Interactivate Style Guide

Table of Contents

Preface

Except for PHP, Javascript is possibly the quirkiest and most annoying language you will come across. However, if done properly, it can be a joy to write code in. This guide will help you avoid common pitfalls and keep the puppy and unicorn fatalities to a minimum.

Compare the folowing two code blocks

General Information

- Create at most one global variable per interactivate project. If you feel the need to use multiple global variables, put them in a namespace.
- Be consistent with the existing coding style. If you are maintaining code that

does not follow the style guide, either fix the old style or keep using the old style. Switching styles in the middle of code can be very confusing to programmers, causing them to jump up and down, screeching like monkeys.

Variables

- ALWAYS declare variables with `var` the first time they are used. If you do not do this, the variable will become a global variable and a unicorn will die

```
//bad. A unicorn just died
function square(x) {
    y = x * x; //throw new UnicornDiedException('You awful person!');
    return y;
}

alert(square(3));
alert(y); // :(

//good. The unicorn is saved
function square(x) {
    var y = x * x;
    return y;
}

alert(square(3));
alert(y); //y is not defined
```

- Put all variable declarations at the top of a file or function. They can either be in the same command or different ones. If you put them in one statement, each new variables goes in its own line.

```
    //bad
    for(var i = 0; i < 10; i++) {
        //do some code
    }

    var num = 4;

    for(var j = 0; j < 10; j++) {
        //do some more code
    }

    //good, multiple var statements

    var num = 4;
    var i;
    var j;
    for(i = 0; i < 10; i++) { //no var
        //some code
    }

    for(j = 0; j < 10; j++) {
        //more code
    }

    //good, single var statement

    var num = 4,
        i,
        j;


    for(i = 0; i < 10; i++) { //no var
        //some code
    }

    for(j = 0; j < 10; j++) {
        //more code
    }
```

Primitives

In Javascript the primitive types are:

- `string`
- `number`
- `boolean`

When you assign values to these, use the literal values

```
//bad
var num = new Number(3);
console.log(num); // => Number {}
var bool = new Boolean(true);
console.log(bool); // => Boolean {}
var str = new String("Hello, World");
console.log(str); // => String {0: "H", 1: "e", 2: "l", 3: "l", 4: "o", 5: ","
, 6: " ", 7: "W", 8: "o", 9: "r", 10: "l", 11: "d", length: 12}
//Clearly, these are exactly what you wanted.

//good
var num = 3;
console.log(num); // => 3
var bool = true;
```

```
console.log(bool); // => true
var str = "Hello, World";
console.log(str); // => "Hello, World"
```

Conditional Statements and Comparison

- When you are comparing two values, unless one of the values is `null`, ALWAYS use `===` instead of `==`. If you use `===`, it can lead to some very odd results. `===` checks if both the type and the value are the same, `==` only checks the value

```
//BAD CODE.
'\r\n\t' == 0;      // true. \r\n\t is a newline and a tab.
'' == '0';          // false
0 == ''             // true
0 == '0'            // true. What happened to the transitive property?

//GOOD CODE.

'\r\n\t' === 0      // false
'' === 0            // false
0 === ''            // false
0 === '0'           // false
```

- Those are only some of a few examples of pitfalls of using `==`. Use `===` and you will be fine
- Use shortcuts in your `if` statements.
- To test if a value is true or false, use `if(value)` and `if(!value)`

```
//bad
if(bool === true) {
    //do something if bool is true;
}
if(bool === false) {
    //do something if bool is false
}

//good

if(bool) {
    //do something if true
}

if(!bool) {
    //do something if false
}
```

- To check if a variable exists, use `if(variable)`. This is often used to provide built-in functions in old browsers that don’t support them

```
//bad
if (typeof variable === 'undefined') {
    //define variable here
}

//good

if(variable) {
    //define variable here
}
```

Arrays

- Similarly to primitives, use the literal `[]` instead of `new Array`

```
//bad
var array = new Array('Apple', 'Banana', 'Orange');

//good
var array = ['Apple', 'Banana', 'Orange'];
```

- To check equality in arrays, iterate through the array and check if the values are the same

```
//bad
var arr1 = [1, 2, 3, 4];
var arr2 = [1, 2, 3, 4];

arr1 == arr2;   // false
arr1 === arr2;  // false

//good

function arrayEquals(array1, array2) {
    var i;

    if(array1.length !== array2.length) {
        return false;
    }

    for(i = 0; i < array1.length; i++) {
        if(array1[i] !== array2[i]) {
            return false;
        }
    }

    return true;
}

arrayEquals([1, 2, 3, 4], [1, 2, 3, 4]); //true. Note that if you have objects
or arrays inside the array, you might need to make a deep equals
```

- All arrays should contain the same type. If you feel the need to use strings and numbers or any other combination of types in the same array, your code can probably be refactored to fix it.

- To iterate through the array, use a regular `for` loop instead of a `for .. in` loop. This will save you lots of trouble if some elements of the array are `undefined`

```
//bad
var array = [1, 2, 3, 4, 5];
for(var i in array) {
    print(array[i]);
}

//good

var array = [1, 2, 3, 4, 5];
for(var i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

Objects

- Use the literal `{}` instead of the `new Object()` constructor.

```
//bad
var obj = new Object();
obj.prop1 = 'Hi';
obj.prop2 = 'Bye';

//good
var obj = {
    prop1: 'Hi',
    prop2: 'Bye'
};
```

- Use dot notation(`obj.propertyName`) for accessing array members when you know the property name.

```
//bad

var obj = {
    prop1: 'Hi',
    prop2: 'Bye'
};

alert(obj['prop1']);

//good

var obj = {
    prop1: 'Hi',
    prop2: 'Bye'
};

alert(obj.prop1);
```

- Use bracket notation(`obj['propertyName']`) for when you don't know the property name or when the property name is stored in a variable

```
//bad.
var obj = {
    prop1: 'Hi',
    prop2: 'Bye'
};
var propName = 'prop1';

var property = eval('obj.' + propName); //Doing this is an awful idea. Don't d
o this. This is bad. This is evil. Never do this. This is awful.  h̍–is unḩo̅ly
radiańcé destro□ying all enlíĝhtenment, evals lea̎kịn ̇g fr–oṃ ¸your eyeș/ ˑl̍ike l
iquid pain, the song of eval parsing will extinguish the voices of mortal man
from the sphere I can see it can you see͜□ít̋ ̰□ it is beautiful the final snuff
ing of the lies of Man ALL IS LOŠ□Ṭ ALL IS LOST the pon⁄y he comes he c–o̡mes he
comes the □ichor peŗmeates all MY FACE MY FACE ᵒh god no NO NOO̳OO N0 stop the a
n*–□g□□ţ□□□□̱es a̤□r̉e n͝ot rȅ□ă□ĩ□̣ ZA͠LGO IS□□□□̦ TO̡N~ȳ̲ THÉ□ P̃O ̍Ň̊Y¸ H□̧⁄Ẹ□□́̆* □□–̦
C□□□⁄ ̦□O□M̗ʻ. If yoṷ Ê□~Ş□̣


//good code

var obj = {
    prop1: 'Hi',
    prop2: 'Bye'
};
var propName = 'prop1';
return obj[propName];
```

- Make sure to use `hasOwnProperty(prop)` when using a `for...in` loop. This makes sure to check that the property is of that object and that object only and not inherited from a superclass

```
//bad
var obj = {
    a: 1,
    b: 2
};

for(var prop in obj) {
    console.log(prop + ": " + obj[prop]);
}

//good
var obj = {
    a: 1,
    b: 2
};

for(var prop in obj) {
    if(obj.hasOwnProperty(prop)) {
        console.log(prop + ": " + obj[prop]);
    }
}
```

Functions

- Functions inside functions must be declared with `var fn = function() {}`

```
//bad

function outer() {
    function inner() { //is not cross-browser
        console.log('hi');
    }
    inner();
}

//good

function outer() {
    var inner = function() {
        console.log('hi');
    }
}
```

- When you want to specify default parameters, use the following syntax

```
function power(base, exponent) {
    var defaultBase = 2;
    var defaultExponent = 3;

    //if the parameters are undefined, set it to the default
    base = typeof base !== 'undefined' ? base : defaultBase;
    exponent = typeof exponent !== 'undefined' ? exponent: defaultExponent;

    return Math.pow(base, exponent);
}
power();        // 8
power(3, 4);    // 81
power(3);       // 3
```

Brackets and Whitespace

- Use 4 spaces or tabs. Tabs will be converted to 4 spaces upon build.
- Curly braces go on the same line as the statement, separated by one space

```
//bad
function fn(x){
var i;
for(i = 0; i < 10; i++){
console.log(i);
}
}

//worse
function fn(x)
{
var i;
for(i = 0; i < 10; i++)
{
console.log(i);
}
}

//good

function fn(x) {
    var i;
    for(i = 0; i < 10; i++) {
        console.log(i);
    }
}
```

- Always use braces, even when they are not necessary

```
//bad
var i;
for(i = 0; i < 10; i++)
    console.log(i); //what happens when you add another line to this?

//good
var i;
for(i = 0; i < 10; i++) {
    console.log(i);
}
```

- Place a space between binary operators and operands, but not between unary operators and operands

```
//bad
var a=1;
var b=2;
console.log(a+b);
console.log(a ++);

//good
var a = 1;
var b = 2;
console.log(a + b);
console.log(a++);
```

- End all files with newlines. You must do this because files get added to each other and without that newline syntax errors could arise

Comments

- Comments are necessary in code to provide information for the IDE, other developers, and future you. I have written code that I completely understand but didn't document, and I came back to it six months later. I had absolutely no clue what things did because I did not provide proper documentation. Write your code primarily for other people to read, and secondarily for a computer to read.
- A good general guideline is that the code explains the *how*, and comments explain the *why*. If you feel that you need to use comments to explain how something works, in most cases it is better to refactor the code to make it more readable.

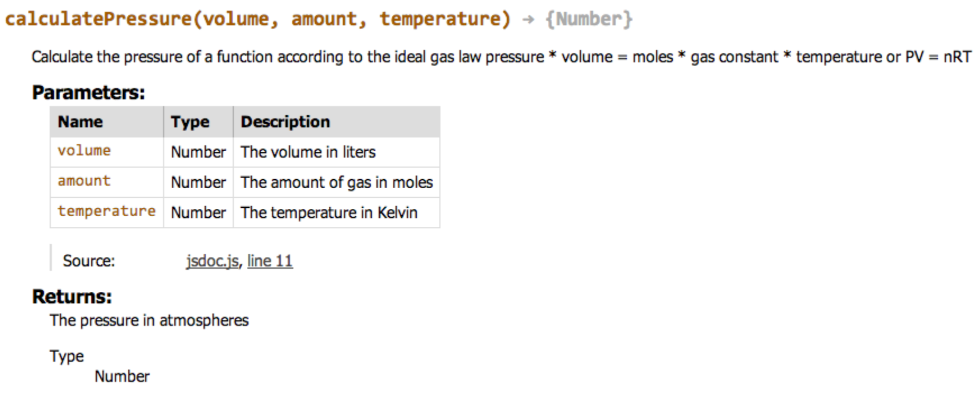
```
//bad
```

```
function calculatePressure(v, n, t) {
    var r = 0.082;
    return r * n * t / v;
}

//This is valid code, but if you don't already know the formula, it is nearly
impossible to understand. Compare it to the following code

//good

//calculate pressure according to the ideal gas law pressure * volume = moles
* gas constant * temperature or PV = nRT
function calculatePressure(volume, moles, temperature) {
    var gasConstant = 0.082;
    return moles * gasConstant * temperature / volume;
}

//In the above example the code explains what it is doing, and the comments ex
plain why it works and why we are using it
```

- JSDoc is a way to provide API documentation. It is used to generate a webpage giving an outline of the code with descriptions and parameter lists of functions and the like. ALL PUBLIC METHODS AND VARIABLES MUST BE DOCUMENTED WITH JSDOC. Learn how to provide JSDoc documentation at http://usejsdoc.org/. Consider the example of calculating pressure with JSDoc

```
/**
* Calculate the pressure of a function according to the ideal gas law pressure
* volume = moles * gas constant * temperature or PV = nRT
*
* @param {Number} volume The volume in liters
* @param {Number} amount The amount of gas in moles
* @param {Number} temperature The temperature in Kelvin
*
* @returns {Number} The pressure in atmospheres
*/

function calculatePressure(volume, moles, temperature) {
    var gasConstant = 0.082;
    return moles * gasConstant * temperature / volume;
}
```

- The JSDoc generated from the above code might look like



calculatePressure(volume, amount, temperature) → {Number}

Calculate the pressure of a function according to the ideal gas law pressure * volume = moles * gas constant * temperature or PV = nRT

Parameters:

Name	Type	Description
volume	Number	The volume in liters
amount	Number	The amount of gas in moles
temperature	Number	The temperature in Kelvin

Source: jsdoc.js, line 11

Returns:

The pressure in atmospheres

Type
Number

Semicolons

- Every time you don't use a semicolon, a cute little puppy is bludgeoned to death with a giant semicolon. Save the puppies.

Commas

- For declaring multiple variables, objects, and arrays, the comma goes on the previous line.

```
//bad
var person = {
    firstName: 'Bob'
  , lastName: 'Joe'
};

//good

var i = 0,
    j = 3,
    k = 4;

var person = {
    firstName: 'Bob',
    lastName: 'Joe'
};
```

- Commas preceding other code are separated by one space

```
//bad
function(param1,param2,param3) {
    //do some stuff
}

//good
function(param1, param2, param3) {
    //do some stuff
}
```

Naming Conventions

- Be descriptive with your variable and function names, and avoid single character names
- use camelCase when naming functions, objects, instances, and regular variables
- use PascalCase when naming classes and constructors
- use ALL_CAPS_WITH_UNDERSCORES when naming constants
- put an underscore before private members in a class

```
//good
function Person(name) {

    //private variables prefixed by an underscore, but they aren't really priv
ate
    this._name = name;
}

Person.prototype.sayHi = function() {
    alert('Hi, my name is ' + this._name);
}
var person = new Person('Alice');
person.sayHi(); //alerts 'Hi, my name is Alice';
```

Use Strict

- Putting `'use strict';` at the top of a Javascript function before any other commands are run sets the function into strict mode. This means that some unexpected behaviors which would fail silently now thrash around and throw an error and actually tell you what is going on. You must put this at the top of every global function, like this

```
//good
function strict() {
    'use strict';
    //some strict code here
}
```

Linting

- All your code must pass the JSHint code quality test without any warnings or errors. These tests are run automatically whenever you save a file or build.

The excellent zalgo text taken from bobince