JavaScript Coding Standards

- Code Refactoring
- Spacing
  - Object Declarations
  - Arrays and Function Calls
  - Examples of Good Spacing
- Semicolons
- Indentation and Line Breaks
  - Blocks and Curly Braces
  - Multi-line Statements
  - Chained Method Calls
- Assignments and Globals
  - Declaring Variables with const and let
  - Declaring Variables With var
  - Globals
  - Common Libraries
- Naming Conventions
  - Abbreviations and Acronyms
  - Class Definitions
  - Constants
- Comments
- Equality
- Type Checks
- Strings
- Switch Statements
- Best Practices
  - Arrays
  - Objects
  - Iteration

    - Underscore.js Collection Functions
    - Iterating Over jQuery Collections
- JSHint
    - Installing and Running JSHint
    - JSHint Settings
    - Target A Single File
    - JSHint Overrides: Ignore Blocks
- Credits

JavaScript has become a critical component in developing WordPress-based applications (themes and plugins) as well as WordPress core. Standards are needed for formatting and styling JavaScript code to maintain the same code consistency as the WordPress standards provide for core PHP, HTML, and CSS code.

> All code in any code-base should look like a single person typed it, no matter how many people contributed. – Principles of Writing Consistent, Idiomatic JavaScript

The WordPress JavaScript Coding Standards are adapted from the jQuery JavaScript Style Guide. Our standard differs from the jQuery guidelines in the following ways:

- WordPress uses single quotation marks for string declarations.
- Case statements are indented within switch blocks.
- Function contents are consistently indented, including full-file closure wrappers.
- Some whitespace rules differ, for consistency with the WordPress PHP coding standards.
- jQuery's 100-character hard line limit is encouraged, but not strictly enforced.

Many of the examples below have been adapted directly from the jQuery style guide; these differences have all been integrated into the examples on this page. Any of the below standards and examples should be considered best practice for WordPress code, unless explicitly noted as anti-patterns.

Code Refactoring

>

> “Code refactoring should not be done just because we can.” – Lead Developer Andrew Nacin

Many parts of the WordPress code structure for JavaScript are inconsistent in their style. WordPress is working to gradually improve this, so the code will be clean and easy to read at a glance. While the coding standards are important, refactoring older .js files simply to conform to the standards is not an urgent issue. “Whitespace-only” patches for older files are strongly discouraged. All new or updated JavaScript code will be reviewed to ensure it conforms to the standards, and passes JSHint.

Spacing

Use spaces liberally throughout your code. “When in doubt, space it out.” These rules encourage liberal spacing for improved developer readability. The minification process creates a file that is optimized for browsers to read and process.

- Indentation with tabs.
- No whitespace at the end of line or on blank lines.
- Lines should usually be no longer than 80 characters, and should not exceed 100 (counting tabs as 4 spaces). *This is a “soft” rule, but long lines generally indicate unreadable or disorganized code.*
- `if`/`else`/`for`/`while`/`try` blocks should always use braces, and always go on multiple lines.
- Unary special-character operators (e.g., `++`, `--`) must not have space next to their operand.
- Any `,` and `;` must not have preceding space.
- Any `;` used as a statement terminator must be at the end of the line.
- Any `:` after a property name in an object definition must not have preceding space.
- The `?` and `:` in a ternary conditional must have space on both sides.
- No filler spaces in empty constructs (e.g., `{}`, `[]`, `fn()`).
- There should be a new line at the end of each file.
- Any `!` negation operator should have a following space.*
- All function bodies are indented by one tab, even if the entire file is wrapped in a closure.*
- Spaces may align code within documentation blocks or within a line, but only tabs should be used at the start of a line.*

*: The WordPress JavaScript standards prefer slightly broader whitespace rules than the jQuery style guide. These deviations are for consistency between the PHP and JavaScript files in the WordPress codebase. Whitespace can easily accumulate at the end of a line – avoid this, as trailing whitespace is caught as an error in JSHint. One way to catch whitespace buildup is enabling visible whitespace characters within your text editor.

Object Declarations

Object declarations can be made on a single line if they are short (remember the line length guidelines). When an object declaration is too long to fit on one line, there must be one property per line and each line ended by a comma. Property names only need to be quoted if they are reserved words or contain special characters: Arrays can be declared on a single line if they are short (remember the line length guidelines). When an array is too long to fit on one line, each member must be placed on its own line and each line ended by a comma.

```
// Preferred
var obj = {
    ready: 9,
    when: 4,
    'you are': 15,
};
var arr = [
    9,
    4,
    15,
];

// Acceptable for small objects and arrays
var obj = { ready: 9, when: 4, 'you are': 15 };
var arr = [ 9, 4, 15 ];

// Bad
var obj = { ready: 9,
    when: 4, 'you are': 15 };
var arr = [ 9,
    4, 15 ];
```

Arrays and Function Calls

Always include extra spaces around elements and arguments:

```
array = [ a, b ];

foo( arg );
```

```
foo( 'string', object );

foo( options, object[ property ] );

foo( node, 'property', 2 );

prop = object[ 'default' ];

firstArrayElement = arr[ 0 ];
```

Examples of Good Spacing

```
var i;

if ( condition ) {
    doSomething( 'with a string' );
} else if ( otherCondition ) {
    otherThing( {
        key: value,
        otherKey: otherValue
    } );
} else {
    somethingElse( true );
}

// Unlike jQuery, WordPress prefers a space after the ! negation operator.
// This is also done to conform to our PHP standards.
while ( ! condition ) {
    iterating++;
}

for ( i = 0; i < 100; i++ ) {
    object[ array[ i ] ] = someFn( i );
    $( '.container' ).val( array[ i ] );
}

try {
    // Expressions
```

```
} catch ( e ) {
    // Expressions
}
```

Semicolons

Use them. Never rely on Automatic Semicolon Insertion (ASI).

Indentation and Line Breaks

Indentation and line breaks add readability to complex statements. Tabs should be used for indentation. Even if the entire file is contained in a closure (i.e., an immediately invoked function), the contents of that function should be indented by one tab:

```
( function ( $ ) {
    // Expressions indented

    function doSomething() {
        // Expressions indented
    }
} )( jQuery );
```

Blocks and Curly Braces

`if`, `else`, `for`, `while`, and `try` blocks should always use braces, and always go on multiple lines. The opening brace should be on the same line as the function definition, the conditional, or the loop. The closing brace should be on the line directly following the last statement of the block.

```
var a, b, c;

if ( myFunction() ) {
    // Expressions
} else if ( ( a && b ) || c ) {
    // Expressions
} else {
    // Expressions
```

```
}
```

Multi-line Statements

When a statement is too long to fit on one line, line breaks must occur after an operator.

```
// Bad
var html = '<p>The sum of ' + a + ' and ' + b + ' plus ' + c
    + ' is ' + ( a + b + c ) + '</p>';

// Good
var html = '<p>The sum of ' + a + ' and ' + b + ' plus ' + c +
    ' is ' + ( a + b + c ) + '</p>';
```

Lines should be broken into logical groups if it improves readability, such as splitting each expression of a ternary operator onto its own line, even if both will fit on a single line.

```
// Acceptable
var baz = ( true === conditionalStatement() ) ? 'thing 1' : 'thing 2';

// Better
var baz = firstCondition( foo ) && secondCondition( bar ) ?
    qux( foo, bar ) :
    foo;
```

When a conditional is too long to fit on one line, each operand of a logical operator in the boolean expression must appear on its own line, indented one extra level from the opening and closing parentheses.

```
if (
    firstCondition() &&
    secondCondition() &&
    thirdCondition()
) {
    doStuff();
}
```

Chained Method Calls

When a chain of method calls is too long to fit on one line, there must be one call per line, with the first call on a separate line from the object the methods are called on. If the method changes the context, an extra level of indentation must be used.

```
elements
    .addClass( 'foo' )
    .children()
        .html( 'hello' )
    .end()
    .appendTo( 'body' );
```

Assignments and Globals

Declaring Variables with `const` and `let`

For code written using ES2015 or newer, `const` and `let` should always be used in place of `var`. A declaration should use `const` unless its value will be reassigned, in which case `let` is appropriate. Unlike `var`, it is not necessary to declare all variables at the top of a function. Instead, they are to be declared at the point at which they are first used.

Declaring Variables With `var`

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`, that variable can leak into an outer scope (which is frequently the global scope, a worst-case 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. Objects and functions that occupy more than a handful of lines should be assigned outside of the `var` statement, to avoid over-indentation.

```
// Good
var k, m, length,
    // Indent subsequent lines by one tab
    value = 'WordPress';

// Bad
var foo = true;
var bar = false;
```

```
var a;
var b;
var c;
```

Globals

In the past, WordPress core made heavier use of global variables. Since core JavaScript files are sometimes used within plugins, existing globals should not be removed. All globals used within a file should be documented at the top of that file. Multiple globals can be comma-separated. This example would make `passwordStrength` an allowed global variable within that file:

```
/* global passwordStrength:true */
```

The “true” after `passwordStrength` means that this global is being defined within this file. If you are accessing a global which is defined elsewhere, omit `:true` to designate the global as read-only.

Common Libraries

Backbone, jQuery, Underscore, and the global `wp` object are all registered as allowed globals in the root `.jshintrc` file. Backbone and Underscore may be accessed directly at any time. jQuery should be accessed through `$` by passing the `jQuery` object into an anonymous function:

```
( function ( $ ) {
    // Expressions
} )( jQuery );
```

This will negate the need to call `.noConflict()`, or to set `$` using another variable. Files which add to, or modify, the `wp` object must safely access the global to avoid overwriting previously set properties:

```
// At the top of the file, set "wp" to its existing value (if present)
window.wp = window.wp || {};
```

Naming Conventions

Variable and function names should be full words, using camel case with a lowercase first letter.

This is an area where this standard differs from the [WordPress PHP coding standards]. Names should be descriptive, but not excessively so. Exceptions are allowed for iterators, such as the use

of `i` to represent the index in a loop.

Abbreviations and Acronyms

Acronyms must be written with each of its composing letters capitalized. This is intended to reflect that each letter of the acronym is a proper word in its expanded form. All other abbreviations must be written as camel case, with an initial capitalized letter followed by lowercase letters. If an abbreviation or an acronym occurs at the start of a variable name, it must be written to respect the camelcase naming rules covering the first letter of a variable or class definition. For variable assignment, this means writing the abbreviation entirely as lowercase. For class definitions, its initial letter should be capitalized.

```
// "Id" is an abbreviation of "Identifier":
const userId = 1;

// "DOM" is an acronym of "Document Object Model":
const currentDOMDocument = window.document;

// Acronyms and abbreviations at the start of a variable name are consistent
// with camelcase rules covering the first letter of a variable or class.
const domDocument = window.document;
class DOMDocument {}
class IdCollection {}
```

Class Definitions

Constructors intended for use with `new` should have a capital first letter (UpperCamelCase). A `class` definition must use the UpperCamelCase convention, regardless of whether it is intended to be used with `new` construction.

```
class Earth {
    static addHuman( human ) {
        Earth.humans.push( human );
    }

    static getHumans() {
        return Earth.humans;
    }
}
```

```
Earth.humans = [];
```

All `@wordpress/element` Components, including stateless function components, should be named using Class Definition naming rules, both for consistency and to reflect the fact that a component may need to be transitioned from a function to a class without breaking compatibility.

Constants

An exception to camel case is made for constant values which are never intended to be reassigned or mutated. Such variables must use the SCREAMING_SNAKE_CASE convention. In almost all cases, a constant should be defined in the top-most scope of a file. It is important to note that JavaScript's `const` assignment is conceptually more limited than what is implied here, where a value assigned by `const` in JavaScript can in-fact be mutated, and is only protected against reassignment. A constant as defined in these coding guidelines applies only to values which are expected to never change, and is a strategy for developers to communicate intent, more than it is a technical restriction.

Comments

Comments come before the code to which they refer, and should always be preceded by a blank line. Capitalize the first letter of the comment, and include a period at the end when writing full sentences. There must be a single space between the comment token (`//`) and the comment text.

```
someStatement();

// Explanation of something complex on the next line
$( 'p' ).doSomething();

// This is a comment that is long enough to warrant being stretched
// over the span of multiple lines.
```

JSDoc comments should use the `/**` multi-line comment opening. Refer to the JavaScript Documentation Standards for more information. Inline comments are allowed as an exception when used to annotate special arguments in formal parameter lists:

```
function foo( types, selector, data, fn, /* INTERNAL */ one ) {
    // Do stuff
```

```
}
```

Equality

Strict equality checks (`===`) must be used in favor of abstract equality checks (`==`).

Type Checks

These are the preferred ways of checking the type of an object:

- String: `typeof object === 'string'`
- Number: `typeof object === 'number'`
- Boolean: `typeof object === 'boolean'`
- Object: `typeof object === 'object'` or `_.isObject( object )`
- Plain Object: `jQuery.isPlainObject( object )`
- Function: `_.isFunction( object )` or `jQuery.isFunction( object )`
- Array: `_.isArray( object )` or `jQuery.isArray( object )`
- Element: `object.nodeType` or `_.isElement( object )`
- null: `object === null`
- null or undefined: `object == null`
- undefined:
  - Global Variables: `typeof variable === 'undefined'`
  - Local Variables: `variable === undefined`
  - Properties: `object.prop === undefined`
  - Any of the above: `_.isUndefined( object )`

Anywhere Backbone or Underscore are already used, you are encouraged to use Underscore.js's type checking methods over jQuery's.

Strings

Use single-quotes for string literals:

```
var myStr = 'strings should be contained in single quotes';
```

When a string contains single quotes, they need to be escaped with a backslash (`\`):

```
// Escape single quotes within strings:
'Note the backslash before the \'single quotes\'';
```

Switch Statements

The usage of `switch` statements is generally discouraged, but can be useful when there are a large number of cases – especially when multiple cases can be handled by the same block, or fall-through logic (the `default` case) can be leveraged. When using `switch` statements:

- Use a `break` for each case other than `default`. When allowing statements to “fall through,” note that explicitly.
- Indent `case` statements one tab within the `switch`.

```
switch ( event.keyCode ) {
    // ENTER and SPACE both trigger x()
    case $.ui.keyCode.ENTER:
    case $.ui.keyCode.SPACE:
        x();
        break;
    case $.ui.keyCode.ESCAPE:
        y();
        break;
    default:
        z();
}
```

It is not recommended to return a value from within a switch statement: use the `case` blocks to set values, then `return` those values at the end.

```
function getKeyCode( keyCode ) {
    var result;

    switch ( event.keyCode ) {
        case $.ui.keyCode.ENTER:
        case $.ui.keyCode.SPACE:
            result = 'commit';
            break;
        case $.ui.keyCode.ESCAPE:
            result = 'exit';
```

```
            break;
        default:
            result = 'default';
    }

    return result;
}
```

Best Practices

Arrays

Creating arrays in JavaScript should be done using the shorthand `[]` constructor rather than the `new Array()` notation.

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

You can initialize an array during construction:

```
var myArray = [ 1, 'WordPress', 2, 'Blog' ];
```

In JavaScript, associative arrays are defined as objects.

Objects

There are many ways to create objects in JavaScript. Object literal notation, `{}`, is both the most performant, and also the easiest to read.

```
var myObj = {};
```

Object literal notation should be used unless the object requires a specific prototype, in which case the object should be created by calling a constructor function with `new`.

```
var myObj = new ConstructorMethod();
```

Object properties should be accessed via dot notation, unless the key is a variable or a string that would not be a valid identifier:

```
prop = object.propertyName;
prop = object[ variableKey ];
prop = object['key-with-hyphens'];
```

Iteration

When iterating over a large collection using a `for` loop, it is recommended to store the loop's max value as a variable rather than re-computing the maximum every time:

```
// Good & Efficient
var i, max;

// getItemCount() gets called once
for ( i = 0, max = getItemCount(); i < max; i++ ) {
    // Do stuff
}

// Bad & Potentially Inefficient:
// getItemCount() gets called every time
for ( i = 0; i < getItemCount(); i++ ) {
    // Do stuff
}
```

Underscore.js Collection Functions

Learn and understand Underscore's collection and array methods. These functions, including `_.each`, `_.map`, and `_.reduce`, allow for efficient, readable transformations of large data sets. Underscore also permits jQuery-style chaining with regular JavaScript objects:

```
var obj = {
    first: 'thing 1',
    second: 'thing 2',
    third: 'lox'
};

var arr = _.chain( obj )
    .keys()
    .map( function ( key ) {
```

```
        return key + ' comes ' + obj[ key ];
    } )
    // Exit the chain
    .value();

// arr === [ 'first comes thing 1', 'second comes thing 2', 'third comes lox' ]
```

Iterating Over jQuery Collections

The only time jQuery should be used for iteration is when iterating over a collection of jQuery objects:

```
$tabs.each( function ( index, element ) {
    var $element = $( element );

    // Do stuff to $element
} );
```

Never use jQuery to iterate over raw data or vanilla JavaScript objects.

JSHint

JSHint is an automated code quality tool, designed to catch errors in your JavaScript code. JSHint is used in WordPress development to quickly verify that a patch has not introduced any logic or syntax errors to the front-end.

Installing and Running JSHint

JSHint is run using a tool called Grunt. Both JSHint and Grunt are programs written in Node.js. The `package.json` configuration file that comes with the WordPress development code allows you to install and configure these tools. To install Node.js, click the Install link on the Node.js website. The correct install file for your operating system will be downloaded. Follow the installation steps for your operating system to install the program. Once Node.js is installed, open a command line window and navigate to the directory where you checked out a copy of the WordPress SVN repository (use `cd ~/directoryname`. You should be in the root directory which contains the `package.json` file. Next, type `npm install` into the command line window. This will download and install all the Node packages used in WordPress development. You should now be able to type `npm`

`run grunt jshint` to have Grunt check all the WordPress JavaScript files for syntax and logic errors. To only check core code, type `npm run grunt jshint:core`; to only check unit test .js files, type `npm run grunt jshint:tests`.

JSHint Settings

The configuration options used for JSHint are stored within a `.jshintrc` in the WordPress SVN repository. This file defines which errors JSHint should flag if it finds them within the WordPress source code.

Target A Single File

To specify a single file for JSHint to check, add `--file=filename.js` to the end of the command. For example, this will only check the file named “admin-bar.js” within WordPress’s core JavaScript files: `npm run grunt jshint:core --file=admin-bar.js` And this would only check the “password-strength-meter.js” file within the unit tests directory: `npm run grunt jshint:tests --file=password-strength-meter.js` Limiting JSHint to a single file can be useful if you are only working on one or two specific files and don’t want to wait for JSHint to process every single file each time it runs.

JSHint Overrides: Ignore Blocks

In some situations, parts of a file should be excluded from JSHint. As an example, the script file for the admin bar contains the minified code for the jQuery HoverIntent plugin – this is third-party code that should not pass through JSHint, even though it is part of a WordPress core JavaScript file. To exclude a specific file region from being processed by JSHint, enclose it in JSHint directive comments:

```
/* jshint ignore:start */
if ( typeof jQuery.fn.hoverIntent === 'undefined' ) {
    // hoverIntent r6 - Copy of wp-includes/js/hoverIntent.min.js
    (function(a){a.fn.hoverIntent=..............
}
/* jshint ignore:end */
```

Credits

- The jQuery examples are adapted from the jQuery JavaScript Style Guide, which is made available under the MIT license.

版本号 #2

由 壳壳虫 创建于 2026-10-31 18:41:46 UTC

由 壳壳虫 更新于 2026-10-31 18:41:48 UTC