Airbnb JS Style Guide

Full Stack Web Development

Naming Conventions

Comments

Whitespace

Commas

Semicolons

References

Variables

Arrays

Strings

Objects

Properties

Destructuring

Blocks

Control Statements

Functions

Arrow Functions

Modules

References

Variables

Arrays

References

References

- 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;
```

References

- 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;
}
```

- 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.

Variables

Variables

- 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();
```

- 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';
```

Variables

- 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;
```

Variables

- 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;
}
```

Variables

- 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`
```

Variables

- 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;
```

Variables

- 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 =
  superLongLongLongLongLongLongLongLongFunctionName();

// bad
const foo
  = 'superLongLongLongLongLongLongLongLongString';

// good
const foo = (
  superLongLongLongLongLongLongLongLongFunctionName()
);

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

Variables

- 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

var some_unused_var = 42;

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

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

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

// good

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

var x = 1;
var 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.
var { type, ...coords } = data;
// 'coords' is now the 'data' object without its 'type' property.
```

Variables

- 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

var some_unused_var = 42;

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

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

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

// good

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

var x = 1;
var 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.
var { type, ...coords } = data;
// 'coords' is now the 'data' object without its 'type' property.
```

Arrays

Arrays

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

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

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

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

```
const someStack = [];

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

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

Arrays

- 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];
```

- 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];
```

Arrays

- 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);
```

- 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);
```

Arrays

- 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. 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;
});
```

Arrays

- 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,
];
```

Airbnb JS Style Guide

Full Stack Web Development