Coding Conventions

Hardik Patel
Mindstix Labs

Code Conventions for JS

- This is a set of coding conventions and rules for use in JavaScript programming.

- 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.

JavaScript Files

- JavaScript programs should be stored in and delivered as **‘.js’** files.

- **Use lowercase file names.** Most web servers (Apache) and operating systems (Unix, Linux) are case sensitive about file names. So, ‘myProgram.js’ cannot be accessed as ‘myprogram.js’.

- Other web servers (IIS) and operating systems are not case sensitive. So, ‘MyProgram.js’ can be accessed as ‘myProgram.js’ or ‘myprogram.js’.

- If you use a mix of upper and lower case, you have to be extremely consistent.

- If you move from a case insensitive, to a case sensitive server, even small errors can break your program. To avoid these problems, always use lower case file names (if possible).

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 file size, but the size is not significant over local networks, and the difference is eliminated by minification.

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.**

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 humour 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.

Comments

- 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.

- **JSDoc Comments 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.
 */
```

Comments

- **JSDoc Comments Indentation**

```
/**
 * 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.
 */
var myFunction = function(foo) {
    return 5;
};
```

- **Top/File Level Comments**

- A copyright notice and author information are optional.

- File overviews are generally recommended whenever a file consists of more than a single class definition. The top level comment is designed to orient readers unfamiliar with the code to what is in this file. If present, it should provide a description of the file's contents and any dependencies or compatibility information. As an example:

Comments

```
/**
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 */
```

- **Class Comments**

- Classes must be documented with a description

```
/**
 * 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.
 */
var myClass = function(arg1, arg2) {
    // ...
};
```

Comments

- **Method 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 {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 myFunction(obj) {
    // ...
}
```

Comments

- **Property Comments**

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

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 minimised.
- 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.

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.

```
  function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
    }

    return inner(0, 1);
}
```

Function Declarations

- 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 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;
}
```

Function Declarations

- 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.

```
var myFunction = function (e) {
    return false;
};
```

- Use of global functions should be minimised.

- 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.

Function Declarations

```
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);
            }
        }
    };
}());
```

Names

- Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 .. 9), and _ (underscore). 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 _ (underscore) 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 capitalisation convention is the only defence 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.)

Statements

- **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.

Statements

- **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.

Statements

- **Labels**

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

- **`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.

- **`if` Statement**

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

Statements

```
if (condition) {
    statements
}

if (condition) {
    statements
} else {
    statements
}

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

Statements

- **`for` Statement**
  - A `for` class statements should have the following form:

```
for (initialisation; 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
    }
}
```

Statements

- **`while` Statement**
  - A `while` statement should have the following form:

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

- **`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).

Statements

- **`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.

- **`try` Statement**

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

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

Statements

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

- **`continue` Statement**
  - Avoid use of the `continue` statement. It tends to obscure the control flow of the function.
- **`with` Statement**
  - The with statement should not be used.

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

- **`{}` 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.
- **`,` (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.)
- **Block Scope**
  - In JavaScript blocks do not have scope. Only functions have scope. Do not use blocks except as required by the compound statements.

Bonus Suggestions

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

Bonus Suggestions

- **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 `++`.

- **`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`.