Javascript Style Guide

> Just because you can doesn't mean you should

Adapted from

- A meta style guide for JavaScript - Dr. Axel Rauschmayer (http://www.2ality.com/2013/07/meta-style-guide.html)
- Maintainable Javascipt - Nicholas C. Zakas (http://shop.oreilly.com/product/0636920025245.do)
- Crockford (http://javascript.crockford.com/code.html)
- Idiomatic Javascript (https://github.com/rwaldron/idiomatic.js/)

Indentation

An indent is equal to one tab. This allows the developer to configure how many spaces to show for each tab in their editor. Indent with tabs, and space with spaces. Spaces should never occur on a line in front of a statement. After a statement begins, feel free to use spaces to align variables or make code easier to read.

```
// good
function indent() {
    var noSpaces = true;
    var onlyUseTabs = true;
}

// bad
function badIndent() {
    var onlySpaces = false;
   var onlyTabs = false;
}
```

- Indentation settings in VS2010 (http://goo.gl/rU5hPo)
- Indentation settings in VS2012 (http://msdn.microsoft.com/en-us/library/vstudio/7sffa753.aspx)
- Indentation settings in Sublime Text 2 (http://www.sublimetext.com/docs/2/indentation.html)

Operator Spacing

Operators with two operands must be preceded and followed by a single space to make the expression clear. Operators include assignments and logical operators.

```
// bad
for (var i=0; i<50; i++) {
}

// good
for (var i = 0; i < 50; i++) {
}

// good
var spacedCorrectly = (50 < 100 && count === 0);
```

Parenthesis Spacing

There should be no whitespace after an opening paren or before a closing paren.

```
// bad
if ( sad ) {
    obj.fn( sad );
}

// good
if (sad) {
    obj.fn(sad);
}
```

Brackets

Brackets should be preceded by a space unless being used to pass an object literal as an argument to a function, in which case it should not be preceded by a space.

```
// bad
if(true){
}

// good
if (true) {
}

// bad
obj.test( {
    row: 0,
    cell: 0
} );

// good
obj.test({
    row: 0,
    cell: 0
});
```

Always use brackets with `if`, `for`, `while`, and other compound statements. `else` should always be on the same line as the closing bracket of the preceding `if` statement.

```
// bad
while (i--) obj.test();

// good
while (i--) {
    obj.test();
}

// good
if (1 < 2) {
    obj.test1();
} else {
    obj.test2();
}

// bad
if (1 < 2)
    obj.test1();
else
    obj.test2()

// bad
if (test) doSomthing();
else doSomethingElse();
```

Naming

Name variables clearly so that another developer could understand them. Avoid single character names at all costs. Avoid snake

case (http://en.wikipedia.org/wiki/Snake_case). Use these methods for naming:

- variables, objects, properties, non-constructor functions: camel case
- constansts: upper case, separating words with underscores
- constructors: pascal case

```
/*
 * Good
 */
(function (undefined) {
    var ONE = 1;
    var TRUE = true;
    var UNDEF = undefined;
    var EMPTY_GUID = '00000000-0000-0000-0000-000000000000';

    function Creation(genes) {
        this.kind = genes.kind;
    }

    var creations = [];
    var create = function (numCreations) {
        for (var i = 0; i < numCreations; i++) {
            creations.push(new Creation({
                kind: 'A ' + i + ' kind.'
            }));
        }
    }

    exampleNamespace.theCreator = {
        create: create
    }
})();

exampleNamespace.theCreator.create(1904);

/*
 * Bad
 */
(function (undefined) {
    var one = 1;
    var True = true;
    var u = undefined;
    var e_g = '00000000-0000-0000-0000-000000000000';

    function creation(genes) {
        this.kind = genes.kind;
    }

    var Creations = [];
    function Create(num_creations) {
        for (var i = 0; i < num_creations; i++) {
            Creations.push(new creation({
                kind: 'A ' + i + ' kind.'
            }));
        }
    }

    ExampleNamespace.Creator = {
        create: Create
    }
})();

ExampleNamespace.Creator.create(1904);
```

Functions

Named functions should have no space between the function name and opening paren for the parameter list. Anonymous (unnamed)

functions should have one space between `function` and the opening paren. This way, anonymous functions will appear like "functions with no name".

```
// bad
function namedFunction (a, b, c) {
}
var fn = function(a, b, c) {
};
obj.test(function(a, b, c) {
});

// good
function namedFunction(a, b, c) {
}
var fn = function (a, b, c) {
};
obj.test(function (a, b, c) {
});
```

Consider the benefits of named function expressions. They are able to call themselves and will show up as named functions when debugging.

```
// good
var factorial = function factorial(number) {
    if (number < 2) {
        return 1;
    }
    return (number * factorial(number - 1));
};

// good
$.ajax(url).done(function ajaxDone() {

});
```

Semicolons

There should be at most one statement per line. Every statement should end with a semicolon. It is better to be safe than to trust that ASI (https://brendaneich.com/2012/04/the-infernal-semicolon/) will cover your mistakes.

```
// bad
function createError() {
    var one = 1;
    var two = 2
    return
    {
        msg: 'uh oh'
    };
}

// good
function noError() {
    var one = 1;
    var two = 2;
    return {
        msg: 'ok'
    };
}
```

Variables

Variable declarations should occur where needed. It is not necessary to have all variable declarations at the top of each function. Be aware of hoisting (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Scope_Cheatsheet), but do not go overboard and declare every single locally scoped variable at the top of a function. A good example of when not to declare a variable at the top of a function is when defining `for` loops.

```
// bad
function loopOverItems(items) {
    var i = 0, l = items.length, item;
    if (someNamespace.canDo()) {
        for (;i<l;i++) {
            item = items[i];
        }
    }
}

// good
function loopOverItems(items) {
    if (someNamespace.canDo()) {
        for (var i = 0, l = items.length; i < l; i++) {
            var item = items[i];
        }
    }
}
```

Each variable should be declared on a separate line. The comma operator (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator) should not be used when declaring variables, except when defining `for` loops. This allows adding and removing variable declarations trivial.

```
// bad
function test() {
    var one = 1,
        two = 2,
        three;
}

// good
function test() {
    var one = 1;
    var two = 2;
}
```

Strings

Never mix single and double quotes in the same file when defining String literals. Prefer single over double quotes for String literals. Never use a slash to create a new line in a string, because this will result in a `SyntaxError` if there are any whitespace characters after the slash.

```
// bad
var help = "I'm coming!\
    and I'm bringing a friend.";
var me   = 'I\'m not ok.';

// good
var help = 'I\'m coming! and I\'m bringing a friend.';
var me   = 'I\'m going to be ok.';
var templ = '<div class="helper"></div>';
```

Numbers

Never use octal literals, leading decimals, or hanging decimals.

```
// bad
var octal = 010;
var leading = .1;
var hanging = 1.;
```

Null

Only use null in these situations:

- To initialize a variable that may later be assigned to an object value
- To compare against an initialized variable that may or may not have an object value
- To compare against both null and undefined at once
- To pass into a function where an object is expected
- To return from a function where an object is expected

```
// good
obj.test(null, null, i);

// bad
obj.test(undefined, undefined, i);

// good
var testRetValue = obj.test();
if (testRetValue == null) {
    // testRetValue must be nothing
}

// good
var maybe = null;
if (somethingIsOk()) {
    maybe = true;
}
```

Undefined

Never use the value `undefined`. It is fragile because it can be redefined (http://us6.campaign-archive1.com/?u=2cc20705b76fa66ab84a6634f&id=3ef2f3d32b). To check if a variable has been initialized use the `typeof` operator against the string `'undefined'`, or by checking equality (non-strict) with `null`. Avoid the use of `void 0` (http://us6.campaign-archive1.com/?u=2cc20705b76fa66ab84a6634f&id=d56bf8ad4f) since it is not easily understood by beginners. Avoid checking for truthiness since the variable might hold a `Boolean` or `0`. Prefer `typeof`.

```
// good
if (typeof test === 'undefined') {
}
if (test == null) {
}

// bad
if (test === void 0) {
}

// bad
if (test === undefined) {
}

// bad (when checking for initialitation)
if (test) {
}
```

Checking for properties of an object

To check if an object has a property, use the `in` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) operator, or `Object.prototype.hasOwnProperty()`. Do not use `.hasOwnProperty()` on DOM elements, because they do not have this method in IE7/8.

```
// good
if ('test' in obj) {
}

// good
if (obj.hasOwnProperty('test')) {
}

// bad
var domElement = document.getElementById('testId');
if (domElement.hasOwnProperty('textContent')) { // error in IE7/8
}

// ok
if ({}.prototype.hasOwnProperty.call(domElement, 'textContent')) {
}

// better
if ('textContent' in domElement) {
}
```

for ... in

When using `for ... in`, *always* check if the object has each property. This is especially necessary for looping over an `Array` because `for ... in` checks all the way up the prototype chain.

```
// good
for (var prop in obj) {
    if (obj.hasOwnProperty(prop)) {
        var val = obj[prop];
    }
}

// bad
for (var prop in obj) {
    var val = obj[prop];
}
```

Object literals

- Opening brace should be on the same line as the containing statement
- Each property value should be indendented once on the line underneath the opening brace
- Try not quote property names. This ensures that all property names to be valid javascript variable names.
- Do not insert a space preceding the colon between property names and values
- The closing brace should be on the last line by itself, followed by a semicolon

```
// good
var obj = {
    one: 1,
    isTwo: function (num) {
        return (num === 2);
    },
    three: 3,
    four: 4
};

// bad - all on same line, no semicolon, quoted property name
var obj = { one: 1, 'two': 2 }
```

Literals vs. Constructors

Use literals and native coersion instead of native Object contructors. Basically, never use the native Object constructors. One main reason for this is that `typeof 'test' === 'string'`, but `typeof new String('test') === 'object'`. An exeption to this is `RegExp` because sometimes it is necessary in order to dynamically create string based regular expressions at runtime.

```
var obj = new Object(); // bad
var obj = {}; // good

var arr = new Array(); // bad
var arr = []; // good

var arr = new Array('a', 'b', 'c'); // bad
var arr = ['a', 'b', 'c']; // good

var regex = new RegExp('abc'); // ok
var regex = /abc/; // good

var str = ('' + value); // good
var str = new String(value); // bad

var now = (+'2'); // good
var now = new Number('2'); // bad
```

Comments

- Comment frequently to help other developers understand your code.
- Keep comments updated. Comments pertaining to deleted code, and misleading comments are worse than no comments.
- Document code using JSDoc (http://usejsdoc.org/). Document constuctors, objects, and methods. Use sublime-jsdocs (https://github.com/spadgos/sublime-jsdocs) when editing in Sublime Text to generate the documentation stubs for you.
- All comments on their own line and documentation comments should be preceded by an empty line.

Single Line Comments

```
// bad
function test() {
    var value1;
    // this should be preceded by an empty line
    var value2;
}

// good
var obj1 = 1; // comments on the same line are ok
var obj = {

    // the is readable
    prop1: value1,

    // so is this
    prop2: value2
};
```

Multiline Comments

Use `/*` to begin multiline comments the are not documentation instead of the JSDoc standard of `/**`. There should be no comment on the first line after the first asterisk. All asterisks should be vertically aligned, and the last line should end with `*/`.

```
//good

/**
 * this function will test something
 *
 * @param {Array} arg1 - an array
 */
function test(arg1) {

    /*
     * this array will do bla-bla
     * and be used for bla-bla
     */
    var arr = arg1;
}
```