JavaScript Programming Coding Conventions

Version 1.1

Table of contents:

Revisions

Version	Date	Content	Author
0.1	2016.04.15	Init. Create from the Swift Code Convention guide. 5.1 ~ 5.10	Leo Cao
1.0	2016.04.18	Insert new 5.1 section. Finish all.	Leo Cao
1.1	2021.08.17	Update organization	Dai Xiao Ying

2 Goal / Purpose

One of the best ways to promote source code maintainability is the adherence to some programming guidelines which describe the coding conventions of files, classes, methods and variables, so that our source code looks alike, regardless of the original author.

These coding conventions are meant as regulation for the standardized formal appearance of our source files and are also meant as common guidelines when programming in JavaScript.

They are not intended to restrict the creativeness and the intuition of the SW developers and should leave them enough room for creativity.

The basic ideas behind programming guidelines are as follows:

- Decrease developing times due to clear and consistent guidelines
- Simplify module, integration and system test (e.g. automatic insertion of human readable version numbers in every file)
- Decrease integration time (e.g. by avoidance of naming conflicts)
- Increase source code readability for both the author and others
- Observe copyright rules
- Best practice sharing of common design and implementation guidelines

Handling common tasks in a common way (which reduces the risk of problems)

This document describes the programming guidelines for SLC CT DD DS CN SZ projects, which will be written under the programming language JavaScript(ECMAScript 6).

3 Glossary of Terms

Example	Examples provide help of usage in form of JavaScript code snippets. Many examples, especially for naming conventions, are taken from the Base Classes guidelines
Hint	Hints offer support for specific cases
Recommendation	Recommendations are conventions, which should be observed. They are not mandatory, but offer a sensible guide
Rule	A rule marks a mandatory convention. If such a rule cannot be observed in some exceptional cases, this has to be reasoned in the design document.
NOT	Anti-Rule cannot to do
LOC	Lines Of Code

4 Reference

http://www.ibm.com/developerworks/cn/web/1008_wangdd_jscodingrule/index.html
http://www.w3schools.com/js/js_conventions.asp
http://google.github.io/styleguide/javascriptguide.xml
http://javascript.crockford.com/code.html
[Recommendation, ES6]https://github.com/yuche/javascript

5 Coding Conventions

The JavaScript coding conventions described in this chapter has to be used in all projects writing managed code. They assure a common structure and naming for all developers, which is an important aspect of software quality.

5.1 Basic & Important Principles

Rule

Keeping the naming style **CONSISTENTLY**.

It means you should name things in one style, not in mixing way.

Rule

Name everything **MEANINGFULLY.**

Not too long, not too short, naming in Semantic corresponding the Designed Purpose.

5.2 File Names

For JavaScript source code file, the filename should follow:

Rule

First word with the first letter in lowercase, other word with the first letter in uppercase, don't use underscore or dash char to join words.

For release file, use ".min.js" as postfix.

The format should be UTF-8 without BOM.

Example

```
fileName.js          // debug version

fileName.min.js      // release version in compressed
```

NOT

```
FileName.js          // first letter is uppercase
file_Name.js         // use underscore to split words
file-Name.js         // use dash to split words
file-name.js         // a mixing way
WKIRM.js             // use abbrev format, Who Knows It Real Means?
```

5.3 Source Code Formatting

These source code rules might look at first sight as if they would domineer over the developers' creative freedom, but a common way of formatting ensures common readability and replaceability of source code snippets.

Rule

JavaScript source files have the following ordering:

- Beginning comments
- Function(as a class, or normal function) or Variable/Object declarations

Rule

All source files should begin with comment that lists the File name, Project name, Author(s), date, and copyright notice:

```
/**
 * fileName.js
 *
 * [Description for main function of this type]
 *
 * Copyright details
 */
```

5.4 Comment Style

Rule

Use C-Style block comment style.

Example

```
/*
 *
 */
```

Rule

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

Example

```
/**
 * Here is a block comment.
 * [More description here]
 */
function funcName() {
}
```

Rule
Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see Rule 1-5). A blank line should precede a single-line comment. Here's an example of a single-line comment in JavaScript code:

Example

```
/**
 * Here is a block comment.
 * [More description here]
 */
function funcName() {
  var samples = []; // A single line comment, place it following the
statement

  // A single line comment, place it on above of statement
  for (i = 0; i < 10; i++) {

  }
}
```

5.5 Code Complexity

Recommendation
Classes/files should not exceed 2000 Lines Of Code.

Recommendation
Classes and interfaces should comply with the rule of 8 +/- 2 publicly visible methods.

Recommendation
Methods should not exceed 100 LOC.

Recommendation
Methods should not have more than five parameters.

5.6 Leading Tab

Check your using IDE's setting

Rule

Convert Tab to Space in 2 in the IDE Setting

Example

```
for (i = 0; i < 10; i++) {
  console.log(i);   // before console, there are 2 spaces
}
```

NOT

```
for (i = 0; i < 10; i++) {
    console.log(i); // before console, there are 4 spaces
}
```

5.7 Blank lines

To improve readability for code blocks, you should use a blank line or blank lines to split code blocks. The relative code lines should be in a block without blank line.

Rule

Use reasonable Blank line(s) to split logical code blocks

Example

```
var samples = [0, 1, 2, 3, 4, 5, 6];
for (i = 0; i < samples.length; i++) { // there is no blank line between
above line
  console.log(“index: ” + i);
}

samples = [6, 5, 4, 3, 2, 1];    // in above, there is a blank line
for (i = 0; i < samples.length; i++) {
  console.log(“index: ” + i);
}
```

NOT

```
var samples = [0, 1, 2, 3, 4, 5, 6];

for (i = 0; i < samples.length; i++) // there is blank line in above, but
samples is used here
  console.log(“index: ” + i);
}
Samples = [6, 5, 4, 3, 2, 1];    // no blank line in above, but it should
for (i = 0; i < samples.length; i++) {
  console.log(“index: ” + i);
}
```

5.8 expression and statement

Rule

Use ";" in the end of a statement

Example

```
console.log(i);
```

NOT

```
console.log(i)
```

Rule

Add "{" following the expression or declaration

Example

```
for (i = 0; i < 10; i++) {
  console.log(i);
}
```

NOT

```
for (i = 0; i < 10; i++)
{
  console.log(i);
}
```

Rule

- Operators should be surrounded by a space character.
- JavaScript reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

Example

```
var samples = [0, 1, 2, 3, 4, 5, 6]; // space in the array; Space after “:”;
space around “=”

function funcName() { // space before “{”
}

function funcName(arg1, arg2) { // space after “,”
}

for (i = 0; i < 10; i++) { // spaces around “=”, “<”; space after “;”

}
```

NOT

```
var samples = [0,1,2,3,4,5,6]; // no space in the array

var samples=[0, 1, 2, 3, 4, 5, 6]; // no space arround “=”

function funcName(){ // no space before “{”
}

function funcName(arg1,arg2) { // no space after “,”
}

for (i=0;i<10;i++) { // no spaces around “=”, “<”; no space after “;”

}
```

5.9 Conditionals

Rule

Use “{}” for condition, even it is a single line

Example

```
if (0 == i) {
  console.log(“It is OK.”);
}

If (0 == i) { console.log(“It is OK.”); } // this is fine too
```

NOT

```
if 0 == i
  console.log(“It is OK.”);
```

Recommendation

Use “===” and “!==” in prefer than “==” and “!=”

Example

```
if (0 === i) {
  console.log(“It is OK.”);
}

if (i !== 0) {
  console.log(“It is OK.”);
}
```

NOT

```
if (0 == i) {
  console.log(“It is OK.”);
}

if (i != 0) {
  console.log(“It is OK.”);
}
```

Recommendation

Place constant value in the before when using “==”

Example

```
if (0 == i) {              // “0” before i
  console.log(“It is OK.”);
}

If (0 == i) { console.log(“It is OK.”); }
```

NOT

```
if i == 0 {         // “0” after i
  console.log(“It is OK.”);
}
```

Rule

Use “()” for multiple conditions

Example

```
if ((0 == i) || (3 == j)) && (2 == k) {
  console.log(“It is OK.”);
}
```

NOT

```
if (0 == i || 3 == j) && (2 == k) {
  console.log(“It is OK.”);
}
```

5.10 Declarations and Initialization

Rule

Put a declaration for variable in each single line

Example

```
var i;
var j;
var k;
var samples;
```

NOT

```
var i, j, k; var samples;
```

Recommendation

Don’t use explicit “Array” keyword to initialize variable

Example

```
var samples = [];
```

NOT

```
var samples = new Array();
```

5.11 Type Name

Rule

The defined custom type, include class. The naming should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations.

In ECMAScript 6, we can use “class” keyword, so don’t use old style to define a class in function.

Example

```
class UrlHandler {
  constructor(contents = []) {
    this._queue = […contents];
  }

  // pop
  publicFunction1() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

NOT

```
function UrlHandler() {
  let rh = this;

  // private members
  this._var1 = 0;
  this._samples = [];

  // private functions
  function privateFunction() {
  }
};

UrlHandler.prototype = {
  // public functions
  publicFunction1 : function() {
  }

  publicFunction2 : function(arg1) {
  }
};
```

Rule

In JavaScript, no direct keyword for Interface, we use class for it, but the name must add “I” (i in uppercase) as the prefix letter, and the second letter should be in uppercase.

Example

```
class IProtocolName { }

class IServiceManager { }
```

NOT

```
class Service { }

class Iservice { }  // “s” in lowercase

class IserviceManager { }  // “s” in lowercase
```

Rule

In JavaScript, there is no enum directly, we can use a normal object to simulate it. the naming style is same as normal class. And put each value in a single line.

Example

```
const EnumName = {
  ValueName1: 0,
  ValueName2: 1,
  ValueName3: 2,
  ValueName4: 3,
};
```

NOT

```
const EnumName = {  // like C Const
  VALUE_NAME1: 0,
  VALUE_NAME2: 1,
  VALUE_NAME3: 2,
  VALUE_NAME4: 3,
};

enum HttpMethod = { // define value name in single line
  ValueName1: 0,  ValueName2: 1,  ValueName3: 2,  ValueName4: 3,
};
```

5.12 Variables

Rule

- Local variable defined in a function.
- Public or protected variable defined in a class
- Const or Static variable

  Define the name with the first letter of first word in lowercase, other words with first letter in uppercase.

Example

```
function funcName() {
  var returnVal
}

class ClassName {
  let constVal = 0 // In JavaScript, there is no “const” keyword like c
}

let constVal2 = 0
```

NOT

```
function funcName() {
  var return_Val    // don’t use underscore
  var b-type        // don’t use dash
  var _self         // don’t use underscore as prefix
}

class ClassName {
  let ConstVal = 0  // The first letter in uppercase
}

let ConstVal2 = 0
```

Rule

When define private variable in class. The naming is start with “_” letter, and the first letter of first word in lowercase, other words with first letter in uppercase.

Example

```
class ClassName {
  this._privateVal = 0;
}
```

NOT

```
class ClassName {
  this.privateVal = 0; // no “_” as first letter
}
```

Recommendation

Use “let” “var” correctly according to the operation for this variable. If the variable is only for read, use “let”, not “var”.

Example

```
class ClassName {
  let readonlyVal = 255;
}
```

NOT

```
class ClassName {
  var readonlyVal = 255; // use var, it means can be changed in later.
}
```

Recommendation

When define a local Boolean variable, use “is”, “has”, “can” etc as prefix.

Example

```
class ClassName {
  function funcName() {
    var isOpen = false
    var hasSet = false
    var canDelete = false
  }
}
```

NOT

```
class ClassName {
  function funcName() {
    var opened = false
    var set = false
    var deleted = false

    var isNotOpen = false // defined a Negated bool variable

  }
}
```

5.13 Methods

Rule

When define function in class or class. The naming is with the first letter of first word in lowercase, other words with first letter in uppercase.

Example

```
class ClassName {
  function funcName() { }
}
```

NOT

```
class ClassName {
  function FuncName() { } // The first letter is uppercase
  function function_Name() { } // use underscore
  function function-Name() { } // The use dash
}
```

Rule

To define a function with return value, use “get” as prefix; or set value, use “set” as prefix. Other words with first letter in uppercase.

Example

```
class ClassName {
  function setFuncName() { }

  function getFuncName() { }
}
```

NOT

```
class ClassName {
  function set-FuncName() { } // user dash or underscore

  function get-FuncName() { } // user dash or underscore
}
```

5.14 Source Code Documentation

Rule

It must need add some description in the header of a source file. Ref 4.2.

Rule

It must need add some description for function defined in class

Example

```
class IProtocolName {

  /**
   * Descriptions purpose for this function
   * A: description for this argument
   * B: description for this argument
   * ReturnType: description for return value
   *
   * Other notation to use this function in need
   */
  function funcName(arg1, arg2)

}
```

Rule

When add comment for function or code blocks, explain its design or condition or use case. DON't say any useless content. It means we should add meaningful comments.

Example

```
class IProtocolName {

  // TODO: Change to generic version to sum two value in any type
  function summary(arg1, arg2)

}
```

NOT

```
class IProtocolName {

  // calculate the total value for two interger values // it is useless
  function summary(arg1, arg2)

}
```

Recommendation

Add comment for the type of argument in function declaration.

Example

```
class ClassName {

  function functionName(/* String */arg1, /* Integer */arg2) {}
}
```

Rule

Use JavaScript defined comment keywords: “TODO:” “FIXME:”

Example

```
class ClassName {

  // TODO: Change to generic version to sum two value in any type
  function functionName(arg1, arg2) {}

  // FIXME: this function need be test in unit test codes.
  function functionName(arg1, arg2) {}

}
```