JavaScript Coding Conventions Day 27

Having conventions make things easier to understand since codebases follow the same patterns.

Naming and declaration of variables, functions, classes, and other entities are consistent in their naming scheme. This means that we can understand code easier and makes things more predictable.

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. Diversity of people, culture, and perspective is a good and valueable thing. Diversity of programming styles is a bad thing. It creates friction, impeding the ability of people to work together. It can make it easier for bugs to form and hide.

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, minification, and compression.

Whitespace.

Where possible, these rules are consistent with centuries of good practice with literary style. Deviations from literary style should only be tolerated if there is strong evidence of a significant benefit. Personal preference is not a significant benefit.

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

Blank spaces should always be used in the following circumstances:

•A keyword followed by (*left parenthesis* should be separated by a space. Spaces are used to make things that are not invocations look less like invocations, so for example, there should be space after **if** or **while**.

```
while (true) {
```

• A blank space should not be used between a function value and its invoking (*left parenthesis*. This helps to distinguish between keywords and function invocations.

•The word **function** is always followed with one space.

•No space should separate a unary operator and its operand except when the operator is a word such as **typeof**.

• All binary operators should be separated from their operands by a space on each side except .Period and (left parenthesis and [left bracket.

•Every , *comma* should be followed by a space or a line break.

•Each ; *semicolon* at the end of a statement should be followed with a line break.

•Each ; *semicolon* in the control part of a for statement should be followed with a space.

Every statement should begin aligned with the current indentation. The outermost level is at the left margin. The indentation increases by 4 spaces when the last token on the previous line is { left brace, [left bracket, (left paren. The matching closing token will be the first token on a line, restoring the previous indentation.

The ternary operator can be visually confusing, so wrap the entire ternary expression in parens. The condition, the ? question mark, and the : colon always begins a line, indented 4 spaces.

```
let integer = function (
        value,
        default_value
    ) {
        value = resolve(value);
        return (
            typeof value === "number"
            ? Math.floor(value)
            : (
                typeof value === "string"
                ? value.charCodeAt(0)
                : default_value
            )
        );
    };
```

Clauses (**case**, **catch**, **default**, **else**, **finally**) are not statements and so should not be indented like statements.

Use of tabs invites confusion, argument,and crying, with little compensating value. Do not use tabs. Use space.

Comments

Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly your future self) who will need to understand what you have done and why. 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

```
// Set i to zero.

   I = 0;
```

Use line comments, not block comments. The comments should start at the left margin.

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

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

Function Declarations

All functions should be declared before they are used. Inner functions should come after the outer function's variable declarations. 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) {
        let 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) {
        let results = [];
        walkTheDOM(
            document.body,
            function (node) {
                let array;                    // array of class names
                let 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.

```
let collection = (function ( ) {
    let keys = [ ];
    let values = [ ];

    return {
        get: function (key) {
            let at = keys.indexOf(key);
            if (at >= 0) {
                return values[at];
            }
        },
        set: function (key, value) {
            let at = keys.indexOf(key);
            if (at < 0) {
                at = keys.length;
            }
            keys[at] = key;
            values[at] = value;
        },
        remove: function (key) {
```

```
            let 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 _*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 closure. 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 missing, so the capitalization convention is an important defense.

Global variables should be avoided, but when used should be in ALL_CAPS.

Statements

Simple Statements

Each line should contain at most one statement. Put a ; *semicolon* at the end of every statement that does not end with a {block}. 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, invocations, and delete.

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 should be avoided. Only these statements should be labeled: **while**, **do**, **for**, **switch**.

Return Statement

The return value expression must start on the same line as the **return** keyword in order to avoid semicolon insertion.

If Statement

A **if** statement should have one of these forms:

```
if (condition) {
    statements
}

if (condition) {
    statements
} else {
    statements
}

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

for Statement

The **for** should be avoided, preferring the array methods if possible. When the **for** statement is used, it should one of these forms:

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

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

while Statement

A **while** statement should have the following form:

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

do Statement

A **do** statement should have this form:

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

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

Switch Statement

A **switch** statement should be avoided, but when used should have this 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** tatement should have this form:

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

The **finally** clause should be avoided. If it is used, it should have this form:

```
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.

{ } 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, and parameter lists.) Having a character that is sometimes a separator and sometimes an operator is a source of confusion.

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 produce false positives and false negatives, so they should not be used.

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 + Number(myInput.value);
```

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

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**.

Google JavaScript Style Guide:

https://google.github.io/styleguide/jsguide.html

Best Wishes Lux Tech Academy.