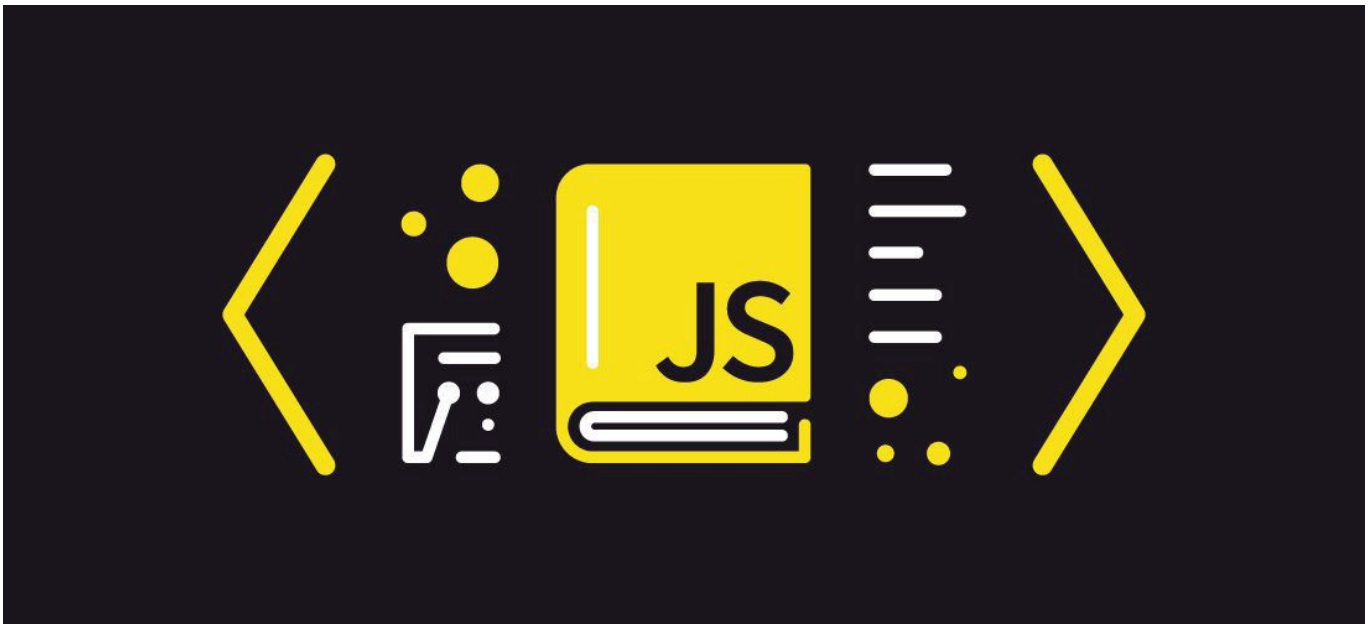
[hdg]: JavaScript Style Guide() {



You can download the bundled versions of it as a HTML file, or PDF this style guide was super long so we broke it up into smaller “Chapters” which you can view in the repo for this guide. We provide a copy on our site in the hopes it will help you as much as it has us. You can find a copy of the Javascript Header we use in conjunction with this guide on our site https://hasidic.dec/style

A mostly reasonable approach to JavaScript

> **Note**: this guide assumes you are using Babel, and requires that you use babel-preset-airbnb or the equivalent. It also assumes you are installing shims/polyfills in your app, with airbnb-browser-shims or the equivalent.

Table of Contents

Types

- 1.1 **Primitives**: When you access a primitive type you work directly on its value.
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`
  - `symbol`
  - `bigint`

```
const foo = 1;
let bar = foo;

bar = 9;

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

  - Symbols and BigInts cannot be faithfully polyfilled, so they should not be used when targeting browsers/environments that don’t support them natively.

- 1.2 **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
```

⬆ back to top

References

- 2.1 Use `const` for all of your references; avoid using `var`. eslint: `prefer-const`, `no-const-assign`

> 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 If you must reassign references, use `let` instead of `var`. eslint: `no-var`

> 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 Note that both `let` and `const` are block-scoped, whereas `var` is function-scoped.

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

In the above code, you can see that referencing `a` and `b` will produce a ReferenceError, while `c` contains the number. This is because `a` and `b` are block scoped, while `c` is scoped to the containing function.

⬆ back to top

Objects

- 3.1 Use the literal syntax for object creation. eslint: `no-new-object`

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

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

- 3.2 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 Use object method shorthand. eslint: `object-shorthand`

```
// 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 Use property value shorthand. eslint: `object-shorthand`

> Why? It is shorter and descriptive.

```
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

- 3.5 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 Only quote properties that are invalid identifiers. eslint: `quote-props`

> 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 Do not call `Object.prototype` methods directly, such as `hasOwnProperty`, `propertyIsEnumerable`, and `isPrototypeOf`. eslint: `no-prototype-builtins`

> 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.
console.log(has.call(object, key));
/* or */
import has from 'has'; // https://www.npmjs.com/package/has
console.log(has(object, key));
/* or */
console.log(Object.hasOwn(object, key)); //
https://www.npmjs.com/package/object.hasown
```

- 3.8 Prefer the object spread syntax over `Object.assign` to shallow-copy objects. Use the object rest parameter syntax to get a new object with certain properties omitted. eslint: `prefer-object-spread`

```
// 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

Arrays

- 4.1 Use the literal syntax for array creation. eslint: `no-array-constructor`

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

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

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

```
const someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- 4.3 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 To convert an iterable object to an array, use spreads `...` instead of `Array.from`

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

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

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

- 4.5 Use `Array.from` for converting an array-like object to an array.

```
const arrLike = { 0: 'foo', 1: 'bar', 2: 'baz', length: 3 };

// bad
const arr = Array.prototype.slice.call(arrLike);

// good
const arr = Array.from(arrLike);
```

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

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

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

- 4.7 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. eslint: `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 `acc` becomes undefined after the
first iteration
[[0, 1], [2, 3], [4, 5]].reduce((acc, item, index) => {
  const flatten = acc.concat(item);
});

// good
[[0, 1], [2, 3], [4, 5]].reduce((acc, item, index) => {
  const flatten = acc.concat(item);
  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;
});
```

- 4.8 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,
];
```

⬆ back to top

Destructuring

- 5.1 Use object destructuring when accessing and using multiple properties of an object. eslint: `prefer-destructuring`

> Why? Destructuring saves you from creating temporary references for those properties, and from repetitive access of the object. Repeating object access creates more repetitive code, requires more reading, and creates more opportunities for mistakes. Destructuring objects also provides a single site of definition of the object structure that is used in the block, rather than requiring reading the entire block to determine what is used.

```
// 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. eslint: `prefer-destructuring`

```
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.

> 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);
```

⬆ back to top

Strings

- 6.1 Use single quotes `''` for strings. eslint: `quotes`

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

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

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

- 6.2 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 When programmatically building up strings, use template strings instead of concatenation. eslint: `prefer-template` `template-curly-spacing`

> 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 Never use `eval()` on a string, it opens too many vulnerabilities. eslint: `no-eval`
- 6.5 Do not unnecessarily escape characters in strings. eslint: `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}'`;
```

⬆ back to top

Functions

- 7.1 Use named function expressions instead of function declarations. eslint: `func-style`

> 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)

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

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

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

- 7.2 Wrap immediately invoked function expressions in parentheses. eslint: `wrap-iife`

> 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 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`

- 7.4 **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 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 Never use `arguments`, opt to use rest syntax `...` instead. eslint: `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 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 Avoid side effects with default parameters.

  > Why? They are confusing to reason about.

```
let b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

- 7.9 Always put default parameters last. eslint: `default-param-last`

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

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

- 7.10 Never use the Function constructor to create a new function. eslint: `no-new-func`

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

```
// bad
const add = new Function('a', 'b', 'return a + b');

// still bad
const subtract = Function('a', 'b', 'return a - b');
```

- 7.11 Spacing in a function signature. eslint: `space-before-function-paren` `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 Never mutate parameters. eslint: `no-param-reassign`

  > 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 Never reassign parameters. eslint: `no-param-reassign`

  > 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 Prefer the use of the spread syntax `...` to call variadic functions. eslint: `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 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. eslint: `function-paren-newline`

```
// 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

Arrow Functions

- 8.1 When you must use an anonymous function (as when passing an inline callback), use arrow function notation. eslint: `prefer-arrow-callback`, `arrow-spacing`

  > 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 If the function body consists of a single statement returning an expression without side effects, omit the braces and use the implicit return. Otherwise, keep the braces and use a `return` statement. eslint: `arrow-parens`, `arrow-body-style`

  > 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 + 1}.`);

// 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 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 Always include parentheses around arguments for clarity and consistency. eslint: `arrow-parens`

  > Why? Minimizes diff churn when adding or removing arguments.

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

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

// bad
[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!`
));

// 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 Avoid confusing arrow function syntax (`=>`) with comparison operators (`<=`, `>=`). eslint: `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;
};
```

- 8.6 Enforce the location of arrow function bodies with implicit returns. eslint: `implicit-arrow-linebreak`

```
// bad
(foo) =>
  bar;

(foo) =>
  (bar);

// good
(foo) => bar;
(foo) => (bar);
(foo) => (
   bar
)
```

⬆ back to top

Classes & Constructors

- 9.1 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 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 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 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 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`

```
// 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 Avoid duplicate class members. eslint: `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; }
}
```

- 9.7 Class methods should use `this` or be made into a static method unless an external library or framework requires using specific non-static methods. Being an instance method should indicate that it behaves differently based on properties of the receiver. eslint: `class-methods-use-this`

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

// good - this is used
class Foo {
  bar() {
    console.log(this.bar);
  }
}

// good - constructor is exempt
class Foo {
  constructor() {
    // ...
  }
}
```

```
// good - static methods aren't expected to use this
class Foo {
  static bar() {
    console.log('bar');
  }
}
```

⬆ back to top

Modules

- 10.1 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 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 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 Only import from a path in one place. eslint: `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 Do not export mutable bindings. eslint: `import/no-mutable-exports`

> 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 In modules with a single export, prefer default export over named export. eslint: `import/prefer-default-export`

> 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 Put all `import`s above non-import statements. eslint: `import/first`

> Why? Since `import`s 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 Multiline imports should be indented just like multiline array and object literals. eslint: `object-curly-newline`

> 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 Disallow Webpack loader syntax in module import statements. eslint: `import/no-webpack-loader-syntax`

> 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';
```

- 10.10 Do not include JavaScript filename extensions eslint: `import/extensions`

> Why? Including extensions inhibits refactoring, and inappropriately hardcodes implementation details of the module you're importing in every consumer.

```
// bad
import foo from './foo.js';
import bar from './bar.jsx';
import baz from './baz/index.jsx';

// good
import foo from './foo';
import bar from './bar';
import baz from './baz';
```

⬆ back to top

Iterators and Generators

- 11.1 Don’t use iterators. Prefer JavaScript’s higher-order functions instead of loops like `for-in` or `for-of`. eslint: `no-iterator` `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 Don’t use generators for now.

  > Why? They don’t transpile well to ES5.

- 11.3 If you must use generators, or if you disregard our advice, make sure their function signature is spaced properly. eslint: `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* () {
  // ...
};
```

⬆ back to top

Properties

- 12.1 Use dot notation when accessing properties. eslint: `dot-notation`

```
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.

```
const luke = {
  jedi: true,
  age: 28,
};

function getProp(prop) {
  return luke[prop];
}

const isJedi = getProp('jedi');
```

- 12.3 Use exponentiation operator `**` when calculating exponentiations. eslint: `no-restricted-properties`.

```
// bad
const binary = Math.pow(2, 10);

// good
const binary = 2 ** 10;
```

⬆ back to top

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 prefer-const`

```
// bad
superPower = new SuperPower();

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

- 13.2 Use one `const` or `let` declaration per variable or assignment. eslint: `one-var`

> 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 Group all your `const`s and then group all your `let`s.

> Why? This is helpful when later on you might need to assign a variable depending on one of the previously 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 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 Don’t chain variable assignments. eslint: `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 Avoid using unary increments and decrements (`++`, `--`). eslint `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;
```

- 13.7 Avoid linebreaks before or after `=` in an assignment. If your assignment violates `max-len`, surround the value in parens. eslint `operator-linebreak`.

> Why? Linebreaks surrounding `=` can obfuscate the value of an assignment.

```
// bad
const foo =
  superLongLongLongLongLongLongLongFunctionName();

// bad
const foo
  = 'superLongLongLongLongLongLongLongString';
```

```
// good
const foo = (
  superLongLongLongLongLongLongLongFunctionName()
);

// good
const foo = 'superLongLongLongLongLongLongLongString';
```

- 13.8 Disallow unused variables. eslint: `no-unused-vars`

> Why? Variables that are declared and not used anywhere in the code are most likely an error due to incomplete refactoring. Such variables take up space in the code and can lead to confusion by readers.

```
// bad

const some_unused_var = 42;

// Write-only variables are not considered as used.
let y = 10;
y = 5;

// A read for a modification of itself is not considered as used.
let z = 0;
z = z + 1;

// Unused function arguments.
function getX(x, y) {
    return x;
}

// good

function getXPlusY(x, y) {
  return x + y;
}

const x = 1;
const y = a + 2;

alert(getXPlusY(x, y));

// 'type' is ignored even if unused because it has a rest property
sibling.
// This is a form of extracting an object that omits the specified
keys.
const { type, ...coords } = data;
// 'coords' is now the 'data' object without its 'type' property.
```

⬆ back to top

Hoisting

- 14.1 `var` declarations get hoisted to the top of their closest enclosing function scope, their assignment does not. `const` and `let` declarations are blessed with a new concept called Temporal Dead Zones (TDZ). It's important to know why typeof is no longer safe.

```
// 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 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 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 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 by Ben Cherry.

⬆ back to top

Comparison Operators & Equality

- 15.1 Use `===` and `!==` over `==` and `!=`. eslint: `eqeqeq`

- 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**

```
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.

```
// 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 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`). eslint: `no-case-declarations`

  > 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 Ternaries should not be nested and generally be single line expressions. eslint: `no-nested-ternary`

```
// 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 Avoid unneeded ternary statements. eslint: `no-unneeded-ternary`

```
// bad
const foo = a ? a : b;
const bar = c ? true : false;
const baz = c ? false : true;
const quux = a != null ? a : b;

// good
const foo = a || b;
const bar = !!c;
const baz = !c;
const quux = a ?? b;
```

- 15.8 When mixing operators, enclose them in parentheses. The only exception is the standard arithmetic operators: `+`, `-`, and `**` since their precedence is broadly understood. We recommend enclosing `/` and `*` in parentheses because their precedence can be ambiguous when they are mixed. eslint: `no-mixed-operators`

> 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
// one may be confused into thinking (a || b) && c
```

```
if (a || b && c) {
  return d;
}

// bad
const bar = a + b / c * 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;
```

⬆ back to top

Blocks

- 16.1 Use braces with all multiline blocks. eslint: `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 If you're using multiline blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace. eslint: `brace-style`

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

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

- 16.3 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`

```
// 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;
  }
}
```

⬆ back to top

Control Statements

- 17.1 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();
}
```

- 17.2 Don't use selection operators in place of control statements.

```
// bad
!isRunning && startRunning();

// good
if (!isRunning) {
  startRunning();
}
```

⬆ back to top

Comments

- 18.1 Use `/** ... */` for multiline 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 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 Start all comments with a space to make it easier to read. eslint: `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 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 Use `// FIXME:` to annotate problems.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

- 18.6 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;
  }
}
```

⬆ back to top

Whitespace

- 19.1 Use soft tabs (space character) set to 2 spaces. eslint: `indent`

```
// bad
function foo() {
····let name;
}

// bad
function bar() {
·let name;
}

// good
function baz() {
··let name;
}
```

- 19.2 Place 1 space before the leading brace. eslint: `space-before-blocks`

```
// 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 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`

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

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

- 19.4 Set off operators with spaces. eslint: `space-infix-ops`

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

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

- 19.5 End files with a single newline character. eslint: `eol-last`

```
// 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 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` `no-whitespace-before-property`

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCo
unt();

// 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);
const svg = leds.enter().append('svg:svg');
svg.classed('led', true).attr('width', (radius + margin) * 2);
const g = svg.append('svg:g');
g.attr('transform', `translate(${radius + margin}, ${radius +
margin})`).call(tron.led);
```

- 19.7 Leave a blank line after blocks and before the next statement.

```

// 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 Do not pad your blocks with blank lines. eslint: `padded-blocks`

```
// bad
function bar() {

  console.log(foo);

}

// bad
if (baz) {

  console.log(quux);
} else {
```

```
  console.log(foo);

}

// bad
class Foo {

  constructor(bar) {
    this.bar = bar;
  }
}

// good
function bar() {
  console.log(foo);
}

// good
if (baz) {
  console.log(quux);
} else {
  console.log(foo);
}
```

- 19.9 Do not use multiple blank lines to pad your code. eslint: `no-multiple-empty-lines`

```
// bad
class Person {
  constructor(fullName, email, birthday) {
    this.fullName = fullName;


    this.email = email;


    this.setAge(birthday);
  }


  setAge(birthday) {
    const today = new Date();


    const age = this.getAge(today, birthday);


    this.age = age;
  }


  getAge(today, birthday) {
    // ..
```

```
  }
}

// good
class Person {
  constructor(fullName, email, birthday) {
    this.fullName = fullName;
    this.email = email;
    this.setAge(birthday);
  }

  setAge(birthday) {
    const today = new Date();
    const age = getAge(today, birthday);
    this.age = age;
  }

  getAge(today, birthday) {
    // ..
  }
}
```

- 19.10 Do not add spaces inside parentheses. eslint: `space-in-parens`

```
// bad
function bar( foo ) {
  return foo;
}

// good
function bar(foo) {
  return foo;
}

// bad
if ( foo ) {
  console.log(foo);
}

// good
if (foo) {
  console.log(foo);
}
```

- 19.11 Do not add spaces inside brackets. eslint: `array-bracket-spacing`

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

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

- 19.12 Add spaces inside curly braces. eslint: `object-curly-spacing`

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

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

- 19.13 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`

> 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;

// better
const foo = 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.'));
```

- 19.14 Require consistent spacing inside an open block token and the next token on the same line. This rule also enforces consistent spacing inside a close block token and previous token on the same line. eslint: `block-spacing`

```
// bad
function foo() {return true;}
if (foo) { bar = 0;}

// good
function foo() { return true; }
if (foo) { bar = 0; }
```

- 19.15 Avoid spaces before commas and require a space after commas. eslint: `comma-spacing`

```
// bad
const foo = 1,bar = 2;
const arr = [1 , 2];

// good
const foo = 1, bar = 2;
const arr = [1, 2];
```

- 19.16 Enforce spacing inside of computed property brackets. eslint: `computed-property-spacing`

```
// bad
obj[foo ]
obj[ 'foo']
const x = {[ b ]: a}
obj[foo[ bar ]]

// good
obj[foo]
obj['foo']
const x = { [b]: a }
obj[foo[bar]]
```

- 19.17 Avoid spaces between functions and their invocations. eslint: `func-call-spacing`

```
// bad
func ();
```

```
func
();

// good
func();
```

- 19.18 Enforce spacing between keys and values in object literal properties. eslint: `key-spacing`

```
// bad
const obj = { foo : 42 };
const obj2 = { foo:42 };

// good
const obj = { foo: 42 };
```

- 19.19 Avoid trailing spaces at the end of lines. eslint: `no-trailing-spaces`
- 19.20 Avoid multiple empty lines, only allow one newline at the end of files, and avoid a newline at the beginning of files. eslint: `no-multiple-empty-lines`

```
// bad - multiple empty lines
const x = 1;


const y = 2;

// bad - 2+ newlines at end of file
const x = 1;
const y = 2;


// bad - 1+ newline(s) at beginning of file

const x = 1;
const y = 2;

// good
const x = 1;
const y = 2;
```

⬆ back to top

Commas

- 20.1 Leading commas: **Nope.** eslint: `comma-style`

```
// 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 Additional trailing comma: **Yup.** eslint: `comma-dangle`

> 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 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
);
```

⬆ back to top

Semicolons

- 21.1 **Yup.** eslint: `semi`

> Why? When JavaScript encounters a line break without a semicolon, it uses a set of rules called Automatic Semicolon Insertion to determine whether it should regard that line break as the end of a statement, and (as the name implies) place a semicolon into your code before the line break if it thinks so. ASI contains a few eccentric behaviors, though, and your code will break if JavaScript misinterprets your line break. These rules will become more complicated as new features become a part of JavaScript. Explicitly terminating your statements and configuring your linter to catch missing semicolons will help prevent you from encountering issues.

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

// bad - raises exception
const reaction = "No! That’s impossible!"
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}())

// bad - returns `undefined` instead of the value on the next line -
always happens when `return` is on a line by itself because of ASI!
function foo() {
  return
```

```
    'search your feelings, you know it to be foo'
}

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

// good
const reaction = 'No! That's impossible!';
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}());

// good
function foo() {
  return 'search your feelings, you know it to be foo';
}
```

Read more.

⬆ back to top

Type Casting & Coercion

- 22.1 Perform type coercion at the beginning of the statement.
- 22.2 Strings: eslint: `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 Numbers: Use `Number` for type casting and `parseInt` always with a radix for parsing strings. eslint: `radix` `no-new-wrappers`

  > Why? The `parseInt` function produces an integer value dictated by interpretation of the contents of the string argument according to the specified radix. Leading whitespace in string is ignored. If radix is `undefined` or `0`, it is assumed to be `10` except when the number begins with the character pairs `0x` or `0X`, in which case a radix of 16 is assumed. This differs from ECMAScript 3, which merely discouraged (but allowed) octal interpretation. Many implementations have not adopted this behavior as of 2013. And, because older browsers must be supported, always specify a radix.

```
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 If for whatever reason you are doing something wild and `parseInt` is your bottleneck and need to use Bitshift for performance reasons, 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 **Note:** Be careful when using bitshift operations. Numbers are represented as 64-bit values, but bitshift operations always return a 32-bit integer (source). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. Discussion. Largest signed 32-bit Int is 2,147,483,647:

```
```

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

- 22.6 Booleans: eslint: `no-new-wrappers`

```
const age = 0;

// bad
const hasAge = new Boolean(age);

// good
const hasAge = Boolean(age);

// best
const hasAge = !!age;
```

⬆ back to top

Naming Conventions

- 23.1 Avoid single letter names. Be descriptive with your naming. eslint: `id-length`

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

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

- 23.2 Use camelCase when naming objects, functions, and instances. eslint: `camelcase`

```
// bad
const OBJEcttssss = {};
const this_is_my_object = {};
function c() {}

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

- 23.3 Use PascalCase only when naming constructors or classes. eslint: `new-cap`

```
// 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 Do not use trailing or leading underscores. eslint: `no-underscore-dangle`

> 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';

// good, in environments where WeakMaps are available
// see https://kangax.github.io/compat-table/es6/#test-WeakMap
const firstNames = new WeakMap();
firstNames.set(this, 'Panda');
```

- 23.5 Don’t save references to `this`. Use arrow functions or Function#bind.

```
// bad
function foo() {
  const self = this;
  return function () {
    console.log(self);
```

```
  };
}

// bad
function foo() {
  const that = this;
  return function () {
    console.log(that);
  };
}

// good
function foo() {
  return () => {
    console.log(this);
  };
}
```

- 23.6 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 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 Use PascalCase when you export a constructor / class / singleton / function library / bare object.

```
const AirbnbStyleGuide = {
  es6: {
  },
};

export default AirbnbStyleGuide;
```

- 23.9 Acronyms and initialisms should always be all uppercased, 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 = [
  // ...
];
```

- 23.10 You may optionally uppercase a constant only if it (1) is exported, (2) is a `const` (it can not be reassigned), and (3) the programmer can trust it (and its nested properties) to never change.

  > Why? This is an additional tool to assist in situations where the programmer would be unsure if a variable might ever change. UPPERCASE_VARIABLES are letting the programmer know that they can trust the variable (and its properties) not to change.

  - What about all `const` variables? - This is unnecessary, so uppercasing should not be used for constants within a file. It should be used for exported constants however.
  - What about exported objects? - Uppercase at the top level of export (e.g. `EXPORTED_OBJECT.key`) and maintain that all nested properties do not change.

```
// bad
const PRIVATE_VARIABLE = 'should not be unnecessarily uppercased
within a file';

// bad
export const THING_TO_BE_CHANGED = 'should obviously not be
uppercased';

// bad
export let REASSIGNABLE_VARIABLE = 'do not use let with uppercase
variables';

// ---

// allowed but does not supply semantic value
export const apiKey = 'SOMEKEY';

// better in most cases
export const API_KEY = 'SOMEKEY';

// ---

// bad - unnecessarily uppercases key while adding no semantic value
export const MAPPING = {
  KEY: 'value'
};
```

```
// good
export const MAPPING = {
  key: 'value',
};
```

⬆ back to top

Accessors

- 24.1 Accessor functions for properties are not required.
- 24.2 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 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 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];
  }
}
```

⬆ back to top

Events

- 25.1 When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass an object literal (also known as 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
});
```

⬆ back to top

jQuery

- 26.1 Prefix jQuery object variables with a `$`.

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');

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

- 26.2 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 For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`. jsPerf

- 26.4 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();
```

⬆ back to top

ECMAScript 5 Compatibility

- 27.1 Refer to Kangax's ES5 compatibility table.

⬆ back to top

ECMAScript 6+ (ES 2015+) Styles

- 28.1 This is a collection of links to the various ES6+ features.

1. Arrow Functions
2. Classes
3. Object Shorthand
4. Object Concise
5. Object Computed Properties
6. Template Strings
7. Destructuring
8. Default Parameters
9. Rest
10. Array Spreads
11. Let and Const
12. Exponentiation Operator
13. Iterators and Generators
14. Modules

- 28.2 Do not use TC39 proposals that have not reached stage 3.

> Why? They are not finalized, and they are subject to change or to be withdrawn entirely. We want to use JavaScript, and proposals are not JavaScript yet.

⬆ back to top

Standard Library

The Standard Library contains utilities that are functionally broken but remain for legacy reasons.

- 29.1 Use `Number.isNaN` instead of global `isNaN`. eslint: `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 Use `Number.isFinite` instead of global `isFinite`. eslint: `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
```

⬆ back to top

Testing

- 30.1 **Yup.**

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

- 30.2 **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` and `jest` at Airbnb. `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

Performance

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Are JavaScript functions like `map()`, `reduce()`, and `filter()` optimized for traversing arrays?
- Loading...

⬆ back to top

Resources

Learning ES6+

- Latest ECMA spec
- ExploringJS
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features
- JavaScript Roadmap

Read This

- Standard ECMA-262

Tools

- Code Style Linters
  - ESlint - Airbnb Style .eslintrc
  - JSHint - Airbnb Style .jshintrc
- Neutrino Preset - @neutrinojs/airbnb

Other Style Guides

- Google JavaScript Style Guide
- Google JavaScript Style Guide (Old)
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript
- StandardJS

Other Styles

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on GitHub - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

Further Reading

- Understanding JavaScript Closures - Angus Croll
- Basic JavaScript for the impatient programmer - Dr. Axel Rauschmayer
- You Might Not Need jQuery - Zack Bloom & Adam Schwartz
- ES6 Features - Luke Hoban
- Frontend Guidelines - Benjamin De Cock

Books

- JavaScript: The Good Parts - Douglas Crockford
- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch
- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- Human JavaScript - Henrik Joreteg
- Superhero.js - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
- JSBooks - Julien Bouquillon
- Third Party JavaScript - Ben Vinegar and Anton Kovalyov
- Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript - David Herman
- Eloquent JavaScript - Marijn Haverbeke
- You Don’t Know JS: ES6 & Beyond - Kyle Simpson

Blogs

- JavaScript Weekly
- JavaScript, JavaScript...
- Bocoup Weblog
- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- nettuts

Podcasts

- JavaScript Air
- JavaScript Jabber

⬆ back to top

Chat With Us About JavaScript

- Find us on Discord.

License

⬆ back to top

Amendments

We encourage you to fork this guide and change the rules to fit your team’s style guide. Below, you may list some amendments to the style guide. This allows you to periodically update your style guide without having to deal with merge conflicts.

};
