Search Wiki Documents

Javascript coding standards

New Document | Edit Document | View History

Last updated 29 days ago by espenv.

Table of Contents

Overview

Scope

This document provides the coding standards and guidelines for developers and teams working for or with VG Multimedia. The subjects covered are:

- Javascript file formatting
- Naming conventions
- Coding style
- Inline documentation
- Errors and exceptions

Goals

Good coding standards are important in any development project, particularly when multiple developers are working on the same project. Having coding standards helps to ensure that the code is of high quality, has fewer bugs, and is easily maintained.

Javascript file formatting

General

All javascript files should create their own scope and only set things that should be globally available on the `window` object. Its very important to end with semicolon `;`, not doing so might mess with Javascript compression and concatination tools.

```
(function(window, $) {
   var codeToExport = 'Foobar';

   window.foobar = codeToExport;
})(this, jQuery);
```

Indentation

Use an indent of 4 spaces with no tab characters. Editors should be configured to treat tabs as spaces in order to prevent injection of tab characters into the source code.

Line termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character, no carriage returns (CR). Lines should not contain trailing spaces. In order to facilitate this convention, most editors can be configured to strip trailing spaces, such as upon a save operation.

Line length

Try to keep line length below 100 characters.

Naming Conventions

All files, classes, functions, variables, parameters, database names, table names etc should be English, and as generic terms as possible. Exceptions are allowed where the name is intended to reflect very specific Norwegian names or concepts, e.g. "skattelister", "rampelys" etc.

Use plural variants where applicable. Try to use verbs followed by the name of the entity.

Examples:

- files: editUser.js, listUsers.js

Filenames

For all other files, only alphanumeric characters are permitted. Use camelCase. Spaces are prohibited.

Declarations

Variables

Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable names but are discouraged.

Like function names, variable names must always start with a lowercase letter and follow the "camelCase" capitalization convention.

Verbosity is encouraged. Variable names should always be as verbose as practical. Terse variable names such as `i` and `n` are discouraged for anything other than the smallest loop contexts. If a loop contains more than 20 lines of code, variables for such indices or counters need to have more descriptive names.

Constants

- Use NAMES_LIKE_THIS
- Never use the const keyword

Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called the "camelCase" method.

Verbosity is encouraged. Function names should be as illustrative as is practical to enhance understanding.

These are examples of acceptable names for functions:

- filterInput()
- getElementById()
- generateLink()

Objects

Object names may only contain alphanumeric characters. Underscores are not permitted.

Object names must always start with a uppercase letter. When a object name consists of more than one word, the first letter of each new word must be capitalized.

Verbosity is encouraged. Object names should be as illustrative as is practical to enhance understanding.

These are examples of acceptable names for objects:

- VGTVFeedParser
- ArticleStyler

Coding style

Nesting

More then three levels of nesting is a code smell and might indicate that the code should be retought or that some of the logic should be seperated out into a seperate functions.

Variables

All variables should **always** be declared using `var`. There are **no** exceptions to this rule!

If possible declare several variables using the same var statement.
BAD:

```
var var1 = true;
var var2 = false;
var var3;
```

GOOD:

```
var var1 = true,
    var2 = false,
    var3;
```

Strings

Prefer `'` over `"`

Comparisons

Just like PHP, Javascript has a type system with a lot of gotchas.

```
(false == 0) === true
```

```
(false == ' ') === true
(true == 1) === true
(true == 2) === false
(true == 'foo') === false
(null == undefined) === true
```

In order to prevent confusion and bugs all comparisons with `null`, boolean values, `0` and `1` and `''` should use the type comparison operators `===` and `!==`.

```
if (variable === true)
if (variable !== false)
if (variable === 0)
if (variable !== 1)
if (variable !== null)
if (variable === ' ')
if (typeof variable == 'undefined')
```

Control statements

if / else / else if

Control statements based on the "if", "else", and "else if" constructs must have a single space before the opening parenthesis of the conditional, and a single space between the closing parenthesis and opening brace.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping of larger conditionals.

The opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented four spaces.

```
if (a != 2) {
    a = 2;
}
```

For "if" statements that include "else if" or "else", the formatting must be as in these examples:

```
if (a != 2) {
    a = 2;
} else {
    a = 7;
}

if (a < 2) {
    a = 2;
} else if (a < 10) {
    a = 7;
} else {
    a = 8;
}
```

Javascript allows for these statements to be written without braces in some circumstances. The coding standard makes no differentiation and all "if", "else if", or "else" statements must use braces.

switch

Control statements written with the "switch" construct must have a single space before the opening parenthesis of the conditional statement, and also a single space between the closing parenthesis and the opening brace.

All content within the "switch" statement must be indented four spaces. Content under each "case" statement must be indented an additional four spaces.

```
switch (numPeople) {
    case 1:
        break;
    case 2:
        break;
    default:
        break;
}
```

The construct "default" may never be omitted from a "switch" statement.

It is sometimes useful to write a "case" statement which falls through to the next case by not including a "break" or "return". To distinguish these cases from bugs, such "case" statements must contain the comment "*break intentionally omitted*".

for

When using a for loop the three parts must be separated by spaces:

```
for (var articleCounter = 0; articleCounter < numArticles; articleCounter++) {
    // display article
}
```

There must also be a single whitespace before the opening parenthesis and one between the closing parenthesis and the opening brace.

try/catch

When wrapping code that can throw exceptions in try/catch blocks, use the following syntax:

```
try {
    someFunctionThatMightThrowAnException();
} catch (error) {
    // Do something clever with error
}
```

Functions

All functions should be bound to a variable using `var` and not declared directly using the `function` keyword. They should also not be named function expressions because of scope pollution:

BAD:

```
function doSomething(inputVar) {
```

```
}

var func = function doSomething(inputVar) {

};
```

GOOD:

```
var doSomething = function(inputVar) {

};
```

Never create block-local function declarations:
BAD:

```
(function() {
    if (true) {
        var func =function(inputVar) {
        };
     }
})();
```

GOOD:

```
(function() {
     var func =function(inputVar) {
     };
})();
```

Objects

Its very important to remember to not leave any dangling commas `,` when declaring objects as this breaks on older IEs and some JS parsers.

```
var myObject = {
    key: 'value',
    key2: 'value', // <-- BAD
}
```

Prototyping

If you are using a library like underscore.js, lodash.js or jQuery that supports extending objects please extend `prototype` with a single object with all the methods.

BAD:

```
var UberCoolObject = function() {
    this.name = 'Unset';
};
```

```
UberCoolObject.prototype.setName = function(name) {
    this.name = name;
};

UberCoolObject.prototype.alertName = function() {
    alert(this.name);
};
```

GOOD:

```
var UberCoolObject = function() {
    this.name = 'Unset';
};

$.extend(UberCoolObject.prototype, {
    setName: function(name) {
        this.name = name;
   },

   alertName: function() {
       alert(this.name);
   }
});
```

Abstract objects

Create/fake abstract objects/classes as an object with all the functions. Then extend the prototype of the implementation object with the "abstract" object.

```
var TwitterAbstract = {
    fetchTweet: function(tweetId) {
           // Do some work here
    },

    deleteTweet: function(tweetId) {
         // Do work here
    }
};

var VGTweets = function() {
     this.accountUsername = 'vgnett';
};

$.extend(VGTweets.prototype, {
    displayVGTweets: function() {
         // Do work here
    }
}, TwitterAbstract);
```