JS Coding Standards

Amendment Record			
Version	**Date**	**Reviewed and Approved by**	**Change Description**
1.0	22-Jul - 2019	Techbench	Initial Version

TABLE OF CONTENTS

1 Purpose

The objective of this document is to describe the rules and recommendations for developing applications and class libraries using the JavaScript Language. The goal is to define guidelines to enforce consistent style and formatting and help developers to avoid common pitfalls and mistakes.

The programming guidelines and best practices described in this document is designed in such a way keeping quality, performance and maintainability in mind.

2 Description

2.1 Scope

The coding standard described in this document applies to all source code files produced in Triassic. It only specifies how the code should look like. How to do the coding and how to manage the coding phase are outside the scope of this standard.

2.2Notational conventions

2.2.1Rule

A rule should be broken only for compelling reasons where no reasonable alternative can be found. The author of the violating code shall consult with at least one knowledgeable colleague and a senior designer to review said necessity. A comment in the code explaining the reason for the violation is mandatory.

2.2.2Recommendation

A recommendation should be followed unless there is good reason to do otherwise. Consultation with a knowledgeable colleague about the validity of the reason is necessary. A comment in the code is recommended.

2.3General Rules

1. Every time a recommendation is not followed, this must have a good reason.

2. Do not mix code from different providers in one file.

2.4JavaScript Language Rules

2.4.1Rules

- Always use var for declarations

When you fail to specify `var`, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable lives (e.g., it could be in the Document or Window just as easily as in the local scope). So always declare with `var`

```
// GOOD:

var good = 'string';

var alsoGood = 'another';

// GOOD:

var good = 'string';

var okay = [

 'hmm', 'a bit', 'better'

];

// BAD:

var good = 'string',

  iffy = [

 'hmm', 'not', 'great'

];
```

Declare variables at the top of the function in which they are first used. This avoids issues with variable hoisting. If a variable is not assigned a value until later in the function then it it okay to define more than one per statement.

```
// BAD: contrived example.
function lowercaseNames(names) {
 var names = [];
 for (var index = 0, length = names.length; index < length; index += 1) {
  var name = names[index];
  names.push(name.toLowerCase());
 }
 var sorted = names.sort();
 return sorted;
}
// GOOD:
function lowerCaseNames(names) {
 var names = [];
 var index, sorted, name;
 for (index = 0, length = names.length; index < length; index += 1) {
  name = names[index];
  names.push(name.toLowerCase());
```

```
  }
  sorted = names.sort();
  return sorted;
}
```

- Always use semicolons at the end of each line

Relying on implicit insertion can cause subtle hard to debug problems. There are a couple of places where missing semicolons are particularly dangerous.

- All JavaScript documents must use **two spaces** for indentation.

Example:

```
var foo = function () {
   return true;
}; // semicolon here.

function foo() {
   return true;
} // no semicolon here.
```

```
//Do not declare functions within blocks
//Example: Do not do this
if (x) {
  function foo() {}
}
```

- White space

Two spaces must be used for indentation at all times. Unlike in idiomatic whitespace must not be used _inside_ parentheses between the parentheses and their Contents.

```
// BAD: Too much whitespace.
function getUrl( full ) {
 var url = '/styleguide/javascript/';


 if ( full ) {

  url = 'http://okfn.github.com/ckan' + url;
 }


 return url;
```

```
}

// GOOD:

function getUrl(full) {
 var url = '/styleguide/javascript/';
 if (full) {
  url = 'http://okfn.github.com/ckan' + 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'
```

```
};
```

While most script engines support Function Declarations within blocks it is not part of ECMAScript (see ECMA-262, clause 13 and 14). Worse implementations are inconsistent with each other and with future ECMAScript proposals. ECMAScript only allows for Function Declarations in the root statement list of a script or function. Instead use a variable initialized with a Function Expression to define a function within a block:
Example:

```
if (x) {
  var foo = function () {};
}
```

1. Do not use Wrapper objects for primitive types

Example: Do not do this

```
var x = new Boolean(false);
if (x) {
  alert('hi');  // Shows 'hi'.
}
```

However type casting is fine.
Example:

```
var x = Boolean(0);
if (x) {
  alert('hi');  // This will never be alerted.
}
typeof Boolean(0) == 'boolean';
typeof new Boolean(0) == 'object';
```

This is very useful for casting things to number, strings and boolean

- Method and property definitions

```
/** @constructor **/
function SomeConstructor() {
  this.someProperty = 1;
}
Foo.prototype.someMethod = function() { ... };
```

While there are several ways to attach methods and properties to an object created via “new”, the preferred style for methods is:

```
Foo.prototype.bar = function() {
  /* ... */
};
```

The preferred style for other properties is to initialize the field in the constructor:

```
/** @constructor */
function Foo() {
  this.bar = value;
}
```

- delete

```
//Prefer
this.foo = null.
Foo.prototype.dispose = function() {
  this.property_ = null;
};
//Instead of:
Foo.prototype.dispose = function() {
  delete this.property_;
};
```

In modern JavaScript engines, changing the number of properties on an object is much slower than reassigning the values. The delete keyword should be avoided except when it is necessary to remove a property from an object's iterated list of keys, or to change the result of `if (key in obj)`.

- Use Closures with care

```
function foo(element, a, b) {
  element.onclick = function() { /* uses a and b */ };
}
```

the function closure keeps a reference to element, a, and b even if it never uses element. Since element also keeps a reference to the closure, we have a cycle that won't be cleaned up by garbage collection. In these situations, the code can be structured as follows:

```
function foo(element, a, b) {
  element.onclick = bar(a, b);
}
function bar(a, b) {
  return function() { /* uses a and b */ };
}
```

- Use ‘this’ only in object constructors, methods and in setting up closures. At times it refers to the global object (in most places), the scope of the caller (in eval), a node in the DOM tree (when attached using an event handler HTML attribute), a newly created object (in a constructor), or some other object (if function was call()ed or apply()ed).

  Because this is so easy to get wrong, limit its use to those places where it is required:

- in constructors

- in methods of objects (including in the creation of closures)
- Use for-in loops only for iterating over keys in an object/map/hash

for-in loops are often incorrectly used to loop over the elements in an Array. This is however very error prone because it does not loop from 0 to length - 1 but over all the present keys in the object and its prototype chain. Here are a few cases where it fails:

```
function printArray(arr) {
   for (var key in arr) {
      print(arr[key]);
   }
}
printArray([0,1,2,3]);  // This works.
var a = new Array(10);
printArray(a);  // This is wrong.
a = document.getElementsByTagName('*');
printArray(a);  // This is wrong.
a = [0,1,2,3];
a.buhu = 'wine';
printArray(a);  // This is wrong again.
a = new Array;
a[3] = 3;
printArray(a);  // This is wrong again.
//Always use normal for loops when using arrays.
```

```
function printArray(arr) {
  var l = arr.length;
  for (var i = 0; i < l; i++) {
    print(arr[i]);
  }
}
```

2.4.2Recommendations

- Standard features are always preferred over non-standard features

For maximum portability and compatibility, always prefer standards features over non-standards features (e.g., **string.charAt (3)** over **string [3]** and element access with DOM functions instead of using application-specific shorthand).

- Multi-level prototype hierarchies are not preferred

This is how JavaScript implements inheritance. You have a multi-level hierarchy if you have a user-defined class D with another user-defined class B as its prototype. These hierarchies are much harder to get right than they first appear!
For that reason, it is best to use goog.inherits() from the Closure Library or a similar library function.

Example:

```
function D() {
  goog.base(this)
}
```

```
goog.inherits(D, B);

D.prototype.method = function() {

  ...

};
```

2.5 Naming Conventions

2.5.1Constants

1. Use NAMES_LIKE_THIS for constant values
2. Use @const to indicate a constant (non-overwritable) *pointer* (a variable or property).
3. Never use the const keyword as it's not supported in Internet Explorer.

Example:

```
/**

* Request timeout in milliseconds.

* @type {number}

*/

goog.example.TIMEOUT_IN_MILLISECONDS = 60;
```

2.5.2Functions:

3 Use camel case notation for naming functions
Correct: sortNamesList (), sortNames_List()

Wrong: SortNameList ()

1. Private methods should be named with a trailing underscore

Example: _calculateSum ();

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

3.1.1Method and function parameter:

4Optional function arguments start with **opt_**

1. Functions that take a variable number of arguments should have the last argument named var_args. You may not refer to var_args in the code; use the arguments array.

4.1.1Getters and Setters

ECMAScript 5 getters and setters for properties are discouraged. However, if they are used, then getters must not change observable state.
Example:

```
/**
* WRONG -- Do NOT do this.
*/
var foo = { get next() { return this.nextId++; } };
```

4.1.2Classes and Enums

Use PascalCase notation for naming classes and enums
Example:

Correct: SampleClass, SortOptions
Wrong: sampleClass, SAMPLECLASS etc

4.1.3File Names

File names should be all small letters: And should contain no punctuation except for - or _ (prefer - to _).

Example:

thisisafilename.txt. this_isafile_name.txt

4.2 JavaScript Language Rules

4.3Custom toString() methods must always succeed without side effects

You can control how your objects string-ify themselves by defining a custom toString() method. Ensure that your method

1. always succeeds
2. does not have side-effects.

Otherwise your method will run into serious problems

4.3.1Always use explicit scope

Always use explicit scope - doing so increases portability and clarity. For example, don't rely on window being in the scope chain. You might want to use your function in another application for which window is not the content window.

4.3.2Code Formatting

4.3.2.1Curly Braces:

Because of implicit semicolon insertion, always start your curly braces on the same line as whatever they're opening.

Example:

```
if (something) {
  // ...
} else {
  // ...
}
```

4.3.2.2Array and object initializers

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

Example:

```
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 initializers and object initializers are indented 2 spaces, with the braces on their own line, just like blocks.
Example:

```
// 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.
Example:

```
CORRECT_Object.prototype = {
  a: 0,
  b: 1,
```

```
  lengthyName: 2
};
```

Not like this:

```
WRONG_Object.prototype = {
  a      : 0,
  b      : 1,
  lengthyName: 2
};
```

4.3.3Use parenthesis only when required

Never use parentheses for unary operators such as delete, typeof andvoid or after keywords such as return, throw as well as others (case, inor new).

4.3.4Single quots(‘) are preferred over double quots(“) when using strings

For consistency single-quotes (') are preferred to double-quotes ("). This is helpful when creating strings that include HTML:
Example:

```
var msg = 'This is some HTML';
```

4.3.5Visibility: Encouraged, use JSDoc annotations.

Use of the JSDoc annotations @private and@protected to indicate visibility levels for classes, functions, and properties are recommended.

4.3.6Comments

All files, classes, methods and properties should be documented with JSDoc comments with the appropriate tags and types . Inline comments should be of the // variety. Complete sentences are recommended but not required. Complete sentences should use appropriate capitalization and punctuation.
Syntax:

```
/**
* A JSDoc comment should begin with a slash and 2 asterisks.
* Inline tags should be enclosed in braces like {@code this}.
* @desc Block tags should always start on their own line.
*/
```

4.3.6.1JSDoc Indentation:

If you have to line break a block tag, you should treat this as breaking a code statement and indent it four spaces (no need to indent @fileoverview and @desc commands).

```
/**
* Illustrates line wrapping for long param/return descriptions.
* @param {string} foo This is a param with a description too long to fit in
*    one line.
* @return {number} This returns something that has a description too long to
*    fit in one line.
*/
project.MyClass.prototype.method = function (foo) {
```

```
  return 5;
};
```

4.3.6.2HTML in JSDoc

Plaintext formatting is not respected. So, don't rely on whitespace to format JSDoc:

```
/**
* Computes weight based on three factors:
*  items sent
*  items received
*  last timestamp
*/
```

It'll come out like this:
Computes weight based on three factors: items sent items received last timestamp
Instead, do this:

```
/**
* Computes weight based on three factors:
* <ul>
* <li>items sent
* <li>items received
```

```
* <li>last timestamp
* </ul>
*/
```

4.3.6.3Class Comments

Classes must be documented with a description and a type tag that identifies the constructor.

```
/**
* Class making something fun and easy.
* @param {string} arg1 An argument that makes this more interesting.
* @param {Array.<number>} arg2 List of numbers to be processed.
* @constructor
* @extends {goog.Disposable}
*/
project.MyClass = function(arg1, arg2) {
  // ...
};
goog.inherits(project.MyClass, goog.Disposable);
```

4.3.6.4Method and Function Comments:

Parameter and return types should be documented. The method description may be omitted if it is obvious from the parameter or return type descriptions. Method descriptions should start with a sentence written in the third person declarative voice.

```
/**
* Operates on an instance of MyClass and returns something.
* @param {project.MyClass} obj Instance of MyClass which leads to a long
*    comment that needs to be wrapped to two lines.
* @return {boolean} Whether something occurred.
*/
function PR_someMethod(obj) {
  // ...
}
```

4.3.6.5Property Comments

```
/** @constructor */
project.MyClass = function () {
/**
* Maximum number of things per pane.
* @type {number}
*/
this.someProperty = 4;
```

```
}
```

4.3.6.6Providing dependencies with goog.provide: Only provide top level symbols

All members defined on a class should be in the same file. So, only top-level classes should be provided in a file that contains multiple members defined on the same class (e.g. enums, inner classes, etc).

Do this:

```
goog.provide('namespace.MyClass');
```

Not this:

```
goog.provide('namespace.MyClass');

goog.provide('namespace.MyClass.Enum');

goog.provide('namespace.MyClass.InnerClass');

goog.provide('namespace.MyClass.TypeDef');

goog.provide('namespace.MyClass.CONSTANT');

goog.provide('namespace.MyClass.staticMethod');
```

Members on namespaces may also be provided:

```
goog.provide('foo.bar');

goog.provide('foo.bar.method');

goog.provide('foo.bar.CONSTANT');
```

4.4 Tips and Tricks

4.4.1True and False boolean Expressions.

The following are all false in boolean expressions:

- null
- undefined
- '' the empty string
- 0 the number

But be careful, because these are all true:

- '0' the string
- [] the empty array
- {} the empty object

This means that instead of this:

```
while (x != null) { }
```

you can write this shorter code (as long as you don't expect x to be 0, or the empty string, or false):

```
while (x) { }
```

And if you want to check a string to see if it is null or empty, you could do this:

```
if (y != null && y != '') {
```

But this is shorter and nicer:

```
if (y) {   }
```

4.4.2Conditional (ternary) Operator(?:):

Instead of this:

```
if (val) {
   return foo();
} else {
   return bar();
}
```

you can write this:

```
return val ? foo() : bar();
```

4.4.3&& and ||

These binary boolean operators are short-circuited, and evaluate to the last evaluated term. "||" has been called the 'default' operator, because instead of writing this:

```
/** @param {*=} opt_win */
```

```
function foo(opt_win) {
  var win;
if (opt_win) {
  win = opt_win;
} else {
  win = window;
}
  // ...
}
```

you can write this:

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

"&&" is also useful for shortening code. For instance, instead of this:

```
if (node) {
```

```
    if (node.kids) {
        if (node.kids[index]) {
            foo(node.kids[index]);
        }
    }
}
```

you could do this:

```
if (node && node.kids && node.kids[index]) {
    foo(node.kids[index]);
}
```

or this:

```
var kid = node && node.kids && node.kids[index];
if (kid) {
    foo(kid);
}
```

4.4.4Iterating over Node Lists

Node lists are often implemented as node iterators with a filter. This means that getting a property like length is O(n), and iterating over the list by re-checking the length will be O(n^2).

```
var paragraphs = document.getElementsByTagName('p');

for (var i = 0; i < paragraphs.length; i++) {

  doSomething(paragraphs[i]);

}
```

It is better to do this instead:

```
var paragraphs = document.getElementsByTagName('p');

for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {

  doSomething(paragraph);

}
```

5 References

1. Google JavaScript Style Guide
https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml