Udacity Frontend Nanodegree Style Guide

Introduction

This style guide acts as the official guide to follow in your projects. Udacity evaluators will use this guide to grade your projects. There are many opinions on the "ideal" style in the world of Front-End Web Development. Therefore, in order to reduce the confusion on what style students should follow during the course of their projects, we urge all students to refer to this style guide for their projects.

General Formatting Rules

Trailing Whitespace

Remove trailing white spaces.

Trailing white spaces are unnecessary and can complicate diffs.

Not Recommended:

```
const name = "John Smith";__
```

Recommended:

```
const name = "John Smith";
```

If using Sublime Text, this can be done automatically each time you save a file by adding the following to your User Settings JSON file (you should be able to find this within Sublime Text's menu):

```
"trim_trailing_white_space_on_save": true
```

Indentation

Indentation should be consistent throughout the entire file. Whether you choose to use tabs or spaces, or 2-spaces vs. 4-spaces - just be consistent!

General Meta Rules

Encoding

Use UTF-8 (no BOM).

Make sure your editor uses UTF-8 as character encoding, without a byte order mark.

Comments

Use comments to explain code: What does it cover, what purpose does it serve, and why is the respective solution used or preferred?

You can optionally document your JavaScript functions with JSDoc, a documentation generator and standard for writing code comments. Its benefits include providing a specification to hold your comments to, and the command line `jsdoc` tool that will generate a website for your documentation. JSDoc provides many annotations you can use to document your code, but we only recommend that you utilize a small subset of the available options:

- @constructor: used to document a class, a.k.a. a function meant to be called with the `new` keyword.
- @description: used to describe your function; this tag allows you to include HTML markup if desired as well.
- @param: used to describe the name, type, and description of a function parameter.
- @returns: document the type and description of a function's return value.

This example shows how to document a class constructor (note the use of `/**` to start the comment block; that's important):

```
/**
* @description Represents a book
* @constructor
* @param {string} title - The title of the book
* @param {string} author - The author of the book
*/
function Book(title, author) {
    ...
}
```

And here is a function with parameters that returns a value; note the lack of description for the parameters, since in this case they're pretty self-explanatory:

```
/**
* @description Adds two numbers
* @param {number} a
* @param {number} b
* @returns {number} Sum of a and b
*/
function sum(a, b) {
    return a + b;
}
```

Feel free to go above and beyond and use more annotations if desired.

Action Items

Mark todos and action items with `TODO:`.

Highlight todos by using the keyword `TODO` only, not other formats like `@@`. Append action items after a colon like this: `TODO: action item`.

Recommended:

```
// TODO: add other fruits
```

JavaScript Language Rules

Variable Declaration

There are three ways to declare a variable in JavaScript:

- `const`
- `let`
- `var`

When declaring variables, you should declare them using the keywords in the order listed above. Declare your variables with `const`, first. If you find that you need to reassign the variable later, use `let`. There isn't a good reason to use the `var` keyword anymore for variable declaration.

Semicolons

Always use semicolons.

Relying on implicit insertion can cause subtle, hard to debug problems. Semicolons should be included at the end of function expressions, but not at the end of function declarations.

Not Recommended:

```
const foo = () => {
    return true // Missing semicolon
} // Missing semicolon

function foo() {
    return true;
}; // Extra semicolon
```

Recommended:

```
const foo = () => {
    return true;
};

function foo() {
    return true;
}
```

Wrapper Objects for Primitive Types

There's no reason to use wrapper objects for primitive types, plus they're dangerous. However, type casting is okay.

Not Recommended:

```
const x = new Boolean(0);
if (x) {
    alert('hi');    // Shows 'hi' because typeof x is truthy object
}
```

Recommended:

```
const x = Boolean(false);
if (x) {
    alert('hi');    // Show 'hi' because typeof x is a falsey boolean
}
```

Closures

Yes, but be careful.

The ability to create closures is perhaps the most useful and often overlooked feature in JavaScript. One thing to keep in mind, however, is that a closure keeps a pointer to its enclosing scope. As a result, attaching a closure to a DOM element can create a circular reference and thus, a memory leak.

Not Recommended:

```
function foo(element, a, b) {
    element.onclick = function() { /* uses a and b */ }
}
```

Recommended:

```
function foo(element, a, b) {
    element.onclick = bar(a, b);
}

function bar(a, b) {
    return function() { /* uses a and b */ }
}
```

for, for-in and forEach

Array

`forEach` or `for` loops are preferred over `for-in` loops when iterating over an array.

Not Recommended:

```
myArray = ['a', 1, 'etc'];
for (const indexNum in myArray) {
    console.log(myArray[indexNum]);
}

const starWars = {
    "creatures": [
        {
            "name": "bantha",
            "face": "furry"
        },
        {
            "name": "loth-cat",
            "face": "toothy"
        }
    ]
};
for (const i in starWars.creatures) {
    console.log(starWars.creatures[i].name);
    console.log(starWars.creatures[i].face);
};
```

Recommended:

```
mySimpleArray = ['a', 1, 'etc'];
mySimpleArray.forEach(function(val) {
    console.log(val);
});

const starWars = {
    "creatures": [
        {
            "name": "bantha",
            "face": "furry"
        },
        {
            "name": "loth-cat",
            "face": "toothy"
        }
    ]
};
starWars.creatures.forEach(function(creature) {
    console.log(creature.name);
    console.log(creature.face);
});
```

// or

```
myArray = ['a', 1, 'etc'];
for (let indexCount = 0; indexCount < myArray.length; indexCount++) {
    console.log(myArray[indexCount]);
};
```

Object

`for-in` loops are used to loop over keys in an object. This can be error prone because `for-in` does not loop from `0` to `length - 1` but over all the present keys in the object and its prototype chain.

If possible, organize data so it is not necessary to iterate over objects. If that isn't possible, wrap the content of the `for-in` loop in a conditional statement to prevent it from from iterating over the prototype chain.

Not Recommended:

```
myObj = {'firstName':'Ada','secondName':'Lovelace'};
for (const key in myObj) {
    console.log(myObj[key]);
}
```

Recommended:

```
myObj = {'firstName':'Ada','lastName':'Lovelace'};
for (const key in myObj) {
    if (myObj.hasOwnProperty(key)) {
        console.log(myObj[key]);
    }
}
```

Multiline String Literals

Do not use.

The whitespace at the beginning of each line can't be safely stripped at compile time; whitespace after the slash will result in tricky errors; and while most script engines support this, it is not part of the specification.

Not Recommended:

```
const myString = 'A rather long string of English text, an error message \
    actually that just keeps going and going -- an error \
    message that is really really long.';
```

Recommended:

```
const myString = 'A rather long string of English text, an error message' +
    'actually that just keeps going and going -- an error' +
    'message that is really really long.';
```

Array and Object Literals

Use Array and Object literals instead of Array and Object constructors.

Not Recommended:

```
const myArray = new Array(x1, x2, x3);

const myObject = new Object();
myObject.a = 0;
```

Recommended:

```
const myArray = [x1, x2, x3];

const myObject = {
    a: 0
};
```

JavaScript Style Rules

Naming

In general, `functionNamesLikeThis`, `variableNamesLikeThis`, `ClassNamesLikeThis`, `methodNamesLikeThis`, `CONSTANT_VALUES_LIKE_THIS` and `filenameslikethis.js`.

Code Formatting

Because of implicit semicolon insertion, always start your curly braces on the same line as whatever they're opening.

Recommended:

```
if (something) {
    // Do something
} else {
    // Do something else
}
```

Single-line array and object initializers are allowed when they fit on one line. There should be no spaces after the opening bracket or before the closing bracket:

Recommended:

```
const array = [1, 2, 3];
const object = {a: 1, b: 2, c: 3};
```

Multiline array and object initializers are indented one-level, with the braces on their own line, just like blocks:

Recommended:

```
const array = [
    'Joe <joe@email.com>',
    'Sal <sal@email.com>',
    'Murr <murr@email.com>',
    'Q <q@email.com>'
];

const object = {
    id: 'foo',
    class: 'foo-important',
    name: 'notification'
};
```

Parentheses

Only where required.

Use sparingly and in general only where required by the syntax and semantics.

Strings

For consistency single-quotes (`'`) are preferred over double-quotes (`"`). This is helpful when creating strings that include HTML:

Recommended:

```
const element = '<button class="btn">Click Me</button>';
```

** Notable exception to this is in JSON objects: double quotes are required per the JSON specification

Tips and Tricks

True and False Boolean Expressions

The following are all false in boolean expressions:

- `null`
- `undefined`
- `''` the empty string
- `0` the number

But be careful, because these are all true:

- `'0'` the string
- `[]` the empty array
- `{}` the empty object

Conditional Ternary Operator

The conditional ternary operator is recommended, although not required, for writing concise code. Instead of this:

Not Recommended:

```
if (val) {
    return foo();
} else {
    return bar();
}
```

You can write this:

Recommended:

```
return val ? foo() : bar();
```

&& and ||

These binary boolean operators are short-circuited and evaluate to the last evaluated term. `||` has been called the default operator because instead of writing this:

Not Recommended:

```
const foo = (name) => {
    const theName;
    if (name) {
        theName = name;
    } else {
        theName = 'John';
    }
};
```

You can write this:

Recommended:

```
const foo = (name) => {
    const theName = name || 'John';
};
```

`&&` is also used for shortening code. For instance, instead of this:

Not Recommended:

```
if (node) {
    if (node.kids) {
        console.log(node.kids);
    }
}
```

You can do this:

Recommended:

```
if (node && node.kids) {
    console.log(node.kids);
}
```