# Airbnb JavaScript Style Guide

## Types

* [1.1](https://github.com/airbnb/javascript" \l "types--primitives) **Primitives**: When you access a primitive type you work directly on its value.
  + string
  + number
  + boolean
  + null
  + undefined
  + symbol
* const foo = 1;
* let bar = foo;
* bar = 9;

console.log(foo, bar); // => 1, 9

* + Symbols cannot be faithfully polyfilled, so they should not be used when targeting browsers/environments that don't support them natively.

* [1.2](https://github.com/airbnb/javascript" \l "types--complex) **Complex**: When you access a complex type you work on a reference to its value.
  + object
  + array
  + function
* const foo = [1, 2];
* const bar = foo;
* bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9

## References

* [2.1](https://github.com/airbnb/javascript" \l "references--prefer-const) Use const for all of your references; avoid using var. eslint: [prefer-const](https://eslint.org/docs/rules/prefer-const.html), [no-const-assign](https://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.

// bad

var a = 1;

var b = 2;

// good

const a = 1;

const b = 2;

* [2.2](https://github.com/airbnb/javascript" \l "references--disallow-var) If you must reassign references, use let instead of var. eslint: [no-var](https://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.

// bad

var count = 1;

if (true) {

count += 1;

}

// good, use the let.

let count = 1;

if (true) {

count += 1;

}

* [2.3](https://github.com/airbnb/javascript" \l "references--block-scope) Note that both let and const are block-scoped.
* // 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

## Objects

* [3.1](https://github.com/airbnb/javascript" \l "objects--no-new) Use the literal syntax for object creation. eslint: [no-new-object](https://eslint.org/docs/rules/no-new-object.html)
* // bad
* const item = new Object();
* // good

const item = {};

* [3.2](https://github.com/airbnb/javascript" \l "es6-computed-properties) Use computed property names when creating objects with dynamic property names.

Why? They allow you to define all the properties of an object in one place.

function getKey(k) {

return `a key named ${k}`;

}

// bad

const obj = {

id: 5,

name: 'San Francisco',

};

obj[getKey('enabled')] = true;

// good

const obj = {

id: 5,

name: 'San Francisco',

[getKey('enabled')]: true,

};

* [3.3](https://github.com/airbnb/javascript" \l "es6-object-shorthand) Use object method shorthand. eslint: [object-shorthand](https://eslint.org/docs/rules/object-shorthand.html) jscs: [requireEnhancedObjectLiterals](http://jscs.info/rule/requireEnhancedObjectLiterals)
* // bad
* const atom = {
* value: 1,
* addValue: function (value) {
* return atom.value + value;
* },
* };
* // good
* const atom = {
* value: 1,
* addValue(value) {
* return atom.value + value;
* },

};

* [3.4](https://github.com/airbnb/javascript" \l "es6-object-concise) Use property value shorthand. eslint: [object-shorthand](https://eslint.org/docs/rules/object-shorthand.html) jscs: [requireEnhancedObjectLiterals](http://jscs.info/rule/requireEnhancedObjectLiterals)

Why? It is shorter to write and descriptive.

const lukeSkywalker = 'Luke Skywalker';

// bad

const obj = {

lukeSkywalker: lukeSkywalker,

};

// good

const obj = {

lukeSkywalker,

};

* [3.5](https://github.com/airbnb/javascript" \l "objects--grouped-shorthand) Group your shorthand properties at the beginning of your object declaration.

Why? It’s easier to tell which properties are using the shorthand.

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,

};

* [3.6](https://github.com/airbnb/javascript" \l "objects--quoted-props) Only quote properties that are invalid identifiers. eslint: [quote-props](https://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.

// bad

const bad = {

'foo': 3,

'bar': 4,

'data-blah': 5,

};

// good

const good = {

foo: 3,

bar: 4,

'data-blah': 5,

};

* [3.7](https://github.com/airbnb/javascript" \l "objects--prototype-builtins) 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)).

// 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));

* [3.8](https://github.com/airbnb/javascript" \l "objects--rest-spread) 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.
* // 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 }

## Arrays

* [4.1](https://github.com/airbnb/javascript" \l "arrays--literals) Use the literal syntax for array creation. eslint: [no-array-constructor](https://eslint.org/docs/rules/no-array-constructor.html)
* // bad
* const items = new Array();
* // good

const items = [];

* [4.2](https://github.com/airbnb/javascript" \l "arrays--push) 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.
* const someStack = [];
* // bad
* someStack[someStack.length] = 'abracadabra';
* // good

someStack.push('abracadabra');

* [4.3](https://github.com/airbnb/javascript" \l "es6-array-spreads) 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];

* [4.4](https://github.com/airbnb/javascript" \l "arrays--from) To convert an array-like object to an array, use spreads ... instead of [Array.from](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/from).
* const foo = document.querySelectorAll('.foo');
* // good
* const nodes = Array.from(foo);
* // best

const nodes = [...foo];

* [4.5](https://github.com/airbnb/javascript" \l "arrays--mapping) Use [Array.from](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/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);

* [4.6](https://github.com/airbnb/javascript" \l "arrays--callback-return) Use return statements in array method callbacks. It’s ok to omit the return if the function body consists of a single statement returning an expression without side effects, following [8.2](https://github.com/airbnb/javascript#arrows--implicit-return). eslint: [array-callback-return](https://eslint.org/docs/rules/array-callback-return)
* // good
* [1, 2, 3].map((x) => {
* const y = x + 1;
* return x \* y;
* });
* // good
* [1, 2, 3].map(x => x + 1);
* // bad - no returned value means `memo` becomes undefined after the first iteration
* [[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {
* const flatten = memo.concat(item);
* memo[index] = flatten;
* });
* // good
* [[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {
* const flatten = memo.concat(item);
* memo[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;

});

* [4.7](https://github.com/airbnb/javascript" \l "arrays--bracket-newline) Use line breaks after open and before close array brackets if an array has multiple lines

// bad

const arr = [

[0, 1], [2, 3], [4, 5],

];

const objectInArray = [{

id: 1,

}, {

id: 2,

}];

const numberInArray = [

1, 2,

];

// good

const arr = [[0, 1], [2, 3], [4, 5]];

const objectInArray = [

{

id: 1,

},

{

id: 2,

},

];

const numberInArray = [

1,

2,

];

## Destructuring

* [5.1](https://github.com/airbnb/javascript" \l "destructuring--object) Use object destructuring when accessing and using multiple properties of an object. eslint: [prefer-destructuring](https://eslint.org/docs/rules/prefer-destructuring)jscs: [requireObjectDestructuring](http://jscs.info/rule/requireObjectDestructuring)

Why? Destructuring saves you from creating temporary references for those properties.

// 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](https://github.com/airbnb/javascript" \l "destructuring--array) Use array destructuring. eslint: [prefer-destructuring](https://eslint.org/docs/rules/prefer-destructuring) jscs: [requireArrayDestructuring](http://jscs.info/rule/requireArrayDestructuring)
* const arr = [1, 2, 3, 4];
* // bad
* const first = arr[0];
* const second = arr[1];
* // good

const [first, second] = arr;

* [5.3](https://github.com/airbnb/javascript" \l "destructuring--object-over-array) 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.

// 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);

## Strings

* [6.1](https://github.com/airbnb/javascript" \l "strings--quotes) Use single quotes '' for strings. eslint: [quotes](https://eslint.org/docs/rules/quotes.html) jscs: [validateQuoteMarks](http://jscs.info/rule/validateQuoteMarks)
* // bad
* const name = "Capt. Janeway";
* // bad - template literals should contain interpolation or newlines
* const name = `Capt. Janeway`;
* // good

const name = 'Capt. Janeway';

* [6.2](https://github.com/airbnb/javascript" \l "strings--line-length) Strings that cause the line to go over 100 characters should not be written across multiple lines using string concatenation.

Why? Broken strings are painful to work with and make code less searchable.

// 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](https://github.com/airbnb/javascript" \l "es6-template-literals) When programmatically building up strings, use template strings instead of concatenation. eslint: [prefer-template](https://eslint.org/docs/rules/prefer-template.html)[template-curly-spacing](https://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.

// bad

function sayHi(name) {

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

}

// bad

function sayHi(name) {

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

}

// bad

function sayHi(name) {

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

}

// good

function sayHi(name) {

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

}

* [6.4](https://github.com/airbnb/javascript" \l "strings--eval) Never use eval() on a string, it opens too many vulnerabilities. eslint: [no-eval](https://eslint.org/docs/rules/no-eval)

* [6.5](https://github.com/airbnb/javascript" \l "strings--escaping) Do not unnecessarily escape characters in strings. eslint: [no-useless-escape](https://eslint.org/docs/rules/no-useless-escape)

Why? Backslashes harm readability, thus they should only be present when necessary.

// bad

const foo = '\'this\' \i\s \"quoted\"';

// good

const foo = '\'this\' is "quoted"';

const foo = `my name is '${name}'`;

## Functions

* [7.1](https://github.com/airbnb/javascript" \l "functions--declarations) Use named function expressions instead of function declarations. eslint: [func-style](https://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 explicitly name the expression, regardless of whether or not the name is inferred from the containing variable (which is often the case in modern browsers or when using compilers such as Babel). This eliminates any assumptions made about the Error's call stack. ([Discussion](https://github.com/airbnb/javascript/issues/794))

// bad

function foo() {

// ...

}

// bad

const foo = function () {

// ...

};

// good

// lexical name distinguished from the variable-referenced invocation(s)

const short = function longUniqueMoreDescriptiveLexicalFoo() {

// ...

};

* [7.2](https://github.com/airbnb/javascript" \l "functions--iife) Wrap immediately invoked function expressions in parentheses. eslint: [wrap-iife](https://eslint.org/docs/rules/wrap-iife.html) jscs: [requireParenthesesAroundIIFE](http://jscs.info/rule/requireParenthesesAroundIIFE)

Why? An immediately invoked function expression is a single unit - wrapping both it, and its invocation parens, in parens, cleanly expresses this. Note that in a world with modules everywhere, you almost never need an IIFE.

// immediately-invoked function expression (IIFE)

(function () {

console.log('Welcome to the Internet. Please follow me.');

}());

* [7.3](https://github.com/airbnb/javascript" \l "functions--in-blocks) 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](https://eslint.org/docs/rules/no-loop-func.html)

* [7.4](https://github.com/airbnb/javascript" \l "functions--note-on-blocks) **Note:** ECMA-262 defines a block as a list of statements. A function declaration is not a statement.
* // bad
* if (currentUser) {
* function test() {
* console.log('Nope.');
* }
* }
* // good
* let test;
* if (currentUser) {
* test = () => {
* console.log('Yup.');
* };

}

* [7.5](https://github.com/airbnb/javascript" \l "functions--arguments-shadow) 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) {
* // ...

}

* [7.6](https://github.com/airbnb/javascript" \l "es6-rest) Never use arguments, opt to use rest syntax ... instead. eslint: [prefer-rest-params](https://eslint.org/docs/rules/prefer-rest-params)

Why? ... is explicit about which arguments you want pulled. Plus, rest arguments are a real Array, and not merely Array-like like arguments.

// bad

function concatenateAll() {

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

return args.join('');

}

// good

function concatenateAll(...args) {

return args.join('');

}

* [7.7](https://github.com/airbnb/javascript" \l "es6-default-parameters) 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 || {};
* // ...
* }
* // still bad
* function handleThings(opts) {
* if (opts === void 0) {
* opts = {};
* }
* // ...
* }
* // good
* function handleThings(opts = {}) {
* // ...

}

* [7.8](https://github.com/airbnb/javascript" \l "functions--default-side-effects) Avoid side effects with default parameters.

Why? They are confusing to reason about.

var b = 1;

// bad

function count(a = b++) {

console.log(a);

}

count(); // 1

count(); // 2

count(3); // 3

count(); // 3

* [7.9](https://github.com/airbnb/javascript" \l "functions--defaults-last) Always put default parameters last.
* // bad
* function handleThings(opts = {}, name) {
* // ...
* }
* // good
* function handleThings(name, opts = {}) {
* // ...

}

* [7.10](https://github.com/airbnb/javascript" \l "functions--constructor) Never use the Function constructor to create a new function. eslint: [no-new-func](https://eslint.org/docs/rules/no-new-func)

Why? Creating a function in this way evaluates a string similarly to eval(), which opens vulnerabilities.

// bad

var add = new Function('a', 'b', 'return a + b');

// still bad

var subtract = Function('a', 'b', 'return a - b');

* [7.11](https://github.com/airbnb/javascript" \l "functions--signature-spacing) Spacing in a function signature. eslint: [space-before-function-paren](https://eslint.org/docs/rules/space-before-function-paren) [space-before-blocks](https://eslint.org/docs/rules/space-before-blocks)

Why? Consistency is good, and you shouldn’t have to add or remove a space when adding or removing a name.

// bad

const f = function(){};

const g = function (){};

const h = function() {};

// good

const x = function () {};

const y = function a() {};

* [7.12](https://github.com/airbnb/javascript" \l "functions--mutate-params) Never mutate parameters. eslint: [no-param-reassign](https://eslint.org/docs/rules/no-param-reassign.html)

Why? Manipulating objects passed in as parameters can cause unwanted variable side effects in the original caller.

// bad

function f1(obj) {

obj.key = 1;

}

// good

function f2(obj) {

const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key : 1;

}

* [7.13](https://github.com/airbnb/javascript" \l "functions--reassign-params) Never reassign parameters. eslint: [no-param-reassign](https://eslint.org/docs/rules/no-param-reassign.html)

Why? Reassigning parameters can lead to unexpected behavior, especially when accessing the arguments object. It can also cause optimization issues, especially in V8.

// 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) {

// ...

}

* [7.14](https://github.com/airbnb/javascript" \l "functions--spread-vs-apply) Prefer the use of the spread operator ... to call variadic functions. eslint: [prefer-spread](https://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.

// 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]);

* [7.15](https://github.com/airbnb/javascript" \l "functions--signature-invocation-indentation) 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.
* // bad
* function foo(bar,
* baz,
* quux) {
* // ...
* }
* // good
* function foo(
* bar,
* baz,
* quux,
* ) {
* // ...
* }
* // bad
* console.log(foo,
* bar,
* baz);
* // good
* console.log(
* foo,
* bar,
* baz,

);

## Arrow Functions

* [8.1](https://github.com/airbnb/javascript" \l "arrows--use-them) When you must use an anonymous function (as when passing an inline callback), use arrow function notation. eslint: [prefer-arrow-callback](https://eslint.org/docs/rules/prefer-arrow-callback.html), [arrow-spacing](https://eslint.org/docs/rules/arrow-spacing.html) jscs: [requireArrowFunctions](http://jscs.info/rule/requireArrowFunctions)

Why? It creates a version of the function that executes in the context of this, which is usually what you want, and is a more concise syntax.

Why not? If you have a fairly complicated function, you might move that logic out into its own named function expression.

// 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;

});

* [8.2](https://github.com/airbnb/javascript" \l "arrows--implicit-return) If the function body consists of a single statement returning an [expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) without side effects, omit the braces and use the implicit return. Otherwise, keep the braces and use a return statement. eslint: [arrow-parens](https://eslint.org/docs/rules/arrow-parens.html), [arrow-body-style](https://eslint.org/docs/rules/arrow-body-style.html)jscs: [disallowParenthesesAroundArrowParam](http://jscs.info/rule/disallowParenthesesAroundArrowParam), [requireShorthandArrowFunctions](http://jscs.info/rule/requireShorthandArrowFunctions)

Why? Syntactic sugar. It reads well when multiple functions are chained together.

// 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}.`);

// good

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

const nextNumber = number + 1;

return `A string containing the ${nextNumber}.`;

});

// good

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

[index]: number,

}));

// No implicit return with side effects

function foo(callback) {

const val = callback();

if (val === true) {

// Do something if callback returns true

}

}

let bool = false;

// bad

foo(() => bool = true);

// good

foo(() => {

bool = true;

});

* [8.3](https://github.com/airbnb/javascript" \l "arrows--paren-wrap) In case the expression spans over multiple lines, wrap it in parentheses for better readability.

Why? It shows clearly where the function starts and ends.

// bad

['get', 'post', 'put'].map(httpMethod => Object.prototype.hasOwnProperty.call(

httpMagicObjectWithAVeryLongName,

httpMethod,

)

);

// good

['get', 'post', 'put'].map(httpMethod => (

Object.prototype.hasOwnProperty.call(

httpMagicObjectWithAVeryLongName,

httpMethod,

)

));

* [8.4](https://github.com/airbnb/javascript" \l "arrows--one-arg-parens) If your function takes a single argument and doesn’t use braces, omit the parentheses. Otherwise, always include parentheses around arguments for clarity and consistency. Note: it is also acceptable to always use parentheses, in which case use the [“always” option](https://eslint.org/docs/rules/arrow-parens#always) for eslint or do not include [disallowParenthesesAroundArrowParam](http://jscs.info/rule/disallowParenthesesAroundArrowParam) for jscs. eslint: [arrow-parens](https://eslint.org/docs/rules/arrow-parens.html) jscs: [disallowParenthesesAroundArrowParam](http://jscs.info/rule/disallowParenthesesAroundArrowParam)

Why? Less visual clutter.

// 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 don’t want it to take up space on the .map line!`

));

// 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;

});

* [8.5](https://github.com/airbnb/javascript" \l "arrows--confusing) Avoid confusing arrow function syntax (=>) with comparison operators (<=, >=). eslint: [no-confusing-arrow](https://eslint.org/docs/rules/no-confusing-arrow)
* // 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;

};

## Classes & Constructors

* [9.1](https://github.com/airbnb/javascript" \l "constructors--use-class) Always use class. Avoid manipulating prototype directly.

Why? class syntax is more concise and easier to reason about.

// 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](https://github.com/airbnb/javascript" \l "constructors--extends) Use extends for inheritance.

Why? It is a built-in way to inherit prototype functionality without breaking instanceof.

// 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](https://github.com/airbnb/javascript" \l "constructors--chaining) 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);

* [9.4](https://github.com/airbnb/javascript" \l "constructors--tostring) It’s okay to write a custom toString() method, just make sure it works successfully and causes no side effects.
* class Jedi {
* constructor(options = {}) {
* this.name = options.name || 'no name';
* }
* getName() {
* return this.name;
* }
* toString() {
* return `Jedi - ${this.getName()}`;
* }

}

* [9.5](https://github.com/airbnb/javascript" \l "constructors--no-useless) 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](https://eslint.org/docs/rules/no-useless-constructor)
* // 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';
* }

}

* [9.6](https://github.com/airbnb/javascript" \l "classes--no-duplicate-members) Avoid duplicate class members. eslint: [no-dupe-class-members](https://eslint.org/docs/rules/no-dupe-class-members)

Why? Duplicate class member declarations will silently prefer the last one - having duplicates is almost certainly a bug.

// bad

class Foo {

bar() { return 1; }

bar() { return 2; }

}

// good

class Foo {

bar() { return 1; }

}

// good

class Foo {

bar() { return 2; }

}

## Modules

* [10.1](https://github.com/airbnb/javascript" \l "modules--use-them) Always use modules (import/export) over a non-standard module system. You can always transpile to your preferred module system.

Why? Modules are the future, let’s start using the future now.

// bad

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

module.exports = AirbnbStyleGuide.es6;

// ok

import AirbnbStyleGuide from './AirbnbStyleGuide';

export default AirbnbStyleGuide.es6;

// best

import { es6 } from './AirbnbStyleGuide';

export default es6;

* [10.2](https://github.com/airbnb/javascript" \l "modules--no-wildcard) Do not use wildcard imports.

Why? This makes sure you have a single default export.

// bad

import \* as AirbnbStyleGuide from './AirbnbStyleGuide';

// good

import AirbnbStyleGuide from './AirbnbStyleGuide';

* [10.3](https://github.com/airbnb/javascript" \l "modules--no-export-from-import) And do not export directly from an import.

Why? Although the one-liner is concise, having one clear way to import and one clear way to export makes things consistent.

// bad

// filename es6.js

export { es6 as default } from './AirbnbStyleGuide';

// good

// filename es6.js

import { es6 } from './AirbnbStyleGuide';

export default es6;

* [10.4](https://github.com/airbnb/javascript" \l "modules--no-duplicate-imports) Only import from a path in one place. eslint: [no-duplicate-imports](https://eslint.org/docs/rules/no-duplicate-imports)

Why? Having multiple lines that import from the same path can make code harder to maintain.

// bad

import foo from 'foo';

// … some other imports … //

import { named1, named2 } from 'foo';

// good

import foo, { named1, named2 } from 'foo';

// good

import foo, {

named1,

named2,

} from 'foo';

* [10.5](https://github.com/airbnb/javascript" \l "modules--no-mutable-exports) Do not export mutable bindings. eslint: [import/no-mutable-exports](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/no-mutable-exports.md)

Why? Mutation should be avoided in general, but in particular when exporting mutable bindings. While this technique may be needed for some special cases, in general, only constant references should be exported.

// bad

let foo = 3;

export { foo };

// good

const foo = 3;

export { foo };

* [10.6](https://github.com/airbnb/javascript" \l "modules--prefer-default-export) In modules with a single export, prefer default export over named export. eslint: [import/prefer-default-export](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/prefer-default-export.md)

Why? To encourage more files that only ever export one thing, which is better for readability and maintainability.

// bad

export function foo() {}

// good

export default function foo() {}

* [10.7](https://github.com/airbnb/javascript" \l "modules--imports-first) Put all imports above non-import statements. eslint: [import/first](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/first.md)

Why? Since imports are hoisted, keeping them all at the top prevents surprising behavior.

// bad

import foo from 'foo';

foo.init();

import bar from 'bar';

// good

import foo from 'foo';

import bar from 'bar';

foo.init();

* [10.8](https://github.com/airbnb/javascript" \l "modules--multiline-imports-over-newlines) Multiline imports should be indented just like multiline array and object literals.

Why? The curly braces follow the same indentation rules as every other curly brace block in the style guide, as do the trailing commas.

// bad

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

// good

import {

longNameA,

longNameB,

longNameC,

longNameD,

longNameE,

} from 'path';

* [10.9](https://github.com/airbnb/javascript" \l "modules--no-webpack-loader-syntax) Disallow Webpack loader syntax in module import statements. eslint: [import/no-webpack-loader-syntax](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/no-webpack-loader-syntax.md)

Why? Since using Webpack syntax in the imports couples the code to a module bundler. Prefer using the loader syntax in webpack.config.js.

// bad

import fooSass from 'css!sass!foo.scss';

import barCss from 'style!css!bar.css';

// good

import fooSass from 'foo.scss';

import barCss from 'bar.css';

## Iterators and Generators

* [11.1](https://github.com/airbnb/javascript" \l "iterators--nope) Don’t use iterators. Prefer JavaScript’s higher-order functions instead of loops like for-in or for-of. eslint: [no-iterator](https://eslint.org/docs/rules/no-iterator.html) [no-restricted-syntax](https://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.

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);

* [11.2](https://github.com/airbnb/javascript" \l "generators--nope) Don’t use generators for now.

Why? They don’t transpile well to ES5.

* [11.3](https://github.com/airbnb/javascript" \l "generators--spacing) If you must use generators, or if you disregard [our advice](https://github.com/airbnb/javascript#generators--nope), make sure their function signature is spaced properly. eslint: [generator-star-spacing](https://eslint.org/docs/rules/generator-star-spacing)

Why? function and \* are part of the same conceptual keyword - \* is not a modifier for function, function\* is a unique construct, different from function.

// bad

function \* foo() {

// ...

}

// bad

const bar = function \* () {

// ...

};

// bad

const baz = function \*() {

// ...

};

// bad

const quux = function\*() {

// ...

};

// bad

function\*foo() {

// ...

}

// bad

function \*foo() {

// ...

}

// very bad

function

\*

foo() {

// ...

}

// very bad

const wat = function

\*

() {

// ...

};

// good

function\* foo() {

// ...

}

// good

const foo = function\* () {

// ...

};

## Properties

* [12.1](https://github.com/airbnb/javascript" \l "properties--dot) Use dot notation when accessing properties. eslint: [dot-notation](https://eslint.org/docs/rules/dot-notation.html) jscs: [requireDotNotation](http://jscs.info/rule/requireDotNotation)
* const luke = {
* jedi: true,
* age: 28,
* };
* // bad
* const isJedi = luke['jedi'];
* // good

const isJedi = luke.jedi;

* [12.2](https://github.com/airbnb/javascript" \l "properties--bracket) Use bracket notation [] when accessing properties with a variable.
* const luke = {
* jedi: true,
* age: 28,
* };
* function getProp(prop) {
* return luke[prop];
* }

const isJedi = getProp('jedi');

* [12.3](https://github.com/airbnb/javascript" \l "es2016-properties--exponentiation-operator) Use exponentiation operator \*\* when calculating exponentiations. eslint: [no-restricted-properties](https://eslint.org/docs/rules/no-restricted-properties).
* // bad
* const binary = Math.pow(2, 10);
* // good

const binary = 2 \*\* 10;

## Variables

* [13.1](https://github.com/airbnb/javascript" \l "variables--const) 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](https://eslint.org/docs/rules/no-undef) [prefer-const](https://eslint.org/docs/rules/prefer-const)
* // bad
* superPower = new SuperPower();
* // good

const superPower = new SuperPower();

* [13.2](https://github.com/airbnb/javascript" \l "variables--one-const) Use one const or let declaration per variable. eslint: [one-var](https://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.

// 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](https://github.com/airbnb/javascript" \l "variables--const-let-group) Group all your consts and then group all your lets.

Why? This is helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

// bad

let i, len, dragonball,

items = getItems(),

goSportsTeam = true;

// bad

let i;

const items = getItems();

let dragonball;

const goSportsTeam = true;

let len;

// good

const goSportsTeam = true;

const items = getItems();

let dragonball;

let i;

let length;

* [13.4](https://github.com/airbnb/javascript" \l "variables--define-where-used) Assign variables where you need them, but place them in a reasonable place.

Why? let and const are block scoped and not function scoped.

// 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;

}

* [13.5](https://github.com/airbnb/javascript" \l "variables--no-chain-assignment) Don’t chain variable assignments. eslint: [no-multi-assign](https://eslint.org/docs/rules/no-multi-assign)

Why? Chaining variable assignments creates implicit global variables.

// 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); // throws ReferenceError

console.log(b); // 1

console.log(c); // 1

// good

(function example() {

let a = 1;

let b = a;

let c = a;

}());

console.log(a); // throws ReferenceError

console.log(b); // throws ReferenceError

console.log(c); // throws ReferenceError

// the same applies for `const`

* [13.6](https://github.com/airbnb/javascript" \l "variables--unary-increment-decrement) Avoid using unary increments and decrements (++, --). eslint [no-plusplus](https://eslint.org/docs/rules/no-plusplus)

Why? Per the eslint documentation, unary increment and decrement statements are subject to automatic semicolon insertion and can cause silent errors with incrementing or decrementing values within an application. It is also more expressive to mutate your values with statements like num += 1 instead of num++ or num ++. Disallowing unary increment and decrement statements also prevents you from pre-incrementing/pre-decrementing values unintentionally which can also cause unexpected behavior in your programs.

// bad

const array = [1, 2, 3];

let num = 1;

num++;

--num;

let sum = 0;

let truthyCount = 0;

for (let i = 0; i < array.length; i++) {

let value = array[i];

sum += value;

if (value) {

truthyCount++;

}

}

// good

const array = [1, 2, 3];

let num = 1;

num += 1;

num -= 1;

const sum = array.reduce((a, b) => a + b, 0);

const truthyCount = array.filter(Boolean).length;

## Hoisting

* [14.1](https://github.com/airbnb/javascript" \l "hoisting--about) var declarations get hoisted to the top of their scope, their assignment does not. const and let declarations are blessed with a new concept called [Temporal Dead Zones (TDZ)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Temporal_Dead_Zone_and_errors_with_let). It’s important to know why [typeof is no longer safe](http://es-discourse.com/t/why-typeof-is-no-longer-safe/15).
* // we know this wouldn’t work (assuming there
* // is no notDefined global variable)
* function example() {
* console.log(notDefined); // => throws a ReferenceError
* }
* // creating a variable declaration after you
* // reference the variable will work due to
* // variable hoisting. Note: the assignment
* // value of `true` is not hoisted.
* function example() {
* console.log(declaredButNotAssigned); // => undefined
* var declaredButNotAssigned = true;
* }
* // the interpreter is hoisting the variable
* // declaration to the top of the scope,
* // which means our example could be rewritten as:
* function example() {
* let declaredButNotAssigned;
* console.log(declaredButNotAssigned); // => undefined
* declaredButNotAssigned = true;
* }
* // using const and let
* function example() {
* console.log(declaredButNotAssigned); // => throws a ReferenceError
* console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
* const declaredButNotAssigned = true;

}

* [14.2](https://github.com/airbnb/javascript" \l "hoisting--anon-expressions) Anonymous function expressions hoist their variable name, but not the function assignment.
* function example() {
* console.log(anonymous); // => undefined
* anonymous(); // => TypeError anonymous is not a function
* var anonymous = function () {
* console.log('anonymous function expression');
* };

}

* [14.3](https://github.com/airbnb/javascript" \l "hoisting--named-expresions) Named function expressions hoist the variable name, not the function name or the function body.
* function example() {
* console.log(named); // => undefined
* named(); // => TypeError named is not a function
* superPower(); // => ReferenceError superPower is not defined
* var named = function superPower() {
* console.log('Flying');
* };
* }
* // the same is true when the function name
* // is the same as the variable name.
* function example() {
* console.log(named); // => undefined
* named(); // => TypeError named is not a function
* var named = function named() {
* console.log('named');
* };

}

* [14.4](https://github.com/airbnb/javascript" \l "hoisting--declarations) Function declarations hoist their name and the function body.
* function example() {
* superPower(); // => Flying
* function superPower() {
* console.log('Flying');
* }

}

* For more information refer to [JavaScript Scoping & Hoisting](http://www.adequatelygood.com/2010/2/JavaScript-Scoping-and-Hoisting/) by [Ben Cherry](http://www.adequatelygood.com/).

## Comparison Operators & Equality

* [15.1](https://github.com/airbnb/javascript" \l "comparison--eqeqeq) Use === and !== over == and !=. eslint: [eqeqeq](https://eslint.org/docs/rules/eqeqeq.html)

* [15.2](https://github.com/airbnb/javascript" \l "comparison--if) Conditional statements such as the if statement evaluate their expression using coercion with the ToBooleanabstract 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**
* if ([0] && []) {
* // true
* // an array (even an empty one) is an object, objects will evaluate to true

}

* [15.3](https://github.com/airbnb/javascript" \l "comparison--shortcuts) Use shortcuts for booleans, but explicit comparisons for strings and numbers.
* // bad
* if (isValid === true) {
* // ...
* }
* // good
* if (isValid) {
* // ...
* }
* // bad
* if (name) {
* // ...
* }
* // good
* if (name !== '') {
* // ...
* }
* // bad
* if (collection.length) {
* // ...
* }
* // good
* if (collection.length > 0) {
* // ...

}

* [15.4](https://github.com/airbnb/javascript" \l "comparison--moreinfo) 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](https://github.com/airbnb/javascript" \l "comparison--switch-blocks) Use braces to create blocks in case and default clauses that contain lexical declarations (e.g. let, const, function, and class). eslint: [no-case-declarations](https://eslint.org/docs/rules/no-case-declarations.html)

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.

// 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](https://github.com/airbnb/javascript" \l "comparison--nested-ternaries) Ternaries should not be nested and generally be single line expressions. eslint: [no-nested-ternary](https://eslint.org/docs/rules/no-nested-ternary.html)
* // 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;

* [15.7](https://github.com/airbnb/javascript" \l "comparison--unneeded-ternary) Avoid unneeded ternary statements. eslint: [no-unneeded-ternary](https://eslint.org/docs/rules/no-unneeded-ternary.html)
* // 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;

* [15.8](https://github.com/airbnb/javascript" \l "comparison--no-mixed-operators) Enclose operators in parentheses when they are mixed in a statement. When mixing arithmetic operators, do not mix \*\* and % with themselves or with +, -, \*, & /. eslint: [no-mixed-operators](https://eslint.org/docs/rules/no-mixed-operators.html)

Why? This improves readability and clarifies the developer’s intention.

// bad

const foo = a && b < 0 || c > 0 || d + 1 === 0;

// bad

const bar = a \*\* b - 5 % d;

// bad

if (a || b && c) {

return d;

}

// good

const foo = (a && b < 0) || c > 0 || (d + 1 === 0);

// good

const bar = (a \*\* b) - (5 % d);

// good

if ((a || b) && c) {

return d;

}

// good

const bar = a + b / c \* d;

## Blocks

* [16.1](https://github.com/airbnb/javascript" \l "blocks--braces) Use braces with all multi-line blocks. eslint: [nonblock-statement-body-position](https://eslint.org/docs/rules/nonblock-statement-body-position)
* // bad
* if (test)
* return false;
* // good
* if (test) return false;
* // good
* if (test) {
* return false;
* }
* // bad
* function foo() { return false; }
* // good
* function bar() {
* return false;

}

* [16.2](https://github.com/airbnb/javascript" \l "blocks--cuddled-elses) 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](https://eslint.org/docs/rules/brace-style.html) jscs: [disallowNewlineBeforeBlockStatements](http://jscs.info/rule/disallowNewlineBeforeBlockStatements)
* // bad
* if (test) {
* thing1();
* thing2();
* }
* else {
* thing3();
* }
* // good
* if (test) {
* thing1();
* thing2();
* } else {
* thing3();

}

* [16.3](https://github.com/airbnb/javascript" \l "blocks--no-else-return) If an if block always executes a return statement, the subsequent else block is unnecessary. A return in an else if block following an if block that contains a return can be separated into multiple if blocks. eslint: [no-else-return](https://eslint.org/docs/rules/no-else-return)
* // bad
* function foo() {
* if (x) {
* return x;
* } else {
* return y;
* }
* }
* // bad
* function cats() {
* if (x) {
* return x;
* } else if (y) {
* return y;
* }
* }
* // bad
* function dogs() {
* if (x) {
* return x;
* } else {
* if (y) {
* return y;
* }
* }
* }
* // good
* function foo() {
* if (x) {
* return x;
* }
* return y;
* }
* // good
* function cats() {
* if (x) {
* return x;
* }
* if (y) {
* return y;
* }
* }
* //good
* function dogs(x) {
* if (x) {
* if (z) {
* return y;
* }
* } else {
* return z;
* }

}

## Control Statements

* [17.1](https://github.com/airbnb/javascript" \l "control-statements) In case your control statement (if, while etc.) gets too long or exceeds the maximum line length, each (grouped) condition could be put into a new line. The logical operator should begin the line.

Why? Requiring operators at the beginning of the line keeps the operators aligned and follows a pattern similar to method chaining. This also improves readability by making it easier to visually follow complex logic.

// bad

if ((foo === 123 || bar === 'abc') && doesItLookGoodWhenItBecomesThatLong() && isThisReallyHappening()) {

thing1();

}

// bad

if (foo === 123 &&

bar === 'abc') {

thing1();

}

// bad

if (foo === 123

&& bar === 'abc') {

thing1();

}

// bad

if (

foo === 123 &&

bar === 'abc'

) {

thing1();

}

// good

if (

foo === 123

&& bar === 'abc'

) {

thing1();

}

// good

if (

(foo === 123 || bar === "abc")

&& doesItLookGoodWhenItBecomesThatLong()

&& isThisReallyHappening()

) {

thing1();

}

// good

if (foo === 123 && bar === 'abc') {

thing1();

}

## Comments

* [18.1](https://github.com/airbnb/javascript" \l "comments--multiline) Use /\*\* ... \*/ for multi-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
* \*/
* function make(tag) {
* // ...
* return element;

}

* [18.2](https://github.com/airbnb/javascript" \l "comments--singleline) 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.
* // 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;

}

* [18.3](https://github.com/airbnb/javascript" \l "comments--spaces) Start all comments with a space to make it easier to read. eslint: [spaced-comment](https://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;

}

* [18.4](https://github.com/airbnb/javascript" \l "comments--actionitems) 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.

* [18.5](https://github.com/airbnb/javascript" \l "comments--fixme) Use // FIXME: to annotate problems.
* class Calculator extends Abacus {
* constructor() {
* super();
* // FIXME: shouldn’t use a global here
* total = 0;
* }

}

* [18.6](https://github.com/airbnb/javascript" \l "comments--todo) Use // TODO: to annotate solutions to problems.
* class Calculator extends Abacus {
* constructor() {
* super();
* // TODO: total should be configurable by an options param
* this.total = 0;
* }

}

## Whitespace

* [19.1](https://github.com/airbnb/javascript" \l "whitespace--spaces) Use soft tabs (space character) set to 2 spaces. eslint: [indent](https://eslint.org/docs/rules/indent.html) jscs: [validateIndentation](http://jscs.info/rule/validateIndentation)
* // bad
* function foo() {
* ∙∙∙∙let name;
* }
* // bad
* function bar() {
* ∙let name;
* }
* // good
* function baz() {
* ∙∙let name;

}

* [19.2](https://github.com/airbnb/javascript" \l "whitespace--before-blocks) Place 1 space before the leading brace. eslint: [space-before-blocks](https://eslint.org/docs/rules/space-before-blocks.html) jscs: [requireSpaceBeforeBlockStatements](http://jscs.info/rule/requireSpaceBeforeBlockStatements)
* // bad
* function test(){
* console.log('test');
* }
* // good
* function test() {
* console.log('test');
* }
* // bad
* dog.set('attr',{
* age: '1 year',
* breed: 'Bernese Mountain Dog',
* });
* // good
* dog.set('attr', {
* age: '1 year',
* breed: 'Bernese Mountain Dog',

});

* [19.3](https://github.com/airbnb/javascript" \l "whitespace--around-keywords) 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](https://eslint.org/docs/rules/keyword-spacing.html) jscs: [requireSpaceAfterKeywords](http://jscs.info/rule/requireSpaceAfterKeywords)
* // bad
* if(isJedi) {
* fight ();
* }
* // good
* if (isJedi) {
* fight();
* }
* // bad
* function fight () {
* console.log ('Swooosh!');
* }
* // good
* function fight() {
* console.log('Swooosh!');

}

* [19.4](https://github.com/airbnb/javascript" \l "whitespace--infix-ops) Set off operators with spaces. eslint: [space-infix-ops](https://eslint.org/docs/rules/space-infix-ops.html) jscs: [requireSpaceBeforeBinaryOperators](http://jscs.info/rule/requireSpaceBeforeBinaryOperators), [requireSpaceAfterBinaryOperators](http://jscs.info/rule/requireSpaceAfterBinaryOperators)
* // bad
* const x=y+5;
* // good

const x = y + 5;

* [19.5](https://github.com/airbnb/javascript" \l "whitespace--newline-at-end) End files with a single newline character. eslint: [eol-last](https://github.com/eslint/eslint/blob/master/docs/rules/eol-last.md)
* // bad
* import { es6 } from './AirbnbStyleGuide';
* // ...

export default es6;

// bad

import { es6 } from './AirbnbStyleGuide';

// ...

export default es6;↵

↵

// good

import { es6 } from './AirbnbStyleGuide';

// ...

export default es6;↵

* [19.6](https://github.com/airbnb/javascript" \l "whitespace--chains) 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](https://eslint.org/docs/rules/newline-per-chained-call) [no-whitespace-before-property](https://eslint.org/docs/rules/no-whitespace-before-property)
* // 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);

* [19.7](https://github.com/airbnb/javascript" \l "whitespace--after-blocks) Leave a blank line after blocks and before the next statement. jscs: [requirePaddingNewLinesAfterBlocks](http://jscs.info/rule/requirePaddingNewLinesAfterBlocks)
* // 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;

* [19.8](https://github.com/airbnb/javascript" \l "whitespace--padded-blocks) Do not pad your blocks with blank lines. eslint: [padded-blocks](https://eslint.org/docs/rules/padded-blocks.html) jscs: [disallowPaddingNewlinesInBlocks](http://jscs.info/rule/disallowPaddingNewlinesInBlocks)
* // bad
* function bar() {
* console.log(foo);
* }
* // bad
* if (baz) {
* console.log(qux);
* } else {
* console.log(foo);
* }
* // bad
* class Foo {
* constructor(bar) {
* this.bar = bar;
* }
* }
* // good
* function bar() {
* console.log(foo);
* }
* // good
* if (baz) {
* console.log(qux);
* } else {
* console.log(foo);

}

* [19.9](https://github.com/airbnb/javascript" \l "whitespace--in-parens) Do not add spaces inside parentheses. eslint: [space-in-parens](https://eslint.org/docs/rules/space-in-parens.html) jscs: [disallowSpacesInsideParentheses](http://jscs.info/rule/disallowSpacesInsideParentheses)
* // bad
* function bar( foo ) {
* return foo;
* }
* // good
* function bar(foo) {
* return foo;
* }
* // bad
* if ( foo ) {
* console.log(foo);
* }
* // good
* if (foo) {
* console.log(foo);

}

* [19.10](https://github.com/airbnb/javascript" \l "whitespace--in-brackets) Do not add spaces inside brackets. eslint: [array-bracket-spacing](https://eslint.org/docs/rules/array-bracket-spacing.html) jscs: [disallowSpacesInsideArrayBrackets](http://jscs.info/rule/disallowSpacesInsideArrayBrackets)
* // bad
* const foo = [ 1, 2, 3 ];
* console.log(foo[ 0 ]);
* // good
* const foo = [1, 2, 3];

console.log(foo[0]);

* [19.11](https://github.com/airbnb/javascript" \l "whitespace--in-braces) Add spaces inside curly braces. eslint: [object-curly-spacing](https://eslint.org/docs/rules/object-curly-spacing.html) jscs: [requireSpacesInsideObjectBrackets](http://jscs.info/rule/requireSpacesInsideObjectBrackets)
* // bad
* const foo = {clark: 'kent'};
* // good

const foo = { clark: 'kent' };

* [19.12](https://github.com/airbnb/javascript" \l "whitespace--max-len) Avoid having lines of code that are longer than 100 characters (including whitespace). Note: per [above](https://github.com/airbnb/javascript#strings--line-length), long strings are exempt from this rule, and should not be broken up. eslint: [max-len](https://eslint.org/docs/rules/max-len.html) jscs: [maximumLineLength](http://jscs.info/rule/maximumLineLength)

Why? This ensures readability and maintainability.

// 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.'));

## Commas

* [20.1](https://github.com/airbnb/javascript" \l "commas--leading-trailing) Leading commas: **Nope.** eslint: [comma-style](https://eslint.org/docs/rules/comma-style.html) jscs: [requireCommaBeforeLineBreak](http://jscs.info/rule/requireCommaBeforeLineBreak)
* // 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',

};

* [20.2](https://github.com/airbnb/javascript" \l "commas--dangling) Additional trailing comma: **Yup.** eslint: [comma-dangle](https://eslint.org/docs/rules/comma-dangle.html) jscs: [requireTrailingComma](http://jscs.info/rule/requireTrailingComma)

Why? This leads to cleaner git diffs. Also, transpilers like Babel will remove the additional trailing comma in the transpiled code which means you don’t have to worry about the [trailing comma problem](https://github.com/airbnb/javascript/blob/es5-deprecated/es5/README.md#commas) in legacy browsers.

// bad - git diff without trailing comma

const hero = {

firstName: 'Florence',

- lastName: 'Nightingale'

+ lastName: 'Nightingale',

+ inventorOf: ['coxcomb chart', 'modern nursing']

};

// good - git diff with trailing comma

const hero = {

firstName: 'Florence',

lastName: 'Nightingale',

+ inventorOf: ['coxcomb chart', 'modern nursing'],

};

// bad

const hero = {

firstName: 'Dana',

lastName: 'Scully'

};

const heroes = [

'Batman',

'Superman'

];

// good

const hero = {

firstName: 'Dana',

lastName: 'Scully',

};

const heroes = [

'Batman',

'Superman',

];

// bad

function createHero(

firstName,

lastName,

inventorOf

) {

// does nothing

}

// good

function createHero(

firstName,

lastName,

inventorOf,

) {

// does nothing

}

// good (note that a comma must not appear after a "rest" element)

function createHero(

firstName,

lastName,

inventorOf,

...heroArgs

) {

// does nothing

}

// bad

createHero(

firstName,

lastName,

inventorOf

);

// good

createHero(

firstName,

lastName,

inventorOf,

);

// good (note that a comma must not appear after a "rest" element)

createHero(

firstName,

lastName,

inventorOf,

...heroArgs

);

## Semicolons

* [21.1](https://github.com/airbnb/javascript" \l "semicolons--required) **Yup.** eslint: [semi](https://eslint.org/docs/rules/semi.html) jscs: [requireSemicolons](http://jscs.info/rule/requireSemicolons)
* // 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#7365214).

[**⬆ back to top**](https://github.com/airbnb/javascript#table-of-contents)

## Type Casting & Coercion

* [22.1](https://github.com/airbnb/javascript" \l "coercion--explicit) Perform type coercion at the beginning of the statement.

* [22.2](https://github.com/airbnb/javascript" \l "coercion--strings) Strings: eslint: [no-new-wrappers](https://eslint.org/docs/rules/no-new-wrappers)
* // => this.reviewScore = 9;
* // bad
* const totalScore = new String(this.reviewScore); // typeof totalScore is "object" not "string"
* // 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);

* [22.3](https://github.com/airbnb/javascript" \l "coercion--numbers) Numbers: Use Number for type casting and parseInt always with a radix for parsing strings. eslint: [radix](https://eslint.org/docs/rules/radix) [no-new-wrappers](https://eslint.org/docs/rules/no-new-wrappers)
* 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);

* [22.4](https://github.com/airbnb/javascript" \l "coercion--comment-deviations) 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.
* // 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;

* [22.5](https://github.com/airbnb/javascript" \l "coercion--bitwise) **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:
* 2147483647 >> 0; // => 2147483647
* 2147483648 >> 0; // => -2147483648

2147483649 >> 0; // => -2147483647

* [22.6](https://github.com/airbnb/javascript" \l "coercion--booleans) Booleans: eslint: [no-new-wrappers](https://eslint.org/docs/rules/no-new-wrappers)
* const age = 0;
* // bad
* const hasAge = new Boolean(age);
* // good
* const hasAge = Boolean(age);
* // best

const hasAge = !!age;

## Naming Conventions

* [23.1](https://github.com/airbnb/javascript" \l "naming--descriptive) Avoid single letter names. Be descriptive with your naming. eslint: [id-length](https://eslint.org/docs/rules/id-length)
* // bad
* function q() {
* // ...
* }
* // good
* function query() {
* // ...

}

* [23.2](https://github.com/airbnb/javascript" \l "naming--camelCase) Use camelCase when naming objects, functions, and instances. eslint: [camelcase](https://eslint.org/docs/rules/camelcase.html) jscs: [requireCamelCaseOrUpperCaseIdentifiers](http://jscs.info/rule/requireCamelCaseOrUpperCaseIdentifiers)
* // bad
* const OBJEcttsssss = {};
* const this\_is\_my\_object = {};
* function c() {}
* // good
* const thisIsMyObject = {};

function thisIsMyFunction() {}

* [23.3](https://github.com/airbnb/javascript" \l "naming--PascalCase) Use PascalCase only when naming constructors or classes. eslint: [new-cap](https://eslint.org/docs/rules/new-cap.html) jscs: [requireCapitalizedConstructors](http://jscs.info/rule/requireCapitalizedConstructors)
* // 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',

});

* [23.4](https://github.com/airbnb/javascript" \l "naming--leading-underscore) Do not use trailing or leading underscores. eslint: [no-underscore-dangle](https://eslint.org/docs/rules/no-underscore-dangle.html) jscs: [disallowDanglingUnderscores](http://jscs.info/rule/disallowDanglingUnderscores)

Why? JavaScript does not have the concept of privacy in terms of properties or methods. Although a leading underscore is a common convention to mean “private”, in fact, these properties are fully public, and as such, are part of your public API contract. This convention might lead developers to wrongly think that a change won’t count as breaking, or that tests aren’t needed. tl;dr: if you want something to be “private”, it must not be observably present.

// bad

this.\_\_firstName\_\_ = 'Panda';

this.firstName\_ = 'Panda';

this.\_firstName = 'Panda';

// good

this.firstName = 'Panda';

* [23.5](https://github.com/airbnb/javascript" \l "naming--self-this) 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)
* // 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);
* };

}

* [23.6](https://github.com/airbnb/javascript" \l "naming--filename-matches-export) A base filename should exactly match the name of its default export.
* // file 1 contents
* class CheckBox {
* // ...
* }
* export default CheckBox;
* // file 2 contents
* export default function fortyTwo() { return 42; }
* // file 3 contents
* export default function insideDirectory() {}
* // in some other file
* // bad
* import CheckBox from './checkBox'; // PascalCase import/export, camelCase filename
* import FortyTwo from './FortyTwo'; // PascalCase import/filename, camelCase export
* import InsideDirectory from './InsideDirectory'; // PascalCase import/filename, camelCase export
* // bad
* import CheckBox from './check\_box'; // PascalCase import/export, snake\_case filename
* import forty\_two from './forty\_two'; // snake\_case import/filename, camelCase export
* import inside\_directory from './inside\_directory'; // snake\_case import, camelCase export
* import index from './inside\_directory/index'; // requiring the index file explicitly
* import insideDirectory from './insideDirectory/index'; // requiring the index file explicitly
* // good
* import CheckBox from './CheckBox'; // PascalCase export/import/filename
* import fortyTwo from './fortyTwo'; // camelCase export/import/filename
* import insideDirectory from './insideDirectory'; // camelCase export/import/directory name/implicit "index"

// ^ supports both insideDirectory.js and insideDirectory/index.js

* [23.7](https://github.com/airbnb/javascript" \l "naming--camelCase-default-export) Use camelCase when you export-default a function. Your filename should be identical to your function’s name.
* function makeStyleGuide() {
* // ...
* }

export default makeStyleGuide;

* [23.8](https://github.com/airbnb/javascript" \l "naming--PascalCase-singleton) Use PascalCase when you export a constructor / class / singleton / function library / bare object.
* const AirbnbStyleGuide = {
* es6: {
* },
* };

export default AirbnbStyleGuide;

* [23.9](https://github.com/airbnb/javascript" \l "naming--Acronyms-and-Initialisms) Acronyms and initialisms should always be all capitalized, or all lowercased.

Why? Names are for readability, not to appease a computer algorithm.

// bad

import SmsContainer from './containers/SmsContainer';

// bad

const HttpRequests = [

// ...

];

// good

import SMSContainer from './containers/SMSContainer';

// good

const HTTPRequests = [

// ...

];

// also good

const httpRequests = [

// ...

];

// best

import TextMessageContainer from './containers/TextMessageContainer';

// best

const requests = [

// ...

];

## Accessors

* [24.1](https://github.com/airbnb/javascript" \l "accessors--not-required) Accessor functions for properties are not required.

* [24.2](https://github.com/airbnb/javascript" \l "accessors--no-getters-setters) Do not use JavaScript getters/setters as they cause unexpected side effects and are harder to test, maintain, and reason about. Instead, if you do make accessor functions, use getVal() and setVal('hello').
* // bad
* class Dragon {
* get age() {
* // ...
* }
* set age(value) {
* // ...
* }
* }
* // good
* class Dragon {
* getAge() {
* // ...
* }
* setAge(value) {
* // ...
* }

}

* [24.3](https://github.com/airbnb/javascript" \l "accessors--boolean-prefix) If the property/method is a boolean, use isVal() or hasVal().
* // bad
* if (!dragon.age()) {
* return false;
* }
* // good
* if (!dragon.hasAge()) {
* return false;

}

* [24.4](https://github.com/airbnb/javascript" \l "accessors--consistent) It’s okay to create get() and set() functions, but be consistent.
* class Jedi {
* constructor(options = {}) {
* const lightsaber = options.lightsaber || 'blue';
* this.set('lightsaber', lightsaber);
* }
* set(key, val) {
* this[key] = val;
* }
* get(key) {
* return this[key];
* }

}

## Events

* [25.1](https://github.com/airbnb/javascript" \l "events--hash) 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:
* // bad
* $(this).trigger('listingUpdated', listing.id);
* // ...
* $(this).on('listingUpdated', (e, listingId) => {
* // do something with listingId

});

prefer:

// good

$(this).trigger('listingUpdated', { listingId: listing.id });

// ...

$(this).on('listingUpdated', (e, data) => {

// do something with data.listingId

});

## jQuery

* [26.1](https://github.com/airbnb/javascript" \l "jquery--dollar-prefix) Prefix jQuery object variables with a $. jscs: [requireDollarBeforejQueryAssignment](http://jscs.info/rule/requireDollarBeforejQueryAssignment)
* // bad
* const sidebar = $('.sidebar');
* // good
* const $sidebar = $('.sidebar');
* // good

const $sidebarBtn = $('.sidebar-btn');

* [26.2](https://github.com/airbnb/javascript" \l "jquery--cache) Cache jQuery lookups.
* // bad
* function setSidebar() {
* $('.sidebar').hide();
* // ...
* $('.sidebar').css({
* 'background-color': 'pink',
* });
* }
* // good
* function setSidebar() {
* const $sidebar = $('.sidebar');
* $sidebar.hide();
* // ...
* $sidebar.css({
* 'background-color': 'pink',
* });

}

* [26.3](https://github.com/airbnb/javascript" \l "jquery--queries) For DOM queries use Cascading $('.sidebar ul') or parent > child $('.sidebar > ul'). [jsPerf](http://jsperf.com/jquery-find-vs-context-sel/16)

* [26.4](https://github.com/airbnb/javascript" \l "jquery--find) Use find with scoped jQuery object queries.
* // bad
* $('ul', '.sidebar').hide();
* // bad
* $('.sidebar').find('ul').hide();
* // good
* $('.sidebar ul').hide();
* // good
* $('.sidebar > ul').hide();
* // good

$sidebar.find('ul').hide();

## ECMAScript 5 Compatibility

* [27.1](https://github.com/airbnb/javascript" \l "es5-compat--kangax) Refer to [Kangax](https://twitter.com/kangax/)’s ES5 [compatibility table](https://kangax.github.io/es5-compat-table/).

## Standard Library

The [Standard Library](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects) contains utilities that are functionally broken but remain for legacy reasons.

* [29.1](https://github.com/airbnb/javascript" \l "standard-library--isnan) Use Number.isNaN instead of global isNaN. eslint: [no-restricted-globals](https://eslint.org/docs/rules/no-restricted-globals)

Why? The global isNaN coerces non-numbers to numbers, returning true for anything that coerces to NaN. If this behavior is desired, make it explicit.

// bad

isNaN('1.2'); // false

isNaN('1.2.3'); // true

// good

Number.isNaN('1.2.3'); // false

Number.isNaN(Number('1.2.3')); // true

* [29.2](https://github.com/airbnb/javascript" \l "standard-library--isfinite) Use Number.isFinite instead of global isFinite. eslint: [no-restricted-globals](https://eslint.org/docs/rules/no-restricted-globals)

Why? The global isFinite coerces non-numbers to numbers, returning true for anything that coerces to a finite number. If this behavior is desired, make it explicit.

// bad

isFinite('2e3'); // true

// good

Number.isFinite('2e3'); // false

Number.isFinite(parseInt('2e3', 10)); // true

## Testing

* [30.1](https://github.com/airbnb/javascript" \l "testing--yup) **Yup.**
* function foo() {
* return true;

}

* [30.2](https://github.com/airbnb/javascript" \l "testing--for-real) **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.