**Single and Double Quotes**

Use single and double quotes when appropriate. If you’re not evaluating anything in the string, use single quotes. You should almost never have to escape quotes in a string, because you can just alternate your quoting style, like so:

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

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

Text that goes into attributes should be run through esc\_attr() so that single or double quotes do not end the attribute value and invalidate the HTML and cause a security issue. See Data Validation in the Codex for further details.

**Indentation**

Your indentation should always reflect logical structure. Use real tabs and not spaces, as this allows the most flexibility across clients.

Exception: if you have a block of code that would be more readable if things are aligned, use spaces:

[tab]$foo = 'somevalue';

[tab]$foo2 = 'somevalue2';

[tab]$foo34 = 'somevalue3';

[tab]$foo5 = 'somevalue4';

For associative arrays, each item should start on a new line when the array contains more than one item:

$query = new WP\_Query( array( 'ID' => 123 ) );

$query = new WP\_Query( array(

[tab]'post\_type' => 'page',

[tab]'post\_author' => 123,

[tab]'post\_status' => 'publish',) );

Note the comma after the last array item: this is recommended because it makes it easier to change the order of the array, and makes for cleaner diffs when new items are added.

$my\_array = array(

[tab]'foo' => 'somevalue',

[tab]'foo2' => 'somevalue2',

[tab]'foo3' => 'somevalue3',

[tab]'foo34' => 'somevalue3',

);

For switch structures case should indent one tab from the switch statement and break one tab from the case statement.

switch ( $type ) {

[tab]case 'foo':

[tab][tab]some\_function();

[tab][tab]break;

[tab]case 'bar':

[tab][tab]some\_function();

[tab][tab]break;

}

Rule of thumb: Tabs should be used at the beginning of the line for indentation, while spaces can be used mid-line for alignment.

**Brace Style**

Braces shall be used for all blocks in the style shown here:

if ( condition ) {

action1();

action2();

} elseif ( condition2 && condition3 ) {

action3();

action4();

} else {

defaultaction();

}

If you have a really long block, consider whether it can be broken into two or more shorter blocks, functions, or methods, to reduce complexity, improve ease of testing, and increase readability.

Braces should always be used, even when they are not required:

if ( condition ) {

action0();

}

if ( condition ) {

action1();

} elseif ( condition2 ) {

action2a();

action2b();

}

foreach ( $items as $item ) {

process\_item( $item );

}

Note that requiring the use of braces just means that single-statement inline control structures are prohibited. You are free to use the alternative syntax for control structures (e.g. if/endif, while/endwhile)—especially in your templates where PHP code is embedded within HTML, for instance:

<?php if ( have\_posts() ) : ?>

<div class="hfeed">

<?php while ( have\_posts() ) : the\_post(); ?>

<article id="post-<?php the\_ID() ?>" class="<?php post\_class() ?>">

<!-- ... -->

</article>

<?php endwhile; ?>

</div>

<?php endif; ?>

Use elseif, not else if #Use elseif, not else if

else if is not compatible with the colon syntax for if|elseif blocks. For this reason, use elseif for conditionals.

**Regular Expressions**

Perl compatible regular expressions (PCRE, preg\_ functions) should be used in preference to their POSIX counterparts. Never use the /e switch, use preg\_replace\_callback instead.

It’s most convenient to use single-quoted strings for regular expressions since, contrary to double-quoted strings, they have only two metasequences: \' and \\.

Opening and Closing PHP Tags #Opening and Closing PHP Tags

When embedding multi-line PHP snippets within a HTML block, the PHP open and close tags must be on a line by themselves.

Correct (Multiline):

function foo() {

?>

<div>

<?php

echo bar(

$baz,

$bat

);

?>

</div>

<?php

}

Correct (Single Line):

<input name="<?php echo esc\_attr( $name ); ?>" />

Incorrect:

if ( $a === $b ) { ?>

<some html>

<?php }

No Shorthand PHP Tags #No Shorthand PHP Tags

Important: Never use shorthand PHP start tags. Always use full PHP tags.

Correct:

<?php ... ?>

<?php echo $var; ?>

Incorrect:

<? ... ?>

<?= $var ?>

Remove Trailing Spaces #Remove Trailing Spaces

Remove trailing whitespace at the end of each line of code. Omitting the closing PHP tag at the end of a file is preferred. If you use the tag, make sure you remove trailing whitespace.

**Space Usage**

Always put spaces after commas, and on both sides of logical, comparison, string and assignment operators.

x == 23

foo && bar

! foo

array( 1, 2, 3 )

$baz . '-5'

$term .= 'X'

Put spaces on both sides of the opening and closing parenthesis of if, elseif, foreach, for, and switch blocks.

foreach ( $foo as $bar ) { ...

When defining a function, do it like so:

function my\_function( $param1 = 'foo', $param2 = 'bar' ) { ...

When calling a function, do it like so:

my\_function( $param1, func\_param( $param2 ) );

When performing logical comparisons, do it like so:

if ( ! $foo ) { ...

When type casting, do it like so:

foreach ( (array) $foo as $bar ) { ...

$foo = (boolean) $bar;

When referring to array items, only include a space around the index if it is a variable, for example:

$x = $foo['bar']; // correct

$x = $foo[ 'bar' ]; // incorrect

$x = $foo[0]; // correct

$x = $foo[ 0 ]; // incorrect

$x = $foo[ $bar ]; // correct

$x = $foo[$bar]; // incorrect

In a switch block, there must be no space before the colon for a case statement.

switch ( $foo ) {

case 'bar': // correct

case 'ba' : // incorrect

}

Similarly, there should be no space before the colon on return type declarations.

function sum( $a, $b ): float {

return $a + $b;

}

**Formatting SQL statements**

When formatting SQL statements you may break it into several lines and indent if it is sufficiently complex to warrant it. Most statements work well as one line though. Always capitalize the SQL parts of the statement like UPDATE or WHERE.

Functions that update the database should expect their parameters to lack SQL slash escaping when passed. Escaping should be done as close to the time of the query as possible, preferably by using $wpdb->prepare()

$wpdb->prepare() is a method that handles escaping, quoting, and int-casting for SQL queries. It uses a subset of the sprintf() style of formatting. Example :

$var = "dangerous'"; // raw data that may or may not need to be escaped

$id = some\_foo\_number(); // data we expect to be an integer, but we're not certain

$wpdb->query( $wpdb->prepare( "UPDATE $wpdb->posts SET post\_title = %s WHERE ID = %d", $var, $id ) );

%s is used for string placeholders and %d is used for integer placeholders. Note that they are not ‘quoted’! $wpdb->prepare() will take care of escaping and quoting for us. The benefit of this is that we don’t have to remember to manually use esc\_sql(), and also that it is easy to see at a glance whether something has been escaped or not, because it happens right when the query happens.

**Database Queries**

Avoid touching the database directly. If there is a defined function that can get the data you need, use it. Database abstraction (using functions instead of queries) helps keep your code forward-compatible and, in cases where results are cached in memory, it can be many times faster.

If you must touch the database, get in touch with some developers by posting a message to the wp-hackers mailing list. They may want to consider creating a function for the next WordPress version to cover the functionality you wanted.

**Naming Conventions**

Use lowercase letters in variable, action/filter, and function names (never camelCase). Separate words via underscores. Don’t abbreviate variable names unnecessarily; 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 { [...] }

Constants should be in all upper-case with underscores separating words:

define( 'DOING\_AJAX', true );

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

This file-naming standard is for all current and new files with classes. There is one exception for three files that contain code that got ported into BackPress: class.wp-dependencies.php, class.wp-scripts.php, class.wp-styles.php. Those files are prepended with class., a dot after the word class instead of a hyphen.

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 #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 the examples 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', 'quickly' );

When more words are needed to describe the function parameters, an $args array may be a better pattern.

// Even Better

function eat( $what, $args ) {

...

}

eat ( 'noodles', array( 'speed' => 'moderate' ) );

Interpolation for Naming Dynamic Hooks #Interpolation for Naming Dynamic Hooks

Dynamic hooks should be named using interpolation rather than concatenation for readability and discoverability purposes.

Dynamic hooks are hooks that include dynamic values in their tag name, e.g. {$new\_status}\_{$post->post\_type} (publish\_post).

Variables used in hook tags should be wrapped in curly braces { and }, with the complete outer tag name wrapped in double quotes. This is to ensure PHP can correctly parse the given variables’ types within the interpolated string.

do\_action( "{$new\_status}\_{$post->post\_type}", $post->ID, $post );

Where possible, dynamic values in tag names should also be as succinct and to the point as possible. $user\_id is much more self-documenting than, say, $this->id.