# JavaScript Style Guide

*A mostly reasonable approach to JavaScript*

## Table of Contents

1. [Types](#3znysh7)
2. [References](#2et92p0)
3. [Objects](#tyjcwt)
4. [Arrays](#3dy6vkm)
5. [Destructuring](#1t3h5sf)
6. [Strings](#4d34og8)
7. [Functions](#2s8eyo1)
8. [Arrow Functions](#17dp8vu)
9. Classes & Constructors
10. [Iterators](#lnxbz9)
11. [Properties](#35nkun2)
12. [Variables](#1ksv4uv)
13. [Hoisting](#44sinio)
14. Comparison Operators & Equality
15. [Blocks](#z337ya)
16. [Comments](#3j2qqm3)
17. [Whitespace](#1y810tw)
18. [Commas](#4i7ojhp)
19. [Semicolons](#2xcytpi)
20. Type Casting & Coercion
21. [Naming Conventions](#3whwml4)
22. [Accessors](#2bn6wsx)
23. [Events](#qsh70q)
24. [jQuery](#3as4poj)
25. [Testing](#49x2ik5)
26. [Performance](#2p2csry)
27. [Resources](#147n2zr)

## Types

- 1.1 **Primitives**: When you access a primitive type you work directly on its value.

+ `string`  
+ `number`  
+ `boolean`  
+ `null`  
+ `undefined`  
  
```javascript  
const foo = 1;  
let bar = foo;  
  
bar = 9;  
  
console.log(foo, bar); // => 1, 9  
```

- 1.2 **Complex**: When you access a complex type you work on a reference to its value.

+ `object`  
+ `array`  
+ `function`  
  
```javascript  
const foo = [1, 2];  
const bar = foo;  
  
bar[0] = 9;  
  
console.log(foo[0], bar[0]); // => 9, 9  
```

[**⬆ back to top**](#1fob9te)

## References

- 2.1 Use const for all of your references; avoid using var. eslint: [prefer-const](http://eslint.org/docs/rules/prefer-const.html), [no-const-assign](http://eslint.org/docs/rules/no-const-assign.html)

> Why? This ensures that you can't reassign your references, which can lead to bugs and difficult to comprehend code.  
  
```javascript  
// bad  
var a = 1;  
var b = 2;  
  
// good  
const a = 1;  
const b = 2;  
```

- 2.2 If you must reassign references, use let instead of var. eslint: [no-var](http://eslint.org/docs/rules/no-var.html) jscs: [disallowVar](http://jscs.info/rule/disallowVar)

> Why? `let` is block-scoped rather than function-scoped like `var`.  
  
```javascript  
// bad  
var count = 1;  
if (true) {  
 count += 1;  
}  
  
// good, use the let.  
let count = 1;  
if (true) {  
 count += 1;  
}  
```

- 2.3 Note that both let and const are block-scoped.

```javascript  
// const and let only exist in the blocks they are defined in.  
{  
 let a = 1;  
 const b = 1;  
}  
console.log(a); // ReferenceError  
console.log(b); // ReferenceError  
```

[**⬆ back to top**](#1fob9te)

## Objects

- 3.1 Use the literal syntax for object creation. eslint: [no-new-object](http://eslint.org/docs/rules/no-new-object.html)

```javascript  
// bad  
const item = new Object();  
  
// good  
const item = {};  
```

- 3.2 Use property value shorthand. eslint: [object-shorthand](http://eslint.org/docs/rules/object-shorthand.html) jscs: [requireEnhancedObjectLiterals](http://jscs.info/rule/requireEnhancedObjectLiterals)

> Why? It is shorter to write and descriptive.  
  
```javascript  
const lukeSkywalker = 'Luke Skywalker';  
  
// bad  
const obj = {  
 lukeSkywalker: lukeSkywalker,  
};  
  
// good  
const obj = {  
 lukeSkywalker,  
};  
```

- 3.3 Group your shorthand properties at the beginning of your object declaration.

> Why? It's easier to tell which properties are using the shorthand.  
  
```javascript  
const anakinSkywalker = 'Anakin Skywalker';  
const lukeSkywalker = 'Luke Skywalker';  
  
// bad  
const obj = {  
 episodeOne: 1,  
 twoJediWalkIntoACantina: 2,  
 lukeSkywalker,  
 episodeThree: 3,  
 mayTheFourth: 4,  
 anakinSkywalker,  
};  
  
// good  
const obj = {  
 lukeSkywalker,  
 anakinSkywalker,  
 episodeOne: 1,  
 twoJediWalkIntoACantina: 2,  
 episodeThree: 3,  
 mayTheFourth: 4,  
};  
```

- 3.4 Only quote properties that are invalid identifiers. eslint: [quote-props](http://eslint.org/docs/rules/quote-props.html) jscs: [disallowQuotedKeysInObjects](http://jscs.info/rule/disallowQuotedKeysInObjects)

> Why? In general we consider it subjectively easier to read. It improves syntax highlighting, and is also more easily optimized by many JS engines.  
  
```javascript  
// bad  
const bad = {  
 'foo': 3,  
 'bar': 4,  
 'data-blah': 5,  
};  
  
// good  
const good = {  
 foo: 3,  
 bar: 4,  
 'data-blah': 5,  
};  
```

- 3.5 Do not call Object.prototype methods directly, such as hasOwnProperty, propertyIsEnumerable, and isPrototypeOf.

> Why? These methods may be shadowed by properties on the object in question - consider `{ hasOwnProperty: false }` - or, the object may be a null object (`Object.create(null)`).  
  
```javascript  
// bad  
console.log(object.hasOwnProperty(key));  
  
// good  
console.log(Object.prototype.hasOwnProperty.call(object, key));  
  
// best  
const has = Object.prototype.hasOwnProperty; // cache the lookup once, in module scope.  
/\* or \*/  
import has from 'has';  
// ...  
console.log(has.call(object, key));  
```

- 3.6 Prefer the object spread operator over [Object.assign](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) to shallow-copy objects. Use the object rest operator to get a new object with certain properties omitted.

```javascript  
// very bad  
const original = { a: 1, b: 2 };  
const copy = Object.assign(original, { c: 3 }); // this mutates `original` ಠ\_ಠ  
delete copy.a; // so does this  
  
// bad  
const original = { a: 1, b: 2 };  
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }  
  
// good  
const original = { a: 1, b: 2 };  
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }  
  
const { a, ...noA } = copy; // noA => { b: 2, c: 3 }  
```

[**⬆ back to top**](#1fob9te)

## Arrays

- 4.1 Use the literal syntax for array creation. eslint: [no-array-constructor](http://eslint.org/docs/rules/no-array-constructor.html)

```javascript  
// bad  
const items = new Array();  
  
// good  
const items = [];  
```

- 4.2 Use [Array#push](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/push) instead of direct assignment to add items to an array.

```javascript  
const someStack = [];  
  
// bad  
someStack[someStack.length] = 'abracadabra';  
  
// good  
someStack.push('abracadabra');  
```

- 4.3 Use array spreads ... to copy arrays.

```javascript  
// bad  
const len = items.length;  
const itemsCopy = [];  
let i;  
  
for (i = 0; i < len; i += 1) {  
 itemsCopy[i] = items[i];  
}  
  
// good  
const itemsCopy = [...items];  
```

- 4.4 To convert an array-like object to an array, use [Array.from](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/from).

```javascript  
const foo = document.querySelectorAll('.foo');  
const nodes = Array.from(foo);  
```

- 4.5 Use return statements in array method callbacks. It's ok to omit the return if the function body consists of a single statement following 8.2. eslint: [array-callback-return](http://eslint.org/docs/rules/array-callback-return)

```javascript  
// good  
[1, 2, 3].map((x) => {  
 const y = x + 1;  
 return x \* y;  
});  
  
// good  
[1, 2, 3].map(x => x + 1);  
  
// bad  
const flat = {};  
[[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {  
 const flatten = memo.concat(item);  
 flat[index] = flatten;  
});  
  
// good  
const flat = {};  
[[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {  
 const flatten = memo.concat(item);  
 flat[index] = flatten;  
 return flatten;  
});  
  
// bad  
inbox.filter((msg) => {  
 const { subject, author } = msg;  
 if (subject === 'Mockingbird') {  
 return author === 'Harper Lee';  
 } else {  
 return false;  
 }  
});  
  
// good  
inbox.filter((msg) => {  
 const { subject, author } = msg;  
 if (subject === 'Mockingbird') {  
 return author === 'Harper Lee';  
 }  
  
 return false;  
});  
```

**[⬆ back to top](#1fob9te)**

## Destructuring

- 5.1 Use object destructuring when accessing and using multiple properties of an object. jscs: [requireObjectDestructuring](http://jscs.info/rule/requireObjectDestructuring)

> Why? Destructuring saves you from creating temporary references for those properties.  
  
```javascript  
// 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}`;  
}  
```

- 5.2 Use array destructuring. jscs: [requireArrayDestructuring](http://jscs.info/rule/requireArrayDestructuring)

```javascript  
const arr = [1, 2, 3, 4];  
  
// bad  
const first = arr[0];  
const second = arr[1];  
  
// good  
const [first, second] = arr;  
```

- 5.3 Use object destructuring for multiple return values, not array destructuring. jscs: [disallowArrayDestructuringReturn](http://jscs.info/rule/disallowArrayDestructuringReturn)

> Why? You can add new properties over time or change the order of things without breaking call sites.  
  
```javascript  
// 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);  
```

[**⬆ back to top**](#1fob9te)

## Strings

- 6.1 Use double quotes “” for strings. eslint: [quotes](http://eslint.org/docs/rules/quotes.html) jscs: [validateQuoteMarks](http://jscs.info/rule/validateQuoteMarks)

```javascript  
// bad - template literals should contain interpolation or newlines  
const name = `Capt. Janeway`;  
  
// bad  
const name = 'Capt. Janeway';

// good  
const name = "Capt. Janeway";  
```

- 6.2 Strings that cause the line to go over 100 characters should be splitted by sentences or tags.

> Why? Broken strings are painful to work with and make code less searchable.  
  
```javascript  
// 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.';  
```

- 6.3 When programmatically building up strings, use template strings instead of concatenation. eslint: [prefer-template](http://eslint.org/docs/rules/prefer-template.html) [template-curly-spacing](http://eslint.org/docs/rules/template-curly-spacing) jscs: [requireTemplateStrings](http://jscs.info/rule/requireTemplateStrings)

> Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features.  
  
```javascript  
// 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}?`;  
}  
```

- 6.4 Never use eval() on a string, it opens too many vulnerabilities.

- 6.5 Do not unnecessarily escape characters in strings. eslint: [no-useless-escape](http://eslint.org/docs/rules/no-useless-escape)

> Why? Backslashes harm readability, thus they should only be present when necessary.  
  
```javascript  
// bad  
const foo = '\'this\' \i\s \"quoted\"';  
  
// good  
const foo = '\'this\' is "quoted"';  
const foo = `my name is '${name}'`;  
```

[**⬆ back to top**](#1fob9te)

## Functions

- 7.1 Use named function expressions instead of function declarations. eslint: [func-style](http://eslint.org/docs/rules/func-style) jscs: [disallowFunctionDeclarations](http://jscs.info/rule/disallowFunctionDeclarations)

> Why? Function declarations are hoisted, which means that it’s easy - too easy - to reference the function before it is defined in the file. This harms readability and maintainability. If you find that a function’s definition is large or complex enough that it is interfering with understanding the rest of the file, then perhaps it’s time to extract it to its own module! Don’t forget to name the expression - anonymous functions can make it harder to locate the problem in an Error's call stack. ([Discussion](https://github.com/airbnb/javascript/issues/794))  
  
```javascript

// bad  
const foo = function bar() {  
 // ...  
};

// good  
function foo() {  
 // ...  
}

// good  
const foo = function () {  
 // ...  
};  
```

- 7.2 Never declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears. eslint: [no-loop-func](http://eslint.org/docs/rules/no-loop-func.html)

- 7.3 **Note:** ECMA-262 defines a block as a list of statements. A function declaration is not a statement. [Read ECMA-262's note on this issue](http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf#page=97).

```javascript  
// bad  
if (currentUser) {  
 function test() {  
 console.log('Nope.');  
 }  
}  
  
// good  
let test;  
if (currentUser) {  
 test = () => {  
 console.log('Yup.');  
 };  
}  
```

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

```javascript  
// bad  
function foo(name, options, arguments) {  
 // ...  
}  
  
// good  
function foo(name, options, args) {  
 // ...  
}  
```

- 7.5 Use default parameter syntax rather than mutating function arguments.

```javascript

// 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 || {};  
 // ...  
}  
  
// still bad  
function handleThings(opts) {  
 if (opts === void 0) {  
 opts = {};  
 }  
 // ...  
}  
  
// good  
function handleThings(opts = {}) {  
 // ...  
}  
```

- 7.6 Avoid side effects with default parameters.

> Why? They are confusing to reason about.  
  
```javascript  
var b = 1;  
// bad  
function count(a = b++) {  
 console.log(a);  
}  
count(); // 1  
count(); // 2  
count(3); // 3  
count(); // 3  
```

- 7.7 Always put default parameters last.

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

- 7.8 Never use the Function constructor to create a new function. eslint: [no-new-func](http://eslint.org/docs/rules/no-new-func)

> Why? Creating a function in this way evaluates a string similarly to eval(), which opens vulnerabilities.  
  
```javascript  
// bad  
var add = new Function('a', 'b', 'return a + b');  
  
// still bad  
var subtract = Function('a', 'b', 'return a - b');  
```

- 7.9 Prefer the use of the spread operator ... to call variadic functions. eslint: [prefer-spread](http://eslint.org/docs/rules/prefer-spread)

> Why? It's cleaner, you don't need to supply a context, and you can not easily compose `new` with `apply`.  
  
```javascript  
// bad  
const x = [1, 2, 3, 4, 5];  
console.log.apply(console, x);  
  
// good  
const x = [1, 2, 3, 4, 5];  
console.log(...x);  
  
// bad  
new (Function.prototype.bind.apply(Date, [null, 2016, 8, 5]));  
  
// good  
new Date(...[2016, 8, 5]);  
```

- 7.10 Functions with multiline signatures, or invocations, should be indented just like every other multiline list in this guide: with each item on a line by itself, with a trailing comma on the last item.

```javascript  
// bad  
function foo(bar,  
 baz,  
 quux) {  
 // ...  
}  
  
// good  
function foo(  
 bar,  
 baz,  
 quux,  
) {  
 // ...  
}  
  
// bad  
console.log(foo,  
 bar,  
 baz);  
  
// good  
console.log(  
 foo,  
 bar,  
 baz,  
);  
```

[**⬆ back to top**](#1fob9te)

## Arrow Functions

- 8.1 Avoid confusing arrow function syntax (=>) with comparison operators (<=, >=). eslint: [no-confusing-arrow](http://eslint.org/docs/rules/no-confusing-arrow)

```javascript  
// bad  
const itemHeight = item => item.height > 256 ? item.largeSize : item.smallSize;  
  
// bad  
const itemHeight = (item) => item.height > 256 ? item.largeSize : item.smallSize;  
  
// good  
const itemHeight = item => (item.height > 256 ? item.largeSize : item.smallSize);  
  
// good  
const itemHeight = (item) => {  
 const { height, largeSize, smallSize } = item;  
 return height > 256 ? largeSize : smallSize;  
};  
```

[**⬆ back to top**](#1fob9te)

## Classes & Constructors

- 9.1 Always use class. Avoid manipulating prototype directly.

> Why? `class` syntax is more concise and easier to reason about.  
  
```javascript  
// 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;  
 }  
}  
```

- 9.2 Use extends for inheritance.

> Why? It is a built-in way to inherit prototype functionality without breaking `instanceof`.  
  
```javascript  
// 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];  
 }  
}  
```

- 9.3 Methods can return this to help with method chaining.

```javascript  
// 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);  
```

- 9.4 It's okay to write a custom toString() method, just make sure it works successfully and causes no side effects.

```javascript  
class Jedi {  
 constructor(options = {}) {  
 this.name = options.name || 'no name';  
 }  
  
 getName() {  
 return this.name;  
 }  
  
 toString() {  
 return `Jedi - ${this.getName()}`;  
 }  
}  
```

- 9.5 Classes have a default constructor if one is not specified. An empty constructor function or one that just delegates to a parent class is unnecessary. eslint: [no-useless-constructor](http://eslint.org/docs/rules/no-useless-constructor)

```javascript  
// 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';  
 }  
}  
```

[**⬆ back to top**](#1fob9te)

## Iterators

- 11.1 Don't use iterators. Prefer JavaScript's higher-order functions instead of loops like for-in or for-of. eslint: [no-iterator](http://eslint.org/docs/rules/no-iterator.html) [no-restricted-syntax](http://eslint.org/docs/rules/no-restricted-syntax)

> Why? This enforces our immutable rule. Dealing with pure functions that return values is easier to reason about than side effects.  
  
> Use `map()` / `every()` / `filter()` / `find()` / `findIndex()` / `reduce()` / `some()` / ... to iterate over arrays, and `Object.keys()` / `Object.values()` / `Object.entries()` to produce arrays so you can iterate over objects.  
  
```javascript  
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;  
  
// bad  
const increasedByOne = [];  
for (let i = 0; i < numbers.length; i++) {  
 increasedByOne.push(numbers[i] + 1);  
}  
  
// good  
const increasedByOne = [];  
numbers.forEach(num => increasedByOne.push(num + 1));  
  
// best (keeping it functional)  
const increasedByOne = numbers.map(num => num + 1);  
```

[**⬆ back to top**](#1fob9te)

## Properties

- 12.1 Use dot notation when accessing properties. eslint: [dot-notation](http://eslint.org/docs/rules/dot-notation.html) jscs: [requireDotNotation](http://jscs.info/rule/requireDotNotation)

```javascript  
const luke = {  
 jedi: true,  
 age: 28,  
};  
  
// bad  
const isJedi = luke['jedi'];  
  
// good  
const isJedi = luke.jedi;  
```

- 12.2 Use bracket notation [] when accessing properties with a variable.

```javascript  
const luke = {  
 jedi: true,  
 age: 28,  
};  
  
function getProp(prop) {  
 return luke[prop];  
}  
  
const isJedi = getProp('jedi');  
```

[**⬆ back to top**](#1fob9te)

## Variables

- 13.1 Always use const or let to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Captain Planet warned us of that. eslint: [no-undef](http://eslint.org/docs/rules/no-undef) [prefer-const](http://eslint.org/docs/rules/prefer-const)

```javascript  
// bad  
superPower = new SuperPower();  
  
// good  
const superPower = new SuperPower();  
```

- 13.2 Use one const or let declaration per variable. eslint: [one-var](http://eslint.org/docs/rules/one-var.html) jscs: [disallowMultipleVarDecl](http://jscs.info/rule/disallowMultipleVarDecl)

> Why? It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs. You can also step through each declaration with the debugger, instead of jumping through all of them at once.  
  
```javascript  
// bad  
const items = getItems(),  
 goSportsTeam = true,  
 dragonball = 'z';  
  
// bad  
// (compare to above, and try to spot the mistake)  
const items = getItems(),  
 goSportsTeam = true;  
 dragonball = 'z';  
  
// good  
const items = getItems();  
const goSportsTeam = true;  
const dragonball = 'z';  
```

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

> Why? `let` and `const` are block scoped and not function scoped.  
  
```javascript  
// 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;  
}  
```

- 13.4 Don't chain variable assignments.

> Why? Chaining variable assignments creates implicit global variables.  
  
```javascript  
// bad  
(function example() {  
 // JavaScript interprets this as  
 // let a = ( b = ( c = 1 ) );  
 // The let keyword only applies to variable a; variables b and c become  
 // global variables.  
 let a = b = c = 1;  
}());  
  
console.log(a); // undefined  
console.log(b); // 1  
console.log(c); // 1  
  
// good  
(function example() {  
 let a = 1;  
 let b = a;  
 let c = a;  
}());  
  
console.log(a); // undefined  
console.log(b); // undefined  
console.log(c); // undefined  
  
// the same applies for `const`  
```

[**⬆ back to top**](#1fob9te)

## Comparison Operators & Equality

- 15.1 Use === and !== over == and !=. eslint: [eqeqeq](http://eslint.org/docs/rules/eqeqeq.html)

- 15.2 Conditional statements such as the if statement evaluate their expression using coercion with the ToBoolean abstract method and always follow these simple 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\*\*  
  
```javascript  
if ([0] && []) {  
 // true  
 // an array (even an empty one) is an object, objects will evaluate to true  
}  
```

- 15.3 Use shortcuts for booleans, but explicit comparisons for strings and numbers.

```javascript  
// bad  
if (isValid === true) {  
 // ...  
}  
  
// good  
if (isValid) {  
 // ...  
}  
  
// bad  
if (name) {  
 // ...  
}  
  
// good  
if (name !== '') {  
 // ...  
}  
  
// bad  
if (collection.length) {  
 // ...  
}  
  
// good  
if (collection.length > 0) {  
 // ...  
}  
```

- 15.4 For more information see [Truth Equality and JavaScript](https://javascriptweblog.wordpress.com/2011/02/07/truth-equality-and-javascript/#more-2108) by Angus Croll.

- 15.5 Use braces to create blocks in case and default clauses that contain lexical declarations (e.g. let, const, function, and class).

> Why? Lexical declarations are visible in the entire `switch` block but only get initialized when assigned, which only happens when its `case` is reached. This causes problems when multiple `case` clauses attempt to define the same thing.  
  
eslint rules: [`no-case-declarations`](http://eslint.org/docs/rules/no-case-declarations.html).  
  
```javascript  
// bad  
switch (foo) {  
 case 1:  
 let x = 1;  
 break;  
 case 2:  
 const y = 2;  
 break;  
 case 3:  
 function f() {  
 // ...  
 }  
 break;  
 default:  
 class C {}  
}  
  
// good  
switch (foo) {  
 case 1: {  
 let x = 1;  
 break;  
 }  
 case 2: {  
 const y = 2;  
 break;  
 }  
 case 3: {  
 function f() {  
 // ...  
 }  
 break;  
 }  
 case 4:  
 bar();  
 break;  
 default: {  
 class C {}  
 }  
}  
```

- 15.6 Ternaries should not be nested..

eslint rules: [`no-nested-ternary`](http://eslint.org/docs/rules/no-nested-ternary.html).  
  
```javascript  
// bad  
const foo = maybe1 > maybe2  
 ? "bar"  
 : value1 > value2 ? "baz" : null;  
  
// good  
const maybeNull = (value1 > value2 ? 'baz' : null);

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

- 15.7 Avoid unneeded ternary statements.

eslint rules: [`no-unneeded-ternary`](http://eslint.org/docs/rules/no-unneeded-ternary.html).  
  
```javascript  
// 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;  
```

[**⬆ back to top**](#1fob9te)

## Blocks

- 16.1 Use braces with all multi-line blocks.

```javascript  
// bad  
if (test)  
 return false;  
  
// bad  
if (test) return false;

// bad  
function foo() { return false; }  
  
// good  
if (test) {  
 return false;  
}  
  
// good  
function bar() {  
 return false;  
}  
```

- 16.2 If you're using multi-line blocks with if and else, put else on the same line as your if block's closing brace. eslint: [brace-style](http://eslint.org/docs/rules/brace-style.html) jscs: [disallowNewlineBeforeBlockStatements](http://jscs.info/rule/disallowNewlineBeforeBlockStatements)

```javascript

// bad  
if (test) {  
 thing1();  
 thing2();  
} else {  
 thing3();  
}

// good  
if (test) {  
 thing1();  
 thing2();  
}  
else {  
 thing3();  
}  
  
  
```

[**⬆ back to top**](#1fob9te)

## Comments

- 17.1 Use /\*\* ... \*/ for multi-line comments.

```javascript  
// 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  
 \*/  
function make(tag) {  
  
 // ...  
  
 return element;  
}  
```

- 17.2 Use // for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment unless it's on the first line of a block.

```javascript  
// 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;  
}  
  
// also good  
function getType() {  
 // set the default type to 'no type'  
 const type = this.type || 'no type';  
  
 return type;  
}  
```

* 17.3 Start all comments with a space to make it easier to read. eslint: [spaced-comment](http://eslint.org/docs/rules/spaced-comment)
* *// bad*  
  *//is current tab*  
  **const** active = **true**;  
    
  *// good*  
  *// is current tab*  
  **const** active = **true**;  
    
  *// bad*  
  */\*\**  
   *\*make() returns a new element*  
   *\*based on the passed-in tag name*  
   *\*/*  
  **function** make(tag) {  
    
   *// ...*  
    
   **return** element;  
  }  
    
  *// good*  
  */\*\**  
   *\* make() returns a new element*  
   *\* based on the passed-in tag name*  
   *\*/*  
  **function** make(tag) {  
    
   *// ...*  
    
   **return** element;  
  }

- 17.4 Prefixing your comments with FIXME or TODO helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are FIXME: -- need to figure this out or TODO: -- need to implement.

- 17.5 Use // FIXME: to annotate problems.

```javascript  
class Calculator extends Abacus {  
 constructor() {  
 super();  
  
 // FIXME: shouldn't use a global here  
 total = 0;  
 }  
}  
```

- 17.6 Use // TODO: to annotate solutions to problems.

```javascript  
class Calculator extends Abacus {  
 constructor() {  
 super();  
  
 // TODO: total should be configurable by an options param  
 this.total = 0;  
 }  
}  
```

[**⬆ back to top**](#1fob9te)

## Whitespace

- 18.1 Use soft tabs (space character) set to 4 spaces. eslint: [indent](http://eslint.org/docs/rules/indent.html) jscs: [validateIndentation](http://jscs.info/rule/validateIndentation)

```javascript  
// bad  
function bar() {  
∙let name;  
}  
  
// bad  
function baz() {  
∙∙let name;  
}

// good  
function foo() {  
∙∙∙∙let name;  
}  
```

- 18.2 Place 1 space before the opening parenthesis in control statements (if, while etc.). Place no space between the argument list and the function name in function calls and declarations. eslint: [keyword-spacing](http://eslint.org/docs/rules/keyword-spacing.html) jscs: [requireSpaceAfterKeywords](http://jscs.info/rule/requireSpaceAfterKeywords)

```javascript  
// bad  
if(isJedi) {  
 fight ();  
}  
  
// good  
if (isJedi) {  
 fight();  
}  
  
// bad  
function fight () {  
 console.log ('Swooosh!');  
}  
  
// good  
function fight() {  
 console.log('Swooosh!');  
}  
```

- 18.3 Set off operators with spaces. eslint: [space-infix-ops](http://eslint.org/docs/rules/space-infix-ops.html) jscs: [requireSpaceBeforeBinaryOperators](http://jscs.info/rule/requireSpaceBeforeBinaryOperators), [requireSpaceAfterBinaryOperators](http://jscs.info/rule/requireSpaceAfterBinaryOperators)

```javascript  
// bad  
const x=y+5;  
  
// good  
const x = y + 5;  
```

- 18.4 Use indentation when making long method chains (more than 2 method chains). Use a leading dot, which emphasizes that the line is a method call, not a new statement. eslint: [newline-per-chained-call](http://eslint.org/docs/rules/newline-per-chained-call) [no-whitespace-before-property](http://eslint.org/docs/rules/no-whitespace-before-property)

```javascript  
// bad  
$('#items').find('.selected').highlight().end().find('.open').updateCount();  
  
// bad  
$('#items').  
 find('.selected').  
 highlight().  
 end().  
 find('.open').  
 updateCount();  
  
// good  
$('#items')  
 .find('.selected')  
 .highlight()  
 .end()  
 .find('.open')  
 .updateCount();  
  
// bad  
const leds = stage.selectAll('.led').data(data).enter().append('svg:svg').classed('led', true)  
 .attr('width', (radius + margin) \* 2).append('svg:g')  
 .attr('transform', `translate(${radius + margin},${radius + margin})`)  
 .call(tron.led);  
  
// good  
const leds = stage.selectAll('.led')  
 .data(data)  
 .enter().append('svg:svg')  
 .classed('led', true)  
 .attr('width', (radius + margin) \* 2)  
 .append('svg:g')  
 .attr('transform', `translate(${radius + margin},${radius + margin})`)  
 .call(tron.led);  
  
// good  
const leds = stage.selectAll('.led').data(data);  
```

- 18.5 Leave a blank line after blocks and before the next statement. jscs: [requirePaddingNewLinesAfterBlocks](http://jscs.info/rule/requirePaddingNewLinesAfterBlocks)

```javascript  
// bad  
if (foo) {  
 return bar;  
}  
return baz;  
  
// good  
if (foo) {  
 return bar;  
}  
  
return baz;  
  
// bad  
const obj = {  
 foo() {  
 },  
 bar() {  
 },  
};  
return obj;  
  
// good  
const obj = {  
 foo() {  
 },  
  
 bar() {  
 },  
};  
  
return obj;  
  
// bad  
const arr = [  
 function foo() {  
 },  
 function bar() {  
 },  
];  
return arr;  
  
// good  
const arr = [  
 function foo() {  
 },  
  
 function bar() {  
 },  
];  
  
return arr;  
```

- 18.6 Do not pad your blocks with blank lines. eslint: [padded-blocks](http://eslint.org/docs/rules/padded-blocks.html) jscs: [disallowPaddingNewlinesInBlocks](http://jscs.info/rule/disallowPaddingNewlinesInBlocks)

```javascript  
// bad  
function bar() {  
  
 console.log(foo);  
  
}  
  
// also bad  
if (baz) {  
  
 console.log(qux);  
} else {  
 console.log(foo);  
  
}  
  
// good  
function bar() {  
 console.log(foo);  
}  
  
// good  
if (baz) {  
 console.log(qux);  
} else {  
 console.log(foo);  
}  
```

- 18.7 Do not add spaces inside parentheses. eslint: [space-in-parens](http://eslint.org/docs/rules/space-in-parens.html) jscs: [disallowSpacesInsideParentheses](http://jscs.info/rule/disallowSpacesInsideParentheses)

```javascript  
// bad  
function bar( foo ) {  
 return foo;  
}  
  
// good  
function bar(foo) {  
 return foo;  
}  
  
// bad  
if ( foo ) {  
 console.log(foo);  
}  
  
// good  
if (foo) {  
 console.log(foo);  
}  
```

- 18.8 Do not add spaces inside brackets. eslint: [array-bracket-spacing](http://eslint.org/docs/rules/array-bracket-spacing.html) jscs: [disallowSpacesInsideArrayBrackets](http://jscs.info/rule/disallowSpacesInsideArrayBrackets)

```javascript  
// bad  
const foo = [ 1, 2, 3 ];  
console.log(foo[ 0 ]);  
  
// good  
const foo = [1, 2, 3];  
console.log(foo[0]);  
```

- 18.9 Do not add spaces inside curly braces. eslint: [object-curly-spacing](http://eslint.org/docs/rules/object-curly-spacing.html) jscs: [requireSpacesInsideObjectBrackets](http://jscs.info/rule/requireSpacesInsideObjectBrackets)

```javascript

// bad  
const foo = { clark: 'kent' };

// good  
const foo = {clark: 'kent'};  
```

- 18.10 Avoid having lines of code that are longer than 100 characters (including whitespace). Note: per above, long strings are exempt from this rule, and should not be broken up. eslint: [max-len](http://eslint.org/docs/rules/max-len.html) jscs: [maximumLineLength](http://jscs.info/rule/maximumLineLength)

> Why? This ensures readability and maintainability.  
  
```javascript  
// bad  
const foo = jsonData && jsonData.foo && jsonData.foo.bar && jsonData.foo.bar.baz && jsonData.foo.bar.baz.quux && jsonData.foo.bar.baz.quux.xyzzy;  
  
// bad  
$.ajax({ method: 'POST', url: 'https://airbnb.com/', data: { name: 'John' } }).done(() => console.log('Congratulations!')).fail(() => console.log('You have failed this city.'));  
  
// good  
const foo = jsonData  
 && jsonData.foo  
 && jsonData.foo.bar  
 && jsonData.foo.bar.baz  
 && jsonData.foo.bar.baz.quux  
 && jsonData.foo.bar.baz.quux.xyzzy;  
  
// good  
$.ajax({  
 method: 'POST',  
 url: 'https://airbnb.com/',  
 data: { name: 'John' },  
})  
 .done(() => console.log('Congratulations!'))  
 .fail(() => console.log('You have failed this city.'));  
```

[**⬆ back to top**](#1fob9te)

## Commas

- 19.1 Leading commas: **Nope.** eslint: [comma-style](http://eslint.org/docs/rules/comma-style.html) jscs: [requireCommaBeforeLineBreak](http://jscs.info/rule/requireCommaBeforeLineBreak)

```javascript  
// bad  
const story = [  
 once  
 , upon  
 , aTime  
];  
  
// good  
const story = [  
 once,  
 upon,  
 aTime  
];  
  
// bad  
const hero = {  
 firstName: 'Ada'  
 , lastName: 'Lovelace'  
 , birthYear: 1815  
 , superPower: 'computers'  
};  
  
// good  
const hero = {  
 firstName: 'Ada',  
 lastName: 'Lovelace',  
 birthYear: 1815,  
 superPower: 'computers'  
};  
```

[**⬆ back to top**](#1fob9te)

## Semicolons

- 20.1 **Yup.** eslint: [semi](http://eslint.org/docs/rules/semi.html) jscs: [requireSemicolons](http://jscs.info/rule/requireSemicolons)

```javascript  
// bad  
(function () {  
 const name = 'Skywalker'  
 return name  
})()  
  
// good  
(function () {  
 const name = 'Skywalker';  
 return name;  
}());  
  
// good, but legacy (guards against the function becoming an argument when two files with IIFEs are concatenated)  
;((() => {  
 const name = 'Skywalker';  
 return name;  
})());  
```  
  
[Read more](https://stackoverflow.com/questions/7365172/semicolon-before-self-invoking-function/7365214%237365214).

[**⬆ back to top**](#1fob9te)

## Type Casting & Coercion

- 21.1 Perform type coercion at the beginning of the statement.

- 21.2 Strings:

```javascript  
// => this.reviewScore = 9;  
  
// bad  
const totalScore = this.reviewScore + ''; // invokes this.reviewScore.valueOf()  
  
// bad  
const totalScore = this.reviewScore.toString(); // isn't guaranteed to return a string  
  
// good  
const totalScore = String(this.reviewScore);  
```

- 21.3 Numbers: Use Number for type casting and parseInt always with a radix for parsing strings. eslint: [radix](http://eslint.org/docs/rules/radix)

```javascript  
const inputValue = '4';  
  
// bad  
const val = new Number(inputValue);  
  
// bad  
const val = +inputValue;  
  
// bad  
const val = inputValue >> 0;  
  
// bad  
const val = parseInt(inputValue);  
  
// good  
const val = Number(inputValue);  
  
// good  
const val = parseInt(inputValue, 10);  
```

- 21.4 If for whatever reason you are doing something wild and parseInt is your bottleneck and need to use Bitshift for [performance reasons](https://jsperf.com/coercion-vs-casting/3), leave a comment explaining why and what you're doing.

```javascript  
// good  
/\*\*  
 \* parseInt was the reason my code was slow.  
 \* Bitshifting the String to coerce it to a  
 \* Number made it a lot faster.  
 \*/  
const val = inputValue >> 0;  
```

- 21.5 **Note:** Be careful when using bitshift operations. Numbers are represented as [64-bit values](https://es5.github.io/#x4.3.19), but bitshift operations always return a 32-bit integer ([source](https://es5.github.io/#x11.7)). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. [Discussion](https://github.com/airbnb/javascript/issues/109). Largest signed 32-bit Int is 2,147,483,647:

```javascript  
2147483647 >> 0; // => 2147483647  
2147483648 >> 0; // => -2147483648  
2147483649 >> 0; // => -2147483647  
```

- 21.6 Booleans:

```javascript  
const age = 0;  
  
// bad  
const hasAge = new Boolean(age);  
  
// good  
const hasAge = Boolean(age);  
  
// best  
const hasAge = !!age;  
```

[**⬆ back to top**](#1fob9te)

## Naming Conventions

- 22.1 Avoid single letter names. Be descriptive with your naming. eslint: [id-length](http://eslint.org/docs/rules/id-length)

```javascript  
// bad  
function q() {  
 // ...  
}  
  
// good  
function query() {  
 // ...  
}  
```

- 22.2 Use camelCase when naming objects, functions, and instances. eslint: [camelcase](http://eslint.org/docs/rules/camelcase.html) jscs: [requireCamelCaseOrUpperCaseIdentifiers](http://jscs.info/rule/requireCamelCaseOrUpperCaseIdentifiers)

```javascript  
// bad  
const OBJEcttsssss = {};  
const this\_is\_my\_object = {};  
function c() {}  
  
// good  
const thisIsMyObject = {};  
function thisIsMyFunction() {}  
```

- 22.3 Use PascalCase only when naming constructors or classes and public functions. eslint: [new-cap](http://eslint.org/docs/rules/new-cap.html) jscs: [requireCapitalizedConstructors](http://jscs.info/rule/requireCapitalizedConstructors)

```javascript  
// 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',  
});

// Public function

const good = function() {  
 return true;  
};  
```

- 22.4 A name prefixed with an underscore (e.g. \_spam) should be treated as a non-public part of the API. eslint: [no-underscore-dangle](http://eslint.org/docs/rules/no-underscore-dangle.html) jscs:

```javascript  
// bad  
this.firstName = 'Panda'; // private  
  
// good  
this.\_firstName = 'Panda'; // private  
```

- 22.5 Don't save references to this. Use arrow functions or [Function#bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind). jscs: [disallowNodeTypes](http://jscs.info/rule/disallowNodeTypes)

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

[**⬆ back to top**](#1fob9te)

## Accessors

- 23.1 Accessor functions for properties are not required.

- 23.2 If the property/method is a boolean, use isVal() or hasVal().

```javascript  
// bad  
if (!dragon.age()) {  
 return false;  
}  
  
// good  
if (!dragon.hasAge()) {  
 return false;  
}  
```

- 23.3 It's okay to create get() and set() functions.

```javascript  
class Jedi {  
 constructor(options = {}) {  
 const lightsaber = options.lightsaber || 'blue';  
 this.set('lightsaber', lightsaber);  
 }  
  
 set(key, val) {  
 this[key] = val;  
 }  
  
 get(key) {  
 return this[key];  
 }  
}  
```

[**⬆ back to top**](#1fob9te)

## Events

- 24.1 When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass a hash instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

```javascript  
// bad  
$(this).trigger('listingUpdated', listing.id);  
  
// ...  
  
$(this).on('listingUpdated', (e, listingId) => {  
 // do something with listingId  
});  
```  
  
prefer:  
  
```javascript  
// good  
$(this).trigger('listingUpdated', { listingId: listing.id });  
  
// ...  
  
$(this).on('listingUpdated', (e, data) => {  
 // do something with data.listingId  
});  
```

[**⬆ back to top**](#1fob9te)

## jQuery

- 25.1 Cache jQuery lookups.

```javascript  
// bad  
function setSidebar() {  
 $('.sidebar').hide();  
  
 // ...  
  
 $('.sidebar').css({  
 'background-color': 'pink',  
 });  
}

// good  
function setSidebar() {  
 const $sidebar = $('.sidebar');  
 $sidebar.hide();  
  
 // ...  
  
 $sidebar.css({  
 'background-color': 'pink',  
 });  
}  
```

- 25.2 For DOM queries use Cascading $('.sidebar ul') or parent > child $('.sidebar > ul'). [jsPerf](http://jsperf.com/jquery-find-vs-context-sel/16)

- 25.4 Use find with scoped jQuery object queries.

```javascript  
// bad  
$('ul', '.sidebar').hide();  
  
// bad  
$('.sidebar').find('ul').hide();  
  
// good  
$('.sidebar ul').hide();  
  
// good  
$('.sidebar > ul').hide();  
  
// good  
$sidebar.find('ul').hide();  
```

[**⬆ back to top**](#1fob9te)

## Testing

- 28.1 **Yup.**

```javascript  
function foo() {  
 return true;  
}  
```

- 28.2 **No, but seriously**: - Whichever testing framework you use, you should be writing tests! - Strive to write many small pure functions, and minimize where mutations occur. - Be cautious about stubs and mocks - they can make your tests more brittle. - We primarily use [mocha](https://www.npmjs.com/package/mocha) at Airbnb. [tape](https://www.npmjs.com/package/tape) is also used occasionally for small, separate modules. - 100% test coverage is a good goal to strive for, even if it's not always practical to reach it. - Whenever you fix a bug, *write a regression test*. A bug fixed without a regression test is almost certainly going to break again in the future.

[**⬆ back to top**](#1fob9te)

## Performance

* [On Layout & Web Performance](https://www.kellegous.com/j/2013/01/26/layout-performance/)
* [String vs Array Concat](https://jsperf.com/string-vs-array-concat/2)
* [Try/Catch Cost In a Loop](https://jsperf.com/try-catch-in-loop-cost)
* [Bang Function](https://jsperf.com/bang-function)
* [jQuery Find vs Context, Selector](https://jsperf.com/jquery-find-vs-context-sel/13)
* [innerHTML vs textContent for script text](https://jsperf.com/innerhtml-vs-textcontent-for-script-text)
* [Long String Concatenation](https://jsperf.com/ya-string-concat)
* [Are Javascript functions like map(), reduce(), and filter() optimized for traversing arrays?](https://www.quora.com/JavaScript-programming-language-Are-Javascript-functions-like-map-reduce-and-filter-already-optimized-for-traversing-array/answer/Quildreen-Motta)
* Loading...

[**⬆ back to top**](#1fob9te)

## Resources

**Learning ES6**

* [Draft ECMA 2015 (ES6) Spec](https://people.mozilla.org/~jorendorff/es6-draft.html)
* [ExploringJS](http://exploringjs.com/)
* [ES6 Compatibility Table](https://kangax.github.io/compat-table/es6/)
* [Comprehensive Overview of ES6 Features](http://es6-features.org/)

**Read This**

* [Standard ECMA-262](http://www.ecma-international.org/ecma-262/6.0/index.html)

**Tools**

* Code Style Linters
  + [ESlint](http://eslint.org/) - [Airbnb Style .eslintrc](https://github.com/airbnb/javascript/blob/master/linters/.eslintrc)
  + [JSHint](http://jshint.com/) - [Airbnb Style .jshintrc](https://github.com/airbnb/javascript/blob/master/linters/.jshintrc)
  + [JSCS](https://github.com/jscs-dev/node-jscs) - [Airbnb Style Preset](https://github.com/jscs-dev/node-jscs/blob/master/presets/airbnb.json) (Deprecated, please use [ESlint](https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base))
* Neutrino preset - [neutrino-preset-airbnb-base](https://neutrino.js.org/presets/neutrino-preset-airbnb-base/)

**Other Style Guides**

* [Google JavaScript Style Guide](https://google.github.io/styleguide/javascriptguide.xml)
* [jQuery Core Style Guidelines](https://contribute.jquery.org/style-guide/js/)
* [Principles of Writing Consistent, Idiomatic JavaScript](https://github.com/rwaldron/idiomatic.js)

**Other Styles**

* [Naming this in nested functions](https://gist.github.com/cjohansen/4135065) - Christian Johansen
* [Conditional Callbacks](https://github.com/airbnb/javascript/issues/52) - Ross Allen
* [Popular JavaScript Coding Conventions on GitHub](http://sideeffect.kr/popularconvention/#javascript) - JeongHoon Byun
* [Multiple var statements in JavaScript, not superfluous](http://benalman.com/news/2012/05/multiple-var-statements-javascript/) - Ben Alman

**Further Reading**

* [Understanding JavaScript Closures](https://javascriptweblog.wordpress.com/2010/10/25/understanding-javascript-closures/) - Angus Croll
* [Basic JavaScript for the impatient programmer](http://www.2ality.com/2013/06/basic-javascript.html) - Dr. Axel Rauschmayer
* [You Might Not Need jQuery](http://youmightnotneedjquery.com/) - Zack Bloom & Adam Schwartz
* [ES6 Features](https://github.com/lukehoban/es6features) - Luke Hoban
* [Frontend Guidelines](https://github.com/bendc/frontend-guidelines) - Benjamin De Cock

**Books**

* [JavaScript: The Good Parts](https://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742) - Douglas Crockford
* [JavaScript Patterns](https://www.amazon.com/JavaScript-Patterns-Stoyan-Stefanov/dp/0596806752) - Stoyan Stefanov
* [Pro JavaScript Design Patterns](https://www.amazon.com/JavaScript-Design-Patterns-Recipes-Problem-Solution/dp/159059908X) - Ross Harmes and Dustin Diaz
* [High Performance Web Sites: Essential Knowledge for Front-End Engineers](https://www.amazon.com/High-Performance-Web-Sites-Essential/dp/0596529309) - Steve Souders
* [Maintainable JavaScript](https://www.amazon.com/Maintainable-JavaScript-Nicholas-C-Zakas/dp/1449327680) - Nicholas C. Zakas
* [JavaScript Web Applications](https://www.amazon.com/JavaScript-Web-Applications-Alex-MacCaw/dp/144930351X) - Alex MacCaw
* [Pro JavaScript Techniques](https://www.amazon.com/Pro-JavaScript-Techniques-John-Resig/dp/1590597273) - John Resig
* [Smashing Node.js: JavaScript Everywhere](https://www.amazon.com/Smashing-Node-js-JavaScript-Everywhere-Magazine/dp/1119962595) - Guillermo Rauch
* [Secrets of the JavaScript Ninja](https://www.amazon.com/Secrets-JavaScript-Ninja-John-Resig/dp/193398869X) - John Resig and Bear Bibeault
* [Human JavaScript](http://humanjavascript.com/) - Henrik Joreteg
* [Superhero.js](http://superherojs.com/) - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
* [JSBooks](http://jsbooks.revolunet.com/) - Julien Bouquillon
* [Third Party JavaScript](https://www.manning.com/books/third-party-javascript) - Ben Vinegar and Anton Kovalyov
* [Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript](http://amzn.com/0321812182) - David Herman
* [Eloquent JavaScript](http://eloquentjavascript.net/) - Marijn Haverbeke
* [You Don't Know JS: ES6 & Beyond](http://shop.oreilly.com/product/0636920033769.do) - Kyle Simpson

**Blogs**

* [JavaScript Weekly](http://javascriptweekly.com/)
* [JavaScript, JavaScript...](https://javascriptweblog.wordpress.com/)
* [Bocoup Weblog](https://bocoup.com/weblog)
* [Adequately Good](http://www.adequatelygood.com/)
* [NCZOnline](https://www.nczonline.net/)
* [Perfection Kills](http://perfectionkills.com/)
* [Ben Alman](http://benalman.com/)
* [Dmitry Baranovskiy](http://dmitry.baranovskiy.com/)
* [Dustin Diaz](http://dustindiaz.com/)
* [nettuts](http://code.tutsplus.com/?s=javascript)

**Podcasts**

* [JavaScript Air](https://javascriptair.com/)
* [JavaScript Jabber](https://devchat.tv/js-jabber/)

[**⬆ back to top**](#1fob9te)