

# Airbnb JavaScript Style Guide() {

## Tabla de Contenido

---

1. [Tipos](#)
2. [Referencias](#)
3. [Objetos](#)
4. [Arreglos](#)
5. [Destructuring](#)
6. [Cadenas de Texto](#)
7. [Funciones](#)
8. [Notación de Funciones de Flecha](#)
9. [Clases y Constructores](#)
10. [Módulos](#)
11. [Iteradores y Generadores](#)
12. [Propiedades](#)
13. [Variables](#)
14. [Hoisting](#)
15. [Expresiones de comparación e igualdad](#)
16. [Bloques](#)
17. [Comentarios](#)
18. [Espacios en blanco](#)
19. [Comas](#)
20. [Puntos y Comas](#)
21. [Casting de Tipos y Coerción](#)
22. [Convenciones de nomenclatura](#)
23. [Funciones de Acceso](#)
24. [Eventos](#)
25. [jQuery](#)
26. [Compatibilidad con EcmaScript 5](#)
27. [Estilos de EcmaScript6+ \(ES2015+\)](#)
28. [Pruebas](#)
29. [Desempeño](#)
30. [Recursos](#)

- 31. [En la cancha](#)
- 32. [Traducciones](#)
- 33. [La guía de la Guía de Estilos JavaScript](#)
- 34. [Charla con nosotros sobre Javascript](#)
- 35. [Colaboradores](#)
- 36. [Licencia](#)

## Tipos

---

- **Primitivos:** Cuando accedes a un tipo primitivo, manejas directamente su valor

- string
- number
- boolean
- null
- undefined

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

- **Complejo:** Cuando accedes a un tipo complejo, manejas la referencia a su valor.

- object
- array
- function

- `const foo = [1, 2];`
- `const bar = foo;`
- 
- `bar[0] = 9;`
- 
- `console.log(foo[0], bar[0]); // => 9, 9`

[\[↑ regresar a la Tabla de Contenido\]](#)

## Referencias

---

- Usa `const` para todas tus referencias; evita usar `var`.

¿Por qué? Esto asegura que no reasignes tus referencias, lo que puede llevar a bugs y dificultad para comprender el código.

```
// mal
var a = 1;
```

```
var b = 2;

// bien
const a = 1;
const b = 2;
```

- Si vas a reasignar referencias, usa `let` en vez de `var`.

¿Por qué? El bloque `let` es de alcance a nivel de bloque a diferencia del alcance a nivel de función de `var`.

```
// mal
var count = 1;
if (true) {
  count += 1;
}

// bien, usa el let
let count = 1;
if (true) {
  count += 1;
}
```

- Nota que tanto `let` como `const` tienen alcance a nivel de bloque.

```
// const y let solo existen en los bloques donde
// estan definidos
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

## Objetos

- Usa la sintaxis literal para la creación de un objeto.

```
• // mal
• const item = new Object();
•
• // bien
• const item = {};
```

- No uses [palabras reservadas](#) para nombres de propiedades. No funciona en IE8 [Más información](#). No hay problema de usarlo en módulos de ES6 y en código de servidor.

```
• // mal
• const superman = {
•   default: { clark: 'kent' },
•   private: true
• };
```

- 
- `// bien`
- `const superman = {`
- `defaults: { clark: 'kent' },`
- `hidden: true`
- `};`

- Usa sinónimos legibles en lugar de palabras reservadas.

- `// mal`
- `const superman = {`
- `class: 'alien'`
- `};`
- 
- `// mal`
- `const superman = {`
- `klass: 'alien'`
- `};`
- 
- `// bien`
- `const superman = {`
- `type: 'alien'`
- `};`

[\[↑ regresar a la Tabla de Contenido\]](#)

## Arreglos

- Usa la sintaxis literal para la creación de arreglos

- `// mal`
- `const items = new Array();`
- 
- `// bien`
- `const items = [];`

- Usa `Array#push`, en vez de asignación directa, para agregar elementos a un arreglo.

- `const someStack = [];`
- 
- `// mal`
- `someStack[someStack.length] = 'abracadabra';`
- 
- `// bien`
- `someStack.push('abracadabra');`

- Usa `spread de arrays` para copiar arreglos.

- `const len = items.length;`
- `const itemsCopy = [];`
- `let i;`
-

- `// mal`
- `for (i = 0; i < len; i++) {`
- `itemsCopy[i] = items[i];`
- `}`
- 
- `// bien`
- `const itemsCopy = [...items];`

- Para convertir un objeto "array-like" (similar a un arreglo) a un arreglo, usa `Array#from`.

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

[\[↑ regresar a la Tabla de Contenido\]](#)

## Destructuring

- Usa object destructuring cuando accedas y uses múltiples propiedades de un objeto.

¿Por qué? Destructuring te ahorra crear referencias temporales para esas propiedades.

```
// mal
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// bien
function getFullName(user) {
  const { firstName, lastName } = user;
  return `${firstName} ${lastName}`;
}

// mejor
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

- Usa array destructuring.

- `const arr = [1, 2, 3, 4];`
- 
- `// mal`
- `const first = arr[0];`
- `const second = arr[1];`
- 
- `// bien`
- `const [first, second] = arr;`

- Usa object destructuring para múltiple valores de retorno, no array destructuring.

¿Por qué? Puedes agregar nuevas propiedades en el tiempo o cambiar el orden de las cosas sin afectar la forma en que se llama.

```
// mal
function processInput(input) {
  // then a miracle occurs
  return [left, right, top, bottom];
}

// el que llama necesita pensar en el orden de la data de retorno
const [left, __, top] = processInput(input);

// bien
function processInput(input) {
  // then a miracle occurs
  return { left, right, top, bottom };
}

// el que llama elige solo la data que necesita
const { left, top } = processInput(input);
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Cadenas de Texto

- Usa comillas simples ' ' para las cadenas de texto

```
// mal
const name = "Bob Parr";

// bien
const name = 'Bob Parr';
```

- Las cadenas de texto con una longitud mayor a 100 caracteres deben ser escritas en múltiples líneas usando concatenación.

**Nota:** Cuando se usa sin criterio, las cadenas de texto largas pueden impactar en el desempeño. [jsPerf](#) & [Discusión](#)

```
// mal
var 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.';

// bien
var 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.';

// bien
var 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.';
```

- Cuando se crean cadenas de texto de forma programática, usa template strings (cadena de plantillas) en vez de concatenación.

¿Por qué? Los template strings te dan mayor legibilidad, sintaxis concisa con nuevas líneas apropiadas y capacidades de interpolación.

```
// mal
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// mal
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// bien
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

- Nunca uses `eval()` en una cadena de texto, abre una caja de Pandora de vulnerabilidades.

[\[↑ regresar a la Tabla de Contenido\]](#)

## Funciones

---

- Usa declaración de función en vez de expresiones de función.

¿Por qué? Las declaraciones de función son nombradas, por lo que son más sencillas de identificar en las pilas de llamadas. Además todo el contenido de una declaración de función es *hoisted*, mientras que solo la referencia de una expresión de función es *hoisted*. Esta regla hace posible que siempre se usen [Arrow Functions](#) en vez de las funciones de expresión.

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

// bien
function foo() {
}
```

- Nunca declares una función en un bloque que no sea de función (if, while, etc). En vez de ello, asigna la función a una variable. Los navegadores te permitirán hacerlo pero todos ellos lo interpretarán de modo diferente, lo que es lamentable.

**Nota:** ECMA-262 define un bloque como una lista de sentencias. Una declaración de función no es una sentencia. [Lee la nota de ECMA-262 sobre este inconveniente.](#)

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

```
}
}

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

```
};
}
```

- Nunca nombres a un parámetro como `arguments`, esto tendrá precedencia sobre el objeto `arguments` que es brindado en cada ámbito de función.

```
• // mal
• function nope(name, options, arguments) {
•   // ...algo...
• }
•
• // bien
• function yup(name, options, args) {
•   // ...algo...
• }
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Notación de Funciones de Flecha

- Cuando debas usar funciones anónimas (como cuando pasas un callback inline), usa la notación de funciones de flecha.

¿Por qué? Crea una versión de la función que ejecuta en el contexto de `this`, lo que usualmente es lo que deseas, además que tiene una sintaxis más concisa.

¿Por qué no? Si tienes una función complicada, debes mover esa lógica fuera de su expresión de función nombrada.

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

// bien
[1, 2, 3].map((x) => {
```



```
const y = x + 1;
return x * y;
});
```

- Si el cuerpo de la función consiste en una sola sentencia retornando una **expresión** sin efectos colaterales, omite las llaves y usa el retorno implícito. De otro modo, mantén las llaves y usa una sentencia de retorno.

¿Por qué? Un edulcorante sintáctico. Se lee bien cuando múltiples funciones están encadenadas entre sí.

```
// mal
[1, 2, 3].map(number => {
  const nextNumber = number + 1;
  `A string containing the ${nextNumber}.`;
});

// bien
[1, 2, 3].map(number => `A string containing the ${number}.`);

// bien
[1, 2, 3].map((number) => {
  const nextNumber = number + 1;
  return `A string containing the ${nextNumber}.`;
});

// bien
[1, 2, 3].map((number, index) => ({
  [index]: number,
}));

// Sin efectos colaterales para retorno implícito
function foo(callback) {
  const val = callback();
  if (val === true) {
    // Do something if callback returns true
  }
}

let bool = false;

// mal
foo(() => bool = true);

// bien
foo(() => {
  bool = true;
});
```

- En caso que la expresión se expanda en varias líneas, envuélvela en paréntesis para una mejor legibilidad.

¿Por qué? Se observa claramente dónde empieza y termina la función.

```
// mal
```

```

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

// bien
['get', 'post', 'put'].map(httpMethod => (
  Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod,
  )
));

```

- Si tu función tiene un solo argumento y no usa llaves, omite los paréntesis. De otra forma, siempre incluye paréntesis alrededor de los argumentos por claridad y consistencia. Nota: es también aceptable siempre usar paréntesis, en cuyo caso usa la [opción de "always"](#) para eslint o no incluyas [disallowParenthesesAroundArrowParam](#) para jscs. ¿Por qué? Menos basura visual.

```

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

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

// bien
[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!`
));

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

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

```

- Evita confundir la sintaxis de función de flecha (=>) con los operadores de comparación (<=, >=).

```

// mal
const itemHeight = item => item.height > 256 ? item.largeSize :
item.smallSize;

// mal
const itemHeight = (item) => item.height > 256 ? item.largeSize :
item.smallSize;

```

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

[↑ regresar a la Tabla de Contenido](#)

## Clases y Constructores

- Siempre usa `class`. Evita manipular `prototype` directamente.  
¿Por qué? La sintaxis `class` es más concisa y fácil con la cual lidiar.

```
// mal
function Queue(contents = []) {
  this._queue = [...contents];
}
Queue.prototype.pop = function () {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}

// bien
class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

- Métodos pueden retornar `this` para ayudar con el encadenamiento de métodos (*chaining*).

```
// mal
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
```

- 
- `// bien`
- `class Jedi {`
- `jump() {`
- `this.jumping = true;`
- `return this;`
- `}`
- 
- `setHeight(height) {`
- `this.height = height;`
- `return this;`
- `}`
- `}`
- 
- `const luke = new Jedi();`
- 
- `luke.jump()`
- `.setHeight(20);`

- Está bien escribir un método `toString()` personalizado, solo asegúrate que funcione correctamente y no cause efectos colaterales.

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

[\[↑ regresar a la Tabla de Contenido\]](#)

## Módulos

- Siempre usa módulos (`import/export`) antes que un sistema de módulos no estándar. Siempre puedes transpilar a tu sistema de módulos preferido.  
¿Por qué? Los módulos son el futuro, comencemos a usar el futuro en el presente.

```
// mal
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;
```

```
// mejor
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- No uses imports con comodines (asterisco).

¿Por qué? Esto te asegura de tener una única exportación por defecto.

```
// mal
import * as AirbnbStyleGuide from './AirbnbStyleGuide';

// bien
import AirbnbStyleGuide from './AirbnbStyleGuide';
```

- Y no exportes directamente lo que traigas de un import.

¿Por qué? A pesar que hacer las cosas en una línea es conciso, tener un modo claro de importar y un modo claro de exportar, hace las cosas consistentes.

```
// mal
// filename es6.js
export { es6 as default } from './AirbnbStyleGuide';

// bien
// filename es6.js
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- Solo importa de una ruta en un mismo lugar.

¿Por qué? Tener varias líneas que importan de una misma ruta hace al código difícil de mantener.

```
// mal
import foo from 'foo';
// ... some other imports ... //
import { named1, named2 } from 'foo';

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

// bien
import foo, {
  named1,
  named2,
} from 'foo';
```

- No exportes las asociaciones (bindings) mutables.

¿Por qué? La mutación debe ser evitada en general, pero en particular cuando se exportan asociaciones (bindings) mutables. Mientras esta técnica puede ser necesaria para algunos casos especiales, en general solo referencias constantes deben ser exportadas.

```
// mal
let foo = 3;
export { foo };

// bien
const foo = 3;
export { foo };
```

- En módulos con una única exportación, prefiere la exportación por defecto sobre la exportación nombrada.

¿Por qué? Para forzar a que más archivos solo exporten una sola cosa, lo que es mejor para la legibilidad y mantenibilidad.

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

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

- Pon todos los imports encima de las sentencias de no importación.  
¿Por qué? Desde que los imports son elevados (hoisted), mantenerlos en el inicio previene comportamientos sorpresivos.

```
// mal
import foo from 'foo';
foo.init();

import bar from 'bar';

// bien
import foo from 'foo';
import bar from 'bar';

foo.init();
```

- Imports de multi-línea deben ser indentados como los arreglos multi-línea y literales de objeto.

¿Por qué? Las llaves deben seguir las mismas reglas de indentación como en otros bloques de llaves en la guía de estilos, así como las comas finales.

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

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

- No permitas la sintaxis de carga de Webpack en las sentencias de importación de módulos.

¿Por qué? Debido a que usar la sintaxis de Webpack en los imports acopla el código a un ensamblador de módulos. Prefiere usar aquella sintaxis de carga en el archivo de `webpack.config.js`.

```
// mal
import fooSass from 'css!sass!foo.scss';
import barCss from 'style!css!bar.css';

// bien
import fooSass from 'foo.scss';
import barCss from 'bar.css';
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Propiedades

---

- Usa la notación de punto `.` cuando accedas a las propiedades.

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

// mal
const isJedi = luke['jedi'];

// bien
const isJedi = luke.jedi;
```

- Usa la notación subscript `[]` cuando accedas a las propiedades con una variable.

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

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

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

[\[↑ regresar a la Tabla de Contenido\]](#)

## Variables

---

- Siempre usa `const` para declarar constantes o `let` para declarar variables. No hacerlo resultará en variables globales. Debemos evitar contaminar el

espacio global (global namespace). El [Capitán Planeta](#) nos advirtió de eso.

```
• // mal
• superPower = new SuperPower();
•
• // bien
• const superPower = new SuperPower();
•
• o
•
• // bien
• let aPower;
• aPower = new SuperPower(); // esto puede cambiar a otro poder
• posteriormente
```

- Usa una declaración `const` o `let` por variable.  
¿Por qué? Es más fácil agregar nuevas declaraciones de variables de este modo, y no tendrás que preocuparte por reemplazar ; por , o introducir diffs de sólo puntuación .

```
// mal
const items = getItems(),
      goSportsTeam = true,
      dragonball = 'z';

// mal
// (compara con lo de arriba y encuentra el error)
const items = getItems(),
      goSportsTeam = true;
      dragonball = 'z';

// bien
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- Agrupa tus `consts` y luego agrupa tus `lets`.  
¿Por qué? Esto es útil cuando necesitas asignar una variable luego dependiendo de una de las variables asignadas previamente.

```
// mal
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// mal
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// bien
const goSportsTeam = true;
const items = getItems();
let dragonball;
```



```
let i;  
let length;
```

- Asigna las variables cuando las necesites, pero ponlas en un lugar razonable.

¿Por qué? let y const están a nivel de bloque, no a nivel de función.

```
// mal - llamada a funcion innecesaria
```

```
function checkName(hasName) {  
  const name = getName();  
  
  if (hasName === 'test') {  
    return false;  
  }  
  
  if (name === 'test') {  
    this.setName('');  
    return false;  
  }  
  
  return name;  
}
```

```
// bien
```

```
function checkName(hasName) {  
  if (hasName === 'test') {  
    return false;  
  }  
  
  const name = getName();  
  
  if (name === 'test') {  
    this.setName('');  
    return false;  
  }  
  
  return name;  
}
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Hoisting

- Las declaraciones de variables son movidas a la parte superior de su ámbito, sin embargo su asignación no.

```
• // sabemos que esto no funcionara (asumiendo  
• // que no hay una variable global notDefined)  
• function example() {  
•   console.log(notDefined); // => lanza un ReferenceError  
• }  
•  
• // crear una declaracion de variable luego  
• // que referencias a la variable funcionara
```

- // por el hoisting. Nota: A la asignacion
- // del valor `true` no se le aplico hoisting.
- `function example() {`
- `console.log(declaredButNotAssigned); // => undefined`
- `var declaredButNotAssigned = true;`
- `}`
- 
- // El interprete lleva la declaracion de la
- // variable a la parte superior de la funcion.
- // Eso significa que nuestro ejemplo
- // podria ser reescrito como:
- `function example() {`
- `var declaredButNotAssigned;`
- `console.log(declaredButNotAssigned); // => undefined`
- `declaredButNotAssigned = true;`
- `}`

- Expresiones de función anónimas hacen hoisting de su nombre de variable, pero no de la asignación de la función.

- `function example() {`
- `console.log(anonymous); // => undefined`
- 
- `anonymous(); // => TypeError anonymous is not a function`
- 
- `var anonymous = function() {`
- `console.log('anonymous function expression');`
- `};`
- `}`

- Expresiones de función nombradas hacen hoisting de su nombre de variable, pero no del nombre de la función ni del contenido de la función.

- `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');`
- `};`
- `}`
- 
- // lo mismo es cierto cuando el nombre
- // de la funcion es igual al nombre de
- // la variable.
- `function example() {`
- `console.log(named); // => undefined`
- 
- `named(); // => TypeError named is not a function`
- 
- `var named = function named() {`

```

• console.log('named');
• }
• }

```

- Las declaraciones de función hacen hoist de su nombre y del contenido de la función.

```

• function example() {
•   superPower(); // => Flying
•
•   function superPower() {
•     console.log('Flying');
•   }
• }

```

- Para más información lee [JavaScript Scoping & Hoisting](#) por [Ben Cherry](#)

[\[↑ regresar a la Tabla de Contenido\]](#)

## Expresiones de comparación e igualdad

- Usa `===` y `!==` en vez de `==` y `!=` respectivamente.
- Las expresiones condicionales son evaluadas usando coerción con el método `ToBoolean` y siempre obedecen a estas reglas sencillas:
  - **Objects** son evaluados como **true** (se considera así al objeto vacío `{}` y arreglos sin contenido `[]`)
  - **Undefined** es evaluado como **false**
  - **Null** es evaluado como **false**
  - **Booleans** son evaluados como **el valor del booleano**
  - **Numbers** son evaluados como **false** si su valor es **+0**, **-0**, o **NaN**, de otro modo **true**
  - **Strings** son evaluados como **false** si es una cadena de texto vacía `''`, de otro modo son **true**

```

• if ([0] && []) {
•   // true
•   // un arreglo es un objeto (incluso uno vacío), los objetos son
  evaluados como true
• }

```

- Usa atajos.

```

• // mal
• if (name !== '') {
•   // ...cosas...
• }
•
• // bien
• if (name) {
•   // ...cosas...

```

```

• }
•
• // mal
• if (collection.length > 0) {
•   // ...cosas...
• }
•
• // bien
• if (collection.length) {
•   // ...cosas...
• }

```

- Para más información revisa [Truth Equality and JavaScript](#) por Angus Croll
- Usa llaves para crear bloques en cláusulas case y default que contengan declaraciones léxicas (e.g. let, const, function y class).  
¿Por qué? La declaración léxica es visible en todo el bloque switch pero solo se inicializa al ser asignado, lo que solo ocurre cuando el bloque case donde es declarado es alcanzado. Esto causa problemas cuando múltiples bloques case intentan definir la misma variable.

```

// mal
switch (foo) {
  case 1:
    let x = 1;
    break;
  case 2:
    const y = 2;
    break;
  case 3:
    function f() {}
    break;
  default:
    class C {}
}

// bien
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 {}
  }
}

```

```
}  
}
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Bloques

---

- Usa llaves con todos los bloques de múltiples líneas.

```
• // mal  
• if (test)  
•   return false;  
•  
• // bien  
• if (test) return false;  
•  
• // bien  
• if (test) {  
•   return false;  
• }  
•  
• // mal  
• function() { return false; }  
•  
• // bien  
• function() {  
•   return false;  
• }
```

- Si estás usando bloques de muchas líneas con if y else, pon el else en la misma línea que el if.

```
• // mal  
• if (test) {  
•   thing1();  
•   thing2();  
• }  
• else {  
•   thing3();  
• }  
•  
• // bien  
• if (test) {  
•   thing1();  
•   thing2();  
• } else {  
•   thing3();  
• }
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Comentarios

---

- Usa `/** ... */` para comentarios de múltiples líneas. Incluye una descripción, especificación de tipos y valores para todos los parámetros y valores de retorno.

```

• // mal
• // make() returns a new element
• // based on the passed in tag name
• //
• // @param {String} tag
• // @return {Element} element
• function make(tag) {
•
•     // ...stuff...
•
•     return element;
• }
•
• // bien
• /**
•  * make() returns a new element
•  * based on the passed in tag name
•  *
•  * @param {String} tag
•  * @return {Element} element
•  */
• function make(tag) {
•
•     // ...stuff...
•
•     return element;
• }

```

- Usa `//` para comentarios de una sola línea. Ubica los comentarios de una sola línea encima de la sentencia comentada. Deja una línea en blanco antes del comentario, a menos que sea la primera línea de un bloque.

```

• // mal
• const active = true; // is current tab
•
• // bien
• // is current tab
• const active = true;
•
• // mal
• function getType() {
•     console.log('fetching type...');
•     // set the default type to 'no type'
•     const type = this._type || 'no type';
•
•     return type;
• }
•
• // bien
• function getType() {
•     console.log('fetching type...');

```

- `// set the default type to 'no type'`
- `const type = this._type || 'no type';`
- 
- `return type;`
- `}`

- Agregando a tus comentarios los prefijos `FIXME` o `TODO`, ayudará a otros desarrolladores a entender rápidamente si estás apuntando a un problema que precisa ser revisado o si estás sugiriendo una solución al problema que debería ser implementado. Estos son diferentes a comentarios regulares en el sentido que requieren alguna acción. Las acciones son `FIXME -- necesito resolver esto` o `TODO -- necesita implementarse`.

- Usa `// FIXME:` para anotar problemas.
- `class Calculator extends Abacus {`
- `constructor() {`
- `super();`
- 
- `// FIXME: shouldn't use a global here`
- `total = 0;`
- `}`
- `}`

- Usa `// TODO:` para anotar soluciones a los problemas.
- `class Calculator extends Abacus {`
- `constructor() {`
- `super();`
- 
- `// TODO: total should be configurable by an options param`
- `this.total = 0;`
- `}`
- `}`

[\[↑ regresar a la Tabla de Contenido\]](#)

## Espacios en blanco

- Usa indentaciones blandas (sin TAB) establecidas en dos espacios.

- `// mal`
- `function foo() {`
- `....const name;`
- `}`
- 
- `// mal`
- `function bar() {`
- `.const name;`
- `}`
- 
- `// bien`
- `function baz() {`

- `..const name;`  
`}`

- Deja un espacio antes de la llave de apertura.

- `// mal`
- `function test(){`
- `console.log('test');`
- `}`
- `// bien`
- `function test() {`
- `console.log('test');`
- `}`
- `// mal`
- `dog.set('attr',{`
- `age: '1 year',`
- `breed: 'Bernese Mountain Dog'`
- `});`
- `// bien`
- `dog.set('attr', {`
- `age: '1 year',`
- `breed: 'Bernese Mountain Dog'`
- `});`

- Deja un espacio antes del paréntesis de apertura en las sentencias de control (if, while, etc.). No dejes espacios antes de la lista de argumentos en las invocaciones y declaraciones de funciones.

- `// mal`
- `if(isJedi) {`
- `fight ();`
- `}`
- `// bien`
- `if (isJedi) {`
- `fight();`
- `}`
- `// mal`
- `function fight () {`
- `console.log ('Swoosh!');`
- `}`
- `// bien`
- `function fight() {`
- `console.log('Swoosh!');`
- `}`

- Separa a los operadores con espacios.

- `// mal`
- `const x=y+5;`



- 
- `// bien`  
`const x = y + 5;`

- Deja una línea en blanco al final del archivo.

- `// mal`
- `(function(global) {`
- `// ...algo...`
- `})(this);`
- `// mal`
- `(function(global) {`
- `// ...algo...`
- `})(this);d`
- `d`
- `// bien`
- `(function(global) {`
- `// ...algo...`
- `})(this);d`

- Usa indentación cuando uses métodos largos con 'chaining' (más de dos métodos encadenados). Emplea un punto adelante en cada nueva línea, lo que enfatiza que es un método llamado no una nueva sentencia.

- `// mal`
- `$('#items').find('.selected').highlight().end().find('.open').updateCount();`
- 
- `// mal`
- `$('#items').`
- `find('.selected').`
- `highlight().`
- `end().`
- `find('.open').`
- `updateCount();`
- 
- `// bien`
- `$('#items')`
- `.find('.selected')`
- `.highlight()`
- `.end()`
- `.find('.open')`
- `.updateCount();`
- 
- `// mal`
- `const leds =`
- `stage.selectAll('.led').data(data).enter().append('svg:svg').class('led', true)`
- `.attr('width', (radius + margin) * 2).append('svg:g')`
- `.attr('transform', 'translate(' + (radius + margin) + ', ' +`
- `(radius + margin) + ')')`
- `.call(tron.led);`
- 
- `// bien`

```

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

```

- Deja una línea en blanco luego de los bloques y antes de la siguiente sentencia.

```

• // mal
• if (foo) {
•   return bar;
• }
• return baz;
•
• // bien
• if (foo) {
•   return bar;
• }
•
• return baz;
•
• // mal
• const obj = {
•   foo() {
•   },
•   bar() {
•   }
• };
• return obj;
•
• // bien
• const obj = {
•   foo() {
•   },
•
•   bar() {
•   }
• };
•
• return obj;
•
• // mal
• const arr = [
•   function foo() {
•   },
•   function bar() {
•   },
• ];
• return arr;
•

```

```

• // bien
• const arr = [
•   function foo() {
•   },
•
•   function bar() {
•   },
• ];
•
• return arr;

```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Comas

- Comas al inicio de línea: **Nop.**

```

• // mal
• const story = [
•   once
•   , upon
•   , aTime
• ];
•
• // bien
• const story = [
•   once,
•   upon,
•   aTime,
• ];
•
• // mal
• const hero = {
•   firstName: 'Ada'
•   , lastName: 'Lovelace'
•   , birthYear: 1815
•   , superPower: 'strength'
• };
•
• // bien
• const hero = {
•   firstName: 'Ada',
•   lastName: 'Lovelace',
•   birthYear: 1815,
•   superPower: 'computers',
• };

```

- Coma adicional al final: **Sip.**

¿Por qué? Esto lleva a diferenciales en git más claros. Además los transpiladores como Babel removerán la coma del final en el código transpilado lo que

significa que no te tendrás que preocupar del [problema de la coma adicional al final](#) en navegadores antiguos.

```
// mal - git diff sin coma adicional al final
const hero = {
  firstName: 'Florence',
-  lastName: 'Nightingale'
+  lastName: 'Nightingale',
+  inventorOf: ['coxcomb chart', 'modern nursing']
};

// bien - git diff con coma adicional al final
const hero = {
  firstName: 'Florence',
  lastName: 'Nightingale',
+  inventorOf: ['coxcomb chart', 'modern nursing'],
};

// mal
const hero = {
  firstName: 'Dana',
  lastName: 'Scully'
};

const heroes = [
  'Batman',
  'Superman'
];

// bien
const hero = {
  firstName: 'Dana',
  lastName: 'Scully',
};

const heroes = [
  'Batman',
  'Superman',
];
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Puntos y Comas

- **Sip.**

```
• // mal
• (function () {
•   const name = 'Skywalker'
•   return name
• })()
•
• // bien
• (() => {
•   const name = 'Skywalker';
```

- `return name;`
- `}());`
- 
- `// bien, pero arcaico (evita que la funcion se vuelva un argumento`
- `// cuando dos archivos con IIFEs sean concatenados)`
- `;(() => {`
- `const name = 'Skywalker';`
- `return name;`
- `}());`

[\[↑ regresar a la Tabla de Contenido\]](#)

## Casting de Tipos y Coerción

- Ejecuta coerción al inicio de una sentencia.
- Strings:

- `// => this.reviewScore = 9;`
- 
- `// mal`
- `const totalScore = this.reviewScore + ''; // invoca a this.reviewScore.valueOf()`
- 
- `// mal`
- `const totalScore = this.reviewScore.toString(); // no se garantiza que retorne una cadena de texto`
- 
- `// bien`
- `const totalScore = String(this.reviewScore);`

- Números: Usa `Number` para el casting de tipo y `parseInt` siempre con la base numérica para el casting de textos.

- `const inputValue = '4';`
- 
- `// mal`
- `const val = new Number(inputValue);`
- 
- `// mal`
- `const val = +inputValue;`
- 
- `// mal`
- `const val = inputValue >> 0;`
- 
- `// mal`
- `const val = parseInt(inputValue);`
- 
- `// bien`
- `const val = Number(inputValue);`
- 
- `// bien`
- `const val = parseInt(inputValue, 10);`

- Si por alguna razón estás haciendo algo salvaje y `parseInt` es un cuello de botella por lo que necesitaste usar Bitshift por [razones de desempeño](#), deja un comentario explicando la razón y resumen de lo que estás haciendo.

```

• // bien
• /**
•  * 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;

```

**Nota:** Ten mucho cuidado al hacer operaciones de Bitshift. En Javascript los números son representados como [valores de 64-bit](#), sin embargo las operaciones de Bitshift siempre retornan un entero de 32-bits ([fuente](#)). Bitshift puede presentarnos un comportamiento inesperado para valores enteros mayores a 32 bits. [Discusión](#). El mayor entero con signo de 32 bits es 2,147,483,647:

```

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

```

- Booleans:

```

• const age = 0;
•
• // mal
• const hasAge = new Boolean(age);
•
• // bien
• const hasAge = Boolean(age);
•
• // bien
• const hasAge = !!age;

```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Convenciones de nomenclatura

- Evita nombres de una sola letra. Sé descriptivo con tus nombres.

```

• // mal
• function q() {
•   // ...algo...
• }
•
• // bien
• function query() {
•   // ...algo...
• }

```

- Usa camelCase cuando nombres tus objetos, funciones e instancias.

```

• // mal
• const OBJEcttsssss = {};
• const this_is_my_object = {};
• const o = {};
• function c() {}
•
• // bien
• var thisIsMyObject = {};
• function thisIsMyFunction() {}

```

- Usa PascalCase cuando nombres constructores o clases.

```

• // mal
• function user(options) {
•   this.name = options.name;
• }
•
• const bad = new user({
•   name: 'nope'
• });
•
• // bien
• class User {
•   constructor(options) {
•     this.name = options.name;
•   }
• }
•
• const good = new User({
•   name: 'yup',
• });

```

- No uses prefijos ni sufijos de guiones bajo.

¿Por qué? JavaScript no tiene el concepto de privacidad en términos de propiedades o métodos. A pesar que un guión bajo como prefijo es una convención común para indicar que son "privados", la realidad es que estas propiedades son absolutamente públicas, y por ello, parte de tu contrato público de API. La convención del prefijo de guión bajo podría orientar a los desarrolladores a pensar erróneamente que un cambio a aquellos no será de impacto o que los tests no son necesarios.

```

// mal
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';
this._firstName = 'Panda';

// bien
this.firstName = 'Panda';

```

- Nunca guardes referencias a this. Usa funciones arrow o la función `#bind`

- `// mal`
- `function() {`
- `const self = this;`
- `return function() {`
- `console.log(self);`
- `};`
- `}`
- 
- `// mal`
- `function() {`
- `const that = this;`
- `return function() {`
- `console.log(that);`
- `};`
- `}`
- 
- `// bien`
- `function foo() {`
- `return () => {`
- `console.log(this);`
- `};`
- `}`

- El nombre del archivo base debe corresponder exactamente con el nombre de su export por defecto.

```
// contenido archivo 1
class CheckBox {
  // ...
}
export default CheckBox;

// contenido archivo 2
export default function fortyTwo() { return 42; }

// contenido archivo 3
export default function insideDirectory() {}

// en algún otro archivo
// mal
import CheckBox from './checkBox'; // importacion/exportacion PascalCase,
nombre de archivo camelCase
import FortyTwo from './FortyTwo'; // importacion/nombre de archivo
PascalCase, exportacion camelCase
import InsideDirectory from './InsideDirectory'; // importacion/nombre de
archivo PascalCase, exportacion camelCase

// mal
import CheckBox from './check_box'; // importacion/exportacion PascalCase,
nombre de archivo snake_case
import forty_two from './forty_two'; // importacion/nombre de archivo
snake_case, exportacion camelCase
import inside_directory from './inside_directory'; // importacion
snake_case, exportacion camelCase
import index from './inside_directory/index'; // requiere el archivo de
index explícitamente
```



```
import insideDirectory from './insideDirectory/index'; // requiere el
archivo de index explícitamente

// bien
import CheckBox from './CheckBox'; // importacion/exportacion/nombre de
archivo PascalCase
import fortyTwo from './fortyTwo'; // importacion/exportacion/nombre de
archivo camelCase
import insideDirectory from './insideDirectory'; //
importacion/exportacion/nombre directorio/archivo "index" implícito
// ^ soporta tanto insideDirectory.js e insideDirectory/index.js
```

- Usa camelCase cuando exportes por defecto una función. Tu nombre de archivo debe ser idéntico al nombre de tu función.

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- Usa camelCase cuando exportes un objeto constructor / clase / singleton / librería de función / esqueleto.

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

export default AirbnbStyleGuide;
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Funciones de Acceso

- Funciones de acceso para las propiedades no son requeridas.
- No uses getters/setters de JavaScript ya que causan efectos colaterales no esperados y son difíciles de probar, mantener y razonar. En vez de ello, si creas funciones de acceso usa `getVal()` y `setVal('hello')`.

```
• // Maintainable-JavaScript-Nicholas-C-Zakas
• class Dragon {
•   get age() {
•     // ...
•   }
•
•   set age(value) {
•     // ...
•   }
• }
•
• // bien
• class Dragon {
```

```

•   getAge() {
•       // ...
•   }
•
•   setAge(value) {
•       // ...
•   }
• }

```

- Si la propiedad es un booleano, usa `isVal()` o `hasVal()`.

```

• // mal
• if (!dragon.age()) {
•     return false;
• }
•
• // bien
• if (!dragon.hasAge()) {
•     return false;
• }

```

- Está bien crear funciones `get()` y `set()`, pero sé consistente.

```

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

```

[\[↑ regresar a la Tabla de Contenido\]](#)

## Eventos

- Cuando envíes paquetes de datos a los eventos (ya sea con eventos del DOM o algo propietario como los eventos de Backbone), pasa un mapa en vez de un valor directo. Esto permitirá a un próximo colaborador a agregar más datos al paquete de datos sin que tenga que encontrar o actualizar un handler para cada evento. Por ejemplo, en vez de:

```

• // mal
• $(this).trigger('listingUpdated', listing.id);
•
• ...
•
• $(this).on('listingUpdated', (e, listingId) => {

```

- `// hacer algo con listingId`  
`});`

prefiere:

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

...

$(this).on('listingUpdated', (e, data) => {
  // hacer algo con data.listingId
});
```

[\[↑ regresar a la Tabla de Contenido\]](#)

## jQuery

- Nombre las variables de objetos jQuery con un prefijo \$.

```
• // mal
• const sidebar = $('.sidebar');
•
• // bien
• const $sidebar = $('.sidebar');
```

- Guarde en variables los lookups de jQuery que se necesiten posteriormente.

```
• // mal
• function setSidebar() {
•   $('.sidebar').hide();
•
•   // ...algo...
•
•   $('.sidebar').css({
•     'background-color': 'pink'
•   });
• }
•
• // bien
• function setSidebar() {
•   const $sidebar = $('.sidebar');
•   $sidebar.hide();
•
•   // ...algo...
•
•   $sidebar.css({
•     'background-color': 'pink'
•   });
• }
```

- Para consultas de elementos DOM usa el modo Cascada `$('.sidebar ul')` o `parent > child` `$('.sidebar > ul')`. [jsPerf](#)

- Usa find solo con consultas guardadas en variables previamente.

```
• // mal
• $('ul', '.sidebar').hide();
•
• // mal
• $('.sidebar').find('ul').hide();
•
• // bien
• $('.sidebar ul').hide();
•
• // bien
• $('.sidebar > ul').hide();
•
• // bien
• $sidebar.find('ul');
```