Coding Standards

Lezioni alla pari

April 19, 2020

Team Members

Ovidiu Andrioaia
David Cirdan
Luciano Mateias
Zhiyang Xia

Document Control

Change History

Revision	Change Date	Description of changes
V1.0	04/19/2020	Initial release

Document storage

This document is stored in the project's GIT repository at:
https://github.com/KilliKrate/Software-Documentation-G6/blob/master/docs/Coding%20Standards/index.md

Document Owner

Group 6 is responsible for developing and maintaining this document.

Table of contents

Introduction

The "Lezioni alla Pari" coding standards aim to illustrate the way forward for implementing features and changes to the Javascript codebase. Every new piece of code committed the Github repository must strictly adhere to the following rules, since all the code in this code-base should look like a single person typed it, no matter the size of the team. These rules must be followed if you also are interested in collaborating with the team, by implementing changes and features to further develop the project.

This document guarantees a basic level of quality for all the code base, which should be, in order of importance:

- Reliable
- Maintainable
- Efficient

Efficiency comes last, as the most important characteristic is for the code to be easily readable, understandable, and modifiable by all members participating to the project.

Back to Top

Enforcing the standards

Code quality tools

In order to guarantee that the coding standards are respected, we require developers to use the following tools:

- ESLint plugin for VS Code OR ESLint Node Module
- Beautify plugin for VS Code OR js-beautify Node Module

Code reviews

After the rules have been agreed upon, issues not covered by the use of the aforementioned tools can be resolved through various tecniques of code review. We will list the two main methods used by our team in order to guarantee code standard compliance:

- Peer review
  Another team member reviews the code to ensure that the code follows the standards and meets the requirements. The code review will be conducted in pairs and in our case, always by the same pair, because of the team size: the two back-end devs will judge each other, and the front-end devs will do the same

- Architect review
  The architect of the team must review the core modules of the project to ensure they adhere to the architecture and design specifications, so that the code and algorithm therein strictly follow them.

Back to Top

Whitespaces

Indentation

All code MUST be indented using two (2) space charachters, the code MUST NOT indent using tab charachters or trailing whitespaces. This can be reinforced by using an IDE with appropriate indentation settings. Alternatively, ESLint will throw an error if use of whitespaces is not consistent.

```
// bad
if (condition) {
    doSomething();
}

// good
if (condition) {
  doSomething();
}
```

Lastly, lines SHOULD generally be no longer than 80 charachters, and MUST NOT not exceed 100. The only exceptions are Strings and Regex literals.

Control Structures

Control structures MUST always use braces, even in cases where they are not required, for the sake of readability. Control structures MUST have a space between the control keyword and the condition, as to differentiate them from function calls, to increase readability and to decrease the likelihood of logic errors when new lines are added.

```
// bad
if(condition) doSomething();

while(condition) iterating++;

for(var i=0;i<100;i++) someIterativeFn();

// good
if (condition) {
  doSomething();
}

while (condition) {
  iterating++;
}

for (var i = 0; i < 100; i++) {
  someIterativeFn();
}
```

Example `if` statement:

```
if (condition1 || condition2) {
  action1();
}
else if (condition3 && condition4) {
  action2();
}
else {
  defaultAction();
}
```

Example `switch` statement:

```
switch (condition) {
  case 1:
    action1();
    break;

  case 2:
    action2();
    break;

  default:
    defaultAction();
```

Example `try/catch` statement:

```
try {
  // Statements...
}
catch (error) {
  // Error handling...
}
finally {
  // Statements...
}
```

Function Declarations

The function keyword MUST be followed by one space. Named functions MUST NOT have a space between the function name and the following left parenthesis Optional arguments (using default values) SHOULD be defined at the end of the function signature. Every function SHOULD attempt to return a meaningful value.

```

// bad
function funStuff (text) {
  alert(`${text} is very fun`);
}

// good
function funStuff(text) {
  alert(`${text} is very fun`);
  return true;
}
```

Function Calls

Functions MUST be called with no spaces between the function name and its parameters. There MUST be one space between commas and each parameter, and there MUST NOT be a space between the last parameter, the closing parenthesis, and the semicolon.

```
// bad
var foobar=foo (bar,baz,quux) ;

// good
var foobar = foo(bar, baz, quux);
```

Back to Top

Semicolons

Although Javascript allows for optional semi-colons, all code in the "Lezioni alla Pari" repository MUST be followed by semi-colons. Therefore, all statements (except for, function, if, switch, try, while and possibly other control structures) MUST be followed by a semi-colon. This is in order to avoid errors in the code, where ASI fails to fill them in correctly.

```
// bad - raises exception
const luke = {}
const leia = {}
[luke, leia].forEach((jedi) => jedi.father = 'vader')

// good
const luke = {};
const leia = {};
[luke, leia].forEach((jedi) => {
  jedi.father = 'vader';
});
```

Back to Top

Naming Conventions

Functions and Variables

All variables and functions MUST use camelCase, with the exception of classes and constructors. The first letter of the word MUST be lowercase, while all the first letters of subsequent words MUST be uppercased. There MUST NOT be underscores between the words, as that would be snake_case.

```
// bad
function q(s) {
  return document.querySelectorAll(s);
}
var i,a=[],els=q("#foo");
for(i=0;i<els.length;i++){a.push(els[i]);}

// good
function query( selector ) {
  return document.querySelectorAll( selector );
}

var idx = 0,
  elements = [],
  matches = query("#foo"),
  length = matches.length;

for ( ; idx < length; idx++ ) {
  elements.push( matches[ idx ] );
}
```

Classes and Constructors

As mentioned above, there is an exception in the case of classes and constructors. The programmer, in this situation, MUST use PascalCase.

```
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}

const good = new User({
  name: 'yup',
});
```

Back to Top

Comments

JSDoc Standard

Single line and multiline comments are not only allowed, but recommended. The team MUST adopt the standards of JSDoc 3 for the proper documentation of code. Following those rules, all files, classes and functions MUST be documented. For more in-depth perspectives on code documentation, you should head over to the official JSDoc Documentation

Non-JSDoc Comments

One should note that comments not adherent to the JSDoc standard are recommended where files, functions, classes are not concerned. In particular, when one is explaining the logic surrounding an algorithm, one MUST write comments so that who tries to comprehend a particularly complex piece of logic can understand it even without reading it directly.

Generally speaking, all single line comments MUST use `//`, placed on a newline above the subject of the comment. An empty line MUST be inserted before the comment unless it's on the first line of a block.

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}
```

Lastly, all multiline comments MUST use `/* ... */`. Multiline comments MUST NOT use `//`, as that is reserved for single-line comments.

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 *
 * @param {String} tag
 * @return {Element} element
 */
function make(tag) {

  // ...

  return element;
}
```

Back to Top

Variables

Variable Declaration

Variables MUST be declared using the `let` or `const` keywords. The `var` keyword MUST NOT be used. Using `var` might result in variable scoper errors, which are often confusing and sometimes result in global variables. Variables MUST be declared on separate lines, by repeating the `let` or `const` keyword.

```
// bad
const items = getItems(),
  goSportsTeam = true,
  dragonball = 'z';

let lambSauceLocation = getLambSauce(),
  lambSauceFound = false,
  hotel = 'trivago';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';

let lambSauceLocation = getLambSauce();
let lambSauceFound = false;
let hotel = 'trivago';
```

Declaration Location

All variables MUST be declared at a sensible location in their respective scope, which is inside the curly braces that contain them. Forget any previous consideration about scope, since this is where `var` is fundamentally different `let` or `const`: The former is function-scoped, so placing them at the beginning of the function was the most sensible approach, while the latter are block-scoped, meaning they exist only in their block. Most often, sensible declaration choices will avoid things like unnecessary function calls.

```
// bad - unnecessary function call
function checkName(hasName) {
  const name = getName();

  if (hasName === 'test') {
    return false;
  }

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}

// good
function checkName(hasName) {
  if (hasName === 'test') {
    return false;
  }

  const name = getName();

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}
```

Unused Variables

Lastly, you SHOULD try to avoid creating unused variables: variables created but never read, or read for the sole purpose of modifying them (like in an increment) MUST NOT be present in the code. This also includes unused function arguments. ESLint will prevent this in some cases, but try to avoid it yourself.

Back to Top

Arrays

Single-line Arrays

Arrays MUST be formatted with one space separating each element, right after the comma

```
// bad
let someArray = ['hello','world'];

// good
let someArray = ['hello', 'world'];
```

Multi-line Arrays

If the line is longer than 80 characters, each element MUST be broken into its own line, and indented one level. You SHOULD use a trailing comma, as doing so will simplify adding and removing items into the array, and also results in cleaner git diffs.

```
// bad
let fruits = ['apples', 'banana', 'pineapple', 'watermelon', 'mango', 'orange',
'blueberry'];

// good
let fruits = [
  'apples',
  'banana',
  'pineapple',
  'watermelon',
  'mango',
  'orange',
  'blueberry',
];
```

Back to Top

Strings

Quotes

Although there is no actual difference to how Javascript interprets them, single quotes MUST be used everywhere, for the sake of uniformity.

```
// bad
let message = "Hello World";

// good
let message = 'Hello World';
```

Concatenation

Furthermore, strings are an exception to the 80-character rule: in case a string is longer than 80 characters, it MUST NOT be written across multiple lines using string concatenation, because broken strings are harder to work with.

```
// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// bad
const errorMessage = 'This is a super long error that was thrown because ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';

// good
const errorMessage = 'This is a super long error that was thrown because of Batman.
```

```
When you stop to think about how Batman had anything to do with this, you would get
nowhere fast.';
```

Template Strings MUST be used when programmatically building up strings, since it provides a more readable, concise syntax with proper newlines and string interpolation features.

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// also bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

Back to Top

Comparison Operators & Equality

Comparison Operators

Strict equality operators (`===` and `==!`) MUST be used when comparing two values, one MUST NOT use their non-strict counterparts (`==` and `=!`). This is because the latter offer type coercion, which can lead to unexpected errors.

```
// bad

// this condition is true, because all the values are converted
if (false == null == 0) {
  return "Something's not right..."
}

// good

// this, on the other hand, is false because the values are not converted
if (false === null === 0) {
  return "Problem solved"
}
```

Equality

Conditional statements, such as the `if` statement, evaluate their condition using type coercion, using these rules:

- Objects evaluate to true
- Undefined evaluates to false

- Null evaluates to false
- Booleans evaluate to the value of the boolean
- Numbers evaluate to false if +0, -0, or NaN, otherwise true
- Strings evaluate to false if an empty string '', otherwise true

```
if ([0] && []) {
  // true
  // an array (even an empty one) is an object, objects will evaluate to true
}
```

So as a rule of thumb, one MUST use shortcuts for booleans, but MUST NOT use them for strings or numbers: in this case, use explicit comparisons with the aforementioned strict equality operators.

```
// bad
if (isValid === true) {
  // ...
}

// good
if (isValid) {
  // ...
}

// bad
if (name) {
  // ...
}

// good
if (name !== '') {
  // ...
}

// bad
if (collection.length) {
  // ...
}

// good
if (collection.length > 0) {
  // ...
}
```

Lastly, ternary statements MUST NOT be nested, and MUST be single expressions. In case multiple conditions need to be evaluated, split them across multiple expressions

```
// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;
```

```
// split into 2 separated ternary expressions
const maybeNull = value1 > value2 ? 'baz' : null;

// better
const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull;

// best
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

Back to Top

Functions

Function Declaration

Functions MUST always have a sensible name that summarizes their use, just like variables. When declaring functions, one MUST NOT declare them in non-function blocks, like `if` and `while`: assign them to a variable outside of the block instead (in this case it is preferable to use arrow functions instead).

```
// bad
if (currentUser) {
  function abc() {
    console.log('Nope.');
  }
}

// good
let isUserVerified;
if (currentUser) {
  isUserVerified = () => {
    console.log('Yup.');
  };
}
```

Function Parameters

Parameters MUST NOT be mutated inside the function. When evaluating conditions on parameters at the start of a function, use default parameters instead.

```
// bad
function f1(a) {
  a = 1;
  // ...
}

function f2(a) {
  if (!a) { a = 1; }
  // ...
}
```

```
// good
function f3(a) {
  const b = a || 1;
  // ...
}

function f4(a = 1) {
  // ...
}
```

Default parameters shall also never reference external variables, since that can (and probably will) introduce bugs. Lastly, default parameters shall be put last.

```
// bad

// if a is redefined, so will the default parameter
let a = {}
function handleThings(opts = a, name) {
  // ...
}

// still bad
function handleThings(opts = {}, name) {
  // ...
}

// good
function handleThings(name, opts = {}) {
  // ...
}
```

Back to Top

Arrow Functions

Arrow Functions vs Functions

The use of arrow functions is preferred when creating and calling anonymous functions. This is better than using normal funcitons, because it is more concise and also excludes the context of `this`. Use normal functions when you have complicated logic that would deserve a named function of its own.

```
// bad
[1, 2, 3].map(function (x) {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

Implicit Returns

If the arrow function consists of a single statement, than you SHOULD omit the braces and the `return` statement, and use the implicit return. On the other hand, if the function spans across multiple lines, one MUST use braces.

```
// bad
[1, 2, 3].map((number) => {
  const nextNumber = number + 1;
  `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map((number) => `A string containing the ${number + 1}.`);
```

Style and Consistency Considerations

Also, even when you have a single parameter, one MUST include parentheses around arguments for clarity and consistency

```
// bad
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].map((x) => x * x);
```

Lastly, when using comparison operators, like `<=` or `>=`, wrap the statement in parentheses, so as to avoid confusion with the arrow syntax.

```
// bad
const itemHeight = (item) => item.height >= 256 ? item.largeSize : item.smallSize;

// good
const itemHeight = (item) => (item.height <= 256 ? item.largeSize : item.smallSize);
```

Back to Top

Classes

Class Declaration

The `class` syntax MUST be used when creating classes. The previous `prototype` manipulation tecnique used for creating classes is no longer valid, because `class` is more concise and easier to debug. Furthermore, use the `extends` keyword for inheritance, instead of the `inherits()` method.

```
// bad
const inherits = require('inherits');
function PeekableQueue(contents) {
  Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function () {
  return this.queue[0];
};

// good
class PeekableQueue extends Queue {
  peek() {
    return this.queue[0];
  }
}
```

Methods & Method Chaining

There MUST NOT be duicate class members, so never give the same name to the same two properties. Furthermore, when creating setters and getters, consider returning `this`, to help with method chaining.

```
class Jedi {
  jump() {
    this.jumping = true;
    return this;
  }

  setHeight(height) {
    this.height = height;
    return this;
  }
}

const luke = new Jedi();

// jump and setHeight can be chained because jump() returns `this`
luke.jump()
  .setHeight(20);
```

Constructors

Classes have a default `constructor` if one is not specified. This MUST be used when dealing with empty constructors, or constructors that delegate to the parent class, because it makes the code less redundant.

```
// bad
class Jedi {
  constructor() {}

  getName() {
    return this.name;
  }
}

// bad
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
  }
}

// good
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
    this.name = 'Rey';
  }
}
```

Static Methods

Lastly, if a method doesn't use `this`, it MUST be made static. Being an instance method should indicate that it behaves differently based on the properties of the object it's executed from.

```
// bad
class Foo {
  bar() {
    console.log('bar');
  }
}

// good
class Foo {
  static bar() {
    console.log('bar');
  }
}
```

Back to Top

Objects

Object Initialization

When creating a new object, always use the literal syntax rather than the curly braces syntax. This makes it clearer that you are creating a new object.

```
// bad
const item = new Object();

// good
const item = {};
```

Object Shorthands

Shorthands MUST be used when creating methods or when defining properties where the value variable is the same as the key name. Lastly, when using shorthand properties, always group them at the beginning of the object notation, in order to make it easier to tell which properties are using the shorthand.

```
// bad
const atom = {
  name: name,
  periodicTablePosition: periodicTablePosition,
  weight: 89,
  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  name,
  periodicTablePosition,
  addValue(value) {
    return atom.value + value;
  },
};
```

Back to Top

Python

For developing the Python back-end we did not create our own coding standard. We have, instead, opted to adopt the extensive and commonly used PEP8 standard. You can take a look at the standard's documentation by following this link

Back to Top