JavaScript Coding Standards

1. **Objects**

- **Using Literal syntax for object creation**

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

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

- **Using Object method shorthand**

```
//Bad
const atom = {
  value: 1,
  addValue: function (value) {
            return atom.value + value;
            },
  };

//Good
const atom = {
  value: 1,
  addValue(value) {
            return atom.value + value;
            },
  };
```

- **Using Property value shorthand**

```
const skywalker = 'Sky walker';
//Bad
const obj = {
  skywalker: skywalker,
  };

//Good
const obj = {
  skywalker,
  };
```

- **Group Shorthand properties at the begining of your object declaration because it is easy to tell which properties are using the shorthand.**

```
const skywalker = 'sky walker';
const lskywalker = 'lsky walker';
//Bad
  const obj = {
     one: 1,
```

```
    two: 2,
    skywalker,
    lskywalker
  }

//Good
const obj = {
  skywalker,
  lskywalker,
  one: 1,
  two: 2
}
```

- **Only quote properties that are invalid identifiers.**

```
// Bad
const bad = {
'foo': 3,
'bar': 4,
'data-blah': 5,
};

 // Good
 const good = {
 foo: 3,
 bar: 4,
 'data-blah': 5,
 };
```

2. **Arrays**

- **Use literal syntax for array creation**

```
// Bad
const items = new Array();

// Good
const items = [];
```

- **Use Array#push instead of direct assignment to add items to an array.**

```
const someStack = [];
  // Bad
  someStack[someStack.length] = 'value';

  // Good
  someStack.push('value');
```

- **Use Array spreads '. . .' to copy arrays**

```
  // Bad
  const len = items.length;
  const itemsCopy = [];
  let i;

  for (i = 0; i < len; i += 1) {
```

```
        itemsCopy[i] = items[i];
}

// Good
const itemsCopy = [...items];
```

- **To convert an array-like object to an array, use spreads '...' instead of Array.from.**

```
const foo = document.querySelectorAll('.foo');

// Good
const nodes = Array.from(foo);

// Best
const nodes = [...foo];
```

- **Use Array.from instead of spread '...' for mapping over iterables, because it avoids creating an intermediate array.**

```
// Bad
const baz = [...foo].map(bar);

// Good
const baz = Array.from(foo, bar);
```

3. **Destructuring**

- **Use object destructuring when accessing and using multiple properties of an object.**

```
// Bad
function getFullName(user) {
const firstName = user.firstName;
const lastName = user.lastName;

return `${firstName} ${lastName}`;
}

// Good
function getFullName(user) {
const { firstName, lastName } = user;
return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
return `${firstName} ${lastName}`;
}
```

- **Use array destructuring.**

```
const arr = [1, 2, 3, 4];

// Bad
```

```
const first = arr[0];
const second = arr[1];

// Good
const [first, second] = arr;
```

4. **Strings**

- **Use single quotes ''**

```
// Bad
const name = "Adam";

// Good
const name = 'Adam';
```

- **Use String concatenation for multiline strings**
- **Never use `eval()` on a string, it opens too many vulnerabilities.**
- **Do not unnecessarily escape characters in strings**

```
// Bad
const foo = '\'this\' \i\s \"quoted\"';

// Good
const foo = '\'this\' is "quoted"';
const foo = `my name is '${name}'`;
```

5. **Functions**

- **Use named function expressions instead of function declarations.**

```
// Bad
function foo() {
        // ...
}

// Good
// lexical name distinguished from the variable-referenced invocation(s)
const short = function longUniqueMoreDescriptiveLexicalFoo() {
        // ...
};
```

- **Never declare a function in a non-function block (`if`, `while`, etc). Assign the function to a variable instead.**

```
// Bad
if (currentUser) {
        function test() {
        console.log('Nope.');
        }
```

```
}

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

- **Never name a parameter arguments. This will take precedence over the arguments object that is given to every function scope.**

```
// Bad
function foo(name, options, arguments) {
      // ...
}

//Good
function foo(name, options, args) {
      // ...
}
```

- **Always put default parameters last.**

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

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

- **Never mutate parameters because manipulating objects passed in as parameters can cause unwanted side effects in variables.**

```
// Bad
function f1(obj) {
        obj.key = 1;
}

// Good
function f2(obj) {
        const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key : 1;
}
```

- **Never reassign passed in parameters.**

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

```
}

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

- **Prefer the use of the spread operator `...` to call variadic functions.**

```
// Bad
const x = [1, 2, 3, 4, 5];
console.log.apply(console, x);

// Good
const x = [1, 2, 3, 4, 5];
console.log(...x);
```

6. **Arrow Functions**

- **When using an anonymous function (as when passing an inline callback), use arrow function notation.**

```
// 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;
});
```

- **Avoid confusing arrow function syntax (=>) with comparison operators (<=, >=).**

```
//Bad
const itemHeight = item => item.height > 256 ? item.largeSize : item.smallSize;

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

7. **Classes & Constructors**

- **Always use class. Avoid manipulating prototype directly because `class` syntax is more precise and easier**

```
// Bad
function Queue(contents = []) {
  this.queue = [...contents];
}
Queue.prototype.pop = function () {
  const value = this.queue[0];
  this.queue.splice(0, 1);
  return value;
```

```
};

// Good
class Queue {
  constructor(contents = []) {
    this.queue = [...contents];
  }
  pop() {
    const value = this.queue[0];
    this.queue.splice(0, 1);
    return value;
  }
}
```

- **Use `extends` for inheritance.**

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

- **Write a custom toString() method**

```
//Good
class MyClass {
  toString() {
    return `MyClass - ${this.getName()}`;
  }
}
```

8. **Modules**

- **Do not use wildcard imports.**

```
  // Bad
 import * as MyModules from './MyModules';

  // Good
  import MyModel from './MyModules';
```

- **Do not export directly from an import.**

```
// Bad
// filename es6.js
export { es6 as default } from './Lib';

// Good
// filename es6.js
import { es6 } from './Lib';
export default es6;
```

- **Only import from a path in one place.**

```
 // Bad
 import foo from 'foo';
```

```
// … some other imports … //
import { named1, named2 } from 'foo';

// Good
import foo, { named1, named2 } from 'foo';
```

- **In modules with a single export, prefer default export over named export.**

```
// Bad
export function foo() {}

// Good
export default function foo() {}
```

- **Put all imports above non-import statements.**

```
// Bad
import foo from 'foo';
foo.init();

import bar from 'bar';

// Good
import foo from 'foo';
import bar from 'bar';
```

- **Multiline imports should be indented just like multiline array and object literals.**

```
// Bad
import {longNameA, longNameB, longNameC, longNameD, longNameE} from 'path';

// Good
import {
        longNameA,
        longNameB,
        longNameC,
        longNameD,
        longNameE,
} from 'path';
```

9. **Naming Conventions**

- **Avoid single letter names. Be descriptive with your naming.**

```
// Bad
function q() {
        // ...
}

// Good
function query() {
        // ...
}
```

- **Use camelCase when naming objects, functions, and instances.**

```
// Good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

- **Do not use trailing or leading underscores.**

```
// Bad
this.__firstName__ = 'Panda';

// Good
this.firstName = 'Panda';
```

- **Use camelCase when you export-default a function.**

```
function myStyleGuide() {
  // ...
}

export default myStyleGuide;
```