Clean Code cho Javascript

Edit

Add topics

45 commits | 1 branch | 0 releases | 3 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is even with hienvd:master. Pull request Compare

hienvd committed on GitHub Merge pull request #14 from quangphuc789/master ... Latest commit 44dff03 on Mar 1

.gitattributes	Create .gitattributes	2 months ago
LICENSE	Create LICENSE	3 months ago
README.md	updated Chinese Version	a month ago

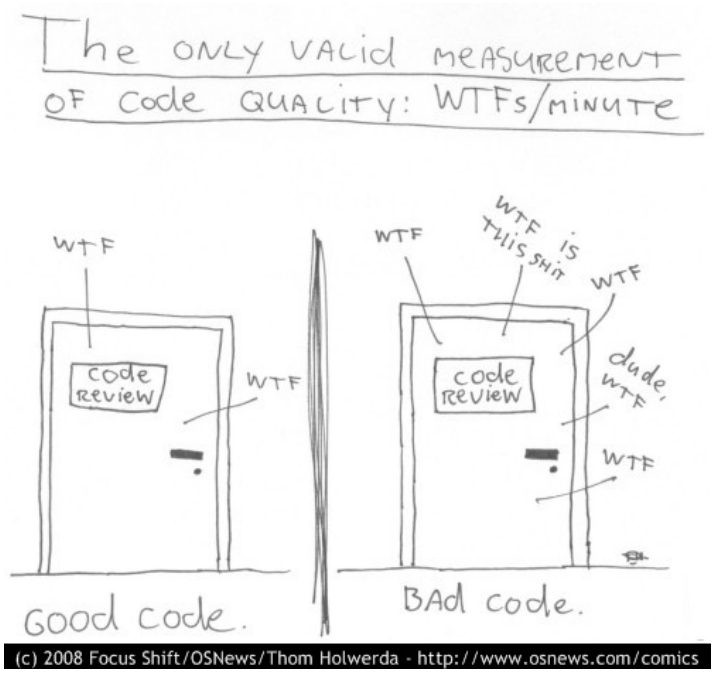
README.md

Original Repository: ryanmcdermott/clean-code-javascript

clean-code-javascript

Mục lục

Giới thiệu



Những nguyên tắc kỹ thuật phần mềm, từ cuốn sách *Clean Code* của Robert C. Martin's, được áp dụng cho ngôn ngữ JavaScript. Đây không phải là một hướng dẫn về cách viết code Javascript mà là hướng dẫn về cách viết các đoạn code dễ đọc hiểu, tái sử dụng và tái cấu trúc được trong Javascript.

Không phải mọi nguyên tắc ở đây phải được tuân thủ một cách nghiêm ngặt, và thậm chí chỉ có một ít trong số đó được sử dụng phổ biến. Ở đây, nó chỉ là một hướng dẫn - không hơn không kém, nhưng chúng được hệ thống hóa thông qua kinh nghiệm thu thập được qua nhiều năm của các tác giả của cuốn sách *Clean Code*

Ngành kỹ thuật phần mềm chỉ phát triển được hơn 50 năm, và chúng ta vẫn đang học rất nhiều. Một khi kiến trúc phần mềm trở thành phổ biến, có lẽ sau đó chúng ta sẽ có thêm nhiều luật lệ khó hơn phải tuân theo. Còn giờ đây, hãy để những hướng dẫn này như là một tiêu chuẩn để đánh giá chất lượng các đoạn code Javascript mà bạn và team của bạn tạo ra.

Biết những hướng dẫn này thôi sẽ không thể ngay lập tức làm bạn trở thành một lập trình viên phần mềm tốt hơn được, và làm việc với chúng trong nhiều năm cũng không có nghĩa bạn sẽ không gặp bất cứ sai lầm nào. Mỗi đoạn code bắt đầu như một bản thảo đầu tiên, giống như đất sét được nặn nhào và cho tới cuối cùng thì nó sẽ lộ diện hình hài. Cuối cùng, chúng ta gọt tỉa những khuyết điểm khi chúng ta xem xét lại nó cùng với các đồng nghiệp. Đừng để bản thân bạn bị đánh bại bởi những bản thảo đầu tiên, thứ mà vẫn cần phải được chỉnh sửa. Thay vào đó hãy đánh bại những dòng code.

Biến

Sử dụng tên biến có nghĩa và dễ phát âm

Không tốt:

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

Tốt:

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

về đầu trang

Sử dụng cùng từ vựng cho cùng loại biến

Không tốt:

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

Tốt:

```
getUser();
```

về đầu trang

Sử dụng các tên có thể tìm kiếm được

Chúng ta sẽ đọc code nhiều hơn là viết chúng. Điều quan trọng là code chúng ta viết có thể đọc được và tìm kiếm được. Việc đặt tên các biến *không* có ngữ nghĩa so với chương trình, chúng ta có thể sẽ làm người đọc code bị tổn thương tinh thần. Hãy làm cho các tên biến của bạn có thể tìm kiếm được. Các công cụ như buddy.js và ESLint có thể giúp nhận ra các hằng chưa được đặt tên.

Không tốt:

```
// 86400000 là cái quái gì thế?
setTimeout(blastOff, 86400000);
```

Tốt:

```
// Khai báo chúng như một biến global.
const MILLISECONDS_IN_A_DAY = 86400000;

setTimeout(blastOff, MILLISECONDS_IN_A_DAY);
```

về đầu trang

Sử dụng những biến có thể giải thích được

Không tốt:

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

Tốt:

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

về đầu trang

Tránh hại não người khác

Tường minh thì tốt hơn là ẩn.

Không tốt:

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((l) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  // Khoan, `l` làm cái gì vậy?
  dispatch(l);
});
```

Tốt:

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

về đầu trang

Đừng thêm những ngữ cảnh không cần thiết

Nếu tên của lớp hay đối tượng của bạn đã nói lên điều gì đó rồi, đừng lặp lại điều đó trong tên biến nữa.

Không tốt:

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

function paintCar(car) {
  car.carColor = 'Red';
}
```

Tốt:

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

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

về đầu trang

Sử dụng những tham số mặc định thay vì kiểm tra các điều kiện lòng vòng

Không tốt:

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

Tốt:

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

về đầu trang

Hàm

Đối số của hàm (lý tưởng là ít hơn hoặc bằng 2)

Giới hạn số lượng param của hàm là một điều cực kì quan trọng bởi vì nó làm cho hàm của bạn trở nên dễ test hơn. Trường hợp có nhiều hơn 3 params có thể dẫn đến việc bạn phải test hàng tấn test case khác nhau với những đối số

riêng biệt.

1 hoặc 2 đối số là trường hợp lý tưởng, còn trường hợp 3 đối số thì nên tránh nếu có thể. Những trường hợp khác (từ 3 params trở lên) thì nên được gộp lại. Thông thường nếu có nhiều hơn 2 đối số thì hàm của bạn đang cố thực hiện quá nhiều việc rồi đấy. Trong trường hợp ngược lại, phần lớn thời gian một đối tượng cấp cao sẽ là đủ để làm đối số.

Kể từ khi Javascript cho phép tạo nhiều đối tượng một cách nhanh chóng, mà không cần nhiều lớp có sẵn, bạn có thể sử dụng một đối tượng nếu bạn cần truyền nhiều đối số.

Để làm cho rõ ràng một hàm mong đợi những thuộc tính gì, bạn có thể sử dụng cấu trúc destructuring của ES6. Điều này có một số ưu điểm:

1. Khi một ai đó nhìn vào hàm, những thuộc tính nào được sử dụng sẽ trở nên rõ ràng ngay lập tức.
2. Destructuring cũng sao chép lại các giá trị ban đầu được chỉ định của đối tượng đối số được truyền vào hàm. Điều này có thể giúp ngăn chặn các ảnh hưởng phụ. Chú ý: các đối tượng và mảng được destructure từ đối tượng đối số thì không được sao chép lại.
3. Linter có thể sẽ cảnh báo bạn về những thuộc tính không sử dụng, điều mà không thể xảy ra nếu không có destructuring.

Không tốt:

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

Tốt:

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

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

Về đầu trang

Hàm chỉ nên giải quyết một vấn đề

Đây là quy định quan trọng nhất của kỹ thuật phần mềm. Khi một hàm thực hiện nhiều hơn 1 việc, chúng sẽ trở nên khó khăn hơn để viết code, test, và suy luận. Khi bạn có thể tách biệt một hàm để chỉ thực hiện một hành động, thì sẽ dễ dàng hơn để tái cấu trúc và code của bạn sẽ dễ đọc hơn nhiều. Nếu bạn chỉ cần làm theo hướng dẫn này thôi mà không cần làm gì khác thì bạn cũng đã giỏi hơn nhiều developer khác rồi.

Không tốt:

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

Tốt:

```
function emailClients(clients) {
  clients
    .filter(isClientActive)
    .forEach(email);
}

function isClientActive(client) {
```

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

Về đầu trang

Tên hàm phải nói ra được những gì chúng làm

Không tốt:

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

const date = new Date();

// Khó để biết được hàm này thêm gì thông qua tên hàm.
addToDate(date, 1);
```

Tốt:

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

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

Về đầu trang

Hàm chỉ nên có một lớp trừu tượng

Khi có nhiều hơn một lớp trừu tượng thì hàm của bạn đang làm quá nhiều. Chia nhỏ các hàm ra sẽ làm cho việc test và tái sử dụng dễ dàng hơn.

Không tốt:

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

Tốt:

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

Về đầu trang

Xóa code trùng lặp

Tuyệt đối tránh những dòng code trùng lặp. Code trùng lặp thì không tốt bởi vì nếu bạn cần thay đổi cùng một logic, bạn phải sửa ở nhiều hơn một nơi.

Hãy tưởng tượng nếu bạn điều hành một nhà hàng và bạn theo dõi hàng tồn kho: bao gồm cà chua, hành tây, tỏi, gia vị, vv.... Nếu bạn có nhiều danh sách quản lý, thì tất cả chúng phải được thay đổi khi bạn phục vụ một món ăn có chứa cà chua. Nếu bạn chỉ có 1 danh sách, thì việc cập nhật ở một nơi thôi.

Thông thường, bạn có những dòng code lặp lại bởi vì bạn có 2 hay nhiều hơn những thứ chỉ khác nhau chút ít, mà chia sẻ nhiều thứ chung, nhưng sự khác nhau của chúng buộc bạn phải có 2 hay nhiều hàm riêng biệt để làm nhiều điều tương tự nhau. Xóa đi những dòng code trùng có nghĩa là tạo ra một abstraction có thể xử lý tập những điểm khác biệt này chỉ với một hàm/module hay class.

Có được một abstraction đúng thì rất quan trọng, đó là lý do tại sao bạn nên tuân thủ các nguyên tắc SOLID được đặt ra trong phần *Lớp*. Những abstraction không tốt có thể còn tệ hơn cả những dòng code bị trùng lặp, vì thế hãy cẩn thận! Nếu bạn có thể tạo ra một abstraction tốt, hãy làm nó! Đừng lặp lại chính mình, nếu bạn không muốn đi cập nhật nhiều nơi bất cứ khi nào bạn muốn thay đổi một thứ gì đó.

Không tốt:

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

Tốt:

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

Về đầu trang

Thiết lập những đối tượng mặc định với Object.assign

Không tốt:

```
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);
```

Tốt:

```
const menuConfig = {
  title: 'Order',
  // User did not include 'body' key
  buttonText: 'Send',
  cancellable: true
};

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

  // config now equals: {title: "Order", body: "Bar", buttonText: "Send", cancellable: true}
  // ...
}
```

```
createMenu(menuConfig);
```

Về đầu trang

Đừng sử dụng các cờ như đối số của hàm

Các biến cờ cho người dùng của bạn biết rằng hàm thực hiện nhiều hơn một việc. Hàm chỉ nên làm một nhiệm vụ. Vì vậy hãy tách hàm của bạn nếu chúng đang làm cho code rẽ nhánh dựa trên một biến boolean.

Không tốt:

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

Tốt:

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

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

Về đầu trang

Tránh những ảnh hưởng phụ (phần 1)

Một hàm tạo ra ảnh hưởng phụ nếu nó làm bất kì điều gì khác hơn là nhận một giá trị đầu vào và trả về một hoặc nhiều giá trị. Ảnh hưởng phụ có thể là ghi một file, thay đổi vài biến toàn cục, hoặc vô tình đưa tất cả tiền của bạn cho một người lạ.

Bây giờ, cũng có khi bạn cần ảnh hưởng phụ trong một chương trình. Giống như ví dụ trước, bạn cần ghi một file. Những gì bạn cần làm là tập trung vào nơi bạn sẽ làm nó. Đừng viết hàm và lớp riêng biệt để tạo ra một file cụ thể. Hãy có một service để viết nó. Một và chỉ một.

Điểm chính là để tránh những lỗi chung như chia sẻ trạng thái giữa những đối tượng mà không có bất kì cấu trúc nào, sử dụng các kiểu dữ liệu có thể thay đổi được mà có thể được ghi bởi bất cứ thứ gì, và không tập trung nơi có thể xảy ra các ảnh hưởng phụ. Nếu bạn có thể làm điều đó, bạn sẽ hạnh phúc hơn so với phần lớn các lập trình viên khác đấy.

Không tốt:

```
// Biến toàn cục được tham chiếu bởi hàm dưới đây.
// Nếu chúng ta có một hàm khác sử dụng name, nó sẽ trở thành một array
let name = 'Ryan McDermott';

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

splitIntoFirstAndLastName();

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

Tốt:

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

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

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

Về đầu trang

Tránh những ảnh hưởng phụ (phần 2)

Trong JavaScript, các kiểu cơ bản được truyền theo giá trị và các đối tượng/mảng được truyền theo tham chiếu. Trong trường hợp các đối tượng và mảng, ví dụ nếu hàm của chúng ta tạo ra thay đổi trong một mảng giỏ mua hàng, ví dụ thêm một sản phẩm để mua, thì bất kì hàm khác mà sử dụng mảng 'giỏ hàng' sẽ bị ảnh hưởng bởi việc thêm này. Điều này có thể tốt, tuy nhiên nó cũng có thể trở nên tồi tệ. Hãy tưởng tượng trường hợp xấu sau:

Người sử dụng nhấp chuột "Mua hàng", nút mua hàng sẽ gọi tới hàm `mua`, cái mà sinh ra một yêu cầu mạng và gửi mảng `giỏ` lên server. Do kết nối chậm, hàm `mua` có thể giữ việc thử lại yêu cầu. Bây giờ, nếu trong thời gian đó người sử dụng vô tình nhấn chuột vào nút "Thêm vào giỏ hàng" ở một sản phẩm mà họ không thực sự muốn trước khi mạng thực hiện yêu cầu? Nếu điều đó xảy ra và mạng bắt đầu gửi yêu cầu thì hàm mua sẽ vô tình thêm một sản phẩm vì nó có một tham chiếu đế mảng giỏ hàng mà hàm `thêm sản phẩm vào giỏ hàng` đã thay đổi bằng cách thêm một sản phẩm mà họ không muốn.

Một giải pháp tốt là hàm `thêm sản phẩm vào giỏ hàng` luôn luôn tạo một bản sao của `giỏ`, thay đổi nó, và trả về bản sao đó. Điều này đảm bảo rằng không một hàm nào có nắm giữ tham chiếu của giỏ mua hàng bị ảnh hưởng bởi bất kì thay đổi.

Hai lưu ý cho cách tiếp cận này:

1. Có thể có những trường hợp mà bạn thực sự muốn thay đổi đối tượng đầu vào, nhưng khi bạn áp dụng phương pháp này bạn sẽ thấy những trường hợp này thì hiếm. Hầu hết các vấn đề có thể được cấu trúc lại để không còn ảnh hưởng phụ.
2. Nhân bản các đối tượng lớn có thể ảnh hưởng đến hiệu năng. May mắn thay, đó không phải là một vấn đề lớn trong thực tế bởi vì có immutable-js cho phép cách tiếp cận này trở nên nhanh và ít tốn bộ nhớ so với khi bạn tự sao chép những đối tượng và mảng.

Không tốt:

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

Tốt:

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

Về đầu trang

Đừng ghi lên những hàm toàn cục

Gây ảnh hưởng đến các biến toàn cục là một bad practice trong JavaScript vì bạn có thể xung đột với các thư viện khác và người dùng API của bạn sẽ không biết trước được cho đến khi cho một lỗi xảy ra trên sản phẩm. Hãy suy nghĩ ví dụ này: điều gì xảy ra nếu bạn muốn mở rộng phương thức của Array trong JavaScript native để có thể có một hàm `diff` chỉ ra sự khác nhau giữa hai mảng? Bạn có thể viết một hàm mới với `Array.prototype`, nhưng nó có thể xung đột với một thư viện khác mà đã làm những điều tương tự. Điều gì xảy ra nếu thư viện đó chỉ sử dụng `diff` để tìm sự khác biệt giữa phần tử đầu tiên và cuối cùng của một mảng? Đó là lý do tại sao sẽ là tốt hơn nhiều khi chỉ sử dụng các lớp ES2015/ES6 và đơn giản mở rộng `Array` toàn cục.

Không tốt:

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

Tốt:

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

Về đầu trang

Ủng hộ lập trình hàm hơn là lập trình mệnh lệnh

JavaScript không phải là ngôn ngữ lập trình hàm giống như là Haskell, nhưng nó có đặc trưng hàm của nó. Những ngôn ngữ lập trình hàm thì gọn gàng hơn và dễ test hơn. Hãy dùng cách lập trình này khi bạn có thể.

Không tốt:

```
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;
}
```

Tốt:

```
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);
```

Về đầu trang

Đóng gói các điều kiện

Không tốt:

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

Tốt:

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

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

Về đầu trang

Tránh điều kiện tiêu cực

Không tốt:

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

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

Tốt:

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

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

Về đầu trang

Tránh điều kiện

Đây dường như là một việc bất khả thi. Khi nghe điều này đầu tiên, hầu hết mọi người đều nói, "Làm sao tôi cần phải làm gì mà không có mệnh đề `if` ?" Câu trả lời là bạn có thể sử dụng tính đa hình để đạt được công việc tương tự trong rất nhiều trường hợp. Câu hỏi thứ hai thường là "Đó là điều tốt nhưng tại sao tôi lại muốn làm điều đó?" Câu trả lời là khái niệm mà ta đã học ở phần trước: một hàm chỉ nên thực hiện một việc. Khi bạn có nhiều lớp và hàm mà có nhiều mệnh đề `if` , bạn đang cho người dùng của bạn biết rằng hàm của bạn đang làm nhiều hơn một việc. Hãy nhớ, chỉ làm một công việc thôi.

Không tốt:

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

Tốt:

```
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();
  }
}
```

Về đầu trang

Tránh kiểm tra kiểu (phần 1)

JavaScript không định kiểu, có nghĩa hàm của bạn có thể nhận bất kì đối số kiểu nào. Đôi khi bạn bị cám dỗ bởi sự tự do này và dễ dẫn đến việc đi kiểm tra kiểu trong hàm của mình. Có nhiều cách để tránh phải làm điều này. Điều đầu tiên là xem xét sử dụng các API nhất quán.

Không tốt:

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

Tốt:

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

Về đầu trang

Tránh kiểm tra kiểu (phần 2)

Nếu bạn làm việc với các kiểu cơ bản như chuỗi, số nguyên và mảng, và bạn không thể sử dụng đa hình nhưng bạn vẫn cảm thấy cần phải kiểm tra kiểu, bạn nên xem xét sử dụng TypeScript. Nó là một phương pháp thay thế tuyệt vời cho JavaScript thường, vì nó cung cấp kiểu tĩnh ngoài cú pháp JavaScript chuẩn. Vấn đề với việc kiểm tra kiểu thủ công là để làm tốt việc này đòi hỏi nhiều sự dài dòng mà "kiểu an toàn" giả này không thay thế được cho việc mất đi tính dễ đọc của code. Hãy giữ code JavaScript của bạn sạch sẽ, viết test tốt và có reviews code tốt. Nếu không thì thực hiện tất cả những điều đó nhưng với TypeScript (giống như tôi đã nói, đó là sự thay thế tốt!).

Không tốt:

```
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');
}
```

Tốt:

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

Về đầu trang

Đừng quá tối ưu

Những trình duyệt hiện đại làm rất nhiều tối ưu hóa bên dưới trong thời gian chạy. Rất nhiều lần, nếu bạn đang tối ưu thì bạn đang làm tốn thời gian của chính mình. Xem ở đây để biết khi nào việc tối ưu hóa là thiếu. Hãy thực hiện những tối ưu đó và cho đến khi chúng được sửa nếu có thể.

Không tốt:

```
// Trên các trình duyệt cũ, mỗi lần lặp với 'list.length` chưa được cache sẽ tốn kém
// vì `list.length` sẽ được tính lại. Trong các trình duyệt hiện đại, điều này đã được tối ưu.
for (let i = 0, len = list.length; i < len; i++) {
  // ...
}
```

Tốt:

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

Về đầu trang

Xóa code chết (dead code)

Dead code cũng tệ như code trùng lặp. Không có lý do gì để giữ chúng lại trong codebase của bạn. Nếu nó không được gọi nữa, hãy bỏ nó đi! Nó vẫn sẽ nằm trong lịch sử phiên bản của bạn nếu bạn vẫn cần nó.

Không tốt:

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

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

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

Tốt:

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

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

Về đầu trang

Đối tượng và Cấu trúc dữ liệu

Sử dụng getter và setter

JavaScript không có interface hoặc kiểu vì vậy rất khó để thực hiện mô hình này, bởi vì chúng ta không có các từ khoá như `public` và `private`. Vì vậy, sử dụng getters và setters để truy cập dữ liệu trên các đối tượng thì tốt hơn là chỉ đơn giản tìm kiếm một thuộc tính trên một đối tượng. Bạn có thể hỏi "Tại sao?". Đây là một danh sách các lí do tại sao:

- Khi bạn muốn thực hiện nhiều hơn việc lấy một thuộc tính của đối tượng, bạn không cần phải tìm kiếm và thay đổi mỗi accessor trong codebase của bạn.
- Làm cho việc thêm các validation đơn giản khi thực hiện trên một `tập hợp`.
- Đóng gói các biểu diễn nội bộ.
- Dễ dàng thêm log và xử lí lỗi khi getting và setting.
- Kế thừa lớp này, bạn có thể override những hàm mặc định.
- Bạn có thể lazy load các thuộc tính của một đối tượng, lấy nó từ server.

Không tốt:

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

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

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

Tốt:

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

  // Một "getter", thiết lập public thông qua đối tượng được trả về dưới đây
  function getBalance() {
    return balance;
  }

  // Một "setter", thiết lập public thông qua đối tượng được trả về dưới đây
  function setBalance(amount) {
    // ... validate before updating the balance
    balance = amount;
  }

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

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

về đầu trang

Làm cho các đối tượng có thành viên private

Điều này có thể được thực hiện thông qua closures (cho ES5 và cũ hơn).

Không tốt:

```
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
```

Tốt:

```
const Employee = function (name) {
  this.getName = function getName() {
    return 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: John Doe
```

về đầu trang

Lớp

Ưu tiên lớp ES2015/ES6 hơn các chức năng thuần ES5

Rất khó khăn để có thể đọc được lớp thừa kế, lớp khởi tạo, và các định nghĩa phương thức trong các lớp ES5 cổ điển. Nếu bạn cần kế thừa (và lưu ý rằng bạn có thể không), tốt hơn là nên sử dụng lớp. Tuy nhiên ưu tiên sử dụng những hàm nhỏ hơn là lớp cho đến khi bạn cần những đối tượng lớn và phức tạp hơn.

Không tốt:

```
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() {};
```

Tốt:

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

về đầu trang

Sử dụng các hàm liên tiếp nhau

Đây là một pattern rất hữu ích trong JavaScript và bạn thấy nó trong rất nhiều thư viện chẳng hạn như jQuery và Lodash. Nó cho phép code của bạn có tính truyền tải và ngắn gọn. Vì lý do đó, theo tôi, sử dụng phương pháp các hàm liên tiếp nhau và hãy xem code của bạn sẽ sạch sẽ như thế nào. Trong các hàm của lớp, đơn giản là trả về `this` ở cuối mỗi hàm, và bạn có thể xâu chuỗi các phương thức khác vào trong nó.

Không tốt:

```
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();
```

Tốt:

```
class Car {
  constructor() {
    this.make = 'Honda';
```

```
    this.model = 'Accord';
    this.color = 'white';
  }

  setMake(make) {
    this.make = make;
    // Ghi chú: Trả về this để xâu chuỗi các phương thức
    return this;
  }

  setModel(model) {
    this.model = model;
    // Ghi chú: Trả về this để xâu chuỗi các phương thức
    return this;
  }

  setColor(color) {
    this.color = color;
    // Ghi chú: Trả về this để xâu chuỗi các phương thức
    return this;
  }

  save() {
    console.log(this.make, this.model, this.color);
    // Ghi chú: Trả về this để xâu chuỗi các phương thức
    return this;
  }
}

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

về đầu trang

Ưu tiên thành phần hơn là kế thừa

Như đã được nhấn mạnh nhiều trong *Design Patterns* của Gang of Four, bạn nên sử dụng cấu trúc thành phần hơn là thừa kế nếu có thể. Có rất nhiều lý do tốt để sử dụng kế thừa cũng như sử dụng thành phần. Điểm nhấn cho phương châm này đó là nếu tâm trí của bạn đi theo bản năng thừa kế, thử nghĩ nếu thành phần có thể mô hình vấn đề của bạn tốt hơn. Trong một số trường hợp nó có thể.

Bạn có thể tự hỏi, "khi nào tôi nên sử dụng thừa kế?" Nó phụ thuộc vào vấn đề trong tầm tay của bạn, nhưng đây là một danh sách manh nha khi kế thừa có ý nghĩa hơn thành phần:

1. Kế thừa của bạn đại diện cho mỗi quan hệ "is-a" và không có mỗi quan hệ "has-a" (Human->Animal vs. User->UserDetails).
2. Bạn có thể sử dụng lại code từ lớp cơ bản (Humans có thể di chuyển giống tất cả Animals).
3. Bạn muốn làm thay đổi toàn cục đến các lớp dẫn xuất bằng cách thay đổi lớp cơ bản. (Thay đổi lượng calo của tất cả animal khi chúng di chuyển)

Không tốt:

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

  // ...
}

// Không tốt bởi vì Employees "có" dữ liệu thuế.
// EmployeeTaxData không phải là một loại của Employee
class EmployeeTaxData extends Employee {
  constructor(ssn, salary) {
    super();
    this.ssn = ssn;
    this.salary = salary;
  }
```

```
  // ...
}
```

Tốt:

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

về đầu trang

SOLID

Nguyên lí đơn trách nhiệm (Single Responsibility Principle)

Như đã được nói đến trong cuốn Clean Code, "Chỉ có thể thay đổi một lớp vì một lí do duy nhất". Thật là hấp dẫn để nhồi nhét nhiều chức năng vào cho một lớp, giống như là khi bạn chỉ có thể lấy một chiếc vali cho chuyến bay vậy. Vấn đề là lớp của bạn sẽ không được hiểu gắn kết về mặt khái niệm của nó và sẽ có rất nhiều lí do để thay đổi. Việc làm giảm thiểu số lần bạn cần phải thay đổi một lớp là một việc quan trọng. Nó quan trọng bởi vì nếu có quá nhiều chức năng trong một lớp và bạn chỉ muốn thay đổi một chút xíu của lớp đó, thì có thể sẽ rất khó để hiểu được việc thay đổi đó sẽ ảnh hưởng đến những module khác trong codebase của bạn như thế nào.

Không tốt:

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

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

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

Tốt:

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

về đầu trang

Nguyên lí đóng mở (Open/Closed Principle)

Betrand Meyer đã nói "có thể thoải mái mở rộng một module, nhưng hạn chế sửa đổi bên trong module đó". Điều đó nghĩa là gì? Nguyên tắc này cơ bản nhấn mạnh rằng bạn phải cho phép người dùng thêm các chức năng mới mà không làm thay đổi các code đang có.

Không tốt:

```
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
}
```

Tốt:

```
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
    });
  }
}
```

về đầu trang

Nguyên lí thay thế Liskov (Liskov Substitution Principle)

Đây là một thuật ngữ đáng sợ cho một khái niệm rất đơn giản. Nó được định nghĩa một cách chính thức là: "Nếu S là một kiểu con của T, thì các đối tượng của kiểu T có thể được thay thế bằng các đối tượng của kiểu S (ví dụ các đối tượng của kiểu S có thể thay thế các đối tượng của kiểu T) mà không làm thay đổi bất kì thuộc tính mong muốn nào của chương trình đó (tính chính xác, thực hiện tác vụ, ..). Đó thậm chí còn là một định nghĩa đáng sợ hơn.

Sự giải thích tốt nhất cho nguyên lí này là, nếu bạn có một lớp cha và một lớp con, thì lớp cơ sở và lớp con có thể được sử dụng thay thế cho nhau mà không làm thay đổi tính đúng đắn của chương trình. Có thể vẫn còn hơi rối ở đây, vậy hãy xem cái ví dụ cổ điển hình vuông-hình chữ nhật (Square-Rectangle) dưới đây. Về mặt toán học, một hình vuông là một hình chữ nhật, tuy nhiên nếu bạn mô hình hoá điều này sử dụng quan hệ "is a" thông qua việc kế thừa, bạn sẽ nhanh chóng gặp phải rắc rối đấy.

Không tốt:

```
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: Will return 25 for Square. Should be 20.
    rectangle.render(area);
  });
}

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

Tốt:

```
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);
```

về đầu trang

Nguyên lí phân tách interface (Interface Segregation Principle)

JavaScript không có interface vì vậy nguyên lí này không áp dụng một cách chặt chẽ như các nguyên lí khác. Tuy nhiên, nó cũng quan trọng và liên quan ngay cả với hệ thống thiếu định kiểu của JavaScript.

Nguyên lí phân tách interface nhấn mạnh rằng "Người dùng không nên bị bắt buộc phải phụ thuộc vào các interfaces mà họ không sử dụng." Interface là những ràng buộc ẩn trong JavaScript bởi vì duck typing.

Một ví dụ tốt để minh hoạ cho nguyên lí này trong JavaScript là các lớp mà yêu cầu cài đặt các đối tượng lớn. Việc không yêu cầu người dùng thiết lập một số lượng lớn các tuỳ chọn là một ích lợi, bởi vì đa số thời gian họ không cần tất cả các cài đặt. Làm cho chúng trở thành tuỳ chọn giúp tránh được việc có một "fat interface".

Không tốt:

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

Tốt:

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

về đầu trang

Nguyên lí đảo ngược dependency (Dependency Inversion Principle)

Nguyên lí này khẳng định hai điều cần thiết sau:

1. Nhưng module cấp cao không nên phụ thuộc vào những module cấp thấp. Cả hai nên phụ thuộc vào abstraction.
2. Abstraction (interface) không nên phụ thuộc vào chi tiết, mà ngược lại.

Điều này có thể khó hiểu lúc ban đầu, nhưng nếu bạn đã từng làm việc với Angular.js, bạn đã thấy một sự hiện thực của nguyên lí này trong dạng của Dependency Injection (DI). Khi chúng không phải là các khái niệm giống nhau, DIP giữ cho module cấp cao không biết chi tiết các module cấp thấp của nó và thiết lập chúng. Có thể đạt được điều này thông qua DI. Một lợi ích to lớn của DIP là nó làm giảm sự phụ thuộc lẫn nhau giữa các module. Sự phụ thuộc lẫn nhau là một kiểu mẫu không tốt, vì nó làm cho việc tái cấu trúc code trở nên khó khăn.

Như đã khẳng định ở trước, JavaScript không có interface vì vậy các abstraction mà bị phụ thuộc là những ràng buộc ẩn. Đó là để nói, các phương thức và thuộc tính mà một đối tượng/lớp làm phơi bày đối tượng/lớp khác. Trong ví dụ bên dưới, sự ràng buộc ẩn là bất cứ module Request cho một `InventoryRequester` sẽ có một phương thức `requestItems`.

Không tốt:

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

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

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

    // Không tốt: chúng ta đã tạo một phụ thuộc vào một hiện thực của một request cụ thể
    // Chúng ta nên có những requestItems phụ thuộc vào một phương thức request `request`
    this.requester = new InventoryRequester();
  }

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

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

Tốt:

```
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) {
```

```
    // ...
  }
}

// Bằng cách xây dựng các phụ thuộc ở ngoài và thêm chúng vào, chúng ta có thể
// dễ dàng thay thế module request bằng một module mới lạ sử dụng WebSockets.
const inventoryTracker = new InventoryTracker(['apples', 'bananas'], new InventoryRequesterV2());
inventoryTracker.requestItems();
```

về đầu trang

Testing

Testing thì quan trọng hơn shipping. Nếu bạn không có test hoặc không đủ, thì mỗi lần ship code bạn sẽ không chắc là mình có làm hư hại thứ gì không. Việc quyết định những gì để tạo thành số lượng test đủ là do team của bạn, nhưng việc có 100% độ bao phủ (tất cả các câu lệnh và rẽ nhánh) là cách để bạn đạt được sự tự tin cao. Điều này có nghĩa ngoài việc có được một framework để test tốt, bạn cũng cần sử dụng một công cụ bao phủ tốt.

Không có lí do gì để không viết test. Có rất nhiều framework test JS tốt, vì thế hãy tìm một framework mà team bạn thích. Khi đã tìm được một cái thích hợp với team của mình, hãy đặt mục tiêu để luôn luôn viết test cho mỗi tính năng hoặc module mới của bạn. Nếu phương pháp test ưa thích của bạn là Test Driven Development (TDD), điều đó thật tuyệt, nhưng điểm quan trọng là phải chắc chắn bạn đạt được mục tiêu về độ bao phủ trước khi launch một tính năng hoặc refactor một tính năng cũ nào đó.

Một khái niệm duy nhất cho mỗi đơn vị test

Không tốt:

```
const assert = require('assert');

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

    date = new MakeMomentJSGreatAgain('1/1/2015');
    date.addDays(30);
    date.shouldEqual('1/31/2015');

    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);
  });
});
```

Tốt:

```
const assert = require('assert');

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

  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);
```

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

về đầu trang

Xử lí đồng thời

Hãy dùng Promise, đừng dùng callback

Callback thì không được 'sạch sẽ' cho lắm, chúng gây ra quá nhiều đoạn code lồng nhau (callback hell). Từ ES2015/ES6, Promise đã được đưa vào Javascript. Hãy sử dụng chúng!

Không tốt:

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

Tốt:

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

về đầu trang

Async/Await thì 'sạch sẽ' hơn Promise

Promise là một sự thay thế 'sạch sẽ' cho callback, nhưng ES2017/ES8 giới thiệu async và await, đó thậm chí còn là một giải pháp tốt hơn Promise nữa. Những gì bạn cần phải làm là một hàm có tiếp đầu ngữ là từ khoá `async` , và bạn có thể viết các lệnh logic mà không cần một chuỗi `then` của các hàm. Hãy sử dụng điều này nếu bạn có thể tận dụng các tính năng của ES2017/ES8 ngay hôm nay!

Không tốt:

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

Tốt:

```
async function getCleanCodeArticle() {
  try {
```

```
    const response = await require('request-promise').get('https://en.wikipedia.org/wiki/Robert_Cecil_Marti
    await require('fs-promise').writeFile('article.html', response);
    console.log('File written');
  } catch(err) {
    console.error(err);
  }
}
```

về đầu trang

Xử lí lỗi

Thông báo lỗi là một điều tốt! Nghĩa là chương trình của bạn nhận dạng được khi có một cái gì đó chạy không đúng và nó sẽ cho bạn biết bằng việc dừng chức năng mà nó đang thực thi, huỷ tiến trình (trong Node), và thông báo cho bạn trong console với một stack để theo dấu.

Đừng bỏ qua những lỗi đã bắt được

Nếu không làm gì với lỗi đã bắt được, bạn sẽ không thể sửa hoặc phản ứng lại được với lỗi đó. Ghi lỗi ra console (`console.log`) cũng không tốt hơn bao nhiêu vì đa số nó có thể bị trôi mất trong một đống những thứ được hiển thị ra ở console. Nếu bạn đặt bất cứ đoạn code nào trong một block `try/catch` , tức là bạn nghĩ một lỗi có thể xảy ra ở đây, do đó bạn nên có một giải pháp hoặc tạo một luồng code để xử lí lỗi khi nó xảy ra.

Không tốt:

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

Tốt:

```
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!
}
```

Đừng bỏ qua những promise bị lỗi (rejected)

Cùng nguyên nhân với phần trên.

Không tốt:

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

Tốt:

```
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!
});
```

về trang chủ

Định dạng

Việc định dạng code mang tính chủ quan. Giống như nhiều quy tắc được trình bày trong tài liệu này, không có quy tắc nào cứng nhắc và nhanh chóng mà bạn bắt buộc phải tuân theo. Điểm chính của phần này là ĐỪNG BAO GIỜ TRANH CÃI về việc định dạng code như thế nào. Có hàng tá công cụ để tự động hoá việc này. Hãy sử dụng một công cụ nào đó! Thật tốn thời gian và tiền bạc chỉ để tranh cãi về vấn đề định dạng code.

Đối với những thứ không thuộc phạm vi của việc tự động định dạng code (thụt đầu dòng, tab và space, nháy đơn và nháy kép,..) hãy xem một số hướng dẫn ở đây.

Sử dụng thống nhất cách viết hoa

Javascript là một ngôn ngữ không định kiểu, vì vậy việc viết hoa sẽ nói lên rất nhiều về các biến, hàm,.. của bạn. Những quy tắc này thì mang tính chủ quan, vì thế team bạn có thể chọn quy tắc nào họ muốn. Tuy nhiên điều quan trọng là dù bạn chọn cách viết như thế nào, thì cũng hãy sử dụng thống nhất nó trong codebase của bạn.

Không tốt:

```
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 {}
```

Tốt:

```
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 {}
```

về đầu trang

Các hàm gọi và hàm được gọi nên nằm gần nhau

Nếu một hàm gọi một hàm khác, hãy giữ những hàm này nằm gần theo chiều dọc trong file. Lí tưởng là, hãy giữ cho hàm gọi ở trên hàm được gọi. Chúng ta có xu hướng đọc code từ trên xuống, giống như đọc báo vậy. Do đó, hãy làm cho code của chúng ta cũng được đọc theo cách đó.

Không tốt:

```
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();
```

Tốt:

```
  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();
```

về đầu trang

Viết chú thích

Chỉ nên viết chú thích cho những thứ có logic phức tạp.

Các chú thích thường là lời xin lỗi, chứ không phải là yêu cầu. Những đoạn code tốt thì *đa số* tự nó đã là tài liệu rồi.

Không tốt:

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

  // Lấy chiều dài của chuỗi
  const length = data.length;

  // Lặp qua mỗi kí tự
  for (let i = 0; i < length; i++) {
    // Lấy mã của kí tự
    const char = data.charCodeAt(i);
    // Gán giá trị cho hash
    hash = ((hash << 5) - hash) + char;
    // Chuyển thành định dạng số nguyên 32 bit
    hash &= hash;
  }
}
```

Tốt:

```
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;

    // Chuyển thành định dạng số nguyên 32 bit
    hash &= hash;
  }
}
```

về đầu trang

Đừng giữ lại những đoạn code bị chú thích trong codebase của bạn.

Những công cụ quản lí phiên bản sinh ra để làm nhiệm vụ của chúng. Hãy để code cũ của bạn nằm lại trong dĩ vãng đi.

Không tốt:

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

Tốt:

```
doStuff();
```

về đầu trang

Đừng viết các chú thích nhật ký.

Hãy nhớ, sử dụng công cụ quản lí phiên bản! Chúng ta không cần những đoạn code vô dụng, bị chú thích và đặc biệt là những chú thích dạng nhật ký... Sử dụng `git log` để xem lịch sử được mà!

Không tốt:

```
/**
 * 2016-12-20: Removed monads, didn't understand them (RM)
```

```
 * 2016-10-01: Improved using special monads (JP)
 * 2016-02-03: Removed type-checking (LI)
 * 2015-03-14: Added combine with type-checking (JR)
 */
function combine(a, b) {
  return a + b;
}
```

Tốt:

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

về đầu trang

Tránh những đánh dấu vị trí

Chúng thường xuyên làm nhiễu code. Hãy để những tên hàm, biến cùng với các định dạng thích hợp tự tạo thành cấu trúc trực quan cho code của bạn.

Không tốt:

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

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

Tốt:

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

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

về đầu trang

Các ngôn ngữ khác

Tài liệu này cũng có sẵn ở các ngôn ngữ sau:

- English: ryanmcdermott/clean-code-javascript
- Brazilian Portuguese: fesnt/clean-code-javascript
- Chinese:
  - alivebao/clean-code-js
  - beginor/clean-code-javascript
- German: marcbruederlin/clean-code-javascript
- Korean: qkraudghgh/clean-code-javascript-ko
- Spanish: andersontr15/clean-code-javascript
- Russian:

- BoryaMogila/clean-code-javascript-ru/
- maksugr/clean-code-javascript

về đầu trang