Front-End Coding Standards

Arthur Nogueira,
last updated September 04, 2015

Front End Coding Standards

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 and CSS code is sent directly to the public. It should always be of publication quality.

Neatness counts.

1 JavaScript

1.1 Namespaces

All functions and objects will belong to namespaces. The namespace should be:

- prefixed with whatever customer root domain is such as “skynet” or “oscorp”;
- reflect the folder structure where the script is located relative to the scripts folder;
- suffixed with the specific page or area of functionality it supports.

```
skynet.widget.datatable
oscorp.util.actions
```

1.2 General Coding Standards

JavaScript Code will, with few exceptions, follow the Java Code standards.

- Hungarian notation will be used for local variables and method parameters.

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.

1.3 Indentation

Use a tab instead of spaces:

- The width of a tab character can be adjusted per editor and the other coders can view your code in the way they feel comfortable with, not in the way you prefer.
- Tabs are easy to select, because that's their sole meaning, spaces on the other hand, have many meanings, so you can't just find & replace space characters.
- **Both** can produce a larger filesize, but the size is not significant over local networks, and the difference is eliminated by minification.

1.4 Line Length

Avoid lines longer than 180 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 from the start of the line.

1.5 Comments and Documentation

All functions and objects will be documented using JSDoc standards. JSDoc will be integrated into the application build process to generate the documentation. The build process should generate the JavaScript documentation.

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 and for commenting out.

1.6 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.

The var statements should be the first statements in the function body.

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

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

Never use reserved words as variable names.

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.

Use of global variables should be minimized. Implied global variables should never be used.

3.1.7 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 line 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.

```
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.

```
function getElementsByClassName(className)
{
      var results = [];

      walkTheDOM(document.body, function (node)
      {
            var a; // array of class names
            var i; // loop counter

            var c = 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 set
            // of results.

            if (c)
            {
                  a = c.split(' ');

                  for (i = 0; i < a.length; i += 1)
                  {
                        if ( a[i] === className )
                        {
                              results.push(node);
                              break;
                        }
                  }
            }
      });

      return results;
}
```

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be wrapped in parentheses 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);
                  }
            }
      };
}());
```

1.8 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 character of a name. It is sometimes used 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.

Constructor functions which 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.)

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

The first lower case letter should be “i” for integers, “s” for strings, “o” for objects, “a” for arrays and “fn” for functions.

1.9 Statements

1.9.1 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 which 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.

1.9.2 Compound Statements

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

-
  - The enclosed statements should be indented four more spaces.
  - Braces will always be placed on a separate line.
  - 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.

1.9.3 Labels

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

1.9.4 **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.

When returned an Object the brackets should stay in line with the return statement.

```
return value;

return {
```

1.9.5 **if** Statement

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

```
if (condition)
{
      statements;
}

if ( condition() )
{
      statements;
}

if ( condition() )
{
      statements;
}
else
{
      statements;
}

if (condition)
{
      statements;
}
else if ( condition() )
{
      statements;
}
else
{
      statements;
}
```

```
if ( (condition) && (condition)
      && (condition) && (condition) )
{
      statements;
}

if (  ( condition() && condition() )
      || ( condition() && condition() )  )
{
      statements;
}
```

1.9.6 **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
    }
}
```

1.9.7 **while** Statement

A `while` statement should have the following form:

```
while (condition)
{
    statements
}
```

1.9.8 **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).

1.9.9 **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.

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

1.9.10 **try** Statement

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

```
try
{
    statements
}
catch (variable)
{
    statements
}
try
{
    statements
}
catch (variable)
{
    statements
}
finally
{
    statements
}
```

1.9.11 **continue** Statement

Avoid use of the `continue` statement. It tends to obscure the control flow of the function. Use if/else logic.

1.9.12 **with** Statement

The `with` statement should not be used.

1.10 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).

1.11 **{}** 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.

```
oObject1 = {
      parameter1       : value,
      param2           : value,
      parameter3       : value
};

aArray : [
      "VALUE1",
      "VALUE2",
      "VALUE3"
];

oObject2 = {
      aArray1 : aArray,
      aArray2 : [
            "VALUE1",
            "VALUE2",
            "VALUE3"
      ]
}
```

1.12 , (comma) Operator

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

1.13 Block Scope

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

1.14 Assignment Expressions

Avoid doing assignments in the condition part of `if` and `while` statements.

Is

```
if (a = b)
{
```

a correct statement? Or was

```
if (a == b)
{
```

Intended? Avoid constructs that cannot easily be determined to be correct.

1.15 === and !== Operators.

It is almost always better to use the `===` and `!==` operators. The `==` and `!=` operators do type coercion. In particular, do not use `==` to compare against false values.

1.16 Confusing Pluses and Minuses

Be careful to not follow a `+` with `+` or `++`. This pattern can be confusing. Insert parentheses between them to make your intention clear. For example,

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

is better written as

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

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

1.17 **eval** is Evil

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

1.18 Object Names

Object names should follow the rules below:

- Start lower case
- Camel Case

2 CSS

2.1 CSS Version

Since we are targeting IE9+, FF4+, Chrome 10+ CSS 3.0 can be used.

2.2 Organization of CSS Files

CSS code must be placed in separate files. Use element styles only when absolutely necessary.

CSS files should be descriptive and meaningful name, using the same standard for *.js and *.jsp files for example, smartpoint.css

2.3 Naming of Classes and Ids

The `class` attribute should be preferred over `id` as CSS identifier. Use `ids` if there is only one instance of that element on a page which needs unique CSS styles. Ids and classes named as follows:

-
  - Meaningful names with (-) as a word separator.
  - Always use lowercase (it may cause browser problems otherwise).

2.4 CSS Syntax

- Add opening curly bracket on the same line as the selector identifier.
- Common designs for inputs, buttons, form and other html elements for whole page or application should be placed at the top of the stylesheet
- Group the rules for same type of elements and its pseudo-elements together.
- Use spacing while describing attribute of a class to improve readability.
- Use empty line between different CSS class to improve readability.
- For visual emphasis use space to separate property and value.
- Properties should be in alphabetical order.
- Try to use shorthand property instead of long property because it will increase readability. Use background-color, background-position, background-image, and background-repeat etc. instead of background.
- Use em instead of px for sizing if possible, line heights etc to allow visitors to use the built-in browser text sizing.
- Use semicolon at the end of each statement even if it is not required.
- Try to avoid presentation specific words as identifier, like black, green etc.

3 HTML

3.1 HTML Syntax

- Always declare a Doctype.
- Use meaningful title tags.
- Use descriptive meta tags.
- Give ids to structure rather than presentation of the document so that it will be easier to refactor like external_link not red_link or main_content rather than left_column.
- Use underscore not hyphen and make all lowercase
- Use Semantic HTML, for example, <h2></h2> not .
- Use an unordered list (<ul>) for Navigation.
- Close all tags.
- Use lower-case markup
- Use alt attributes with images
- Use title attributes with links (when needed).
- Use fieldset and label in web forms.
- Use modular IE fixes.
- Validate HTML code.