Los principios de la ingeniería de software, del libro de Robert C. Martin *Clean Code*, adaptado para JavaScript. Esta no es una guía de estilo, en cambio, es una guía para crear software que sea reutilizable, comprensible y que se pueda mejorar con el tiempo.

No hay que seguir tan estrictamente todos los principios en este libro, y vale la pena mencionar que hacia muchos de ellos habrá controversia en cuanto al consentimiento. Estas son reflexiones hechas después de muchos años de experiencia colectiva de los autores de *Clean Code*.

Nuestra obra de ingeniería de software lleva poco más que 50 años como negocio, y aún estamos aprendiendo. Cuando la arquitectura de software llegue a ser tan vieja como la arquitectura en sí misma, quizás tengamos reglas más estrictas para seguir. Hasta entonces, dejemos que estas guías sirvan como ejemplo para medir la calidad del código en JavaScript que tú y tu equipo producen.

Una cosa más: saber esto no te hará un mejor ingeniero inmediatamente, y tampoco trabajar con estas herramientas durante muchos años garantiza que nunca te equivocarás. Cualquier código empieza primero como un borrador, como arcilla mojada moldeándose en su forma final. Por último, arreglamos las imperfecciones cuando lo repasamos con nuestros compañeros de trabajo. No seas tan duro contigo mismo por los borradores iniciales que aún necesitan mejorar. ¡Trabaja más duro para mejorar el programa!

Variables

Utiliza nombres significativos y pronunciables para las variables

Mal hecho:

```
const yyyymmdstr = moment().format('YYYY/MM/DD');
```

Bien hecho:

```
const fechaActual = moment().format('YYYY/MM/DD');
```

Utiliza el vocabulario igual para los variables del mismo tipo

Mal hecho:

```
conseguirInfoUsuario();
conseguirDataDelCliente();
conseguirRecordDelCliente();
```

Bien hecho:

```
conseguirUsuario();
```

Utiliza nombres buscables

Nosotros leemos mucho más código que jamás escribiremos. Es importante que el código que escribimos sea legible y buscable. Cuando faltamos nombrar a los variables de manera buscable y legible, acabamos confundiendo a nuestros lectores. Echa un vistazo a las herramientas para ayudarte: buddy.js and ESLint

Mal hecho:

```
// Para que rayos sirve 86400000?
setTimeout(hastaLaInfinidadYMasAlla, 86400000);
```

Bien hecho:

```
// Declaralos como variables globales de 'const'.
const MILISEGUNDOS_EN_UN_DIA = 8640000;

setTimeout(hastaLaInfinidadYMasAlla, MILISEGUNDOS_EN_UN_DIA);
```

Utiliza variables explicativas

Mal hecho:

```
const direccion = 'One Infinite Loop, Cupertino 95014';
const codigoPostalRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
saveCityZipCode(direccion.match(codigoPostalRegex)[1],
direccion.match(codigoPostalRegex)[2]);
```

Bien hecho:

```
const direccion = 'One Infinite Loop, Cupertino 95014';
const codigoPostalRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
const [, ciudad, codigoPostal] = direccion.match(codigoPostalRegex) || [];
guardarcodigoPostal(ciudad, codigoPostal);
```

Evitar el mapeo mental

El explícito es mejor que el implícito.

Mal hecho:

```
const ubicaciones = ['Austin', 'New York', 'San Francisco'];
ubicaciones.forEach((u) => {
  hazUnaCosa();
  hasMasCosas()
  // ...
  // ...
  // ...
  // Espera, para que existe la 'u'?
  ejecuta(u);
});
```

Bien hecho:

```
const ubicaciones = ['Austin', 'New York', 'San Francisco'];
ubicaciones.forEach((ubicacion) => {
  hazUnaCosa();
  hasMasCosas()
  // ...
  // ...
  // ...
  ejecuta(ubicacion);
});
```

No incluyas contexto innecesario

Si tu clase / nombre tu objeto te dice algo, no lo repitas en el nombre de variable también.

Mal hecho:

```
const Coche = {
  cocheMarca: 'Honda',
  cocheModelo: 'Accord',
  cocheColor: 'Blue'
};

function pintarCoche(coche) {
  coche.cocheColor = 'Red';
}
```

Bien hecho:

```
const Coche = {
  marca: 'Honda',
  modelo: 'Accord',
  color: 'Blue'
};

function pintarCoche(coche) {
  coche.color = 'Red';
}
```

Utiliza argumentos predefinidos en vez de utilizar condicionales

Los argumentos predefinidos muchas veces son más organizados que utilizar los condicionales. Se consciente que si tú los usas, tu función sólo tendrá valores para los argumentos de `undefined`. Los demás valores de 'falso' como `''`, `""`, `false`, `null`, `0`, y `NaN`, no se reemplazan con un valor predefinido.

Mal hecho:

```
function crearMicroCerveceria(nombre) {
  const nombreDelMicroCerveceria = nombre || 'Hipster Brew Co.';
  // ...
}
```

Bien hecho:

```
function crearMicroCerveceria(nombreDelMicroCerveceria = 'Hipster Brew Co.')
{
  // ...
}
```

Funciones

Argumentos de funciones (2 o menos idealmente)

Limitar la cantidad de parámetros de tus funciones es increíblemente importante ya que hace que tus pruebas del código sean más fáciles. Al pasar los 3 argumentos, llegarás a un escenario de una explosión combinatoria en que hay que comprobar con pruebas muchos casos únicos con un argumento separado.

Uno o dos argumentos es la situación ideal, y más que eso uno debe evitar si es posible. Todo lo que se puede consolidar se debe consolidar. Normalmente, si tienes más que dos argumentos, tu función sirve para hacer demasiado. En otros casos, es mejor refactorizar y hacerlo un objeto para encapsular las funciones extras.

Ya que JavaScript te deja crear objetos cuando quieras sin incorporar la arquitectura de 'clases', se puede usar un objeto si necesitas muchos argumentos.

Para hacerlo más obvio cuáles argumentos espera la función, se puede usar la sintaxis de ES2015/ES6: 'destructuración'. Esta sintaxis tiene varias ventajas:

1. Cuando alguien se fija en el firme de la función, es inmediatamente claro cuáles argumentos se usan.
2. Destructurar también copia los valores específicos y primitivos del objeto argumento que se le pasa a la función. Esto puede evitar los efectos extras. Ojo: objetos y arreglos que se destructuran del objeto argumento NO se copian.
3. Los 'linters' te pueden avisar cuales argumentos / propiedades no se usan, lo cual sería imposible sin destructurar.

Mal hecho:

```
function crearMenu(titulo, contexto, textoDelBoton, cancelable) {
  // ...
}
```

Bien hecho:

```
function crearMenu({ titulo, contexto, textoDelBoton, cancelable }) {
  // ...
}

crearMenu({
  titulo: 'Foo',
  contexto: 'Bar',
  textoDelBoton: 'Baz',
  cancelable: true
});
```

Las funciones deben tener una sola responsabilidad

Esta regla por mucho es la más importante en la ingeniería de software. Cuando las funciones sirven para hacer más que una sola cosa, se dificultan las pruebas, la composición y el entender. Cuando puedes aislar una función hasta tener solo una acción, se pueden mejorar más fácil y tu código llegue a ser mucho más limpio. Si solamente entiendes una cosa de esta guía, entiende esta regla y estarás adelantado de muchos desarrolladores.

Mal hecho:

```
function escribirClientes(clients) {
  clientes.forEach((cliente) => {
    const recordDelCliente = database.busca(cliente);
    if (recordDelCliente.esActivo()) {
      escribir(cliente);
    }
  });
}
```

Bien hecho:

```
function escribirClientes(clientes) {
  clientes
    .filter(esActivoElCliente)
    .forEach(email);
}

function esActivoElCliente(cliente) {
```

```
  const recordDelCliente = database.busca(cliente);
  return recordDelCliente.esActivo();
}
```

Los nombres de las funciones deben explicar lo que hacen

Mal hecho:

```
function adelantarLaFechaPorUnDia(fecha, mes) {
  // ...
}

const fecha = new Date();
// Es difícil entender del nombre lo que hace la función
adelantarLaFechaPorUnDia(fecha, 1);
```

Bien hecho:

```
function agregarMesAlDia(mes, fecha) {
  // ...
}

const fecha = new Date();
agregarMesAlDia(1, fecha);
```

Las funciones deben tener solo un nivel de abstracción

Cuando tienes más que un nivel de abstracción tu función suele servir para hacer demasiado. Crear varias funciones más pequeñas se debe a mejor reutilización y comprobación más fácil.

Mal hecho:

```
function parseBetterJSAlternative(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      // ...
    });
```

```
  });

  const ast = [];
  tokens.forEach((token) => {
    // lex...
  });

  ast.forEach((node) => {
    // parse...
  });
}
```

Bien hecho:

```
function tokenize(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      tokens.push( /* ... */ );
    });
  });

  return tokens;
}

function lexer(tokens) {
  const ast = [];
  tokens.forEach((token) => {
    ast.push( /* ... */ );
  });

  return ast;
}

function parseBetterJSAlternative(code) {
  const tokens = tokenize(code);
  const ast = lexer(tokens);
  ast.forEach((node) => {
    // parse...
  });
}
```

Eliminar el código duplicado

Haz tanto como puedas para evitar código duplicado. El código duplicado es malo ya que significa que hay varios lugares donde hay que actualizar algo si un cambio es necesario en tu lógico.

Imagínate que estás en un restaurante y necesitas organizar tu inventario: todos tus tomates, cebolla, pimientos y tal. Si tienes varias listas donde organizas el inventario, cada lista se tendrá que actualizar en cuanto se baja tu inventario. En cambio, si logras tener una sola lista, solo se actualizará en un lugar a la hora de apuntar el inventario.

Muchas veces tienes código duplicado se debe al hecho de tener dos o más cosas semejantes. Estos archivos pueden comparten varias cosas, pero sus diferencias te obligan separarlos para tener dos o más funciones que hacen cosas muy similares. Remover el código duplicado significa que se puede hacer la misma cosa que un solo función/módulo/clase.

Obtener la abstracción correcta es crítica y por eso debes de adherir a los principios de SOLID que se explican en la sección de Clases. Las malas abstracciones pueden ser aún peores que el código duplicado, ¡así que ten cuidado! Es decir, si puedes hacer una buena abstracción, ¡hazla! No te repitas, si no te darás cuenta de que andas actualizando mucho código en varios lugares a la hora de implementar un cambio.

Mal hecho:

```
function showDeveloperList(developers) {
  developers.forEach((developer) => {
    const expectedSalary = developer.calculateExpectedSalary();
    const experience = developer.getExperience();
    const githubLink = developer.getGithubLink();
    const data = {
      expectedSalary,
      experience,
      githubLink
    };

    render(data);
  });
}

function showManagerList(managers) {
  managers.forEach((manager) => {
    const expectedSalary = manager.calculateExpectedSalary();
    const experience = manager.getExperience();
    const portfolio = manager.getMBAProjects();
```

```
    const data = {
      expectedSalary,
      experience,
      portfolio
    };

    render(data);
  });
}
```

Bien hecho:

```
function showEmployeeList(employees) {
  employees.forEach((employee) => {
    const expectedSalary = employee.calculateExpectedSalary();
    const experience = employee.getExperience();

    let portfolio = employee.getGithubLink();

    if (employee.type === 'manager') {
      portfolio = employee.getMBAProjects();
    }

    const data = {
      expectedSalary,
      experience,
      portfolio
    };

    render(data);
  });
}
```

Crear objetos predefinidos con Object.assign

Mal hecho:

```
const menuConfig = {
  title: null,
  body: 'Bar',
  buttonText: null,
  cancellable: true
};

function createMenu(config) {
  config.title = config.title || 'Foo';
  config.body = config.body || 'Bar';
  config.buttonText = config.buttonText || 'Baz';
```

```
  config.cancellable = config.cancellable === undefined ? config.cancellable
: true;
}

createMenu(menuConfig);
```

Bien hecho:

```
const menuConfig = {
  title: 'Order',
  // El usuario no tenía la clave 'body'
  buttonText: 'Send',
  cancellable: true
};

function createMenu(config) {
  config = Object.assign({
    title: 'Foo',
    body: 'Bar',
    buttonText: 'Baz',
    cancellable: true
  }, config);
  // el variable 'config' ahora iguala: {title: "Order", body: "Bar",
buttonText: "Send", cancellable: true}
  // ...
}

createMenu(menuConfig);
```

No utilices 'marcadores' como parámetros de las funciones

Los marcadores existen para decirle a tu usuario que esta función hace más que una sola cosa. Como se ha mencionado antes las funciones deben hacer una sola cosa. Divide tus funciones en varias funciones más pequeñas si se adhieren a distintos métodos basados en un booleano.

Mal hecho:

```
function createFile(name, temp) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
}
```

Bien hecho:

```
function createFile(name) {
  fs.create(name);
}

function createTempFile(name) {
  createFile(`./temp/${name}`);
}
```

Evitar que las funciones produzcan efectos extras (parte 1)

Una función produce un efecto extra si hace cualquier cosa más que solo tomar un valor y volverlo/los). Un efecto extra podría ser escribir a un archivo, modificar un variable global, o accidentalmente enviar todo tu dinero a un desconfiado.

Bueno, las funciones necesitan tener efectos extras a menudo. Como el ejemplo anterior, puede que sea necesario escribir hasta un archivo. En ese caso, hay que centralizar en el 'por qué' de lo que estás haciendo. No tengas varias funciones y clases que escriben hasta un archivo particular. En cambio, crea un 'servicio' que se dedica a eso: uno y solo un servicio.

El punto clave aquí es evitar las equivocaciones comunes como compartir 'estado' entre objeto sin ninguna estructura, utilizar tipos de data mutables que se pueden escribir hasta lo que sea, y no centralizar donde se ocurren los efectos extras. Si puedes conseguir esto, serás más feliz que la mayoría de los demás programadores.

Mal hecho:

```
// Global variable referenced by following function.
// If we had another function that used this name, now it'd be an array and
it could break it.
let name = 'Ryan McDermott';

function splitIntoFirstAndLastName() {
  name = name.split(' ');
}

splitIntoFirstAndLastName();

console.log(name); // ['Ryan', 'McDermott'];
```

Bien hecho:

```
function splitIntoFirstAndLastName(name) {
  return name.split(' ');
}

const name = 'Ryan McDermott';
const newName = splitIntoFirstAndLastName(name);

console.log(name); // 'Ryan McDermott';
console.log(newName); // ['Ryan', 'McDermott'];
```

Evitar los efectos extras(parte 2)

En JavaScript, los primitivos se pasan por valores y los objetos/arrays se pasan por referencia. En el caso de los objetos y los array, si tu función hace un cambio en la shopping cart array, por ejemplo, con agregar una cosa a la hora de comprar, resulta que todas las demás funciones que utilizan este array estarán afectadas. Esto puede ser bueno o malo. Imaginemos una situación mala:

El usuario le da click a "Comprar", un botón que invoca la función de "comprar". Esta función hace una solicitud del red y envía el array de 'cart' hasta el servidor. Debido a la conexión mala del red, la función sigue intentando invocarse para mandar la solicitud. Ahora, que pasa mientras tanto cuando el usuario le da click otra vez al botón en una cosa que no querían antes de que empezase la solicitud del red? Bueno, si pasa eso y comienza la solicitud del red, la función de 'comprar' mandara sin querer la cosa que estaba agregada accidentalmente ya que tiene una referencia al array de 'shopping cart' que la función 'addItemToCart' modifico con agregar una cosa no deseada.

Una buena solución seria que la función 'addItemToCart' siempre copiara la 'carta', editarla, y devolvérsela a la copia. Esto asegura que ninguna otra función relacionada se afectará por estos cambios.

Dos cosas para mencionar con esta solución:

1. Puede que existan escenarios donde de verdad quieres modificar el objeto de input, pero cuando adoptas esta práctica de programar, te darás cuentas de que estos casos son bastante únicos.
2. Copiar objetos grandes pueden ser muy caros en cuanto a la velocidad y calidad de tu programa. Afortunadamente, no hay mucho problema con esto ya que existen muchas recursos que nos dejan lograr el copiar de objetos y arrays sin perder actuación.

Mal hecho:

```
const addItemToCart = (cart, item) => {
  cart.push({ item, date: Date.now() });
};
```

Bien hecho:

```
const addItemToCart = (cart, item) => {
  return [...cart, { item, date : Date.now() }];
};
```

No intentes cambiar las funciones globales

Polucionar las construcciones globales no es buen costumbre en JavaScript ya que se puede afrontar con otra biblioteca y el usuario de tu API no se daría cuenta hasta que reciba una excepción cuando ya está en producción el código. Pensemos en un ejemplo: que pasaría si quisieras extender los métodos nativos de la clase Array para tener un método de 'diff' en que se podría mostrar la diferencia entre dos arrays? Podrías escribir una nueva función hasta el prototipo del `Array.prototype`, pero eso también podría causar problemas con otra biblioteca que contenía el método igual. Bueno, ¿qué pasaría si la otra biblioteca solamente usaba ‘diff’ para averiguar la diferencia entre el primer elemento y el último elemento del array? Por eso hay que utilizar las clases de ES2015/ES6 y extender el global de `Array`.

Mal hecho:

```
Array.prototype.diff = function diff(comparisonArray) {
  const hash = new Set(comparisonArray);
  return this.filter(elem => !hash.has(elem));
};
```

Bien hecho:

```
class SuperArray extends Array {
  diff(comparisonArray) {
    const hash = new Set(comparisonArray);
    return this.filter(elem => !hash.has(elem));
  }
}
```

Favorece a la programación funcional en vez de la programación imperativa

JavaScript no es un idioma funcional tal como es Haskell, pero tiene su propio sabor funcional. Los idiomas funcionales son más limpios y fáciles de comprobar. Favorece este estilo de programar cuando puedes.

Mal hecho:

```
const programmerOutput = [
  {
    name: 'Uncle Bobby',
    linesOfCode: 500
  }, {
    name: 'Suzie Q',
    linesOfCode: 1500
  }, {
    name: 'Jimmy Gosling',
    linesOfCode: 150
  }, {
    name: 'Gracie Hopper',
    linesOfCode: 1000
  }
];

let totalOutput = 0;

for (let i = 0; i < programmerOutput.length; i++) {
  totalOutput += programmerOutput[i].linesOfCode;
}
```

Bien hecho:

```
const programmerOutput = [
  {
    name: 'Uncle Bobby',
    linesOfCode: 500
  }, {
    name: 'Suzie Q',
    linesOfCode: 1500
  }, {
    name: 'Jimmy Gosling',
    linesOfCode: 150
  }, {
    name: 'Gracie Hopper',
    linesOfCode: 1000
  }
];
```

```
const INITIAL_VALUE = 0;

const totalOutput = programmerOutput
  .map((programmer) => programmer.linesOfCode)
  .reduce((acc, linesOfCode) => acc + linesOfCode, INITIAL_VALUE);
```

Encapsular los condicionales

Mal hecho:

```
if (fsm.state === 'fetching' && isEmpty(listNode)) {
  // ...
}
```

Bien hecho:

```
function shouldShowSpinner(fsm, listNode) {
  return fsm.state === 'fetching' && isEmpty(listNode);
}

if (shouldShowSpinner(fsmInstance, listNodeInstance)) {
  // ...
}
```

Evitar los condicionales negativos

Mal hecho:

```
function isDOMNodeNotPresent(node) {
  // ...
}

if (!isDOMNodeNotPresent(node)) {
  // ...
}
```

Bien hecho:

```
function isDOMNodePresent(node) {
  // ...
}

if (isDOMNodePresent(node)) {
  // ...
```

```
}
```

Evitar los condicionales

Esto parece ser un reto imposible. Al escuchar esto por primera vez, la mayoría de la gente dirá: "como se supone que hago sin una declaración de 'if'?" Bueno, la respuesta es que puedes utilizar para lograr los mismos retos en muchos escenarios. La segunda pregunta suele ser: "bueno, eso está bien, pero por qué voy a querer hacer eso?" La respuesta yace en un concepto anterior que ya hemos aprendido: una función solo debe hacer una sola cosa. Cuando tienes clases y funciones que contienen declaraciones de `if`, le dices al usuario que tu función hace más que una sola cosa. Recuerda, solo haz una cosa.

Mal hecho:

```
class Airplane {
  // ...
  getCruisingAltitude() {
    switch (this.type) {
      case '777':
        return this.getMaxAltitude() - this.getPassengerCount();
      case 'Air Force One':
        return this.getMaxAltitude();
      case 'Cessna':
        return this.getMaxAltitude() - this.getFuelExpenditure();
    }
  }
}
```

Bien hecho:

```
class Airplane {
  // ...
}

class Boeing777 extends Airplane {
  // ...
  getCruisingAltitude() {
    return this.getMaxAltitude() - this.getPassengerCount();
  }
}

class AirForceOne extends Airplane {
  // ...
  getCruisingAltitude() {
    return this.getMaxAltitude();
```

```
  }
}

class Cessna extends Airplane {
  // ...
  getCruisingAltitude() {
    return this.getMaxAltitude() - this.getFuelExpenditure();
  }
}
```

Evitar la comprobación de tipos (parte 1)

JavaScript es un idioma no tecleado, por lo cual significa que tus funciones pueden aceptar cualquier tipo de argumento. A veces te aprovechas de esta libertad y tienes ganas de hacer comprobación de `tipos` dentro de tus funciones. Hay muchas maneras de evitar tener que hacer esto. La primeras cosas para considerar son APIs consistentes.

Mal hecho:

```
function travelToTexas(vehicle) {
  if (vehicle instanceof Bicycle) {
    vehicle.pedal(this.currentLocation, new Location('texas'));
  } else if (vehicle instanceof Car) {
    vehicle.drive(this.currentLocation, new Location('texas'));
  }
}
```

Bien hecho:

```
function travelToTexas(vehicle) {
  vehicle.move(this.currentLocation, new Location('texas'));
}
```

Evitar la comprobación de tipos (parte 2)

Si estás trabajando con los valores primitivos básicos como `strings`, enteros y arrays y que no puedes utilizar polimorfismo, pero existe la necesidad de comprobar los `tipos`, debes considerar utilizando TypeScript. Es un alternativo excelente a JavaScript, y te provee con los tipos estáticos encima del sintaxis estándar de JavaScript. El problema con comprobar los tipos en JavaScript es que para hacerlo bien resulta en mucho más verbos que no vale la pena al lado de la legibilidad

disminuida que viene a junto con esta solución. Intenta mantener limpio tu código de JavaScript, escribe buenas pruebas, y haz buenas revisiones de código. Eso dicho, haz todo lo de arriba, pero con TypeScript (por lo cual, como dije, es buen alternativo).

Mal hecho:

```
function combine(val1, val2) {
  if (typeof val1 === 'number' && typeof val2 === 'number' ||
      typeof val1 === 'string' && typeof val2 === 'string') {
    return val1 + val2;
  }

  throw new Error('Must be of type String or Number');
}
```

Bien hecho:

```
function combine(val1, val2) {
  return val1 + val2;
}
```

No optimices demasiado

Los navegadores modernos hacen mucha optimización en el fondo a la hora de ejecutar. Muchas veces, malgastas tu tiempo si optimizas. Hay buenos recursos para esto para ver donde carece de optimizar tu código. Enfócate en esos huecos donde puedes optimizar, hasta que se puedan arreglar si es posible.

Mal hecho:

```
// On old browsers, each iteration with uncached `list.length` would be
costly
// because of `list.length` recomputation. In modern browsers, this is
optimized.
for (let i = 0, len = list.length; i < len; i++) {
  // ...
}
```

Bien hecho:

```
for (let i = 0; i < list.length; i++) {
  // ...
}
```

Eliminar el código muerto

El código muerto es tan elegante como el código duplicado. No hay razón para guardarlo. Si no se usa, ¡elimínalo! Aun estará en tu historia del control versión si de verdad lo necesitas.

Mal hecho:

```
function oldRequestModule(url) {
  // ...
}

function newRequestModule(url) {
  // ...
}

const req = newRequestModule;
inventoryTracker('apples', req, 'www.inventory-awesome.io');
```

Bien hecho:

```
function newRequestModule(url) {
  // ...
}

const req = newRequestModule;
inventoryTracker('apples', req, 'www.inventory-awesome.io');
```

Objetos y estructuras de data

Utiliza getters y setters

Utilizar los getters y setters para acceder data dentro de los objetos puede ser mejor que simplemente buscar una propiedad. "Por qué?" Bueno, aquí te dejo con una lista desorganizada de las razones:

- Cuando quieres hacer algo más allá de acceder una propiedad de objeto, no tienes qué buscar todos los accesorios en tu programa.
- Hace que implementar validación sea más fácil cuando construyes un `set`
- Encapsula la representación internal

- Facilita la incorporación de apuntar errores de acceder y crear
- Puedes cargar de manera vaga las propiedades del objeto, digamos de un servidor por ejemplo

Mal hecho:

```
function makeBankAccount() {
  // ...

  return {
    balance: 0,
    // ...
  };
}

const account = makeBankAccount();
account.balance = 100;
```

Bien hecho:

```
function makeBankAccount() {
  // this one is private
  let balance = 0;

  // a "getter", made public via the returned object below
  function getBalance() {
    return balance;
  }

  // a "setter", made public via the returned object below
  function setBalance(amount) {
    // ... validate before updating the balance
    balance = amount;
  }

  return {
    // ...
    getBalance,
    setBalance,
  };
}

const account = makeBankAccount();
account.setBalance(100);
```

Haz que los objetos tengan miembros privados

Esto se puede lograr con `closures` (con ES5 y antes)

Mal hecho:

```
const Employee = function(name) {
  this.name = name;
};

Employee.prototype.getName = function getName() {
  return this.name;
};

const employee = new Employee('John Doe');
console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Employee name: undefined
```

Bien hecho:

```
function makeEmployee(name) {
  return {
    getName() {
      return name;
    },
  };
}

const employee = makeEmployee('John Doe');
console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe
```

Clases

Prefiere ES2015/ES6 clases en vez de funciones normales de ES5

Es muy difícil para obtener una herencia legible de las clases, las construcción y las definiciones de los métodos para las clases de ES5. Si necesitas la herencia (y puede que no la vayas a necesitar), entonces prefiere a las clases de ES2015/ES6.

Sin embargo, prefiere funciones pequeñas hasta que necesites objetos más grandes y complejos.

Mal hecho:

```
const Animal = function(age) {
  if (!(this instanceof Animal)) {
    throw new Error('Instantiate Animal with `new`');
  }

  this.age = age;
};

Animal.prototype.move = function move() {};

const Mammal = function(age, furColor) {
  if (!(this instanceof Mammal)) {
    throw new Error('Instantiate Mammal with `new`');
  }

  Animal.call(this, age);
  this.furColor = furColor;
};

Mammal.prototype = Object.create(Animal.prototype);
Mammal.prototype.constructor = Mammal;
Mammal.prototype.liveBirth = function liveBirth() {};

const Human = function(age, furColor, languageSpoken) {
  if (!(this instanceof Human)) {
    throw new Error('Instantiate Human with `new`');
  }

  Mammal.call(this, age, furColor);
  this.languageSpoken = languageSpoken;
};

Human.prototype = Object.create(Mammal.prototype);
Human.prototype.constructor = Human;
Human.prototype.speak = function speak() {};
```

Bien hecho:

```
class Animal {
  constructor(age) {
    this.age = age;
  }

  move() { /* ... */ }
}
```

```
class Mammal extends Animal {
  constructor(age, furColor) {
    super(age);
    this.furColor = furColor;
  }

  liveBirth() { /* ... */ }
}

class Human extends Mammal {
  constructor(age, furColor, languageSpoken) {
    super(age, furColor);
    this.languageSpoken = languageSpoken;
  }

  speak() { /* ... */ }
}
```

Utiliza la agregación de métodos

Este modelo es muy útil en JavaScript y puede que lo veas en muchas bibliotecas como jQuery y Lodash. También permite que tu código sea expresivo y menos verboso. Por eso, digo, utiliza el encadenamiento de métodos y échale un vistazo a lo limpio que llega a ser tu código. En tus funciones de clases, simplemente devuelve el `this` al final de cada función y asi puedes seguir encadenando los metodos de tu clase.

Mal hecho:

```
class Car {
  constructor() {
    this.make = 'Honda';
    this.model = 'Accord';
    this.color = 'white';
  }

  setMake(make) {
    this.make = make;
  }

  setModel(model) {
    this.model = model;
  }

  setColor(color) {
    this.color = color;
```

```
  }

  save() {
    console.log(this.make, this.model, this.color);
  }
}

const car = new Car();
car.setColor('pink');
car.setMake('Ford');
car.setModel('F-150');
car.save();
```

Bien hecho:

```
class Car {
  constructor() {
    this.make = 'Honda';
    this.model = 'Accord';
    this.color = 'white';
  }

  setMake(make) {
    this.make = make;
    // NOTE: Returning this for chaining
    return this;
  }

  setModel(model) {
    this.model = model;
    // NOTE: Returning this for chaining
    return this;
  }

  setColor(color) {
    this.color = color;
    // NOTE: Returning this for chaining
    return this;
  }

  save() {
    console.log(this.make, this.model, this.color);
    // NOTE: Returning this for chaining
    return this;
  }
}

const car = new Car()
  .setColor('pink')
  .setMake('Ford')
  .setModel('F-150')
```

```
  .save();
```

Prefiere composición en vez de la herencia

Como se ha dicho antes famosamente en el libro de *Design Patterns* escrito por el `Gang of Four`, debes preferir composición en vez de la herencia cuando puedas. Hay muchas razones para utilizar estos dos modelos. El punto importante aquí es que tu mente naturalmente quiere utilizar la herencia, así que intenta pensar si composición también podría resolver tu problema. En algunos casos, puede que sea la solución.

Puede que te preguntes, ¿”cuando debería de utilizar la herencia?" Bueno, depende de tu problema del momento, pero esta sería una lista decente de cuando tiene más sentido utilizarla que la composición

1. Tu herencia representa una relación de "es-un" y no un "tener-un" (Humano->Animal vs Usuario->Detalles del Usuario)
2. Puedes reutilizar tu código de las clases bases (Los humanos pueden moverse como todos los animales)
3. Quieres hacer cambios globales a las clases derivadas con cambiar una clase base. (Cambiar el gasto calórico de todos los animales cuando se mueven)

Mal hecho:

```
class Employee {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }

  // ...
}

// Bad because Employees "have" tax data. EmployeeTaxData is not a type of
Employee
class EmployeeTaxData extends Employee {
  constructor(ssn, salary) {
    super();
    this.ssn = ssn;
    this.salary = salary;
  }

  // ...
```

```
}
```

Bien hecho:

```
class EmployeeTaxData {
  constructor(ssn, salary) {
    this.ssn = ssn;
    this.salary = salary;
  }

  // ...
}

class Employee {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }

  setTaxData(ssn, salary) {
    this.taxData = new EmployeeTaxData(ssn, salary);
  }
  // ...
}
```

SOLID

El principio único de responsabilidad (SRP)

Como se menciona en *Clean Code*, "Nunca debe existir más que una sola razón para cambiar una clase". Vale la pena decir que es normal querer llenar una 'clase' con muchas funciones, igual que cuando solo te permiten llevar una maleta en el vuelo. El problema existe en que tu 'clase' no estará cohesiva conceptualmente y existirá muchas razones para cambiarse. Minimizar la cantidad de veces que necesitas cambiar una clase es importante. Es importante ya que con demasiada funcionalidad viene dificultad de modificarlo y entender cómo afecta a otros módulos dependientes en tu programa.

Mal hecho:

```
class UserSettings {
  constructor(user) {
    this.user = user;
  }
```

```
  changeSettings(settings) {
    if (this.verifyCredentials()) {
      // ...
    }
  }

  verifyCredentials() {
    // ...
  }
}
```

Bien hecho:

```
class UserAuth {
  constructor(user) {
    this.user = user;
  }

  verifyCredentials() {
    // ...
  }
}


class UserSettings {
  constructor(user) {
    this.user = user;
    this.auth = new UserAuth(user);
  }

  changeSettings(settings) {
    if (this.auth.verifyCredentials()) {
      // ...
    }
  }
}
```

Principio de abierto/cerrado (OCP)

Como dijo Bertrand Meyer, "las entidades de software (clases, módulos, funciones, etc.) deben abrirse para extensión, pero cerrarse para modificación. ¿Qué significa eso? Bueno, este principio básicamente nos dice que debes permitir que tus usuarios introduzcan nuevas funcionalidades sin cambiar el código existente.

Mal hecho:

```
class AjaxAdapter extends Adapter {
```

```
  constructor() {
    super();
    this.name = 'ajaxAdapter';
  }
}

class NodeAdapter extends Adapter {
  constructor() {
    super();
    this.name = 'nodeAdapter';
  }
}

class HttpRequester {
  constructor(adapter) {
    this.adapter = adapter;
  }

  fetch(url) {
    if (this.adapter.name === 'ajaxAdapter') {
      return makeAjaxCall(url).then((response) => {
        // transform response and return
      });
    } else if (this.adapter.name === 'httpNodeAdapter') {
      return makeHttpCall(url).then((response) => {
        // transform response and return
      });
    }
  }
}

function makeAjaxCall(url) {
  // request and return promise
}

function makeHttpCall(url) {
  // request and return promise
}
```

Bien hecho:

```
class AjaxAdapter extends Adapter {
  constructor() {
    super();
    this.name = 'ajaxAdapter';
  }

  request(url) {
    // request and return promise
  }
}
```

```
class NodeAdapter extends Adapter {
  constructor() {
    super();
    this.name = 'nodeAdapter';
  }

  request(url) {
    // request and return promise
  }
}

class HttpRequester {
  constructor(adapter) {
    this.adapter = adapter;
  }

  fetch(url) {
    return this.adapter.request(url).then((response) => {
      // transform response and return
    });
  }
}
```

El principio de sustitución de Liskov (LSP)

Este es un término espantoso para un concepto muy simple. Formalmente se define como "Si S es un subtipo de T que los objetos de T se reemplazan con los objetos de tipo S". (Es decir, los objetos de tipo S se pueden substituir con los objetos de tipo T sin alterar las propiedades deseables del programa (precisión, actuación, etc.). Esa si es una definición aún más espantosa.

La mejor explanación para este concepto es si tienes una clase `padre` y una clase `hijo`, luego la clase base y la clase `hijo` se pueden intercambiar sin tener resultados que carecen de precisión. Puede que aun estas confundido, así que miremos al modelo clásico de rectángulo-cuadro. Matemáticamente, un cuadro es un rectángulo, pero si lo modelas como una relación de "es-un" con la herencia, te meterás en problemas rápidamente.

Mal hecho:

```
class Rectangle {
  constructor() {
    this.width = 0;
    this.height = 0;
  }
```

```
  setColor(color) {
    // ...
  }

  render(area) {
    // ...
  }

  setWidth(width) {
    this.width = width;
  }

  setHeight(height) {
    this.height = height;
  }

  getArea() {
    return this.width * this.height;
  }
}

class Square extends Rectangle {
  setWidth(width) {
    this.width = width;
    this.height = width;
  }

  setHeight(height) {
    this.width = height;
    this.height = height;
  }
}

function renderLargeRectangles(rectangles) {
  rectangles.forEach((rectangle) => {
    rectangle.setWidth(4);
    rectangle.setHeight(5);
    const area = rectangle.getArea(); // BAD: Returns 25 for Square. Should
be 20.
    rectangle.render(area);
  });
}

const rectangles = [new Rectangle(), new Rectangle(), new Square()];
renderLargeRectangles(rectangles);
```

Bien hecho:

```
class Shape {
  setColor(color) {
    // ...
  }

  render(area) {
    // ...
  }
}

class Rectangle extends Shape {
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }

  getArea() {
    return this.width * this.height;
  }
}

class Square extends Shape {
  constructor(length) {
    super();
    this.length = length;
  }

  getArea() {
    return this.length * this.length;
  }
}

function renderLargeShapes(shapes) {
  shapes.forEach((shape) => {
    const area = shape.getArea();
    shape.render(area);
  });
}

const shapes = [new Rectangle(4, 5), new Rectangle(4, 5), new Square(5)];
renderLargeShapes(shapes);
```

El principio de segregación en cuanto a los interfaces (ISP)

JavaScript no tiene interfaces así que este principio no se aplica tanto como en otros idiomas. Sin embargo, es importante y es relevante, aunque JavaScript no tenga un sistema de `tipos`.

ISP declara que "Los clientes no se deben forzar para depender en interfaces que no implementan". Los interfaces son contratos implícitos en JavaScript debido al `teclear de duck`.

Un buen ejemplo que demuestra este principio en JavaScript es para las clases que necesitan objetos grandes de composición. Con no requerir que los clientes se encarguen de muchas opciones, puedes beneficiar ya que la mayoría del tiempo no hace falta todo lo extra. Cuando haces que las opciones del contratos sean opcionales, evitas un "interfaz gordo".

Mal hecho:

```
class DOMTraverser {
  constructor(settings) {
    this.settings = settings;
    this.setup();
  }

  setup() {
    this.rootNode = this.settings.rootNode;
    this.animationModule.setup();
  }

  traverse() {
    // ...
  }
}

const $ = new DOMTraverser({
  rootNode: document.getElementsByTagName('body'),
  animationModule() {} // Most of the time, we won't need to animate when
traversing.
  // ...
});
```

Bien hecho:

```
class DOMTraverser {
  constructor(settings) {
    this.settings = settings;
    this.options = settings.options;
    this.setup();
  }
```

```
  setup() {
    this.rootNode = this.settings.rootNode;
    this.setupOptions();
  }

  setupOptions() {
    if (this.options.animationModule) {
      // ...
    }
  }

  traverse() {
    // ...
  }
}

const $ = new DOMTraverser({
  rootNode: document.getElementsByTagName('body'),
  options: {
    animationModule() {}
  }
});
```

El principio de la inversión de las dependencias (DIP)

Este principio declara dos cosas esenciales:

1. Los módulos de nivel alto no deben depender en los módulos de nivel bajo. Los dos deben dependerse en las abstracciones.
2. Las abstracciones no deben dependerse en las detalles. Las detalles deben dependerse en las abstracciones.

Esto ha de entender la primera vez, pero si has trabajado con AngularJS, has visto una implementación de este principio en la forma de la inyección de dependencias (DI). Mientras que no son conceptos idénticos, el DIP mantiene que los módulos de nivel alto no sepan las detalles de los módulos de nivel bajo y también se encarga de ellos. Esto se puede conseguir con DI. Un beneficio enorme de esto es que reduce la convivencia entre los módulos. La convivencia es un modelo muy malo en cuanto al desarrollo de software ya dificulta la posibilidad de refactorizar.

Como se ha mencionado previamente, JavaScript no tiene interfaces así que las abstracciones de las que se dependen son contratos implícitos. Es decir, los métodos y las propiedades que un objeto/clase expone hasta otro objeto/clase. En

el ejemplo más abajo, el contrato implícito es que cualquier módulo de Request que utilizar el `InventoryTracker` debe tener un método de `requestItems`.

Mal hecho:

```
class InventoryRequester {
  constructor() {
    this.REQ_METHODS = ['HTTP'];
  }

  requestItem(item) {
    // ...
  }
}

class InventoryTracker {
  constructor(items) {
    this.items = items;

    // BAD: We have created a dependency on a specific request
implementation.
    // We should just have requestItems depend on a request method: `request`
    this.requester = new InventoryRequester();
  }

  requestItems() {
    this.items.forEach((item) => {
      this.requester.requestItem(item);
    });
  }
}

const inventoryTracker = new InventoryTracker(['apples', 'bananas']);
inventoryTracker.requestItems();
```

Bien hecho:

```
class InventoryTracker {
  constructor(items, requester) {
    this.items = items;
    this.requester = requester;
  }

  requestItems() {
    this.items.forEach((item) => {
      this.requester.requestItem(item);
    });
  }
}
```

```
class InventoryRequesterV1 {
  constructor() {
    this.REQ_METHODS = ['HTTP'];
  }

  requestItem(item) {
    // ...
  }
}

class InventoryRequesterV2 {
  constructor() {
    this.REQ_METHODS = ['WS'];
  }

  requestItem(item) {
    // ...
  }
}

// By constructing our dependencies externally and injecting them, we can
easily
// substitute our request module for a fancy new one that uses WebSockets.
const inventoryTracker = new InventoryTracker(['apples', 'bananas'], new
InventoryRequesterV2());
inventoryTracker.requestItems();
```

Pruebas

Comprobar nuestro código es más importante que enviarlo. Si no tienes pruebas o tienes una cantidad inadecuada, cada vez que envías tu código tendrás dudas en cuanto el saber de cuantos errores aún existen en tus programas. Deducir en lo que constituye una cantidad adecuada es la responsabilidad del equipo, pero tener cobertura 100% (todos las declaraciones y ramos) es como se logra una confianza alta y una tranquilidad de mente. Esto significa que encima de utilizar una estructura de pruebas, también necesitas usar una buena herramienta de cobertura.

No existe excusa para no escribir pruebas. Hay muchas estructuras buenas de pruebas para JS, así que busca una que le guste tu equipo. Cuando encuentras una que tu equipo le gusta, enfócate en siempre escribir pruebas para cada nueva característica/módulo que introduces. Si tu método preferido es el Test Driven Development (TDD), eso está bien, pero el punto principal es que te aseguras de llegar a tus objetivos de cobertura antes de enviar el código o refactorizar una prueba ya existente.

Un solo concepto para cada prueba

Mal hecho:

```
import assert from 'assert';

describe('MakeMomentJSGreatAgain', () => {
  it('handles date boundaries', () => {
    let date;

    date = new MakeMomentJSGreatAgain('1/1/2015');
    date.addDays(30);
    assert.equal('1/31/2015', date);

    date = new MakeMomentJSGreatAgain('2/1/2016');
    date.addDays(28);
    assert.equal('02/29/2016', date);

    date = new MakeMomentJSGreatAgain('2/1/2015');
    date.addDays(28);
    assert.equal('03/01/2015', date);
  });
});
```

Bien hecho:

```
import assert from 'assert';

describe('MakeMomentJSGreatAgain', () => {
  it('handles 30-day months', () => {
    const date = new MakeMomentJSGreatAgain('1/1/2015');
    date.addDays(30);
    assert.equal('1/31/2015', date);
  });

  it('handles leap year', () => {
    const date = new MakeMomentJSGreatAgain('2/1/2016');
    date.addDays(28);
    assert.equal('02/29/2016', date);
  });

  it('handles non-leap year', () => {
    const date = new MakeMomentJSGreatAgain('2/1/2015');
    date.addDays(28);
    assert.equal('03/01/2015', date);
  });
});
```

Concurrencia

Utiliza las promesas y no utilices los callbacks

Los callbacks no son limpos y utilizan una cantidad excesiva de encajamiento. Con ES2015/ES6, las Promesas son un tipo ya nativo del idioma. Utilizalas!

Mal hecho:

```
import { get } from 'request';
import { writeFile } from 'fs';

get('https://en.wikipedia.org/wiki/Robert_Cecil_Martin', (requestErr,
response) => {
  if (requestErr) {
    console.error(requestErr);
  } else {
    writeFile('article.html', response.body, (writeErr) => {
      if (writeErr) {
        console.error(writeErr);
      } else {
        console.log('File written');
      }
    });
  }
});
```

Bien hecho:

```
import { get } from 'request';
import { writeFile } from 'fs';

get('https://en.wikipedia.org/wiki/Robert_Cecil_Martin')
  .then((response) => {
    return writeFile('article.html', response);
  })
  .then(() => {
    console.log('File written');
  })
  .catch((err) => {
    console.error(err);
  });
```

Async/Await son aún más limpios que las Promesas

Las Promesas son una alternativa muy limpia a los callbacks, pero ES2017/E8 incluye `async` y `await` que ofrecen una solución aún más limpia. Todo lo que necesitas es una función que empieza con la palabra `async`, y luego puedes escribir tu lógico imperativamente sin una fila de funciones de `then`. Utiliza esto si puedes aprovecharte de los beneficios de ES2017/E8 hoy!

Mal hecho:

```
import { get } from 'request-promise';
import { writeFile } from 'fs-promise';

get('https://en.wikipedia.org/wiki/Robert_Cecil_Martin')
  .then((response) => {
    return writeFile('article.html', response);
  })
  .then(() => {
    console.log('File written');
  })
  .catch((err) => {
    console.error(err);
  });
```

Bien hecho:

```
import { get } from 'request-promise';
import { writeFile } from 'fs-promise';

async function getCleanCodeArticle() {
  try {
    const response = await
get('https://en.wikipedia.org/wiki/Robert_Cecil_Martin');
    await writeFile('article.html', response);
    console.log('File written');
  } catch(err) {
    console.error(err);
  }
}
```

Resolver los errores

¡Los errores que emergen en tus programas son buenos! Significan que tu ejecución ha tenido éxito a la hora de identificar un error en tu programa y te avisa con detener

la ejecución del 'stack' actual, matando el proceso (en Node), y notificarse en el 'console' con un reporte de 'stack trace'

No les ignores a los errores pillados

Hacer nada cuando existe un error pillado no te da la habilidad de arreglar o resolverlo. Apuntar el error al console (`console.log`) no es mucho mejor ya que muchas veces se puede perder en un mar de cosas que se apuntan al console. Si metes tu código en un `try/catch` significa que un error puede ocurrir allí y así que deberías de tener un plan, o crear una solución por si acaso.

Mal hecho:

```
try {
  functionThatMightThrow();
} catch (error) {
  console.log(error);
}
```

Bien hecho:

```
try {
  functionThatMightThrow();
} catch (error) {
  // One option (more noisy than console.log):
  console.error(error);
  // Another option:
  notifyUserOfError(error);
  // Another option:
  reportErrorToService(error);
  // OR do all three!
}
```

No le ignores a las promesas rechazadas

Igual que no debes ignorar a los errores no pillados de un 'try/catch'

Mal hecho:

```
getdata()
  .then((data) => {
    functionThatMightThrow(data);
  })
  .catch((error) => {
    console.log(error);
  });
```

Bien hecho:

```
getdata()
  .then((data) => {
    functionThatMightThrow(data);
  })
  .catch((error) => {
     // Una opción (más ruidoso que console.log):
    console.error(error);
    // Otra opción:
    notifyUserOfError(error);
    // Otra opción
    reportErrorToService(error);
    // O haz las tres!
  });
```

Formatear

Formatear es subjetivo. Como muchas reglas en esta guía, no hay que seguirlas 100%. El punto clave es: NO DISCUTAS sobre formatear. Hay muchas herramientas para facilitar esto. ¡Utiliza una de estas herramientas! Te desperdicias de tu propio tiempo y el tiempo de los demás cuando discuten sobre formatear.

Para las cosas que no tienen relevancia al formateo automático (indentación, tabulos y espacios, comillas dobles y sencillas, etc.), busca aquí para aconsejarte.

Utiliza capitalización consistente

JavaScript es un idioma no tecleado, así que la capitalización puede decirte muchas cosas sobre tus variables, funciones, etc. Estas reglas son subjetivas, así que tu equipo puede escoger lo que quiera. El punto es, sin importar lo que escojas, se consistente.

Mal hecho:

```
const DAYS_IN_WEEK = 7;
const daysInMonth = 30;

const songs = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];
const Artists = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {}
function restore_database() {}

class animal {}
```

```
class Alpaca {}
```

Bien hecho:

```
const DAYS_IN_WEEK = 7;
const DAYS_IN_MONTH = 30;

const songs = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];
const artists = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {}
function restoreDatabase() {}

class Animal {}
class Alpaca {}
```

Los llamadores y llamantes de las funciones deben existir cercas

Si una función le llama a otra, mantiene esa funcionas verticalmente cerca en su archivo de fuente. Idealmente, mantiene el llamador justo encima del llamante. Solemos leer código arriba-abajo, como un periódico. Debido a esto, haz que tus programas sean legibles así.

Mal hecho:

```
class PerformanceReview {
  constructor(employee) {
    this.employee = employee;
  }

  lookupPeers() {
    return db.lookup(this.employee, 'peers');
  }

  lookupManager() {
    return db.lookup(this.employee, 'manager');
  }

  getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }

  perfReview() {
    this.getPeerReviews();
    this.getManagerReview();
```

```
    this.getSelfReview();
  }

  getManagerReview() {
    const manager = this.lookupManager();
  }

  getSelfReview() {
    // ...
  }
}

const review = new PerformanceReview(user);
review.perfReview();
```

Bien hecho:

```
class PerformanceReview {
  constructor(employee) {
    this.employee = employee;
  }

  perfReview() {
    this.getPeerReviews();
    this.getManagerReview();
    this.getSelfReview();
  }

  getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }

  lookupPeers() {
    return db.lookup(this.employee, 'peers');
  }

  getManagerReview() {
    const manager = this.lookupManager();
  }

  lookupManager() {
    return db.lookup(this.employee, 'manager');
  }

  getSelfReview() {
    // ...
  }
}

const review = new PerformanceReview(employee);
```

```
review.perfReview();
```

Comentarios

Solamente comenta las cosas que tienen lógico complejo.

Los comentarios existen para pedir perdón, pero no son un requisito. El código bueno más que nada se documenta sí mismo.

Mal hecho:

```
function hashIt(data) {
  // El hash
  let hash = 0;

  // Length of string
  const length = data.length;
  // Iterar cada caracter en la data
  for (let i = 0; i < length; i++) {
     // Conseguir el código del caracter
    const char = data.charCodeAt(i);
    // Crear el hash
    hash = ((hash << 5) - hash) + char;
    // Conviertelo hasta 32-bit
    hash &= hash;
  }
}
```

Bien hecho:

```
function hashIt(data) {
  let hash = 0;
  const length = data.length;

  for (let i = 0; i < length; i++) {
    const char = data.charCodeAt(i);
    hash = ((hash << 5) - hash) + char;
    hash &= hash;
  }
}
```

No dejes código inutilizado en tus archivos

El control de versión existe para una razón. Deja el código viejo en tu historia (git).

Mal hecho:

```
hazAlgo()
// hazMasCosas();
// hazAunMasCosas();
// hazTantasOtrasCosas();
```

Bien hecho:

```
doStuff();
```

No escribas comentarios de jornada

Ojo: ¡utiliza el control de versión (git)! No hay necesidad para el código no utilizado, comentado, y especialmente comentarios de jornada. En cambio, utiliza 'git log' para recuperar una historia de lo que has hecho.

Mal hecho:

```
/**
 * 2016-12-20: Remover monads, no los entendia bien (RM)
 * 2016-10-01: Mejorar utilizando los monads especiales (JP)
 * 2016-02-03: Remover la comprobacion de tipos de data (LI)
 * 2015-03-14: Agregar la funcion combinar (JR)
 */
function combinar(a, b) {
  return a + b;
}
```

Bien hecho:

```
function combinar(a, b) {
  return a + b;
}
```

Evitar los marcadores posicionales

Los marcadores posicionales suelen dificultar las cosas. Deja que las funciones, los nombres de tus variables, la indentación adecuada y el formatear creen una estructura visual para tu código.

Mal hecho:

```
////////////////////////////////////////////////////////////////////////////////
///
// Instanciacion del modelo de Scope
////////////////////////////////////////////////////////////////////////////////
///
$scope.modelo = {
  menu: 'foo',
  nav: 'bar'
};

////////////////////////////////////////////////////////////////////////////////
///
// Iniciar de acciones
////////////////////////////////////////////////////////////////////////////////
///
const acciones = function() {
  // ...
};
```

Bien hecho:

```
$scope.modelo = {
  menu: 'foo',
  nav: 'bar'
};

const acciones = function() {
  // ...
};
```