places in core, but several lines must still be updated.

- Login or register to post comments

Just found a small code error

Posted by net02 on *October 24, 2011 at 4:43pm*

Just found a small code error which won't be misleading being just a coding standard example, but it would be a 5 secs fix.
In the "Line lenght and wrapping" section, the last example code has a small error. Within the example context this variable

```
$is_valid_query = (isset($user->cache) ? $user->cache == ip_address() :
FALSE);
```

should be called

```
$is_valid_cache = (isset($user->cache) ? $user->cache == ip_address() :
FALSE);
```

as it's mentioned in the following line.

- Login or register to post comments

Extra White Space

Posted by gamelodge on *November 10, 2011 at 12:15am*

Under the operators section it mentions that white spaces should be added before and after, is there a limit to the white space before?

For example I like to align any consecutive assignments for readability, this introduces extra white spaces and it fails the code sniffer test.

```
// This may not come out as expected here, but the ideas is it should line
up.
$foo     = '4';
$foobar  = '7';
$foob    = '2';
```

I have added a check to my codesniffer code to allow for extra white spaces... (SpaceOperatorSniff.php)

```
public function process(PHP_CodeSniffer_File $phpcsFile, $stackPtr)
    {
        $tokens = $phpcsFile->getTokens();
// Check if there is at least one space there may be more
if(ltrim($tokens[($stackPtr - 1)]['content']) != $tokens[($stackPtr - 1)]
['content']) {
$whitespace = true;
}

        if ($tokens[($stackPtr + 1)]['code'] !== T_WHITESPACE
            || $tokens[($stackPtr + 1)]['content'] != ' '
        ) {
            $error = 'A opeator statement must be followed by a single
space';
            $phpcsFile->addError($error, $stackPtr);
        }
        if ($tokens[($stackPtr - 1)]['code'] !== T_WHITESPACE
            || !$whitespace
        ) {
            $error = 'There must be a single space befora a opeator
statement';
            $phpcsFile->addError($error, $stackPtr);
        }

    }//end process()
```

- Login or register to post comments

btw, I think it's a bad idea

Posted by Jax on *November 10, 2011 at 8:52am*

btw, I think it's a bad idea to align consecutive statements because once you introduce a variable with a name that is longer than the previous ones you have to re-align all the other lines as well which reduces the readability of the patch you create afterwards.

- Login or register to post comments

Newline at file end

Posted by doitDave on *November 18, 2011 at 3:23pm*

Since there was more than one recent discussion on that in the project appications review queue: The rule defined in this article obviously causes some confusion.

While the article says

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.

it obviously means that, at the end of any file, a ***blank line*** should exist. Actually a blank line, in UNIX text files, means ***two newlines***:

```
First line\n
Second line\n
....
Last line\n
\n
```

So I urgently suggest clarification in the document. Many people consider "newline" as an ASCII symbol (\n) and not as a synonym for a blank line.

- Login or register to post comments

No. What the article says is

Posted by TR on *November 19, 2011 at 5:21pm*

No. What the article says is exactly what it means - "All text files should end in a single newline (\n)." That does not mean TWO \n characters, it means one. The explanation given in the article makes it clear what why this is desired. When in doubt, look at all the core files and you will see without exception that they all end in exactly one \n. There should never be a blank line at the end of any file.

`<tr>.`

- Login or register to post comments

This should be discussed with

Posted by doitDave on *November 19, 2011 at 5:47pm*

This should be discussed with the folks working on drupalcs module then. Since it throws errors otherwise. It leads to many confusion in the project applications queue atm (that's why I commented here ;))

(Much core code wouldn't pass hardly any current coding standard IMO, btw, so I would not really make it a role model... but however, there seems to be a big need of discussion on this.)

Edit: I have raised this here #1346946: Newlines at file endings. My only goal is an end of this confusion, so hopefully this will soon be clear.

- Login or register to post comments

We can probably update that

Posted by jthorson on *November 19, 2011 at 9:36pm*

EDIT: I believe wrongly. :)

We can probably clarify that page ... but I the intention is that all files should have "only one sequential newline character at the end of the file".

In other words, files should *NOT* end with "\n" alone on the final line.

- Login or register to post comments

missing link to http://groups.drupal.org/coding-standards-and-be

Posted by Thomas_Zahreddin on *December 9, 2011 at 6:00pm*

telling people the place to join the discussion:

join http://groups.drupal.org/coding-standards-and-best-practices

Thomas

Combining IT and arts to organize
http://it-arts.org

- Login or register to post comments

Coding standards in template files

Posted by IRuslan on *January 5, 2012 at 3:45pm*

Are there any standards for templates?
I didn't found information about it.
If this page exists, i think it should be accessible from this one.

E.g. how we should write this lines:

```
<?php if (!empty($some_var)): ?>
  <h2>This is some var value</h2>
  <div><?php print check_plain($some_var); ?></div>
<?php endif; ?>
```

Or like this (note a space before colon):

```
<?php if (!empty($some_var)) : ?>
  <h2>This is some var value</h2>
  <div><?php print check_plain($some_var); ?></div>
<?php endif; ?>
```

D7 core was contained both variants prior last releases.
As i saw last release contains only "spaceless" option, but it's not documented here.

- Login or register to post comments

Page status

No known problems

Log in to edit this page

About this page

Drupal version
Drupal 7.x, Drupal 8.x
Audience

Developers and coders

Develop for Drupal

- Setting up a development environment
- Module developer's guide
- Working with the Drupal API
- Related programs and applications
- SimpleTest
- Standards, security and best practices
    - Coding standards
        - Doxygen and comment formatting conventions
        - Namespaces
        - Object-oriented code
        - PHP Exceptions
        - SQL coding conventions
        - Temporary placeholders and delimiters
        - Use Drupal Unicode functions for strings
        - Write E_ALL compliant code
        - Drupal SimpleTest coding standards
        - Drupal Markup Style Guide
        - CSS coding standards
        - JavaScript coding standards
    - Writing secure code
    - Core Theme Candidate Requirements
    - Documentation of Projects
    - Programming best practices
- User interface standards
- Developing installation profiles
- Guidelines for SQL