HTML CSS JavaScript Git

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:

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

Recommended:

```
var 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

var

Always declare variables with `var`.

When you fail to specify `var`, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable lives.

Constants

If a value is intended to be constant and immutable, it should be given a name in all capital letters, like `CONSTANT_VALUE`. Never use the `const` keyword as it's not supported by all browsers at this time.

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:

```
var foo = function() {
    return true // Missing semicolon
} // Missing semicolon

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

Recommended:

```
var foo = function() {
    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:

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

Recommended:

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

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 nd b */ }
}
```

for-in loop

Only for iterating over keys in an object/map/hash.

`for-in` loops are often incorrectly used to loop over the elements in an array. This is however very error prone because it does not loop from `0` to `length - 1` but over all the present keys in the object and its prototype chain.

Not Recommended:

```
for (var key in arr) {
console.log(arr[key]);
}
```

Recommended:

```
var len = array.length;
for (var i = 0; i < len; i++) {
    console.log(array[i]);
}

// or...

array.forEach(function(val) {
    console.log(val);
});
```

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:

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

Recommended:

```
var myString = 'A rather long string of English text, an error
    '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:

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

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

Recommended:

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

var 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:

```
var array = [1, 2, 3];
var 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:

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

var 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:

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

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:

```
function foo(name) {
    var theName;
    if (name) {
        theName = name;
```

```
    } else {
        theName = 'John';
    }
}
```

You can write this:

Recommended:

```
function foo(name) {
    var 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);
}
```