BuildTrack®

Javascript Standards

Created By: Mahendra Deshmukh

Created Date: 20th Nov 2019
Version: 1.0

Revision History:

Version	**Date**	**Name**	**Author**	**Revision Notes**
1.0	20th Nov 2019	JS Coding Standards	Mahendra Deshmukh	Initial Draft

This document comprises of BuildTarck's coding standards for source code written in JavaScript language. Issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the hard-and-fast rules that we follow universally, and avoids giving advice that isn't clearly enforceable.

File name

File names must be all lowercase and may include underscores (`_`) or dashes (`-`), but no additional punctuation.

Source file structure

Files consist of the following, in order:

1. License or copyright information (Default: Surmount Energy Solutions)
2. `@auther` statement file (Default: BuildTrack Development Team)
3. `@description` statement of a file
4. ES `import` statements, if an ES module
5. The file's implementation

Exactly one blank line separates each section that is present, except the file's implementation, which may be preceded by 1 or 2 blank lines.

A) Formatting

block-like construct refers to the body of a class, function, method, or brace-delimited block of code.

Braces

Braces are required for all control structures (i.e. `if`, `else`, `for`, `do`, `while`, as well as any others), even if the body contains only a single statement. The first statement of a non-empty block must begin on its own line.

Disallowed:

```
if (someVeryLongCondition())
  doSomething();

for (let i = 0; i < foo.length; i++) bar(foo[i]);
```

Exception: A simple if statement that can fit entirely on a single line with no wrapping (and that doesn’t have an else) may be kept on a single line with no braces when it improves readability. This is the only case in which a control structure may omit braces and newlines.

```
if (shortCondition()) foo();
```

Nonempty blocks: K&R style

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace *if* that brace terminates a statement or the body of a function or class statement, or a class method. Specifically, there is *no* line break after the brace if it is followed by `else`, `catch`, `while`, or a comma, semicolon, or right-parenthesis.

Example:

```
class InnerClass {
  constructor() {}

  /** @param {number} foo */
  method(foo) {
    if (condition(foo)) {
      try {
        // Note: this might fail.
        something();
      } catch (err) {
        recover();
      }
    }
  }
}
```

Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters, space, or line break in between (i.e. `{}`), unless it is a part of a *multi-block statement* (one that directly contains multiple blocks: `if`/`else` or `try`/`catch`/`finally`).

Example:

```
function doNothing() {}
```

Disallowed:

```
if (condition) {
  // …
} else if (otherCondition) {} else {
  // …
}

try {
  // …
} catch (e) {}
```

Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

Array literals

Any array literal may optionally be formatted as if it were a “block-like construct.” For example, the following are all valid (not an exhaustive list):

```
const a = [
  0,
  1,
  2,
];


const b =
    [0, 1, 2];
```

```
const c = [0, 1, 2];


someMethod(foo, [
  0, 1, 2,
], bar);
```

Other combinations are allowed, particularly when emphasizing semantic groupings between elements, but should not be used only to reduce the vertical size of larger arrays.

Object literals

Any object literal may optionally be formatted as if it were a “block-like construct.”

```
const a = {
  a: 0,
  b: 1,
};

const b =
    {a: 0, b: 1};
```

```
const c = {a: 0, b: 1};

someMethod(foo, {
  a: 0, b: 1,
}, bar);
```

Class literals

Class literals (whether declarations or expressions) are indented as blocks. Do not add semicolons after methods, or after the closing brace of a class *declaration.*

Example:

```
class Foo {
  constructor() {
    /** @type {number} */
    this.x = 42;
  }

  /** @return {number} */
  method() {
    return this.x;
  }
}
Foo.Empty = class {};
```

```
/** @extends {Foo<string>} */
foo.Bar = class extends Foo {
  /** @override */
  method() {
    return super.method() / 2;
  }
};

/** @interface */
class Frobnicator {
  /** @param {string} message */
  frobnicate(message) {}
}
```

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' );
```

Switch statements

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

A blank line is optional between a `break` and the following case.

Example:

```
switch (animal) {
  case Animal.BANDERSNATCH:
    handleBandersnatch();
    break;

  case Animal.JABBERWOCK:
    handleJabberwock();
    break;

  default:
    throw new Error('Unknown animal');
}
```

B) Spacing

Use spaces liberally throughout your code. “When in doubt, space it out.” These rules encourage liberal spacing for improved developer readability.

- Indentation with 2 spaces.
- No whitespace at the end of line or on blank lines.
- Lines should usually be no longer than 80 characters. This is a “soft” rule, but long lines generally indicate unreadable or disorganized code.
- There should be a new line at the end of each file.
- All function bodies are indented by 2 sapces, even if the entire file is wrapped in a closure.

Statements

We are following one statement per line. Each statement is followed by a line-break.

Semicolons

Every statement must be terminated with a semicolon. Never rely on Automatic Semicolon Insertion (ASI).

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();
}
```

Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line, unless it falls under the rules of block indentation. When there are multiple continuation lines, indentation may be varied beyond +4 as appropriate.

Whitespace

1) Vertical whitespace

A single blank line appears:

1. Between consecutive methods in a class or object literal
2. Within method bodies, sparingly to create *logical groupings* of statements. Blank lines at the start or end of a function body are not allowed.

2) Horizontal whitespace

Use of horizontal whitespace depends on location, and falls into three broad categories: *leading* (at the start of a line), *trailing* (at the end of a line), and *internal*. Leading whitespace (i.e., indentation) is already addressed. Trailing whitespace is forbidden or not allowed. Horizontal whitespaces appear in the following places only.

1. Separating any reserved word (such as `if`, `for`, or `catch`) except for `function` and `super`, from an open parenthesis (`(`) that follows it on that line.
2. Separating any reserved word (such as `else` or `catch`) from a closing curly brace (`}`) that precedes it on that line.
3. Before any open curly brace (`{`), with following exception:
   a. Before an object literal that is the first argument of a function or the first element in an array literal (e.g. `foo({a: [{c: d}]})`).
4. On both sides of any binary or ternary operator.
5. After a comma (`,`) or semicolon (`;`). Note that spaces are *never* allowed before these characters.
6. After the colon (`:`) in an object literal.
7. On both sides of the double slash (`//`) that begins an end-of-line comment.

Horizontal alignment: discouraged

Horizontal alignment is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

Allowed :

```
{
  tiny: 42, // this is great
  longer: 435, // this too
};
```

Not allowed :

```
{
  tiny:   42,  // permitted, but future edits
  longer: 435, // may leave it unaligned
};
```

Function arguments

Prefer to put all function arguments on the same line as the function name. If doing so would exceed the 80-column limit, the arguments must be line-wrapped in a readable way.

C) Language features

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;
```

Naming Conventions

Package names

Package names are all lowerCamelCase. For example, `my.exampleCode.deepSpace`, but not `my.examplecode.deepspace` or `my.example_code.deep_space`.

Method names

Method names are written in `lowerCamelCase` convention.

Class Definitions

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 = [];
```

Enum names

Enum names are written in `UpperCamelCase`, similar to classes and should generally be singular nouns. Individual items within the enum are named in `CONSTANT_CASE`.

Constants

An exception to camel case is made for constant values which are never intended to be reassigned or mutated. Such variables names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. In almost all cases, a constant should be defined in the top-most scope of a file.

Parameter names

Parameter names are written in `lowerCamelCase`. Note that this applies even if the parameter expects a constructor.

Local variable names

Local variable names are written in `lowerCamelCase`, except for module-local (top-level) constants, as described above. Constants in function scopes are still named in `lowerCamelCase`. Note that `lowerCamelCase` is used even if the variable holds a constructor.

Local aliases

Local aliases should be used whenever they improve readability over fully-qualified names. Aliases may also be used within functions. Aliases must be `const`.

```
const staticHelper = importedNamespace.staticHelper;
const CONSTANT_NAME = ImportedClass.CONSTANT_NAME;
const {assert, assertInstanceof} = asserts;
```

Non-constant field names

Non-constant field names (static or otherwise) are written in `lowerCamelCase`, with a trailing underscore for private fields. These names are typically nouns or noun phrases. For example, `computedValues` or `index_`.

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 {}
```

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.
```

Block comment

Block comments are indented at the same level as the surrounding code. They may be in `/* … */` or `//`-style. For multi-line `/* … */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line, to make comments obvious with no extra context.

Equality

Strict equality checks (`===`) must be used in favor of abstract equality checks (`==`). The *only* exception is when checking for both `undefined` and `null` by way of `null`.

```
// Check for both undefined and null values, for some important reason.
if ( undefOrNull == null ) {
    // Expressions
}
```

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\'';
```

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'];
```

Switch Statements

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;
}
```

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
}
```