Repositório original: ryanmcdermott/clean-code-javascript

clean-code-javascript

Índice

Introdução

Princípios da Engenharia de Software, do livro de Robert C. Martin *Código Limpo*, adaptados para JavaScript. Isto não é um guia de estilos. É um guia para se produzir código legível, reutilizável e refatorável em JavaScript.

Nem todo princípio demonstrado deve ser seguido rigorosamente, e ainda menos são um consenso universal. Estes princípios são orientações e nada mais, entretanto, foram codificados durante muitos anos de experiência coletiva pelos autores de *Código limpo*.

Nosso ofício de engenharia de software tem pouco mais de 50 anos e ainda estamos aprendendo muito. Quando a arquitetura de software for tão velha quanto a própria arquitetura, talvez então tenhamos regras mais rígidas para seguir. Por enquanto, deixe que estas orientações sirvam como critério para se avaliar a qualidade de código JavaScript que tanto você e o seu time produzirem.

Mais uma coisa: aprender isto não irá lhe transformar imediatamente em um desenvolvedor de software melhor, trabalhar com eles por muitos anos não quer dizer que você não cometerá erros. Toda porção de código começa com um rascunho, como argila molhada sendo moldada em sua forma final. Finalmente, talhamos as imperfeições quando revisamos com nossos colegas. Não se sinta culpado pelos primeiros rascunhos que ainda precisam de melhorias. Ao invés, desconte em seu código.

Variáveis

Use nomes de variáveis que tenham significado e sejam pronunciáveis

Ruim:

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

Bom:

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

⬆ voltar ao topo

Use o mesmo vocabulário para o mesmo tipo de variável

Ruim:

```
getUserInfo();
getClientData();
getCustomerRecord();
```

Bom:

```
getUser();
```

⬆ voltar ao topo

Use nomes pesquisáveis

Nós iremos ler mais código que escrever. É importante que o código que escrevemos seja legível e pesquisável. *Não* dando nomes em variáveis que sejam significativos para entender nosso programa, machucamos nossos leitores. Torne seus nomes pesquisáveis. Ferramentas como buddy.js e ESLint podem ajudar a identificar constantes sem nome.

Ruim:

```
// Para que diabos serve 86400000?
setTimeout(blastOff, 86400000);
```

Bom:

```
// Declare-as como `const` global em letras maiúsculas.
const MILLISECONDS_PER_DAY = 86400000;

setTimeout(blastOff, MILLISECONDS_PER_DAY);
```

⬆ voltar ao topo

Use variáveis explicativas

Ruim:

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

Bom:

```
const address = 'One Infinite Loop, Cupertino 95014';
const cityZipCodeRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
const [, city, zipCode] = address.match(cityZipCodeRegex) || [];
saveCityZipCode(city, zipCode);
```

⬆ voltar ao topo

Evite Mapeamento Mental

Explicito é melhor que implícito.

Ruim:

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((l) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  // Espera, para que serve o `l` mesmo?
  dispatch(l);
});
```

Bom:

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((location) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  dispatch(location);
});
```

⬆ voltar ao topo

Não adicione contextos desnecessários

Se o nome de sua classe/objeto já lhe diz alguma coisa, não as repita nos nomes de suas variáveis.

Ruim:

```
const Car = {
  carMake: 'Honda',
  carModel: 'Accord',
  carColor: 'Blue'
};

function paintCar(car, color) {
  car.carColor = color;
}
```

Bom:

```
const Car = {
  make: 'Honda',
  model: 'Accord',
  color: 'Blue'
};

function paintCar(car, color) {
  car.color = color;
}
```

⬆ voltar ao topo

Use argumentos padrões ao invés de curto circuitar ou usar condicionais

Argumentos padrões são geralmente mais limpos do que curto circuitos. Esteja ciente que se você usá-los, sua função apenas irá fornecer valores padrões para argumentos `undefined`. Outros valores "falsos" como `''`, `""`, `false`, `null`, `0`, e `NaN`, não serão substituidos por valores padrões.

Ruim:

```
function createMicrobrewery(name) {
  const breweryName = name || 'Hipster Brew Co.';
  // ...
}
```

Bom:

```
function createMicrobrewery(breweryName = 'Hipster Brew Co.') {
  // ...
}
```

⬆ voltar ao topo

Funções

Argumentos de funções (idealmente 2 ou menos)

Limitar a quantidade de parâmetros de uma função é incrivelmente importante porque torna mais fácil testá-la. Ter mais que três leva a uma explosão combinatória onde você tem que testar muitos casos diferentes com cada argumento separadamente.

Um ou dois argumentos é o caso ideal, e três devem ser evitados se possível. Qualquer coisa a mais que isso deve ser consolidada. Geralmente, se você tem mais que dois argumentos então sua função está tentando fazer muitas coisas. Nos casos em que não está, na maioria das vezes um objeto é suficiente como argumento.

Já que JavaScript lhe permite criar objetos instantaneamente, sem ter que escrever muita coisa, você pode usar um objeto se você se pegar precisando usar muitos argumentos.

Para tornar mais óbvio quais as propriedades que as funções esperam, você pode usar a sintaxe de desestruturação (destructuring) do ES2015/ES6. Ela possui algumas vantagens:

1. Quando alguém olha para a assinatura de uma função, fica imediatamente claro quais propriedades são usadas.
2. Desestruturação também clona os valores primitivos específicos do objeto passado como argumento para a função. Isso pode ajudar a evitar efeitos colaterais. Nota: objetos e vetores que são desestruturados a partir do objeto passado por argumento NÃO são clonados.
3. Linters podem te alertar sobre propriedades não utilizadas, o que seria impossível sem usar desestruturação.

Ruim:

```
function createMenu(title, body, buttonText, cancellable) {
  // ...
}
```

Bom:

```
function createMenu({ title, body, buttonText, cancellable }) {
  // ...
}

createMenu({
  title: 'Foo',
  body: 'Bar',
  buttonText: 'Baz',
  cancellable: true
});
```

⬆ voltar ao topo

Funções devem fazer uma coisa

Essa é de longe a regra mais importante em engenharia de software. Quando funções fazem mais que uma coisa, elas se tornam difíceis de serem compostas, testadas e raciocinadas. Quando você pode isolar uma função para realizar apenas uma ação, elas podem ser refatoradas facilmente e seu código ficará muito mais limpo. Se você não levar mais nada desse guia além disso, você já estará na frente de muitos desenvolvedores.

Ruim:

```
function emailClients(clients) {
  clients.forEach((client) => {
    const clientRecord = database.lookup(client);
    if (clientRecord.isActive()) {
      email(client);
    }
  });
}
```

Bom:

```
function emailActiveClients(clients) {
  clients
    .filter(isActiveClient)
    .forEach(email);
}

function isActiveClient(client) {
  const clientRecord = database.lookup(client);
  return clientRecord.isActive();
}
```

⬆ voltar ao topo

Nomes de funções devem dizer o que elas fazem

Ruim:

```
function addToDate(date, month) {
  // ...
}

const date = new Date();

// É difícil dizer pelo nome da função o que é adicionado
addToDate(date, 1);
```

Bom:

```
function addMonthToDate(month, date) {
  // ...
}

const date = new Date();
addMonthToDate(1, date);
```

⬆ voltar ao topo

Funções devem ter apenas um nível de abstração

Quando você tem mais de um nível de abstração sua função provavelmente esta fazendo coisas demais. Dividir suas funções leva a reutilização e testes mais fáceis.

Ruim:

```
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...
  });
}
```

Bom:

```
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...
  });
}
```

⬆ voltar ao topo

Remova código duplicado

Faça absolutamente seu melhor para evitar código duplicado. Código duplicado quer dizer que existe mais de um lugar onde você deverá alterar algo se precisar mudar alguma lógica.

Imagine que você é dono de um restaurante e você toma conta do seu estoque: todos os seus tomates, cebolas, alhos, temperos, etc. Se você tem multiplas listas onde guarda estas informações, então você terá que atualizar todas elas quando servir um prato que tenha tomates. Se você tivesse apenas uma lista, teria apenas um lugar para atualizar!

Frequentemente, você possui código duplicado porque você tem duas ou mais coisas levemente diferentes, que possuem muito em comum, mas suas diferenças lhe forçam a ter mais duas ou três funções que fazem muito das mesmas coisas. Remover código duplicado significa criar uma abstração que seja capaz de lidar com este conjunto de coisas diferentes com apenas uma função/módulo/classe.

Conseguir a abstração correta é crítico, por isso que você deveria seguir os princípios SOLID descritos na seção *Classes*. Abstrações ruins podem ser piores do que código duplicado, então tome cuidado! Dito isto, se você puder fazer uma boa abstração, faça-a! Não repita a si mesmo, caso contrário você se pegará atualizando muitos lugares toda vez que precisar mudar qualquer coisinha.

Ruim:

```
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);
  });
}
```

Bom:

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

    const data = {
      expectedSalary,
      experience
    };

    switch(employee.type){
      case 'manager':
        data.portfolio = employee.getMBAProjects();
        break;
      case 'developer':
        data.githubLink = employee.getGithubLink();
        break;
    }

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

⬆ voltar ao topo

Defina (set) objetos padrões com Object.assign

Ruim:

```
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);
```

Bom:

```
const menuConfig = {
  title: 'Order',
  // Usuário não incluiu a chave 'body'
  buttonText: 'Send',
  cancellable: true
};

function createMenu(config) {
  config = Object.assign({
    title: 'Foo',
    body: 'Bar',
    buttonText: 'Baz',
    cancellable: true
  }, config);

  // configuração agora é: {title: "Order", body: "Bar", buttonText: "Send", cancellable: true}
  // ...
}

createMenu(menuConfig);
```

⬆ voltar ao topo

Não use flags como parâmetros de funções

Flags falam para o seu usuário que sua função faz mais de uma coisa. Funções devem fazer apenas uma coisa. Divida suas funções se elas estão seguindo caminhos de código diferentes baseadas em um valor boleano.

Ruim:

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

Bom:

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

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

⬆ voltar ao topo

Evite Efeitos Colaterais (parte 1)

Uma função produz um efeito colateral se ela faz alguma coisa que não seja receber um valor de entrada e retornar outro(s) valor(es). Um efeito colateral pode ser escrever em um arquivo, modificar uma variável global, ou acidentalmente transferir todo seu dinheiro para um estranho.

Agora, você precisa de efeitos colaterais ocasionalmente no seu programa. Como no exemplo anterior, você pode precisar escrever em um arquivo. O que você quer fazer é centralizar aonde está fazendo isto. Não tenha diversas funções e classes que escrevam para um arquivo em particular. Tenha um serviço que faça isso. Um e apenas um.

O ponto principal é evitar armadilhas como compartilhar o estado entre objetos sem nenhuma estrutura, usando tipos de dados mutáveis que podem ser escritos por qualquer coisa, e não centralizando onde seu efeito colateral acontece. Se você conseguir fazer isto, você será muito mais feliz que a grande maioria dos outros programadores.

Ruim:

```
// Variável global referenciada pela função seguinte
// Se tivéssemos outra função que usa esse nome, então seria um vetor (array) e poderia quebrar seu código
let name = 'Ryan McDermott';

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

splitIntoFirstAndLastName();

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

Bom:

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

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

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

⬆ voltar ao topo

Evite Efeitos Colaterais (parte 2)

Em JavaScript, tipos primitivos são passados por valor e objetos/vetores são passados por referência. No caso de objetos e vetores, se sua função faz uma mudança em um vetor de um carrinho de compras, por exemplo, adicionando um item para ser comprado, então qualquer outra função que use o vetor `cart` também será afetada por essa adição. Isso pode ser ótimo, mas também pode ser ruim. Vamos imaginar uma situação ruim:

O usuário clica no botão "Comprar", botão que invoca a função `purchase` que dispara uma série de requisições e manda o vetor `cart` para o servidor. Devido a uma conexão ruim de internet, a função `purchase` precisa fazer novamente a requisição. Agora, imagine que nesse meio tempo o usuário acidentalmente clique no botão `Adicionar ao carrinho` em um produto que ele não queria antes da requisição começar. Se isto acontecer e a requisição for enviada novamente, então a função `purchase` irá enviar acidentalmente o vetor com o novo produto adicionado porque existe uma referência para o vetor `cart` que a função `addItemToCart` modificou adicionando um produto indesejado.

Uma ótima solução seria que a função `addCartToItem` sempre clonasse o vetor `cart`, editasse-o, e então retornasse seu clone. Isso garante que nenhuma outra função que possua uma referência para o carrinho de compras seja afetada por qualquer mudança feita.

Duas ressalvas desta abordagem:

1. Podem haver casos onde você realmente quer mudar o objeto de entrada, mas quando você adota este tipo de programação, você vai descobrir que estes casos são bastante raros. A maioria das coisas podem ser refatoradas para não terem efeitos colaterais.
2. Clonar objetos grandes pode ser bastante caro em termos de desempenho. Com sorte, na prática isso não é um problema, porque existem ótimas bibliotecas que permitem que este tipo de programação seja rápida e não seja tão intensa no uso de memória quanto seria se você clonasse manualmente objetos e vetores.

Ruim:

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

Bom:

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

Não escreva em funções globais

Poluir globais é uma pratica ruim em JavaScript porque você pode causar conflito com outra biblioteca e o usuário da sua API não faria a menor ideia até que ele tivesse um exceção sendo levantada em produção. Vamos pensar em um exemplo: e se você quisesse estender o método nativo Array do JavaScript para ter um método `diff` que poderia mostrar a diferença entre dois vetores? Você poderia escrever sua nova função em `Array.prototype`, mas poderia colidir com outra biblioteca que tentou fazer a mesma coisa. E se esta outra biblioteca estava apenas usando `diff` para achar a diferença entre o primeiro e último elemento de um vetor? É por isso que seria muito melhor usar as classes padrões do ES2015/ES6 e apenas estender o `Array` global.

Ruim:

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

Bom:

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

⬆ voltar ao topo

Favoreça programação funcional sobre programação imperativa

JavaScript não é uma linguagem funcional da mesma forma que Haskell é, mas tem um toque de funcional em si. Linguagens funcionais são mais limpas e fáceis de se testar. Favoreça esse tipo de programação quando puder.

Ruim:

```
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;
}
```

Bom:

```
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);
```

⬆ volta ao topo

Encapsule condicionais

Ruim:

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

Bom:

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

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

⬆ voltar ao topo

Evite negações de condicionais

Ruim:

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

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

Bom:

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

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

⬆ voltar ao topo

Evite condicionais

Esta parece ser uma tarefa impossível. Da primeira vez que as pessoas escutam isso, a maioria diz, “como eu supostamente faria alguma coisa sem usar `if` ? ” A resposta é que você pode usar polimorfismo para realizar a mesma tarefa em diversos casos. A segunda questão é geralmente, “bom, isso é ótimo, mas porque eu deveria fazer isso?” A resposta é um conceito de código limpo aprendido previamente: uma função deve fazer apenas uma coisa. Quando você tem classes e funções que tem declarações `if`, você esta dizendo para seu usuário que sua função faz mais de uma coisa. Relembre-se, apenas uma coisa.

Ruim:

```
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();
    }
  }
}
```

Bom:

```
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();
  }
}
```

⬆ voltar ao topo

Evite checagem de tipos (parte 1)

JavaScript não possui tipos, o que significa que suas funções podem receber qualquer tipo de argumento. Algumas vezes esta liberdade pode te morder, e se torna tentador fazer checagem de tipos em suas funções. Existem muitas formas de evitar ter que fazer isso. A primeira coisa a se considerar são APIs consistentes.

Ruim:

```
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'));
  }
}
```

Bom:

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

⬆ voltar ao topo

Evite checagem de tipos (parte 2)

Se você estiver trabalhando com valores primitivos básicos como strings e inteiros, e você não pode usar polimorfismo, mas ainda sente a necessidade de checar o tipo, você deveria considerar usar TypeScript. É uma excelente alternativa para o JavaScript normal, já que fornece uma tipagem estática sobre a sintaxe padrão do JavaScript. O problema com checagem manual em JavaScript é que para se fazer bem feito requer tanta verborragia extra que a falsa "tipagem-segura" que você consegue não compensa pela perca de legibilidade. Mantenha seu JavaScript limpo, escreve bons testes, e tenha boas revisões de código. Ou, de outra forma, faça tudo isso mas com TypeScript (que, como eu falei, é uma ótima alternativa!).

Ruim:

```
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');
}
```

Bom:

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

⬆ voltar ao topo

Não otimize demais

Navegadores modernos fazem muitas otimizações por debaixo dos panos em tempo de execução. Muitas vezes, se você estiver otimizando, está apenas perdendo o seu tempo. Existem bons recursos para se verificar onde falta otimização. Foque nesses por enquanto, até que eles sejam consertados caso seja possível.

Ruim:

```
// Em navegadores antigos, cada iteração de `list.length` não cacheada seria custosa
// devido a recomputação de `list.length`. Em navegadores modernos, isto é otimizado.
for (let i = 0, len = list.length; i < len; i++) {
  // ...
}
```

Bom:

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

⬆ voltar ao topo

Remova código morto

Código morto é tão ruim quanto código duplicado. Não existe nenhum motivo para deixá-lo em seu código. Se ele não estiver sendo chamado, livre-se dele. Ele ainda estará a salvo no seu histórico de versionamento se ainda precisar dele.

Ruim:

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

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

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

Bom:

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

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

⬆ voltar ao topo

Objetos e Estruturas de Dados

Use getters e setters

Usar getters e setters para acessar dados em objetos é bem melhor que simplesmente procurar por uma propriedade em um objeto. "Por quê?", você deve perguntar. Bem, aqui vai uma lista desorganizada de motivos:

- Quando você quer fazer mais além de pegar (get) a propriedade de um objeto, você não tem que procurar e mudar todos os acessores do seu código;
- Torna mais fácil fazer validação quando estiver dando um `set`;
- Encapsula a representação interna;

- Mais fácil de adicionar logs e tratamento de erros quando dando get and set;
- Você pode usar lazy loading nas propriedades de seu objeto, digamos, por exemplo, pegando ele de um servidor.

Ruim:

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

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

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

Bom:

```
function makeBankAccount() {
  // este é privado
  let balance = 0;

  // um "getter", feito público através do objeto retornado abaixo
  function getBalance() {
    return balance;
  }

  // um "setter", feito público através do objeto retornado abaixo
  function setBalance(amount) {
    // ... validate before updating the balance
    balance = amount;
  }

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

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

⬆ voltar ao topo

Faça objetos terem membros privados

Isto pode ser alcançado através de closures (para ES5 e além).

Ruim:

```
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
```

Bom:

```
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
```

⬆ voltar ao topo

Classes

Prefira classes do ES2015/ES6 ao invés de funções simples do ES5

É muito difícil conseguir que herança de classe, construtores, e definições de métodos sejam legíveis para classes de ES5 clássicas. Se você precisa de herança (e esteja ciente que você talvez não precise), então prefira classes ES2015/ES6. Entretanto, prefira funções pequenas ao invés de classes até que você precise de objetos maiores e mais complexos.

Ruim:

```
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() {};
```

Bom:

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

⬆ voltar ao topo

Use encadeamento de métodos

Este padrão é muito útil em JavaScript e você o verá em muitas bibliotecas como jQuery e Lodash. Ele permite que seu código seja expressivo e menos verboso. Por esse motivo, eu digo, use encadeamento de métodos e dê uma olhada em como o seu código ficará mais limpo. Em suas funções de classes, apenas retorne `this` no final de cada função, e você poderá encadear mais métodos de classe nele.

Ruim:

```
class Car {
  constructor(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
  }

  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('Ford','F-150','red');
car.setColor('pink');
car.save();
```

Bom:

```
class Car {
  constructor(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
  }

  setMake(make) {
    this.make = make;
    // NOTA: Retorne this para encadear
    return this;
  }

  setModel(model) {
    this.model = model;
    // NOTA: Retorne this para encadear
    return this;
  }

  setColor(color) {
    this.color = color;
    // NOTA: Retorne this para encadear
    return this;
  }

  save() {
    console.log(this.make, this.model, this.color);
    // NOTA: Retorne this para encadear
    return this;
  }
}

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

⬆ voltar ao topo

Prefira composição ao invés de herança

Como dito famosamente em *Padrão de projeto* pela Gangue dos Quatro, você deve preferir composição sobre herança onde você puder. Existem muitas boas razões para usar herança e muitas boas razões para se usar composição. O ponto principal para essa máxima é que se sua mente for instintivamente para a herança, tente pensar se composição poderia modelar melhor o seu problema. Em alguns casos pode.

Você deve estar pensando então, "quando eu deveria usar herança?" Isso depende especificamente do seu problema, mas essa é uma lista decente de quando herança faz mais sentido que composição:

1. Sua herança representa uma relação de "isto-é" e não uma relação de "isto-tem" (Human→Animal vs. User->UserDetails)
2. Você pode reutilizar código de classes de base (Humanos podem se mover como todos os animais).
3. Você quer fazer mudanças globais para classes derivadas mudando apenas a classe base. (Mudar o custo calórico para todos os animais quando se movem).

Ruim:

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

  // ...
}

// Ruim porque Employees (Empregados) "tem" dados de impostos. EmployeeTaxData não é um tipo de Employee
class EmployeeTaxData extends Employee {
  constructor(ssn, salary) {
    super();
    this.ssn = ssn;
    this.salary = salary;
  }

  // ...
}
```

Bom:

```
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);
  }
  // ...
}
```

⬆ voltar ao topo

SOLID

Princípio da Responsabilidade Única (SRP)

Como dito em Código Limpo, "Nunca deveria haver mais de um motivo para uma classe ter que mudar". É tentador empacotar uma classe em excesso com muitas funcionalidades, como quando você pode levar apenas uma mala em seu voo. O problema com isso é que sua classe não será conceitualmente coesa e dar-lhe-á diversos motivos para mudá-la. Minimizar o número de vezes que você precisa mudar uma classe é importante, porque, se muitas funcionalidades estão em uma classe e você mudar uma porção dela, pode ser difícil entender como isto afetará outras módulos que dependem dela no seu código.

Ruim:

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

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

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

Bom:

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

⬆ voltar ao topo

Princípio do Aberto/Fechado (OCP)

Como foi dito por Bertrand Meyer, "entidades de software (classes, módulos, funções, etc.) devem se manter abertas para extensões, mas fechadas para modificações." Mas o que isso significa? Esse princípio basicamente diz que você deve permitir que usuários adicionem novas funcionalidades sem mudar código já existente.

Ruim:

```
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) => {
        // transforma a resposta e retorna
      });
    } else if (this.adapter.name === 'httpNodeAdapter') {
      return makeHttpCall(url).then((response) => {
        // transforma a resposta e retorna
      });
    }
  }
}

function makeAjaxCall(url) {
  // faz a request e retorna a promessa
}

function makeHttpCall(url) {
  // faz a request e retorna a promessa
}
```

Bom:

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

  request(url) {
    // faz a request e retorna a promessa
  }
}

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

  request(url) {
    // faz a request e retorna a promessa
  }
}

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

  fetch(url) {
    return this.adapter.request(url).then((response) => {
      // transforma a resposta e retorna
    });
  }
}
```

⬆ voltar ao topo

Princípio de Substituição de Liskov (LSP)

Esse é um termo assustador para um conceito extremamente simples. É formalmente definido como “Se S é um subtipo de T, então objetos do tipo T podem ser substituídos por objetos com o tipo S (i.e., objetos do tipo S podem substituir objetos do tipo T) sem alterar nenhuma das propriedades desejáveis de um programa (corretude, desempenho em tarefas, etc.).” Esta é uma definição ainda mais assustadora.

A melhor explicação para este conceito é se você tiver uma classe pai e uma classe filha, então a classe base e a classe filha pode ser usadas indistintamente sem ter resultados incorretos. Isso ainda pode ser confuso, então vamos dar uma olhada no exemplo clássico do Quadrado-Retângulo (Square-Rectangle). Matematicamente, um quadrado é um retângulo, mas se você modelá-lo usando uma relação “isto-é” através de herança, você rapidamente terá problemas.

Ruim:

```
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(); // RUIM: Retorna 25 para o Quadrado. Deveria ser 20.
    rectangle.render(area);
  });
}

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

Bom:

```
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);
```

⬆ voltar ao topo

Princípio da Segregação de Interface (ISP)

JavaScript não possui interfaces então esse princípio não se aplica estritamente como os outros. Entretanto, é importante e relevante até mesmo com a falta de um sistema de tipos em JavaScript.

ISP diz que "Clientes não devem ser forcados a depender de interfaces que eles não usam." Interfaces são contratos implícitos em JavaScript devido a sua tipagem pato (duck typing).

Um bom exemplo para se observar que demonstra esse princípio em JavaScript é de classes que requerem objetos de configurações grandes. Não pedir para clientes definirem grandes quantidades de opções é benéfico, porque na maioria das vezes eles não precisarão de todas as configurações. Torná-las opcionais ajuda a prevenir uma “interferência gorda”.

Ruim:

```
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() {} //  Na maioria das vezes, não precisamos animar enquanto atravessamos (traversing).
  // ...
});
```

Bom:

```
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() {}
  }
});
```

⬆ voltar ao topo

Princípio da Inversão de Dependência (DIP)

Este princípio nos diz duas coisas essenciais: 1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. 2. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Isso pode ser difícil de entender a princípio, mas se você já trabalhou com AngularJS, você já viu uma implementação deste princípio na forma de injeção de dependência (DI). Apesar de não serem conceitos idênticos, DIP não deixa módulos de alto nível saber os detalhes de seus módulos de baixo nível, assim como configurá-los. Isso pode ser alcançado através de DI. Um grande beneficio é que reduz o acoplamento entre os módulos. Acoplamento é um padrão de desenvolvimento muito ruim porque torna seu código mais difícil de ser refatorado.

Como dito anteriormente, JavaScript não possui interfaces, então as abstrações que são necessárias são contratos implícitos. Que quer dizer que, os métodos e as classes que um objeto/classe expõe para outros objeto/classe. No exemplo abaixo, o contrato implícito é que qualquer módulo de Request para `InventoryTracker` terá um método `requestItems`:

Ruim:

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

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

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

    // Ruim: Nós criamos uma dependência numa implementação de request especifica.
    // Nós deveriamos apenas ter requestItems dependendo de um método de request: `request`
    this.requester = new InventoryRequester();
  }

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

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

Bom:

```
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) {
    // ...
  }
}

// Construindo nossas dependências externamente e injetando-as, podemos facilmente
// substituir nosso módulo de request por um novo mais chique que usa WebSockets
const inventoryTracker = new InventoryTracker(['apples', 'bananas'], new InventoryRequesterV2());
inventoryTracker.requestItems();
```

⬆ voltar ao topo

Testes

Testes são mais importantes que entregas. Se você não possui testes ou um quantidade inadequada, então toda vez que você entregar seu código você não terá certeza se você não quebrou alguma coisa. Decidir o que constitui uma quantidade adequada é responsabilidade do seu time, mas ter 100% de cobertura (todas as sentenças e branches) é a maneira que se alcança uma alta confiança e uma paz de espirito em desenvolvimento. Isso quer dizer que além de ter um ótimo framework de testes, você também precisa usar uma boa ferramenta de cobertura.

Não existe desculpa para não escrever testes. Existem diversos frameworks de testes em JS ótimos, então encontre um que seu time prefira. Quando você encontrar um que funciona para seu time, então tenha como objetivo sempre escrever testes para cada nova funcionalidade/módulo que você introduzir. Se seu método preferido for Desenvolvimento Orientado a Testes (TDD), isso é ótimo, mas o ponto principal é apenas ter certeza que você está alcançado suas metas de cobertura antes de lançar qualquer funcionalidade, ou refatorar uma já existente.

Um conceito por teste

Ruim:

```
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);
  });
});
```

Bom:

```
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);
  });
});
```

⬆ voltar ao topo

Concorrência

Use Promessas, não callbacks

Callbacks não são limpos, e eles causam uma quantidade excessiva de aninhamentos. A partir de ES2015/ES6, Promessas são um tipo nativo global. Use-as!

Ruim:

```
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');
      }
    });
  }
});
```

Bom:

```
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);
  });
```

⬆ voltar ao topo

Async/Await são ainda mais limpas que Promessas

Promessas são uma alternativa bem mais limpa que callbacks, mas o ES2017/ES8 traz `async` e `await` que oferecem uma solução ainda mais limpa. Tudo o que você precisa é uma função que tem como prefixo a palavra-chave `async`, e então você pode escrever sua logica imperativamente sem usar `then` para encadear suas funções. Use isto se você puder tirar vantagem das funcionalidades do ES2017/ES8 hoje!

Ruim:

```
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);
  });
```

Bom:

```
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);
  }
}
```

⬆ voltar ao topo

Tratamento de Erros

`throw error` é uma coisa boa! Eles significam que o programa identificou com sucesso quando algo deu errado e está permitindo que você saiba parando a execução da função no processo atual, fechando o processo (em Node), e notificando você no console com a pilha de processos.

Não ignore erros capturados

Não fazer nada com um erro capturado não te dá a habilidade de resolvê-lo ou reagir ao erro informado. Exibir um log no console(`console.log`) não é muito melhor porque muitas vezes ele pode ficar perdido entre um monte de outras coisas impressas no console. Se você envolver qualquer pedaço de código em um `try/catch` isso significa que você acredita que um erro pode ocorrer lá e então você deveria ter um plano, ou criar caminho de código para quando isso ocorrer.

Ruim:

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

Bom:

```
try {
  functionThatMightThrow();
} catch (error) {
  // Uma opção (mais chamativa que console.log):
  console.error(error);
  // Outra opção:
  notifyUserOfError(error);
  // Outra opção:
  reportErrorToService(error);
  // OU as três!
}
```

Não ignore promessas rejeitadas

Pela mesma razão que você não deveria ignorar erros caputados de `try/catch`

Ruim:

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

Bom:

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

⬆ voltar ao topo

Formatação

Formatação é subjetiva. Como muitas regras aqui, não há nenhuma regra fixa e rápida que você precisa seguir. O ponto principal é NÃO DISCUTA sobre formatação. Existem muitas ferramentas para automatizar isso. Utilize uma! É um desperdicio de tempo e dinheiro para engenheiros discutirem sobre formatação.

Para coisas que não possam utilizar formatação automática (identação, tabs vs. espaços, aspas simples vs. duplas, etc.) olhe aqui para alguma orientação.

Utilize capitalização consistente

JavaScript não é uma linguagem tipada, então a capitalização diz muito sobre suas variáveis, funções, etc. Estas regras são subjetivas, então sua equipe pode escolher o que quiserem. O ponto é, não importa o que vocês todos escolham, apenas seja consistente.

Ruim:

```
 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 {}
```

Bom:

```
 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 {}
```

⬆ voltar ao topo

Funções e chamadas de funções devem estar próximas

Se uma função chamar outra, mantenha estas funções verticalmente próximas no arquivo fonte. Em um cenário ideal, manter a chamada logo acima da função. Nós tendemos a ler códigos de cima para baixo, como num jornal. Por causa disso, faça o seu código desta maneira.

Ruim:

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

Bom:

```
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();
```

⬆ voltar ao topo

Comentários

Apenas comente coisas que tenham complexidade de lógica de negócio.

Comentários são uma desculpa, não um requisito. Um bom código documenta-se, *a maior parte*, por si só.

Ruim:

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

  // Tamanho da string
  const length = data.length;

  // Loop em cada caracter da informação
  for (let i = 0; i < length; i++) {
    // Pega o código do caracter.
    const char = data.charCodeAt(i);
    // Cria a hash
    hash = ((hash << 5) - hash) + char;
    // Converte para um integer 32-bit
    hash &= hash;
  }
}
```

Bom:

```
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;

    // Converte para um integer 32-bit
    hash &= hash;
  }
}
```

⬆ voltar ao topo

Não deixe código comentado na sua base de código

Controle de versão existe por uma razão. Deixar códigos velhos no seu histórico.

Ruim:

```
doStuff();
// doOtherStuff();
// doSomeMoreStuff();
// doSoMuchStuff();
```

Bom:

```
doStuff();
```

⬆ voltar ao topo

Não comente registro de alterações

Lembre-se, utilize controle de versão! Não tem necessidade em deixar códigos inutlizados, códigos comentados e especialmente registros de alterações. Utilize `git log` para pegar o histórico!

Ruim:

```
/**
 * 2016-12-20: Removidas monads, não entendia elas (RM)
 * 2016-10-01: Melhoria utilizando monads especiais (JP)
 * 2016-02-03: Removido checagem de tipos (LI)
 * 2015-03-14: Adicionada checagem de tipos (JR)
 */
function combine(a, b) {
  return a + b;
}
```

Bom:

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

⬆ voltar ao topo

Evite marcadores de posição

Eles geralmente criam ruídos. Deixe que as funções e nomes de variáveis em conjunto com a devida identação e formatação deem a estrutura visual para o seu código.

Ruim:

```
////////////////////////////////////////////////////////////////////////////////
// Intanciação do Scope Model
////////////////////////////////////////////////////////////////////////////////
$scope.model = {
  menu: 'foo',
  nav: 'bar'
};

////////////////////////////////////////////////////////////////////////////////
// Configuração da Action
////////////////////////////////////////////////////////////////////////////////
const actions = function() {
  // ...
};
```

Bom:

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

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

⬆ voltar ao topo

Traduções

Existem traduções disponíveis em outras linguas:

- **Inglês**: ryanmcdermott/clean-code-javascript
- **Espanhol**: andersontr15/clean-code-javascript
- **Chinês**:
  - alivebao/clean-code-js
  - beginor/clean-code-javascript
- **Alemão**: marcbruederlin/clean-code-javascript
- **Coreano**: qkraudghgh/clean-code-javascript-ko
- **Polaco**: greg-dev/clean-code-javascript-pl
- **Russo**:
  - BoryaMogila/clean-code-javascript-ru/
  - maksugr/clean-code-javascript
- **Vietnamita**: hienvd/clean-code-javascript/
- **Japonês**: mitsuruog/clean-code-javascript/
- **Indonésio**: andirkh/clean-code-javascript/