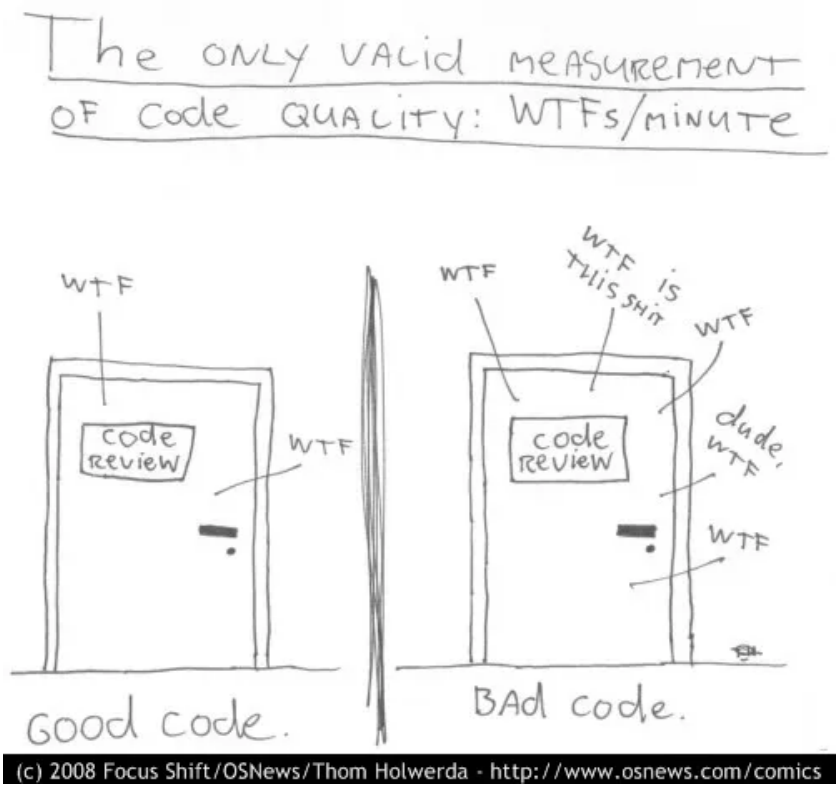
Por: *Jesús Gabriel Rivera*

En Github como `grChad`

Me he basado en el repositorio de `andersontr15/clean-code-javascript-es`

Clean Code JavasCript

1. Introducción



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*.

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!

2. 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 las 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 las variables de manera buscable y legible, acabamos confundiendo a nuestros lectores. Echa un vistazo a las herramientas para ayudarte: buddy.js y 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();
  hazMasCosas()
  // ...
  // ...
  // ...
  ejecuta(ubicacion);
});
```

No incluyas contexto innecesario

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

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 crearEmpresa(nombre) {
  const nombreEmpresa = nombre || 'Tacos S.A';
  // ...
}
```

Bien Hecho:

```
function crearEmpresa(nombreEmpresa = 'Tacos S.A') {
  // ...
}
```

3. 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 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 dificil entender del nombre lo que hace la funcion
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 tenia 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 , 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 Çomprar", un botón que invoca la función de çomprar". 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 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

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
  }
];

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)

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`, `number` y `Array` y que no puedes utilizar polimorfismo, pero existe la necesidad de comprobar los `tipos`, debes considerar utilizando **TypeScript**. Es un alternativo excelente a , 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');
```

4. 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() {
  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
```

5. 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() { /* ... */ }
}
```

Mal Hecho:

Bien Hecho: