Prepared By	Krishna Prasad Timilsina
Date	April 30, 2011
Description	Maintain Javascript code quality and standard

How To Write Better Code In Javascript

1. Use === instead of ==

JavaScript uses two different types of equality operators: === | !== and == | !=
When two operands are of the same type and have the same value, then === produces ***true*** and !== produces ***false***.
Eg.

```
" == '0'  // false
0 == "  // true
0 == '0'  // true

false == 'false'  // false
false == '0'  // true

false == undefined  // false
false == null  // false
null == undefined  // true
```

2. Variables

2.1. Use meaningful and pronounceable variable names

Whenever you use a variable, the name of the variable should be useful and understandable for everyone.
Bad:

```
const yyyymmdstr = moment().format('YYYY/MM/DD');
```

Good:

```
const currentDate = moment().format('YYYY/MM/DD');
```

2.2. Do not add unneeded context

If your class/object name tells you something, do not repeat that in your variable name.
Bad:

```
const Car = {
  carMake: 'Honda',
  carModel: 'Accord',
  carColor: 'Blue'
};
```

```
function paintCar(car) {
        car.carColor = 'Red';
}
```

Good:

```
const Car = {
  make: 'Honda',
  model: 'Accord',
  color: 'Blue'
};

function paintCar(car) {
        car.color = 'Red';  // Use only car.color instead of car.carColor
}
```

3. Function

Functions are one of the essential fundamental building blocks of JavaScript.

3.1. Function arguments

A function should not have more than three arguments. Keep it as low as possible.

Bad:

```
function createMenu(title, body, buttonText, cancellable) {   // ... }
```

Good:

```
function createMenu({
  title,
  body,
  buttonText,
  cancellable
}) {
        // ...
}

createMenu({
  title: 'Foo',
  body: 'Bar',
  buttonText: 'Baz',
  cancellable: true
});
```

3.2. Function names should say what they do

If you create any function, the function name should tell you the working of it. By doing this you can understand it in future, it helps you and your co-worker or anyone else.

Bad:

```
/**
 * Invite a new user with its email address
 * @param {String} user email address
 */
function inv(user) { /* implementation */ }
```

Good:

```
function inviteUser(emailAddress) { /* implementation */ }
```

3.3 Arrow function shorthand

Arrow functions are more compact and have shorter syntax than function expression, and it utilizes **=>**, that looks like a fat arrow.

Longhand:

```
function sayHello(name) {
	console.log('Hello', name);
}

setTimeout(function() {
	console.log('Loaded')
}, 2000);

list.forEach(function(item) {
	console.log(item);
});
```

Shorthand:

```
sayHello = name => console.log('Hello', name);

setTimeout(() => console.log('Loaded'), 2000);

list.forEach(item => console.log(item));
```

4. Shorthand for if

This might be trivial, but worth it. While doing ***if***, assignment operators sometimes be omitted.

Longhand:

```
if (likeJavaScript === true) {
	// do something...
}
```

Shorthand:

```
if (likeJavaScript) {
	// do something...
```

```
}
```

Here is another example. If “a” is NOT equal to true, then do something.

Longhand:

```
let a;
if (a !== true) {
    // do something...
}
```

Shorthand:

```
let a;
if (!a) {
    // do something...
}
```

5. Object property shorthand

ES6 provides an even more natural way of assigning properties to objects. If the property name is the same as the key name, you can take advantage of the shorthand notation.

Longhand:

```
const obj = { x:x, y:y };
```

Shorthand:

```
const obj = { x, y };
```

6. Short-Circuit Conditionals

Longhand:

```
if (connected) {
    login();
}
```

Use a combination of a variable (which will be verified) and a function using the ***&&*** (AND operator) between them.

Shorthand:

```
connected && login();
```

7. Implicit return shorthand

We use return keyword to return the final result of a function. Arrow function with a single statement will return the result of its evaluation. The function omits the braces ***{}***. For multi-line statements, it’s necessary to use ***()*** instead of ***{}*** to wrap your function body.

Longhand:

```
function calcCircumference(diameter) {
    return Math.PI * diameter;
```

```
}
```

Shorthand:

```
calcCircumference = diameter => (
    Math.PI * diameter;
)
```

8. Declare variables outside of the for statement

While executing lengthy statements, do not make the engine work any harder than it must. Bad:

```
for (var i = 0; i < items.length; i++) {
    var container = document.getElementById('container');
    container.innerHtml += 'my number: ' + i;
    console.log(i);
}
```

Good:

```
var container = document.getElementById('container');
for (var i = 0, len = items.length; i < len; i++) {
    container.innerHtml += 'my number: ' + i;
    console.log(i);
}
```

9. Reduce globals

Having many global variables is always a bad thing because you can easily forget the name you declare the variable and you accidentally declare it somewhere else. Prevent global variables, because global variables are easily overwritten and it may lead to maintenance issues.
Bad:

```
var name = 'Jeffrey';
 var lastName = 'Way';
 function doSomething() { …
 }
 console.log(name);  // Jeffrey -- or window.name
```

Good:

```
var NameSpace = {
    name: 'Jeffrey',
    lastName: 'Way',
    doSomething: function() { ...
  }
}
console.log(NameSpace.name);  // Jeffrey
```

10. Comment your code

Do not comment on every small thing in your code. Just comment on things that have business logic complexity.
Bad:

```
function hashIt(data) {
  // The hash
  let hash = 0;

  // Length of string
  const length = data.length;

  // Loop through every character in data
  for (let a = 0; a < length; a++) {
        // Get character code.
        const char = data.charCodeAt(a);
        // Make the hash
        hash = ((hash << 5) - hash) + char;
        // Convert to 32-bit integer
        hash &= hash;
  }
}
```

Good:

```
function hashIt(data) {
  let hash = 0;
  const length = data.length;

  for (let a = 0; a < length; a++) {
        const char = data.charCodeAt(i);
        hash = ((hash << 5) - hash) + char;

        // Convert to 32-bit integer
        hash &= hash;
  }
}
```

10.1. Do not leave commented out code in your codebase
If there is code which you no longer want, do not comment on it, remove it from the code.
Bad:

```
doStuff();  // doOtherStuff(); // doSomeMoreStuff(); // doSoMuchStuff();
```

Good:

```
doStuff();
```

11. Do not pass a string to “setInterval” or “setTimeOut”

Eg.
setInterval("document.getElementById('container').innerHTML += 'My new number: ' + i", 3000);

Pass a function name instead of passing a string to setInterval and setTimeOut.

```
setInterval(someFunction, 3000);
```

12. Use consistent capitalization

JavaScript is untyped, so capitalization tells you a lot about your variables, functions, etc. These rules are subjective, so your team can choose whatever they want. The point is, no matter what you choose, just be consistent.

Bad:

```
const DAYS_IN_WEEK = 7;
const daysInMonth = 30;

const songs = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];
const Artists = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {
    //...
}

function restore_database() {
    //...
}

class animal {
    //...
}
class Alpaca {
    //...
}
```

Good:

```
const DAYS_IN_WEEK = 7;
const DAYS_IN_MONTH = 30;

const SONGS = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];
const ARTISTS = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {
```

```
        //...
}

function restoreDatabase() {
        //...
}

class Animal {
        //...
}
class Alpaca {
        //...
}
```

13. How to write nice async code?

Use ***Promise***, instead of using nested callbacks. You can use chainable promise calls.

Avoid:

```
asyncFunc1((err, result1) => {
  asyncFunc2(result1, (err, result2) => {
        asyncFunc3(result2, (err, result3) => {
        console.log(result3)
        })
  })
})
```

Prefer:

```
asyncFuncPromise1()
  .then(asyncFuncPromise2)
  .then(asyncFuncPromise3)
  .then((result) => console.log(result))
  .catch((err) => console.error(err))
```

14. Do not repeat yourself

If you are doing the same thing in multiple places, consolidate the duplicate code.

Dirty:

```
const MyComponent = () => (
  <div>
        <OtherComponent type="a" className="colorful" foo={123} bar={456} />
        <OtherComponent type="b" className="colorful" foo={123} bar={456} />
  </div>
);
```

Clean:

```
const MyOtherComponent = ({ type }) => (
  <OtherComponent type={type} className="colorful" foo={123} bar={456} />
);
const MyComponent = () => (
  <div>
        <MyOtherComponent type="a" />
        <MyOtherComponent type="b" />
  </div>
);
```

15. Always consider using *try/catch* when using *JSON.parse* or *JSON.stringify*

In JavaScript, when we pass JSON as input to the ***JSON.parse*** method, it expects a properly formatted JSON as the first argument. If it's formatted incorrectly, it will throw a JSON parse error.

Eg.

```
let json;
try {
        json = JSON.parse(input)
} catch (e) {
        // invalid json input, set to null
        json = null
}
```

16. Converting to Boolean Using the *!!(double-negation operator)* Operator

A simple ***!!variable***, which will automatically convert any kind of data to a boolean and this variable will return ***false*** only if it has some of these values: ***0***, ***null***, ***""***, ***undefined***, or ***NaN***; otherwise, it will return ***true***.

Eg.

```
function Account(cash) {
        this.cash = cash;
        this.hasMoney = !!cash;
}

let account = new Account(100.50);
console.log(account.cash);          // 100.50
console.log(account.hasMoney);      // true

let emptyAccount = new Account(0);
console.log(emptyAccount.cash);     // 0
console.log(emptyAccount.hasMoney); // false
```

17. Caching the *array.length* in the Loop

This is used when processing smaller arrays during a loop.
Eg.

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

If you process large arrays, this code will recalculate the size of an array in every iteration of this loop and this will cause some delays. To avoid it, you can cache the ***array.length*** in a variable to use it, instead of invoking the ***array.length*** every time during the loop.
Eg.

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

To make it smaller, just write this code:
Eg.

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

These tools will help you write clean code

- Prettier
- ESLint
- Husky
- Lint-staged
- With Husky and Lint-staged Combined
- EditorConfig

1. Prettier

Prettier is an opinionated code formatter. It formats code for you in a specific way.

2. ESLint

Now there are 2 popular style guides out there at the moment:

- Google JavaScript Style Guide
- Airbnb JavaScript Style Guide

3. Husky

Husky basically let's you ***Git hook***. That means you can perform certain actions when you are about to commit or when you are about to push code to a branch.

4. Lint-staged

Lint-staged helps you run linters on staged files, so that bad code doesn't get pushed to your branch.

5. With Husky and Lint-staged combined

Every time your commit your code, before committing your code, it will run a script called ***lint-staged*** which will run ***npm run lint:write*** which will lint and format your code—then add it to the staging area and then commit.

6. EditorConfig

In order to keep everyone on the same page in terms of what the defaults, such as ***tab space*** or ***line ending*** should be, we use ***.editorconfig***. This actually helps enforce some certain rule sets.

Resource Link:

1. https://www.atyantik.com/clean-code-practices-javascript/
2. https://github.com/ryanmcdermott/clean-code-javascript