Coding Guidelines | Javascript

**Naming Convention**

**In general, use functionNamesLikeThis, variableNamesLikeThis, ClassNamesLikeThis, EnumNamesLikeThis, metho dNamesLikeThis, CONSTANT\_VALUES\_LIKE\_THIS, foo.namespaceNamesLikeThis.bar, and filenameslikethis.js.**

**Avoid single letter names. Be descriptive with your naming. Properties and methods**

Private properties and methods should be named with a trailing underscore like var test\_

Protected properties and methods should be named without a trailing underscore (like public ones).

**Method and function parameter**

Optional function arguments start with opt\_.

**Events**

Event handlers and callback functions should be prefixed with “on”:



**Boolean variables or methods**

Boolean variables or methods returning boolean functions should prefix the variable name with “is”:

*function isAdmin() {}*

*var canEdit = isUser() && isAdmin();*

**Note :- Alternatives are “has”, “can” and “should” if they make more sense**

**Formatting**

All JavaScript documents must use**4 spaces** for indentation.

**White space :-** white spaces should not be used inside of function parentheses



/ *BAD: Too much whitespace.*

*function getUrl****( full )*** *{*

*var url = '/styleguide/javascript/';*

*if ( full ) {*

*url = '*[*http://okfn.github.com/ckan'*](http://okfn.github.com/ckan%27) *+ url;*

*}*

*return url;*

*}*

/ *GOOD:*

*function getUrl****(full)*** *{*

*var url = '/styleguide/javascript/';*

*if (full) {*

*url = '*[*http://okfn.github.com/ckan'*](http://okfn.github.com/ckan%27) *+ url;*

*}*

*return url;*

*}*

**Quotes :-**Single quotes should be used everywhere unless writing JSON or the string contains them. This makes it easier to create strings containing HTML.

*jQuery('<div id="my-div" />').appendTo('body');*

*Object properties need not be quoted unless required by the interpreter.*

*var object = {*

*name: 'bill',*

*'class': 'user-name'*

*};*

**Curly Braces** :- Because of implicit semicolon insertion, always start your curly braces on the same line as whatever they're opening. For example:

*if (something) {*

/ *...*

*} else {*

/ *...*

*}*

**Array and Object Initializers :-**

Single-line array and object initializes are allowed when they fit on a line:

*var arr = [1, 2, 3]; // No space after [ or before ].*

*var obj = {a: 1, b: 2, c: 3}; // No space after { or before }.*

Multiline array initialisers and object initialisers are indented 2 spaces, with the braces on their own line, just like blocks.

/ *Object initializer. var inset = {*

*top: 10,*

*right: 20,*

*bottom: 15,*

*left: 12*

*};*

Long identifiers or values present problems for aligned initialization lists, so always prefer non-aligned initialization. For example:

*CORRECT\_Object.prototype = { a: 0,*

*b: 1,*

*lengthyName: 2*

*};*

*Not like this:*

*WRONG\_Object.prototype = { a : 0,*

*b : 1,*

*lengthyName: 2*

*};*

**Statement Rule :-** General rules for simple statements: Always end a simple statement with a semicolon. Put the opening bracket at the end of the first line. Use one space before the opening bracket.

Put the closing bracket on a new line, without leading spaces. Do not end a complex statement with a semicolon.

**Object Rules :-** General rules for object definitions:

Place the opening bracket on the same line as the object name. Use colon plus one space between each property and its value. Use quotes around string values, not around numeric values.

Do not add a comma after the last property-value pair.

Place the closing bracket on a new line, without leading spaces. Always end an object definition with a semicolon.

**Function Arguments :-**

When possible, all function arguments should be listed on the same line. If doing so would exceed the 80-column limit, the arguments must be line-wrapped in a readable way. To save space, you may wrap as close to 80 as possible, or put each argument on its own line to enhance readability. The indentation may be either four spaces, or aligned to the parenthesis. Below are the most common patterns for argument wrapping:

/ *Four-space, wrap at 80. Works with very long function names, survives*

/ *renaming without reindenting, low on space. goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(*

*veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo, tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {*

/ *...*

*};*

**Semicolons :-** We always use semicolon to end the statement.

**Comments :-** All files, classes, methods and properties should be comments with the appropriate tags and types. Textual descriptions for properties, methods, method parameters and method return values should be included unless obvious from the property, method, or parameter name.

Inline comments should be of the // variety.

*/\* My method description. Should describe what the method does and where*

* *it should be used.*

\*

* *param1 - The method params, one per line (default: null)*
* *param2 - A default can be provided in brackets at the end.*

\*

* *Example*

*\**

*\* // Indented two spaces. Should give a common example of use.*

* *client.getTemplate('index.html', {limit: 1}, function (html) {*
* *module.el.html(html);*

\* *});*

*\**

* *Returns describes what the object returns.*

*\*/*

**Avoid Global Variables**

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

**Always Declare Local Variables**

All variables used in a function should be declared as **local** variables.

Local variables **must** be declared with the **var** keyword, otherwise they will become global variables.

**Declarations on Top**

It is a good coding practice to put all declarations at the top of each script or function. This will:

Give cleaner code

Provide a single place to look for local variables

Make it easier to avoid unwanted (implied) global variables Reduce the possibility of unwanted re-declarations

**Initialize Variables**

It is a good coding practice to initialise variables when you declare them. This will:

Give cleaner code

Provide a single place to initialise variables Avoid undefined values

Variables created without the keyword var, are always global, even if they are created inside a function.

**Variable Lifetime**

Global variables live as long as your application (your window / your web page) lives.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished. Don't Use new Object()

Use {} instead of new Object() Use "" instead of new String() Use 0 instead of new Number()

Use false instead of new Boolean() Use [] instead of new Array()

Use /()/ instead of new RegExp()

Use function (){} instead of new Function()

***Example***

*var x1 = {}; // new object*

*var x2 = ""; // new primitive string var x3 = 0; // new primitive number*

*var x4 = false; // new primitive boolean var x5 = []; // new array object*

*var*

*x6 = /()/; // new regexp object*

*var x7 = function(){}; // new function object*

**Beware of Automatic Type Conversions**

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

**Use === Comparison**

The == comparison operator always converts (to matching types) before comparison. The === operator forces comparison of values and type:

**End Your Switches with Defaults**

Always end your switch statements with a default. Even if you think there is no need for it.

**Reduce DOM Access**

Accessing the HTML DOM is very slow, compared to other JavaScript statements.

If you expect to access a DOM element several times, access it once, and use it as a local variable:

***Example***

*var obj;*

*obj = document.getElementById("demo"); obj.innerHTML = "Hello";*

**Reduce DOM Size**

Keep the number of elements in the HTML DOM small.

This will always improve page loading, and speed up rendering (page display), especially on smaller devices. Every attempt to search the DOM (like getElementsByTagName) will benefit from a smaller DOM.

**Avoid Unnecessary Variables**

Don't create new variables if you don't plan to save values. Often you can replace code like this:

*var fullName = firstName + " " + lastName; document.getElementById("demo").innerHTML = fullName;*

*With this:*

*document.getElementById("demo").innerHTML = firstName + " " + lastName*

**Delay JavaScript Loading**

Putting your scripts at the bottom of the page body, lets the browser load the page first.

While a script is downloading, the browser will not start any other downloads. In addition all parsing and rendering activity might be blocked.

The HTTP specification defines that browsers should not download more than two components in parallel.

An alternative is to use defer="true" in the script tag. The defer attribute specifies that the script should be executed after the page has finished parsing, but it only works for external scripts.

If possible, you can add your script to the page by code, after the page has loaded:

***Example***

*<script>*

*window.onload = downScripts; function downScripts() {*

*var element = document.createElement("script"); element.src = "myScript.js"; document.body.appendChild(element);*

*}*

*</script>*

eval **is Evil**

The eval function is the most misused feature of JavaScript. Avoid it.

**Inlining the crap out of functions and object literals.**

the nested code like this to be really hard to follow:

*var name = 'Steve Kwan';*

*var company = 'Electronic Arts'; var myFunction = function()*

*{*

*$('form#my-form').submit (*

*function(event)*

*{*

*event.preventDefault();*

*$.ajax (*

*'/some\_service',*

*{*

*type: "POST", data:*

*{*

*name: name, name: company*

*},*

*success: function(data)*

*{*

*callSomeCompletionFunction (*

*{*

*response1: data.value1, response2: data.value2*

*}*

*)*

*},*

*error: function(data)*

*{*

*callSomeErrorHandler (*

*{*

*response1: data.value1,*

*response2: data.value2*

*}*

*)*

*}*

*}*

*);*

*}*

*);*

*};*

This kind of code is problematic because:

It causes major readability and maintainability issues

It prevents you from reusing some of those nested functions Avoid unneeded ternary statements.

*eslint rules: no-unneeded-ternary.*

*// bad*

*const foo = a ? a : b;*

*const bar = c ? true : false; const baz = c ? false : true;*

*// good*

*const foo = a || b; const bar = !!c; const baz = !c;*