JavaScript Coding Standards

Version: 5.11-CO::1

March 6, 2019

DOMINION
VOTING

Our customers come first.

Table of Contents

CHAPTER 1: INTRODUCTION

The aim of this document is to establish the coding standard and set of tools to be used to verify and enforce the usage of the standard for javascript code.

CHAPTER 2: JAVASCRIPT CODE QUALITY TOOL

We are using JSLint.NET as code quality tool. https://jslintnet.codeplex.com/

JSLint.NET is a wrapper for Douglas Crockford's JSLint, the JavaScript code quality tool. It can validate JavaScript anywhere .NET runs.

JSLint is a static code analysis tool used in software development for checking if JavaScript source code complies with coding rules. It is provided primarily as an online tool, but there are also command-line adaptations. It was developed by Douglas Crockford, author of the book "JavaScript: Good Parts".

2.1 Installation

JSLint.net can be installed as Visual Studio Extension, search for "jslint.net" in the Extensions and Updates menu in Visual Studio.
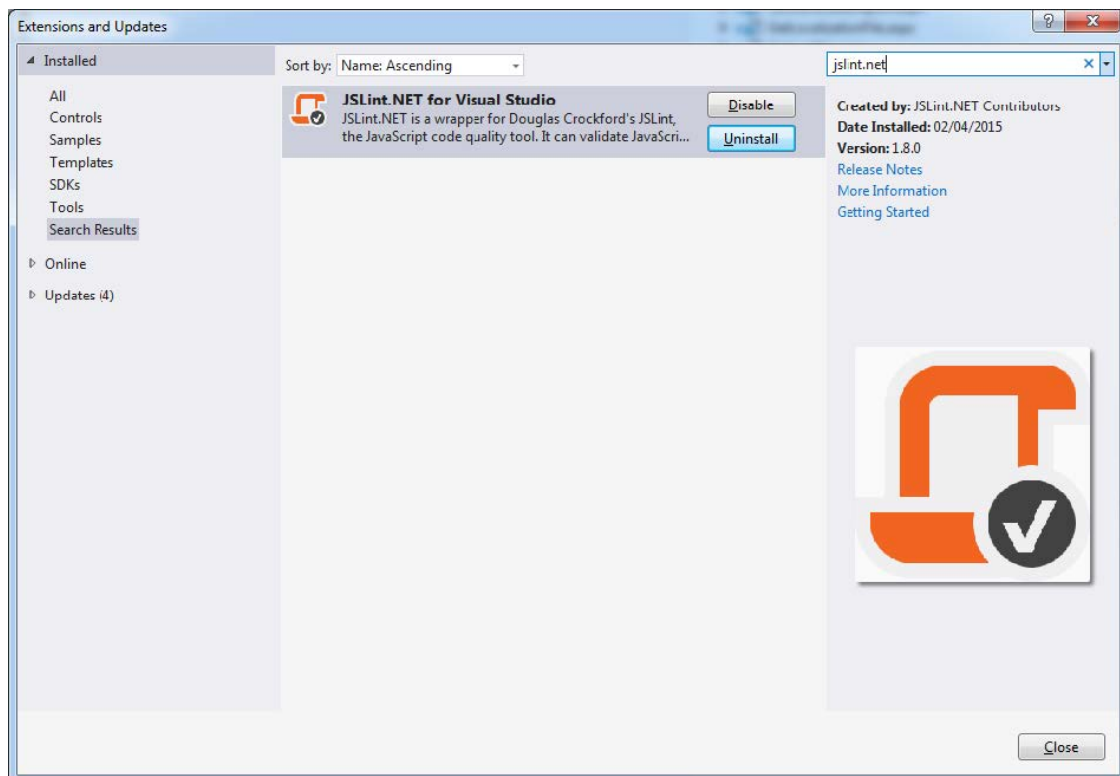
Figure 2-1: JSLint.net Installation

2.2 Configuration

JSLint.net could be configured by having JSLintNet.json file in the root of the project.

Example of the configuration file:

```
{
    "options": {
        "predef": {
            "jQuery": false,
            "$": false
        },
        "browser": true,
        "unparam": true
    },
    "ignore": [
        "\\UI\\vendor\\"
    ],
    "output": "Warning",
    "runOnSave": true,
    "runOnBuild": true,
    "cancelBuild": false
}
```

Figure 2-2: Sample Configuration File

CHAPTER 3: CODE CONVENTIONS FOR THE JAVASCRIPT PROGRAMMING LANGUAGE

NOTE: Original document can be found on http://javascript.crockford.com/code.html

This is a set of coding conventions and rules for use in JavaScript programming. It is inspired by the Sun document Code Conventions for the Java Programming Language. It is heavily modified of course because JavaScript is not Java.

The long-term value of software to an organization is in direct proportion to the quality of the codebase. Over its lifetime, a program will be handled by many pairs of hands and eyes. If a program is able to clearly communicate its structure and characteristics, it is less likely that it will break when modified in the never-too-distant future.

Code conventions can help in reducing the brittleness of programs.

All of our JavaScript code is sent directly to the public. It should always be of publication quality.

Neatness counts.

3.1 JavaScript Files

JavaScript programs should be stored in and delivered as `.js` files.

JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to pageweight with no opportunity for mitigation by caching and compression.

`<script src=filename.js>` tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components. There is no need to use the language or type attributes. It is the server, not the script tag, that determines the MIME type.

3.2 Indentation

The unit of indentation is four spaces. Use of tabs should be avoided because (as of this writing in the 21st Century) there still is not a standard for the placement of tabstops. The use of spaces can produce a larger filesize, but the size is not significant over local networks, and the difference is eliminated by minification.

3.3 Line Length

Avoid lines longer than 80 characters. When a statement will not fit on a single line, it may be necessary to break it. Place the break after an operator, ideally after a comma. A break after an operator decreases the likelihood that a copy-paste

error will be masked by semicolon insertion. The next line should be indented 8 spaces.

3.4 Comments

Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly yourself) who will need to understand what you have done. The comments should be well-written and clear, just like the code they are annotating. An occasional nugget of humor might be appreciated. Frustrations and resentments will not.

It is important that comments be kept up-to-date. Erroneous comments can make programs even harder to read and understand.

Make comments meaningful. Focus on what is not immediately visible. Don't waste the reader's time with stuff like:

```
i = 0; // Set i to zero.
```

Generally, use line comments. Save block comments for formal documentation.

3.5 Variable Declarations

All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals. Implied global variables should never be used. Use of global variables should be minimized.

The `var` statement should be the first statement in the function body.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order if possible.

```
    var currentEntry, // currently selected table entry
level,       // indentation level        size;        // size
of table
```

JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.

3.6 Function Declarations

All functions should be declared before they are used. Inner functions should follow the var statement. This helps make it clear what variables are included in its scope.

There should be no space between the name of a function and the ((left parenthesis) of its parameter list. There should be one space between the) (right parenthesis) and the { (left curly brace) that begins the statement body. The body

itself is indented four spaces. The } (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```javascript
function outer(c, d) {

var e = c * d;

function inner(a, b) {

return (e * a) + b;

}

return inner(0, 1);

}
```

This convention works well with JavaScript because in JavaScript, functions and object

literals can be placed anywhere that an expression is allowed. It provides the best

readability with inline functions and complex structures.

```javascript
function getElementsByClassName(className) {

var results = [];

walkTheDOM(document.body, function (node) {

var array, // array of class names

ncn = node.className; // the node's classname

// If the node has a class name, then split it into a list of
simple names.

// If any of them match the requested name, then append the node
to the list of results.

if (ncn && ncn.split(' ').indexOf(className) >= 0) {

results.push(node);

}

});

return results;

}
```

If a function literal is anonymous, there should be one space between the word `function` and the ((left parenthesis). If the space is omitted, then it can appear that the function's name

is `function`, which is an incorrect reading.

```
div.onclick = function (e) {

return false;

};

that = {

method: function () {

return this.datum;

},

datum: 0

};
```

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be

wrapped in parens so that it is clear that the value being produced is the result of the

function and not the function itself.

```
var collection = (function () {

var keys = [], values = [];

return {

get: function (key) {

var at = keys.indexOf(key);

if (at >= 0) {

return values[at];

}

},

set: function (key, value) {

var at = keys.indexOf(key);

if (at < 0) {

at = keys.length;

}

keys[at] = key;
```

```
values[at] = value;

},

remove: function (key) {

var at = keys.indexOf(key);

if (at >= 0) {

keys.splice(at, 1);

values.splice(at, 1);

}

}

};

}());
```

3.7 Names

Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 .. 9), and _ (underbar). Avoid use of international characters because they may not read well or be understood everywhere. Do not use $ (dollar sign) or \ (backslash) in names.

Do not use _ (underbar) as the first or last character of a name. It is sometimes intended to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide private members. Avoid conventions that demonstrate a lack of competence.

Most variables and functions should start with a lower case letter.

Constructor functions that must be used with the new prefix should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time warning if a required new is omitted. Bad things can happen if new is not used, so the capitalization convention is the only defense we have.

Global variables should be in all caps. (JavaScript does not have macros or constants, so there isn't much point in using all caps to signify features that JavaScript doesn't have.)

3.8 Statements

3.8.1 Simple Statements

Each line should contain at most one statement. Put a ; (semicolon) at the end of every simple statement. Note that an assignment statement that is assigning a function literal or object literal is still an assignment statement and must end with

a semicolon.

JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion. The only expressions that should be used as statements are assignments and invocations.

3.8.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in { } (curly braces).

- The enclosed statements should be indented four more spaces.
- The { (left curly brace) should be at the end of the line that begins the compound statement.
- The } (right curly brace) should begin a line and be indented to align with the beginning of the line containing the matching { (left curly brace).
- Braces should be used around all statements, even single statements, when they are part of a control structure, such as an if or for statement. This makes it easier to add statements without accidentally introducing bugs.

Labels

Statement labels are optional. Only these statements should be labeled: `while, do, for, switch`.

3.8.3 return Statement

A `return` statement with a value should not use ( ) (parentheses) around the value. The return value expression must start on the same line as the `return` keyword in order to avoid semicolon insertion.

3.8.4 if **Statement**

The `if` class of statements should have the following form:

```
if (condition) {

statements

}

if (condition) {

statements

} else {

statements

}
```

```
if (condition) {

statements

} else if (condition) {

Statements

} else {

statements

}
```

3.8.5 for Statement

A for class of statements should have the following form:

```
for (initialization; condition; update) {

statements

}

for (variable in object) {

if (filter) {

statements

}

}
```

The first form should be used with arrays and with loops of a predeterminable number of iterations.

The second form should be used with objects. Be aware that members that are added to the prototype of the object will be included in the enumeration. It is wise to program defensively by using the hasOwnProperty method to distinguish the true members of the object:

for (variable in object) {

if (object.hasOwnProperty(variable)) {

statements

}

}

3.8.6 while Statement

A while statement should have the following form:

```
while (condition) {

statements

}
```

3.8.7 do Statement

A do statement should have the following form:

```
do {

statements

} while (condition);
```

Unlike the other compound statements, the do statement always ends with a ; (semicolon).

3.8.8 switch Statement

A switch statement should have the following form:

```
switch (expression) {

case expression:

statements

default:

statements

}
```

Each case is aligned with the switch. This avoids over-indentation. A case label is not a statement and should not be indented like one.

Each group of statements (except the default) should end with break, return, or throw. Do not fall through.

3.8.9 try Statement

The try class of statements should have the following form:

```
try {

statements

} catch (variable) {

statements

}    try {
```

Statements

} catch (variable) {

Statements

} finally {

statements

}

3.8.10 continue Statement

Avoid use of the `continue` statement. It tends to obscure the control flow of the function.

3.8.11 with Statement

The `with` statement should not be used.

3.9 Whitespace

Blank lines improve readability by setting off sections of code that are logically related.

Blank spaces should be used in the following circumstances:

- A keyword followed by `(` (left parenthesis) should be separated by a space. `while (true) {`
- A blank space should not be used between a function value and its `(` (left parenthesis). This helps to distinguish between keywords and function invocations.
- All binary operators except `.` (period) and `(` (left parenthesis) and `[` (left bracket) should be separated from their operands by a space.
- No space should separate a unary operator and its operand except when the operator is a word such as `typeof`.
- Each `;` (semicolon) in the control part of a `for` statement should be followed with a space.
- Whitespace should follow every `,` (comma).

Bonus Suggestions

3.9.1 {} and []

Use `{}` instead of `new Object()`. Use `[]` instead of `new Array()`.

Use arrays when the member names would be sequential integers. Use objects

when the member names are arbitrary strings or names.

3.9.2 , (comma) Operator

Avoid the use of the comma operator. (This does not apply to the comma separator, which is used in object literals, array literals, `var` statements, and parameter lists.)

3.9.3 Block Scope

In JavaScript blocks do not have scope. Only functions have scope. Do not use blocks except as required by the compound statements.

3.9.4 Assignment Expressions

Avoid doing assignments in the condition part of `if` and `while` statements. Is `if (a = b) {` a correct statement? Or is `if (a == b) {` intended?

Avoid constructs that cannot easily be determined to be correct.

3.9.5 === and !== Operators.

Use the `===` and `!==` operators. The `==` and `!=` operators do type coercion and should not be used.

3.9.6 Confusing Pluses and Minuses

Be careful to not follow a + with + or ++. This pattern can be confusing. Insert parens between them to make your intention clear.

```
total = subtotal + +myInput.value;
```

is better written as

```
total = subtotal + (+myInput.value);
```

so that the + + is not misread as ++.

3.9.7 eval is Evil

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

`eval` has aliases. Do not use the `Function` constructor. Do not pass strings to `setTimeout` or `setInterval`.

REVISION HISTORY

Rev.	Date	Author	Summary
1	02-.21-2019	brian.fitzsimmons	Created 5.11-CO branch from trunk

LIST OF FIGURES

End of Document