CODE GUID FOR ES2015

## 1. References

1.1 Use const for all of your references; avoid using var.

// bad

var a = 1;

var b = 2;

// good

const a = 1;

const b = 2;

1.2 If you must reassign references, use let instead of var.

// bad

var count = 1;

if (true) {

count += 1;

}

// good, use the let.

let count = 1;

if (true) {

count += 1;

}

**2. Objects**

2.1 Use the literal syntax for object creation

// bad

const item = new Object();

// good

const item = {};

2.2 Use 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;

},

};

2.3 Use property value shorthand

const lukeSkywalker = 'Luke Skywalker';

// bad

const obj = {

lukeSkywalker: lukeSkywalker,

};

// good

const obj = {

lukeSkywalker,

};

**3. Arrays**

3.1 Use the literal syntax for array creation

// bad

const items = new Array();

// good

const items = [];

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

const someStack = [];

// bad

someStack[someStack.length] = 'abracadabra';

// good

someStack.push('abracadabra');

3.3 Use array spreads ... to copy arrays.

// bad

const len = items.length;

const itemsCopy = [];

let i;

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

itemsCopy[i] = items[i];

}

// good

const itemsCopy = [...items];

3.4 To convert an array-like object to an array, use Array#from.

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

const nodes = Array.from(foo);

3.5 Use return statements in array method callbacks

// good

[1, 2, 3].map((x) => {

const y = x + 1;

return x \* y;

});

// good

[1, 2, 3].map(x => x + 1);

// good

const flat = {};

[[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {

const flatten = memo.concat(item);

flat[index] = flatten;

return flatten;

});

**4. Destructuring**

4.1 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}`;

}

4.2 Use array destructuring

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

// bad

const first = arr[0];

const second = arr[1];

// good

const [first, second] = arr;

4.3 Use object destructuring for multiple return values, not array destructurin

// bad

function processInput(input) {

// then a miracle occurs

return [left, right, top, bottom];

}

// the caller needs to think about the order of return data

const [left, \_\_, top] = processInput(input);

// good

function processInput(input) {

// then a miracle occurs

return { left, right, top, bottom };

}

// the caller selects only the data they need

const { left, top } = processInput(input);

**5. Strings**

5.1 Use single quotes '' for strings.

// bad

const name = "Capt. Janeway";

// good

const name = 'Capt. Janeway';

5.2 When programmatically building up strings, use template strings instead of concatenation.

// bad

function sayHi(name) {

return 'How are you, ' + name + '?';

}

// bad

function sayHi(name) {

return ['How are you, ', name, '?'].join();

}

// good

function sayHi(name) {

return `How are you, ${name}?`;

}

**6. Functions**

6.1 Use function declarations instead of function expressions

// bad

const foo = function () {

};

// good

function foo() {

}

6.2 Never name a parameter arguments

// bad

function nope(name, options, arguments) {

// ...stuff...

}

// good

function yup(name, options, args) {

// ...stuff...

}

6.3 Never use arguments, opt to use rest syntax ... instead

// bad

function concatenateAll() {

const args = Array.prototype.slice.call(arguments);

return args.join('');

}

// good

function concatenateAll(...args) {

return args.join('');

}

6.4 Use default parameter syntax rather than mutating function arguments

// really bad

function handleThings(opts) {

// No! We shouldn't mutate function arguments.

// Double bad: if opts is falsy it'll be set to an object which may

// be what you want but it can introduce subtle bugs.

opts = opts || {};

// ...

}

// good

function handleThings(opts = {}) {

// ...

}

6.5 Always put default parameters last.

// bad

function handleThings(opts = {}, name) {

// ...

}

// good

function handleThings(name, opts = {}) {

// ...

}

**7. Arrow Functions**

7.1 When you must use function expressions (as when passing an anonymous function), 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;

});

7.2 If your function takes a single argument and doesn’t use braces, omit the parentheses. Otherwise, always include parentheses around arguments

// bad

[1, 2, 3].map((x) => x \* x);

// good

[1, 2, 3].map(x => x \* x);

// good

[1, 2, 3].map(number => (

`A long string with the ${number}. It’s so long that we’ve broken it ` +

'over multiple lines!'

));

// bad

[1, 2, 3].map(x => {

const y = x + 1;

return x \* y;

});

// good

[1, 2, 3].map((x) => {

const y = x + 1;

return x \* y;

});

**8. Constructors**

8.1 Always use class. Avoid manipulating prototype directly

// 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;

}

}

8.2 Use extends for inheritance.

// 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];

}

}

8.3 Methods can return this to help with method chaining.

// bad

Jedi.prototype.jump = function () {

this.jumping = true;

return true;

};

Jedi.prototype.setHeight = function (height) {

this.height = height;

};

const luke = new Jedi();

luke.jump(); // => true

luke.setHeight(20); // => undefined

// good

class Jedi {

jump() {

this.jumping = true;

return this;

}

setHeight(height) {

this.height = height;

return this;

}

}

const luke = new Jedi();

luke.jump()

.setHeight(20);

**9. Modules**

9.1 Always use modules (import/export) over a non-standard module system.

// bad

const AirbnbStyleGuide = require('./AirbnbStyleGuide');

module.exports = AirbnbStyleGuide.es6;

// good

import AirbnbStyleGuide from './AirbnbStyleGuide';

export default AirbnbStyleGuide.es6;

// best

import { es6 } from './AirbnbStyleGuide';

export default es6;

**10. Iterators and Generators**

10.1 Don't use iterators. Prefer JavaScript's higher-order functions like map() and reduce() instead of loops like for-of

const numbers = [1, 2, 3, 4, 5];

// bad

let sum = 0;

for (let num of numbers) {

sum += num;

}

sum === 15;

// good

let sum = 0;

numbers.forEach(num => sum += num);

sum === 15;

// best (use the functional force)

const sum = numbers.reduce((total, num) => total + num, 0);

sum === 15;

## 11. Variables

11.1 Always use const to declare variables.

// bad

superPower = new SuperPower();

// good

const superPower = new SuperPower();

11.2 Assign variables where you need them, but place them in a reasonable place.

// 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;

}

11.3 Avoid unneeded ternary statements.

// bad

const foo = a ? a : b;

const bar = c ? true : false;

const baz = c ? false : true;

// good

const foo = a || b;

const bar = !!c;

const baz = !c;

## 12. Naming Conventions

12.1 Use camelCase when naming objects, functions, and instances

// bad

const OBJEcttsssss = {};

const this\_is\_my\_object = {};

function c() {}

// good

const thisIsMyObject = {};

function thisIsMyFunction() {}

12.2 Use PascalCase only when naming constructors or classes.

// bad

function user(options) {

this.name = options.name;

}

// good

class User {

constructor(options) {

this.name = options.name;

}

}

12.3 Don't save references to this. Use arrow functions or Function#bind

// bad

function foo() {

const self = this;

return function () {

console.log(self);

};

}

// bad

function foo() {

const that = this;

return function () {

console.log(that);

};

}

// good

function foo() {

return () => {

console.log(this);

};

}