

Types

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

- string
- number
- boolean
- null
- undefined

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

```
bar = 9;
```

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

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

- object
- array
- function

```
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` jscs: `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 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
```

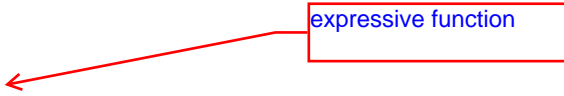
[↑ back to top](#)

Objects

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

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

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

 expressive function

- 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 jscs: 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 Use **property value shorthand**. eslint: object-shorthand jscs: requireEnhancedObjectLiterals

Why? It is shorter to write and descriptive.

```
const lukeSkywalker = 'Luke Skywalker';
```

```

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

```

Good

```

// 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` jscs: `disallowQuotedKeysInObjects`

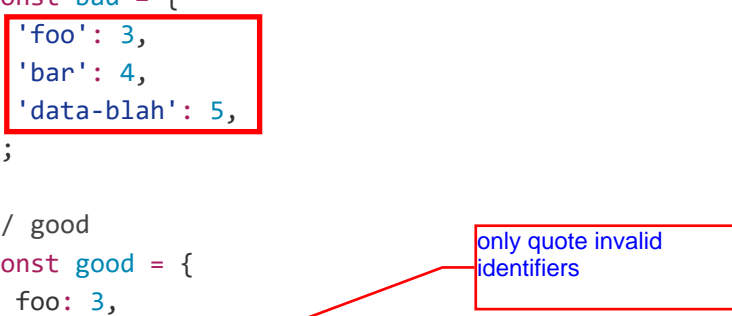
Why? In general we consider it subjectively easier to read. It improves syntax highlighting, and is also more easily optimized by many JS engines.

```

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

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

```



- 3.7 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 Prefer the object **spread operator** over `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` ☹_☹
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++) {
  itemsCopy[i] = items[i];
}

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

- 4.4 To convert an array-like object to an array, use `Array.from`.

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

- 4.5 Use **return statements in array method callbacks.** It's ok to omit the return if the function body consists of a single statement following 8.2. eslint: `array-callback-return`

```
// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});

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

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

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

```

});

// bad
inbox.filter((msg) => {
  const { subject, author } = msg;
  if (subject === 'Mockingbird') {
    return author === 'Harper Lee';
  } else {
    return false;
  }
});

// good
inbox.filter((msg) => {
  const { subject, author } = msg;
  if (subject === 'Mockingbird') {
    return author === 'Harper Lee';
  }

  return false;
});

```

[↑ back to top](#)

⁹ Destructuring

- 5.1 Use **object destructuring** when **accessing and using multiple properties of an object**.
jscs: 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 Use **array destructuring**. jscs: requireArrayDestructuring

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

```
// bad
```

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

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

```
// good
```

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

- 5.3 Use **object destructuring** for **multiple return values**, ~~not~~ **array destructuring**.

jscs: disallowArrayDestructuringReturn

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 };  
}
```

return object destructuring

```
// the caller selects only the data they need
```

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

object destructuring

[↑ back to top](#)

Strings

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

- 6.3 When programmatically building up strings, use **template strings** instead of concatenation. eslint: prefer-templatetemplate-curly-spacing jscs: 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 **Never use eval()** on a string, it opens too many vulnerabilities.
- 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 \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 jscs: requireFunctionDeclarations

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

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

// bad
function foo() {
}

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

seems combine the good parts of function statement and function expression

- 7.2 **Wrap immediately invoked function expressions in parentheses.** eslint: wrap-iife jscs: 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.');
```

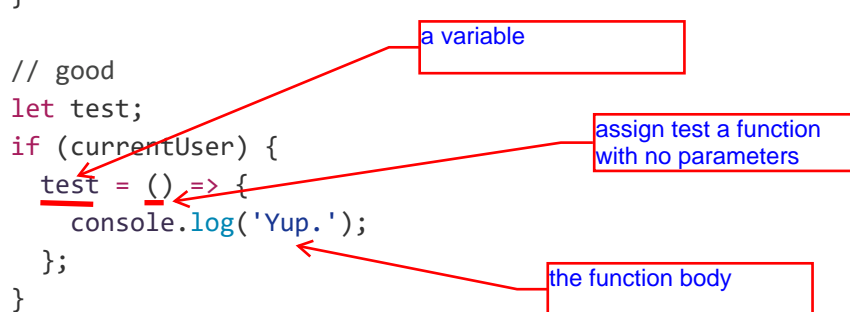
- 7.3 **Never declare a function in a non-function block** (if, while, etc). **Assign the function to a variable instead.** Browsers will allow you to do it, but they all interpret it differently, which is

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. Read [ECMA-262's note on this issue](#).

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

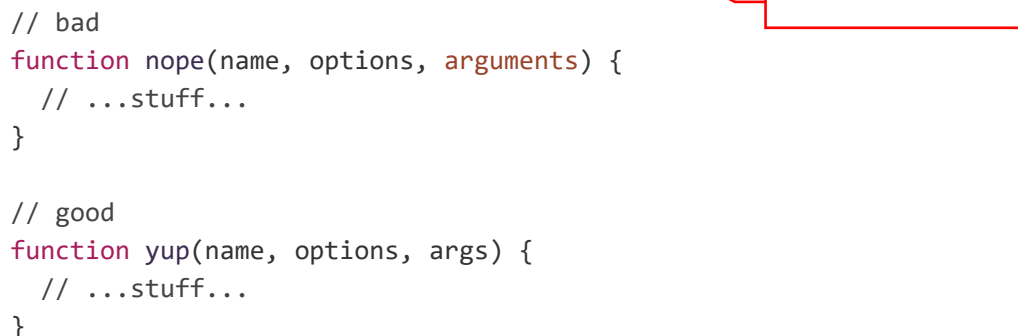
// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
```



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

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

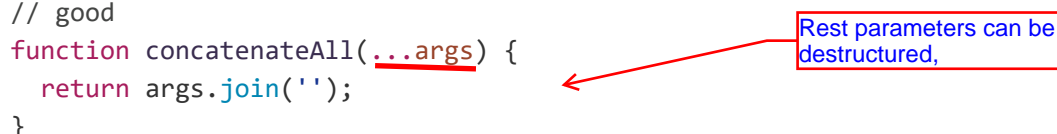


- **7.6** Never use arguments, opt to use rest syntax ... instead. eslint: prefer-rest-params

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.

```
var 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.

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


// still bad
var 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 operator ... 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, 08, 05]));

// good
new Date(...[2016, 08, 05]);

```

- 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.

```

// bad
function foo(bar,
             baz,
             quux) {
    // body
}

// good
function foo(
    bar,
    baz,
    quux,
) {
    // body
}

// bad

```

```

console.log(foo,
  bar,
  baz);

// good
console.log(
  foo,
  bar,
  baz,
);

```

[↑ back to top](#)

↳ Arrow Functions

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

```

// 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 expression, omit the braces and use the implicit return. Otherwise, keep the braces and use a return statement. eslint: `arrow-parens`, `arrow-body-style` jscs: `disallowParenthesesAroundArrowParam`, `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
}));
```

- 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 If your function takes a single argument and doesn't use braces, omit the parentheses. Otherwise, always include parentheses around arguments. eslint: `arrow-parens` jscs: `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
));
```



```
// 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;
};
```

[↑ 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;
  }
}
```

```

        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; }
}
```

[↑ 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`

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

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

- 10.7 Put all imports above non-import statements. eslint: `import/first`

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 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 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';
```

[↑ 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;
```

- 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() {
}

const bar = function * () {
}

const baz = function *() {
}

const quux = function*() {
}

function*foo() {
}

function *foo() {
}

// very bad
function
*
foo() {
}

const wat = function
*
() {
}

// good
function* foo() {
}

const foo = function* () {
}
```

[↑ back to top](#)

Properties

- 12.1 Use dot notation when accessing properties. eslint: [dot-notation](#) jscs: [requireDotNotation](#)


```
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');
```

[↑ back to top](#)

Variables

- 13.1 Always use const 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 declaration per variable. eslint: `one-var` jscs: `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 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 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.

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); // undefined
console.log(b); // 1
console.log(c); // 1

// good
(function example() {
    let a = 1;
    let b = a;
    let c = a;
})();

console.log(a); // undefined
console.log(b); // undefined
console.log(c); // undefined

// the same applies for `const`

```

- 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

let 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

let 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;
```

[↑ back to top](#)

⌘ Hoisting

- 14.1 `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)**. 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: `eql`
- 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) {
  // ...stuff...
}

// good
if (isValid) {
  // ...stuff...
}

// bad
if (name) {
  // ...stuff...
}

// good
if (name !== '') {
  // ...stuff...
}

// bad
if (collection.length) {
  // ...stuff...
}

// good
if (collection.length > 0) {
  // ...stuff...
}
```

- 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).

Why? Lexical declarations are visible in the entire switch block but only get initialized when assigned, which only happens when its case is reached. This causes problems when multiple case clauses attempt to define the same thing.

eslint rules: [no-case-declarations](#).

```
// 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 rules: `no-nested-ternary`.

```

// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;

// better
const maybeNull = value1 > value2 ? 'baz' : null;

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

// best
const maybeNull = value1 > value2 ? 'baz' : null;

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

```

- 15.7 Avoid unneeded ternary statements.

eslint rules: `no-unneeded-ternary`.


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

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

[↑ back to top](#)

Blocks

- 16.1 Use braces with all multi-line blocks.

```
// 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 multi-line blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace. eslint: `brace-style` jscs: `disallowNewlineBeforeBlockStatements`

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

// good
```

```

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

```

[↑ back to top](#)

Comments

- 17.1 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) {

  // ...stuff...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...stuff...

  return element;
}

```

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

```

// bad
const active = true; // is current tab

// good
// is current tab
const active = true;

// bad

```

```

function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}

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

  return type;
}

```

- 17.3 Start all comments with a space to make it easier to read. eslint: [spaced-comment](#)

```

// 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) {

  // ...stuff...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */

```

```
function make(tag) {

  // ...stuff...

  return element;
}
```

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

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

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

- 17.6 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

- 18.1 Use soft tabs set to 2 spaces. eslint: `indent` jscs: `validateIndentation`

```
// bad
function foo() {
  ....const name;
}
```

```
// bad
function bar() {
```

```

•const name;
}

// good
function baz() {
  •const name;
}

```

- 18.2 Place 1 space before the leading brace. eslint: `space-before-blocks` jscs: `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',
});

```

- 18.3 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` jscs: `requireSpaceAfterKeywords`

```

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

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

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

```

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

- 18.4 Set off operators with spaces. eslint: space-infix-ops jscs: requireSpaceBeforeBinaryOperators, requireSpaceAfterBinaryOperators

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

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

- 18.5 End files with a single newline character. eslint: eof-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;↵
```

- 18.6 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').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',
  .attr('width', (radius + margin) * 2).append('svg:g')
  .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin) + ')')
  .call(tron.led);

// good
const leds = stage.selectAll('.led')
  .data(data)
  .enter().append('svg:svg')
  .classed('led', true)
  .attr('width', (radius + margin) * 2)
  .append('svg:g')
  .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin) + ')')
  .call(tron.led);

// good
const leds = stage.selectAll('.led').data(data);

```

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

jscs: `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;

```

- 18.8 Do not pad your blocks with blank lines. eslint: `padded-blocks` jscs: `disallowPaddingNewlinesInBlocks`

```

// bad
function bar() {

  console.log(foo);

}

// also bad
if (baz) {

  console.log(qux);
} else {
  console.log(foo);

}

// good
function bar() {

```



```

    console.log(foo);
}

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

```

- 18.9 Do not add spaces inside parentheses. eslint: `space-in-parens` jscs: `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);
}

```

- 18.10 Do not add spaces inside brackets. eslint: `array-bracket-spacing` jscs: `disallowSpacesInsideArrayBrackets`

```

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

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

```

- 18.11 Add spaces inside curly braces. eslint: `object-curly-spacing` jscs: `requireSpacesInsideObjectBrackets`

```

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

```

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

- 18.12 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` jscs: `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.xzyzy;
```

```
// 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.xzyzy;
```

```
// good
$.ajax({
  method: 'POST',
  url: 'https://airbnb.com/',
  data: { name: 'John' },
})
  .done(() => console.log('Congratulations!'))
  .fail(() => console.log('You have failed this city.'));
```

[↑ back to top](#)

Commas

- 19.1 Leading commas: **Nope**. eslint: `comma-style` jscs: `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',
  };

```

- 19.2 Additional trailing comma: **Yup**. eslint: [comma-dangle](#) jscs: [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](#) 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

- 20.1 Yup. eslint: `semi` jscs: `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  
;(() => {  
  const name = 'Skywalker';  
  return name;  
})();
```



[Read more.](#)

[↑ back to top](#)

⌵ Type Casting & Coercion

- 21.1 Perform type coercion at the beginning of the statement.
- 21.2 Strings:

```
// => this.reviewScore = 9;
```

```
// bad  
const totalScore = this.reviewScore + ''; // invokes this.reviewScore.valueOf()
```

```
// bad  
const totalScore = this.reviewScore.toString(); // isn't guaranteed to return a string
```

```
// good
const totalScore = String(this.reviewScore);
```

- 21.3 Numbers: Use Number for type casting and parseInt always with a radix for parsing strings. eslint: radix

```
const inputValue = '4';

// bad
const val = new Number(inputValue);

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;

// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);

// good
const val = parseInt(inputValue, 10);
```

- 21.4 If for whatever reason you are doing something wild and parseInt is your bottleneck and need to use Bitshift for [performance reasons](#), 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;
```

- 21.5 **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
```

- 21.6 Booleans:

```
const age = 0;

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

// good
const hasAge = Boolean(age);

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

[↑ back to top](#)

³Naming Conventions

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

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

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

- 22.2 Use camelCase when naming objects, functions, and instances.
eslint: `camelcase` jscs: `requireCamelCaseOrUpperCaseIdentifiers`

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

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

- 22.3 Use PascalCase only when naming constructors or classes. eslint: `new-cap` jscs: `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',
});

```

- 22.4 Do not use trailing or leading underscores. eslint: `no-underscore-dangle` jscs: `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';

```

- 22.5 Don't save references to `this`. Use arrow functions or `Function#bind`. jscs: `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);
  };
}

```

- 22.6 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
// ^ supports both insideDirectory.js and insideDirectory/index.js

```

- 22.7 Use camelCase when you export-default a function. Your filename should be identical to your function's name.

```

function makeStyleGuide() {
}

export default makeStyleGuide;

```

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

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

export default AirbnbStyleGuide;
```

- 22.9 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 = [
  // ...
];

// best
import TextMessageContainer from './containers/TextMessageContainer';

// best
const Requests = [
  // ...
];
```

[↑ back to top](#)

Accessors

- 23.1 Accessor functions for properties are not required.
- 23.2 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) {
    // ...
  }
}
```

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

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

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

- 23.4 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

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

```
// 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

- 25.1 Prefix jQuery object variables with a \$. jscs: requireDollarBeforejQueryAssignment

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

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

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

- 25.2 Cache jQuery lookups.

```

// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}

```

- 25.3 For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`.[jsPerf](#)
- 25.4 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

- 26.1 Refer to [Kangax's ES5 compatibility table](#).

[↑ back to top](#)

↳ ECMAScript 6+ (ES 2015+) Styles

- 27.1 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. Iterators and Generators
13. Modules

- 27.2 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](#)

↳ Testing

- 28.1 Yup.

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

- 28.2 No, but seriously:
 - Whichever testing framework you use, you should be writing tests!
 - Strive to write many small pure functions, and minimize where mutations occur.
 - Be cautious about stubs and mocks - they can make your tests more brittle.
 - We primarily use `mocha` 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

- Draft ECMA 2015 (ES6) Spec
- ExploringJS
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features

Read This

- Standard ECMA-262

Tools

- Code Style Linters
 - ESLint - Airbnb Style `.eslintrc`
 - JSHint - Airbnb Style `.jshintrc`
 - JSCS - Airbnb Style Preset

Other Style Guides

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

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](#)
- [Dustin Diaz](#)
- [nettuts](#)

Podcasts

- [JavaScript Air](#)
- [JavaScript Jabber](#)

[↑ back to top](#)

⁹ In the Wild

This is a list of organizations that are using this style guide. Send us a pull request and we'll add you to the list.

- **4Catalyzer**: [4Catalyzer/javascript](#)
- **Aan Zee**: [AanZee/javascript](#)
- **Adult Swim**: [adult-swim/javascript](#)
- **Airbnb**: [airbnb/javascript](#)
- **Apartment**: [apartment/javascript](#)
- **Ascribe**: [ascribe/javascript](#)
- **Avalara**: [avalara/javascript](#)
- **Avant**: [avantcredit/javascript](#)
- **Billabong**: [billabong/javascript](#)
- **Bisk**: [bisk/javascript](#)
- **Blendle**: [blendle/javascript](#)
- **Brainshark**: [brainshark/javascript](#)
- **Chartboost**: [ChartBoost/javascript-style-guide](#)
- **ComparaOnline**: [comparaonline/javascript](#)
- **Compass Learning**: [compasslearning/javascript-style-guide](#)
- **DailyMotion**: [dailymotion/javascript](#)
- **DoSomething**: [DoSomething/eslint-config](#)
- **Digitpaint**: [digitpaint/javascript](#)
- **Ecosia**: [ecosia/javascript](#)
- **Evernote**: [evernote/javascript-style-guide](#)
- **Evolution Gaming**: [evolution-gaming/javascript](#)
- **EvozonJs**: [evozonjs/javascript](#)
- **ExactTarget**: [ExactTarget/javascript](#)
- **Expensify**: [Expensify/Style-Guide](#)
- **Flexberry**: [Flexberry/javascript-style-guide](#)
- **Gawker Media**: [gawkermedia/javascript](#)
- **General Electric**: [GeneralElectric/javascript](#)
- **GoodData**: [gooddata/gdc-js-style](#)
- **Grooveshark**: [grooveshark/javascript](#)
- **How About We**: [howaboutwe/javascript](#)
- **Huballin**: [huballin/javascript](#)
- **HubSpot**: [HubSpot/javascript](#)
- **Hyper**: [hyperoslo/javascript-playbook](#)
- **InfoJobs**: [InfoJobs/JavaScript-Style-Guide](#)
- **Intent Media**: [intentmedia/javascript](#)
- **Jam3**: [Jam3/Javascript-Code-Conventions](#)
- **JeopardyBot**: [kesne/jeopardy-bot](#)

- **JSSolutions:** [JSSolutions/javascript](#)
- **KickorStick:** [kickorstick/javascript](#)
- **Kinetica Solutions:** [kinetica/javascript](#)
- **Lonely Planet:** [lonelyplanet/javascript](#)
- **M2GEN:** [M2GEN/javascript](#)
- **Mighty Spring:** [mightyspring/javascript](#)
- **MinnPost:** [MinnPost/javascript](#)
- **MitocGroup:** [MitocGroup/javascript](#)
- **ModCloth:** [modcloth/javascript](#)
- **Money Advice Service:** [moneyadviceservice/javascript](#)
- **Muber:** [muber/javascript](#)
- **National Geographic:** [natgeo/javascript](#)
- **National Park Service:** [nationalparkservice/javascript](#)
- **Nimbl3:** [nimbl3/javascript](#)
- **Orion Health:** [orionhealth/javascript](#)
- **OutBoxSoft:** [OutBoxSoft/javascript](#)
- **Peerby:** [Peerby/javascript](#)
- **Razorfish:** [razorfish/javascript-style-guide](#)
- **reddit:** [reddit/styleguide/javascript](#)
- **React:** [/facebook/react/blob/master/CONTRIBUTING.md#style-guide](#)
- **REI:** [reidev/js-style-guide](#)
- **Ripple:** [ripple/javascript-style-guide](#)
- **SeekingAlpha:** [seekingalpha/javascript-style-guide](#)
- **Shutterfly:** [shutterfly/javascript](#)
- **Springload:** [springload/javascript](#)
- **StratoDem Analytics:** [stratodem/javascript](#)
- **SteelKiwi Development:** [steelkiwi/javascript](#)
- **StudentSphere:** [studentsphere/javascript](#)
- **SysGarage:** [sysgarage/javascript-style-guide](#)
- **Syzygy Warsaw:** [syzygypl/javascript](#)
- **Target:** [target/javascript](#)
- **TheLadders:** [TheLadders/javascript](#)
- **The Nerdery:** [thenerdery/javascript-standards](#)
- **T4R Technology:** [T4R-Technology/javascript](#)
- **VoxFeed:** [VoxFeed/javascript-style-guide](#)
- **WeBox Studio:** [weboxstudio/javascript](#)
- **Weggo:** [Weggo/javascript](#)
- **Zillow:** [zillow/javascript](#)
- **ZocDoc:** [ZocDoc/javascript](#)

[↑ back to top](#)

Translation

This style guide is also available in other languages:

-  **Brazilian Portuguese:** [armoucar/javascript-style-guide](#)
-  **Bulgarian:** [borislavvv/javascript](#)
-  **Catalan:** [fpmweb/javascript-style-guide](#)
-  **Chinese (Simplified):** [sivan/javascript-style-guide](#)
-  **Chinese (Traditional):** [jigsawye/javascript](#)
-  **French:** [nmussy/javascript-style-guide](#)
-  **German:** [timofurrer/javascript-style-guide](#)
-  **Italian:** [sinkswim/javascript-style-guide](#)
-  **Japanese:** [mitsuruog/javascript-style-guide](#)
-  **Korean:** [tipjs/javascript-style-guide](#)
-  **Polish:** [mjurczyk/javascript](#)
-  **Russian:** [uprock/javascript](#)
-  **Spanish:** [paolocarrasco/javascript-style-guide](#)
-  **Thai:** [lvarayut/javascript-style-guide](#)
-  **Vietnam:** [giangpii/javascript-style-guide](#)

The JavaScript Style Guide Guide

- [Reference](#)

Chat With Us About JavaScript

- Find us on [gitter](#).

Contributors

- [View Contributors](#)

License

(The MIT License)

Copyright (c) 2014-2016 Airbnb

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

[↑ 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.

`};`
