

qkraudghgh / clean-code-javascript-ko Public

forked from ryanmcdermott/clean-code-javascript



Clean Code concepts adapted for JavaScript - 한글 번역판 KR

View license

☆ 1.3k stars    7.7k forks

☆ Star

Watch ▾

Code

Issues

Pull requests

Actions

Projects

Security

Insights

master ▾

...

This branch is 45 commits ahead, 406 commits behind ryanmcdermott:master.

Contribute ▾



qkraudghgh ...

on Feb 8



[View code](#)

README.md

# clean-code-javascript

- Updated date 2020.01.09
- 현재 원문의 1c0b20a 까지 반영되어 있습니다.

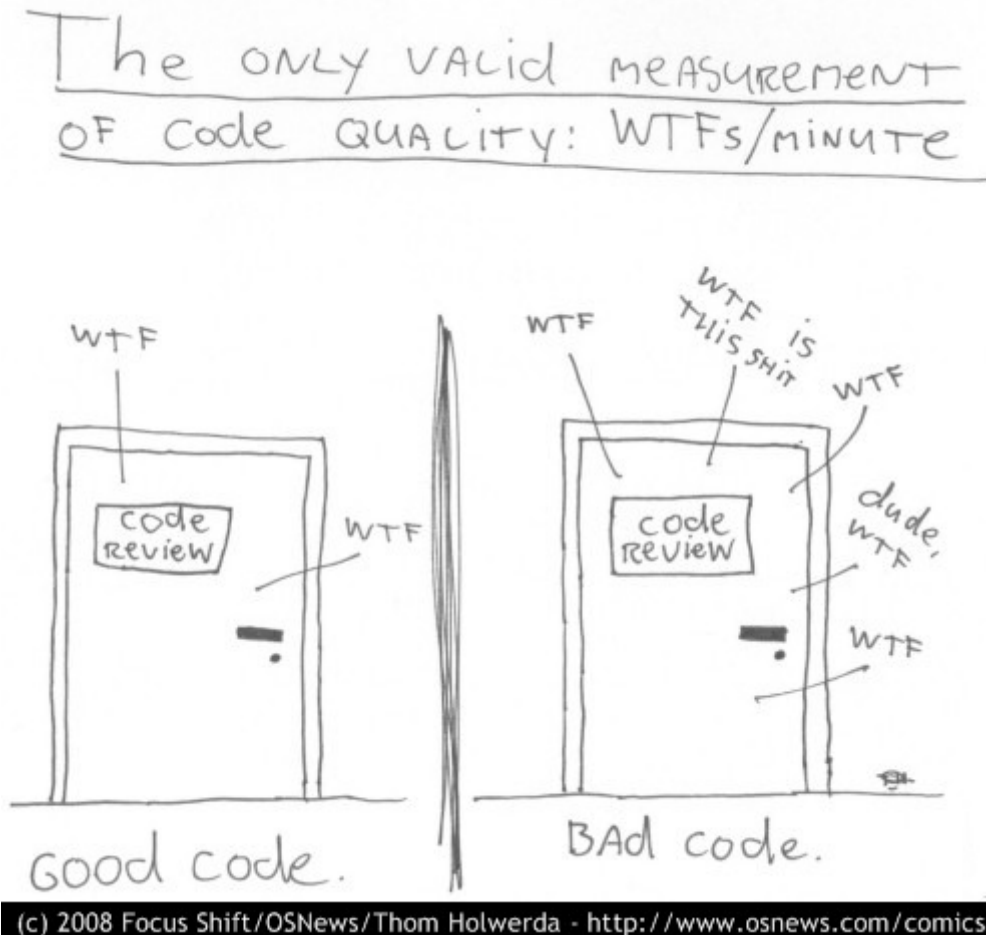
## 목차

1. 소개(Introduction)
2. 변수(Variables)
3. 함수(Functions)
4. 객체와 자료구조(Objects and Data Structures)
5. 클래스(Classess)
6. SOLID
7. 테스트(Testing)
8. 동시성(Concurrency)

- 9. 에러 처리(Error Handling)
- 10. 포매팅(Formatting)
- 11. 주석(Comments)
- 12. 번역(Translation)

## 소개(Introduction)

---



이 글은 소프트웨어 방법론에 관한 책들 중 Robert C. Martin's의 책인 *Clean Code*에 있는 내용을 JavaScript 언어에 적용시켜 적은 글입니다. 이 글은 단순히 Style Guide가 아니라 JavaScript로 코드를 작성할때 읽기 쉽고, 재사용 가능하며 리팩토링 가능하게끔 작성하도록 도와줍니다.

여기 있는 모든 원칙이 엄격히 지켜져야하는 것은 아니며, 보편적으로 통용되는 원칙은 아닙니다. 이것들은 지침일 뿐이며 Clean Code의 저자가 수년간 경험한 내용을 바탕으로 정리한 것입니다.

소프트웨어 엔지니어링 역사는 50년을 조금 넘겼지만 우리는 아직도 많은 것들을 배우고 있습니다. 그리고 소프트웨어 아키텍처가 건축설계 만큼이나 오래되었을 때 우리는 아래 규칙들보다 더 엄격한 규칙들을 따라야 할지도 모릅니다. 하지만 지금 당장은 이 가이드 라인을 당신과 당신 팀이 작성하는 JavaScript 코드의 품질을 평가하는 기준으로 삼으세요.

한가지 더 덧붙이자면, 이 원칙들을 알게된다해서 당장 더 나은 개발자가 되는 것은 아니며 코드를 작성할 때 실수를 하지 않게 해주는 것은 아닙니다. 훌륭한 도자기들이 처음엔 말랑한 점토부터 시작하듯이 모든 코드들은 처음부터 완벽할 수 없습니다. 하지만 당신은 팀원들과 같이 코드를 리뷰하며 점점 완벽하게 만들어가야 합니다. 당신이 처음 작성한 코드를 고칠 때 절대로 자신을 질타하지 마세요. 대신 코드를 부수고 더 나은 코드를 만드세요!

## 변수(Variables)

---

### 의미있고 발음하기 쉬운 변수 이름을 사용하세요

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

### 동일한 유형의 변수에 동일한 어휘를 사용하세요

안좋은 예:

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

좋은 예:

```
getUser();
```

[↑ 상단으로](#)

### 검색가능한 이름을 사용하세요

우리는 작성할 코드보다 읽을 코드가 더 많습니다. 그렇기 때문에 코드를 읽기 쉽고 검색 가능하게 작성해야 합니다. 그렇지 않으면 여러분의 코드를 이해하려고 하는 사람들에게 큰 어려움을 줍니다. 검색가능한 이름으로 만드세요. [buddy.js](#) 그리고 [ESLint](#) 와 같은 도구들이 이름이 정해져있지 않은 상수들을 발견하고 고칠 수 있게 도와줍니다.

안좋은 예:

```
// 대체 86400000 무엇을 의미하는 걸까요?  
setTimeout(blastOff, 86400000);
```

좋은 예

```
// 대문자로 `const` 전역 변수를 선언하세요  
const MILLISECONDS_IN_A_DAY = 86400000;  
setTimeout(blastOff, MILLISECONDS_IN_A_DAY);
```

[↑ 상단으로](#)

## 의도를 나타내는 변수들을 사용하세요

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

## 자신만 알아볼 수 있는 작명을 피하세요

명시적인 것이 암시적인 것보다 좋습니다.

안좋은 예:

```
const locations = ['서울', '인천', '수원'];  
locations.forEach(l => {  
  doStuff();  
  doSomeOtherStuff();  
  // ...  
  // ...  
  // ...  
  // 잠깐, `l`은 또 뭘까요?  
  dispatch(l);  
});
```

좋은 예:

```
const locations = ['서울', '인천', '수원'];
locations.forEach(location => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  dispatch(location);
});
```

[↑ 상단으로](#)

## 문맥상 필요없는 것들을 쓰지 마세요

안좋은 예:

```
const Car = {
  carMake: 'BMW',
  carModel: 'M3',
  carColor: '파란색'
};

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

좋은 예:

```
const Car = {
  make: 'BMW',
  model: 'M3',
  color: '파란색'
};

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

[↑ 상단으로](#)

## 기본 매개변수가 short circuiting 트릭이나 조건문 보다 깔끔합니다

기본 매개변수는 종종 short circuiting 트릭보다 깔끔합니다. 기본 매개변수는 매개변수가 undefined 일때만 적용됩니다. '', '', false, null, 0, NaN 같은 falsy 한 값들은 기본 매개변수가 적용되지 않습니다.

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

## 함수(Functions)

---

### 함수 인자는 2개 이하가 이상적입니다

매개변수의 개수를 제한 하는 것은 함수 테스트를 쉽게 만들어 주기 때문에 중요합니다. 만약 매개변수가 3개 이상일 경우엔 테스트 해야하는 경우의 수가 많아지고 각기 다른 인수들로 여러 사례들을 테스트 해야합니다.

1개나 2개의 인자를 가지고 있는 것이 가장 이상적인 케이스입니다. 그리고 3개의 인자는 가능한 피해야합니다. 그것보다 더 많다면 통합되어야합니다. 만약 당신이 2개 이상의 인자를 가진 함수를 사용한다면 그 함수에게 너무 많은 역할을 하게 만든 것입니다. 그렇지 않은 경우라면 대부분의 경우 상위 객체는 1개의 인자만으로 충분합니다.

JavaScript를 사용할 때 많은 보일러플레이트 없이 바로 객체를 만들 수 있습니다. 그러므로 당신이 만약 많은 인자들을 사용해야 한다면 객체를 이용할 수 있습니다.

함수가 기대하는 속성을 좀더 명확히 하기 위해서 es6의 비구조화(destructuring) 구문을 사용할 수 있고 이 구문에는 몇가지 장점이 있습니다.

1. 어떤 사람이 그 함수의 시그니처(인자의 타입, 반환되는 값의 타입 등)를 볼 때 어떤 속성이 사용되는지 즉시 알 수 있습니다.
2. 또한 비구조화는 함수에 전달된 인수 객체의 지정된 기본타입 값을 복제하며 이는 사이드이펙트가 일어나는 것을 방지합니다. 참고로 인수 객체로부터 비구조화된 객체와 배열은 복제되지 않습니다.
3. Linter를 사용하면 사용하지않는 인자에 대해 경고해주거나 비구조화 없이 코드를 짤 수 없게 할 수 있습니다.

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

## 함수는 하나의 행동만 해야합니다

이것은 소프트웨어 엔지니어링에서 가장 중요한 규칙입니다. 함수가 1개 이상의 행동을 한다면 작성하는 것도, 테스트하는 것도, 이해하는 것도 어려워집니다. 당신이 하나의 함수에 하나의 행동을 정의하는 것이 가능해진다면 함수는 좀 더 고치기 쉬워지고 코드들은 읽기 쉬워질 것입니다. 많은 원칙들 중 이것만 알아간다 하더라도 당신은 많은 개발자들을 앞설 수 있습니다.

안좋은 예:

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

좋은 예:

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

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

## ↑ 상단으로

## 함수명은 함수가 무엇을 하는지 알 수 있어야 합니다

안좋은 예:

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

const date = new Date();

// 월 추가하는 건지 이름만 보고 알아내기 힘듭니다.
AddToDate(date, 1);
```

좋은 예:

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

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

## ↑ 상단으로

## 함수는 단일 행동을 추상화 해야합니다

추상화된 이름이 여러 의미를 내포하고 있다면 그 함수는 너무 많은 일을 하게끔 설계된 것입니다. 함수들을 나누어서 재사용가능하고 테스트하기 쉽게 만드세요.

안좋은 예:

```
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...
});
}

```

좋은 예:

```

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...
  });
}

```

[↑ 상단으로](#)

중복된 코드를 작성하지 마세요

중복된 코드를 작성하지 않기 위해 최선을 다하세요. 중복된 코드가 있다는 것은 어떤 로직을 수정해야 할 일이 생겼을 때 수정 해야할 코드가 한 곳 이상이라는 것을 뜻합니다.

만약 당신이 레스토랑을 운영하면서 토마토나 양파, 마늘, 고추같은 것들의 재고관리를 해야 한다고 생각해보세요. 재고가 적혀있는 종이가 여러장 있다면 토마토나 양파의 재고가 변동 되었을 때 재고가 적혀있는 모든 종이를 수정해야 합니다. 만약 재고를 관리하는 종이가 한 장이었다면 한 장의 재고 목록만 수정하면 됐겠죠!

종종 코드를 살펴보면 사소한 몇몇의 차이점 때문에 중복된 코드를 작성한 경우가 있고 이런 차이점들은 대부분 똑같은 일을 하는 분리된 함수들을 갖도록 강요합니다. 즉 중복 코드를 제거한다는 것은 하나의 함수 / 모듈 / 클래스를 사용하여 이 여러 가지 사소한 차이점을 처리 할 수 있는 추상화를 만드는 것을 의미합니다.

그리고 추상화 할 부분이 남아있는 것은 위험하기때문에 클래스 섹션에 제시된 여러 원칙들을 따라야 합니다. 잘 추상화 하지 못한 코드는 중복된 코드보다 나쁠 수 있으므로 조심하세요. 즉 추상화를 잘 할 수 있다면 그렇게 하라는 말입니다. 코드의 중복을 피한다면 여러분이 원할 때 언제든지 한 곳만 수정해도 다른 모든 코드에 반영되게 할 수 있습니다.

**안좋은 예:**

```
function showDeveloperList(developers) {
  developers.forEach(developers => {
    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.getMBAPProjects();
    const data = {
      expectedSalary,
      experience,
      portfolio
    };

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

좋은 예:

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

    let portfolio = employee.getGithubLink();

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

    const data = {
      expectedSalary,
      experience,
      portfolio
    };

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

## ↑ 상단으로

Object.assign을 사용해 기본 객체를 만드세요

안좋은 예:

```
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);
```

좋은 예:

```
const menuConfig = {
  title: 'Order',
```

```
// 사용자가 'body' key의 value를 정하지 않았다.
buttonText: 'Send',
cancellable: true
};

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

  // config는 이제 다음과 동일합니다: {title: "Order", body: "Bar", buttonText: "Send",
  // ...
}

createMenu(menuConfig);
```

[↑ 상단으로](#)

## 매개변수로 플래그를 사용하지 마세요

플래그를 사용하는 것 자체가 그 함수가 한가지 이상의 역할을 하고 있다는 것을 뜻합니다. boolean 기반으로 함수가 실행되는 코드가 나뉜다면 함수를 분리하세요.

안좋은 예:

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

좋은 예:

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

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

[↑ 상단으로](#)

## 사이드 이펙트를 피하세요 (part 1)

함수는 값을 받아서 어떤 일을 하거나 값을 리턴할 때 사이드 이펙트를 만들어냅니다. 사이드 이펙트는 파일에 쓰여질 수도 있고, 전역 변수를 수정할 수 있으며, 실수로 모든 돈을 다른 사람에게 보낼 수도 있습니다.

당신은 때때로 프로그램에서 사이드 이펙트를 만들어야 할 때가 있습니다. 아까 들었던 예들 중 하나인 파일작성을 할 때와 같이 말이죠. 이 때 여러분이 해야 할 일은 파일 작성을 하는 한 개의 함수를 만드는 일입니다. 파일을 작성하는 함수나 클래스가 여러개 존재하면 안됩니다. 반드시 하나만 있어야 합니다.

즉, 어떠한 구조체도 없이 객체 사이의 상태를 공유하거나, 무엇이든 쓸 수 있는 변경 가능한 데이터 유형을 사용하거나, 같은 사이드 이펙트를 만들어내는 것을 여러개 만들거나하면 안 됩니다. 여러분들이 이러한 것들을 지키며 코드를 작성한다면 대부분의 다른 개발자들보다 행복할 수 있습니다.

안좋은 예:

```
// 아래 함수에 의해 참조되는 전역 변수입니다.
// 이 전역 변수를 사용하는 또 하나의 함수가 있다고 생각해 보세요. 이제 이 변수는 배열이 될 것
let name = 'Ryan McDermott';

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

splitIntoFirstAndLastName();

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

좋은 예:

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

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

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

[↑ 상단으로](#)

## 사이드 이펙트를 피하세요 (part 2)

자바스크립트에서는 기본타입 자료형은 값을 전달하고 객체와 배열은 참조를 전달합니다. 객체와 배열인 경우를 한번 살펴봅시다. 우리가 만든 함수는 장바구니 배열에 변화를 주며 이 변화는 구매목록에 어떤 상품을 추가하는 기능 같은 것을 말합니다. 만약 장바구니 배열을 사용하는 어느 다른 함수가 있다면 이러한 추가에 영향을 받습니다. 이것은 좋을 수도 있지만, 안좋은 수도 있습니다. 안좋은 예를 한번 상상해봅시다.

유저가 구매하기 버튼을 눌러 구매 함수를 호출합니다. 이는 네트워크 요청을 생성하고 서버에 장바구니 배열을 보냅니다. 하지만 네트워크 연결이 좋지않아서 구매 함수는 다시한번 네트워크 요청을 보내야 하는 상황이 생겼습니다. 이때, 사용자가 네트워크 요청이 시작되기 전에 실수로 원하지 않는 상품의 "장바구니에 추가" 버튼을 실수로 클릭하면 어떻게될까요? 실수가 있고난 뒤, 네트워크 요청이 시작되면 장바구니에 추가 함수 때문에 실수로 변경된 장바구니 배열을 서버에 보내게 됩니다.

가장 좋은 방법은 장바구니에 추가 는 항상 장바구니 배열을 복제하여 수정하고 복제본을 반환하는 것입니다. 이렇게하면 장바구니 참조를 보유하고있는 다른 함수가 다른 변경 사항의 영향을 받지 않게됩니다.

이 접근법에대해 말하고 싶은 것이 두가지 있습니다.

1. 실제로 입력된 객체를 수정하고 싶은 경우가 있을 수 있지만 이러한 예제를 생각해보고 적용해보면 그런 경우는 거의 없다는 것을 깨달을 수 있습니다. 그리고 대부분의 것들이 사이드 이펙트 없이 리팩토링 될 수 있습니다.
2. 큰 객체를 복제하는 것은 성능 측면에서 값이 매우 비쌉니다. 운 좋게도 이렇게 큰 문제가 되지는 않습니다. 왜냐하면 이러한 프로그래밍 접근법을 가능하게해줄 **좋은 라이브러리**가 있기 때문입니다. 이는 객체와 배열을 수동으로 복제하는 것처럼 메모리 집약적이지 않게 해주고 빠르게 복제해줍니다.

Bad:

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

Good:

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

 **상단으로**

**전역 함수를 사용하지 마세요**

전역 환경을 사용하는 것은 JavaScript에서 나쁜 관행입니다. 왜냐하면 다른 라이브러리들과의 충돌이 일어날 수 있고, 당신의 API를 쓰는 유저들은 운영환경에서 예외가 발생하기 전까지는 문제를 인지하지 못할 것이기 때문입니다. 예제를 하나 생각해봅시다. JavaScript의 네이티브 Array 메소드를 확장하여 두 배열 간의 차이를 보여줄 수 있는 diff 메소드를 사용하려면 어떻게 해야할까요? 새로운 함수를 Array.prototype 에 쓸 수도 있지만, 똑같은 일을 시도한 다른 라이브러리와 충돌 할 수 있습니다. 다른 라이브러리가 diff 메소드를 사용하여 첫 번째 요소와 마지막 요소의 차이점을 찾으려면 어떻게 될까요? 이것이 그냥 ES2015/ES6의 classes를 사용해서 전역 Array 를 상속해버리는 것이 훨씬 더 나은 이유입니다.

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

## 명령형 프로그래밍보다 함수형 프로그래밍을 지향하세요

JavaScript는 Haskell처럼 함수형 프로그래밍 언어는 아니지만 함수형 프로그래밍처럼 작성할 수 있습니다. 함수형 언어는 더 깔끔하고 테스트하기 쉽습니다. 가능하다면 이 방식을 사용하도록 해보세요.

안좋은 예:

```
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;
}

```

좋은 예:

```

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

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

```

## ↑ 상단으로

## 조건문을 캡슐화 하세요

안좋은 예:

```

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

```

좋은 예:

```

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

if (shouldShowSpinner(fsmInstance, listNodeInstance)) {

```



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

[↑ 상단으로](#)

## 부정조건문을 사용하지 마세요

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

## 조건문 작성을 피하세요

조건문 작성을 피하라는 것은 매우 불가능한 일로 보입니다. 이 얘기를 처음 듣는 사람들은 대부분 "If 문 없이 어떻게 코드를 짜나요?"라고 말합니다. 하지만 다형성을 이용한다면 동일한 작업을 수행할 수 있습니다. 두번째 질문은 보통 "네 좋네요 근데 내가 왜 그렇게 해야 하나요?"이죠. 그에 대한 대답은, 앞서 우리가 공부했던 clean code 컨셉에 있습니다. 함수는 단 하나의 일만 수행하여야 합니다. 당신이 함수나 클래스에 if 문을 쓴다면 그것은 그 함수나 클래스가 한가지 이상의 일을 수행하고 있다고 말하는 것과 같습니다. 기억하세요, 하나의 함수는 딱 하나의 일만 해야합니다.

안좋은 예:

```
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();
    }
}
}

```

좋은 예:

```

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();
    }
}

```

## ↑ 상단으로

## 타입-체킹을 피하세요 (part 1)

JavaScript는 타입이 정해져있지 않습니다. 이는 당신의 함수가 어떤 타입의 인자든 받을 수 있다는 것을 의미합니다. 이런 JavaScript의 자유로움 때문에 여러 버그가 발생했었고 이 때문에 당신의 함수에 타입-체킹을 시도 할 수도 있습니다. 하지만 타입-체킹 말고도 이러한 화를 피할 많은 방법들이 존재합니다. 첫번째 방법은 일관성 있는 API를 사용하는 것입니다.

안좋은 예:

```

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'));
  }
}
```

좋은 예:

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

[↑ 상단으로](#)

## 타입-체킹을 피하세요 (part 2)

당신이 문자열, 정수, 배열 등 기본 자료형을 사용하고 다형성을 사용할 수 없을 때 여전히 타입-체킹이 필요하다고 느껴진다면 TypeScript를 도입하는 것을 고려해보는 것이 좋습니다. TypeScript는 표준 JavaScript 구문에 정적 타입을 제공하므로 일반 JavaScript의 대안으로 사용하기에 좋습니다. JavaScript에서 타입-체킹을 할 때 문제점은 가짜 type-safety 를 얻기 위해 작성된 코드를 설명하기 위해서 많은 주석을 달아야한다는 점입니다. JavaScript로 코드를 작성할때 깔끔하게 코드를 작성하고, 좋은 테스트 코드를 짜야하며 좋은 코드 리뷰를 해야합니다. 그러기 싫다면 그냥 TypeScript(이건 제가 말했듯이, 좋은 대체재입니다!)를 쓰세요.

안좋은 예:

```
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');
}
```

좋은 예:

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

[↑ 상단으로](#)

## 과도한 최적화를 지양하세요

최신 브라우저들은 런타임에 많은 최적화 작업을 수행합니다. 대부분 당신이 코드를 최적화하는 것은 시간낭비일 가능성이 많습니다. 최적화가 부족한 곳이 어딘지를 알려주는 [좋은 자료](#)가 여기 있습니다. 이것을 참조하여 최신 브라우저들이 최적화 해주지 않는 부분만 최적화를 해주는 것이 좋습니다.

안좋은 예:

```
// 오래된 브라우저의 경우 캐시되지 않은 `list.length`를 통한 반복문은 높은 코스트를 가졌습니
// 그 이유는 `list.length`를 매번 계산해야만 했기 때문인데, 최신 브라우저에서는 이것이 최적
for (let i = 0, len = list.length; i < len; i++) {
  // ...
}
```



좋은 예:

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

[↑ 상단으로](#)

## 죽은 코드를 지우세요

죽은 코드는 중복된 코드 만큼이나 좋지 않습니다. 죽은 코드는 당신의 코드에 남아있을 어떠한 이유도 없습니다. 호출되지 않는 코드가 있다면 그 코드는 지우세요! 그 코드가 여전히 필요해도 그 코드는 버전 히스토리에 안전하게 남아있을 것입니다.

안좋은 예:

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

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

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

좋은 예:

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

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

[↑ 상단으로](#)

## 객체와 자료구조(Objects and Data Structures)

---

### getter와 setter를 사용하세요

JavaScript는 인터페이스와 타입을 가지고있지 않고 이러한 패턴을 적용하기가 힘듭니다. 왜냐하면 `public` 이나 `private` 같은 키워드가 없기 때문이죠. 그렇기 때문에 getter 및 setter를 사용하여 객체의 데이터에 접근하는 것이 객체의 속성을 찾는 것보다 훨씬 낫습니다. "왜요?"라고 물으실 수도 있겠습니다. 왜 그런지에 대해서 몇 가지 이유를 두서없이 적어봤습니다.

- 객체의 속성을 얻는 것 이상의 많은 것을 하고싶을 때, 코드에서 모든 접근자를 찾아 바꾸고 할 필요가 없습니다.
- `set` 할때 검증로직을 추가하는 것이 코드를 더 간단하게 만듭니다.
- 내부용 API를 캡슐화 할 수 있습니다.
- `getting` 과 `setting` 할 때 로그를 찾거나 에러처리를 하기 쉽습니다.
- 서버에서 객체 속성을 받아올 때 `lazy load` 할 수 있습니다.

안좋은 예:

```
function makeBankAccount() {  
  // ...  
  
  return {  
    // ...  
    balance: 0  
  };  
}  
  
const account = makeBankAccount();  
account.balance = 100;
```

좋은 예:

```
function makeBankAccount() {  
  // private으로 선언된 변수  
  let balance = 0;  
  
  // 아래 return을 통해 public으로 선언된 "getter"  
  function getBalance() {  
    return balance;  
  }  
}
```

```
// 아래 return을 통해 public으로 선언된 "setter"
function setBalance(amount) {
  // ... balance를 업데이트하기 전 검증로직
  balance = amount;
}

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

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

## ↑ 상단으로

## 객체에 비공개 멤버를 만드세요

클로저를 이용하면 가능합니다. (ES5 이하에서도)

안좋은 예:

```
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
```

좋은 예:

```
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
```

[↑ 상단으로](#)

## 클래스(Classess)

---

### ES5의 함수보다 ES2015/ES6의 클래스를 사용하세요

기존 ES5의 클래스에서 이해하기 쉬운 상속, 구성 및 메소드 정의를 하는 건 매우 어렵습니다. 매번 그런것은 아니지만 상속이 필요한 경우라면 클래스를 사용하는 것이 좋습니다. 하지만 당신이 크고 더 복잡한 객체가 필요한 경우가 아니라면 클래스보다 작은 함수를 사용하세요.

안좋은 예:

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

좋은 예:

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

## ↑ 상단으로

## 메소드 체이닝을 사용하세요

JavaScript에서 메소드 체이닝은 매우 유용한 패턴이며 jQuery나 Lodash같은 많은 라이브러리에서 이 패턴을 찾아볼 수 있습니다. 이는 코드를 간결하고 이해하기 쉽게 만들어줍니다. 이런 이유들로 메소드 체이닝을 쓰는 것을 권하고, 사용해본뒤 얼마나 코드가 깔끔해졌는지 꼭 확인 해보길 바랍니다. 클래스 함수에서 단순히 모든 함수의 끝에 'this'를 리턴해주는 것으로 클래스 메소드를 추가로 연결할 수 있습니다.

안좋은 예:

```
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();

```

좋은 예:

```

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

    setMake(make) {
        this.make = make;
        // 메모: 체이닝을 위해 this를 리턴합니다.
        return this;
    }

    setModel(model) {
        this.model = model;
        // 메모: 체이닝을 위해 this를 리턴합니다.
        return this;
    }

    setColor(color) {
        this.color = color;
        // 메모: 체이닝을 위해 this를 리턴합니다.
        return this;
    }

    save() {
        console.log(this.make, this.model, this.color);
        // 메모: 체이닝을 위해 this를 리턴합니다.
        return this;
    }
}

```

```
}
```

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

## ↑ 상단으로

## 상속보단 조합(composition)을 사용하세요

Gang of four의 *Design Patterns*에서 유명한 전략으로 당신은 가능하다면 상속보다는 조합을 사용해야 합니다. 상속을 사용했을 때 얻을 수 있는 이득보다 조합을 사용했을 때 얻을 수 있는 이득이 많기 때문입니다. 이 원칙의 요점은 당신이 계속 상속을 사용해서 코드를 작성하고 자 할 때, 만약 조합을 이용하면 더 코드를 잘 짤 수 있지 않을까 생각해보라는 것에 있습니다. 때때로는 이것이 맞는 전략이기 때문이죠.

"그럼 대체 상속을 언제 사용해야 되는 건가요?"라고 물어 볼 수 있습니다. 이걸 당신이 직면한 문제 상황에 달려있지만 조합보다 상속을 쓰는게 더 좋을 만한 예시를 몇 개 들어 보겠습니다.

1. 당신의 상속관계가 "has-a" 관계가 아니라 "is-a" 관계일 때 (사람->동물 vs. 유저->유저 정보)
2. 기반 클래스의 코드를 다시 사용할 수 있을 때 (인간은 모든 동물처럼 움직일 수 있습니다.)
3. 기반 클래스를 수정하여 파생된 클래스 모두를 수정하고 싶을 때 (이동시 모든 동물이 소비하는 칼로리를 변경하고 싶을 때)

안좋은 예:

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

// 이 코드가 안좋은 이유는 Employees가 tax data를 "가지고" 있기 때문입니다.  
// EmployeeTaxData는 Employee 타입이 아닙니다.

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

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

좋은 예:

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

[↑ 상단으로](#)

## SOLID

---

### 단일 책임 원칙 (Single Responsibility Principle, SRP)

Clean Code에서 말하길 "클래스를 수정 할 때는 수정 해야하는 이유가 2개 이상 있으면 안됩니다". 이것은 하나의 클래스에 많은 기능을 쑤셔넣는 것이나 다름 없습니다. 마치 비행기를 탈때 가방을 1개만 가지고 탈 수 있을 때 처럼 말이죠. 이 문제는 당신의 클래스가 개념적으로 응집되어 있지 않다는 것이고, 클래스를 바꿔야할 많은 이유가 됩니다. 클래스를 수정하는데 들이는 시간을 줄이는 것은 중요합니다. 왜냐면 하나의 클래스에 너무 많은 기능들이 있고 당신이 이 작은 기능들을 수정할 때 이 코드가 다른 모듈들에 어떠한 영향을 끼치는지 이해하기 어려울 수 있기 때문입니다.

안좋은 예:

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

```

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

verifyCredentials() {
    // ...
}
}

```

좋은 예:

```

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

```

[↑ 상단으로](#)

## 개방/폐쇄 원칙 (Open/Closed Principle, OCP)

Bertrand Meyer에 말에 의하면 "소프트웨어 개체(클래스, 모듈, 함수 등)는 확장을 위해 개방적이어야 하며 수정시엔 폐쇄적이어야 합니다." 이것에 의미는 무엇일까요? 이 원리는 기본적으로 사용자가 .js 소스 코드 파일을 열어 수동으로 조작하지 않고도 모듈의 기능을 확장하도록 허용해야한다고 말합니다.

안좋은 예:

```

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 makeHttpRequest(url).then((response) => {
                // transform response and return
            });
        }
    }
}

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

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

```

좋은 예:

```

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
    });
  }
}

```

## ↑ 상단으로

## 리스코프 치환 원칙 (Liskov Substitution Principle, LSP)

이것은 매우 간단하지만 강력한 원칙입니다. 리스코프 원칙이란 자료형 S가 자료형 T의 하위 형이라면, 프로그램이 갖추어야 할 속성들(정확성, 수행되는 작업 등)의 변경사항 없이, 자료형 T의 객체를 자료형 S의 객체로 교체(치환)할 수 있어야 한다는 원칙입니다.

이 원칙을 예를 들어 설명하자면 당신이 부모 클래스와 자식 클래스를 가지고 있을 때 베이스 클래스와 하위 클래스를 잘못된 결과 없이 서로 교환하여 사용할 수 있습니다. 여전히 이해가 안간다면 정사각형-직사각형 예제를 봅시다. 수학적으로 정사각형은 직사각형이지만 상속을 통해 "is-a" 관계를 사용하여 모델링한다면 문제가 발생합니다.

안좋은 예:

```

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(); // 정사각형일때 25를 리턴합니다. 하지만 20이어야 하
        rectangle.render(area);
    });
}

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

```

좋은 예:

```

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);

```

## ↑ 상단으로

## 인터페이스 분리 원칙 (Interface Segregation Principle, ISP)

JavaScript는 인터페이스가 없기 때문에 다른 원칙들처럼 딱 맞게 적용할 수는 없습니다. 그러나, JavaScript에 타입 시스템이 없다 하더라도 중요하고 관계있는 원칙입니다.

ISP에 의하면 "클라이언트는 사용하지 않는 인터페이스에 의존하도록 강요 받으면 안됩니다." 덕 타이핑 때문에 인터페이스는 JavaScript에서는 암시적인 계약일 뿐입니다.

JavaScript에서 이것을 보여주는 가장 좋은 예는 방대한 양의 설정 객체가 필요한 클래스입니다. 클라이언트가 방대한 양의 옵션을 설정하지 않는 것이 좋습니다. 왜냐하면 대부분의 경우 설정들이 전부 다 필요한 건 아니기 때문입니다. 설정을 선택적으로 할 수 있다면 "무거운 인터페이스(fat interface)"를 만드는 것을 방지할 수 있습니다.

안좋은 예:

```

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() {} // 우리는 대부분의 경우 DOM을 탐색할 때 애니메이션이 필요하지 않습니
    // ...
});

```

좋은 예:

```

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() {}
    }
});

```

[↑ 상단으로](#)

의존성 역전 원칙 (Dependency Inversion Principle, DIP)

이 원칙은 두가지 중요한 요소를 가지고 있습니다.

1. 상위 모듈은 하위 모듈에 종속되어서는 안됩니다. 둘 다 추상화에 의존해야 합니다.
2. 추상화는 세부사항에 의존하지 않습니다. 세부사항은 추상화에 의해 달라져야 합니다.

처음에는 이것을 이해하는데 어려울 수 있습니다. 하지만 만약 Angular.js로 작업해본적이 있다면 의존성 주입(Dependency Injection) 형태로 이 원리를 구현한 것을 보았을 것입니다. DIP는 동일한 개념은 아니지만 상위 모듈이 하위 모듈의 세부사항을 알지 못하게 합니다. 이는 의존성 주입을 통해 달성할 수 있습니다. DI의 장점은 모듈 간의 의존성을 감소시키는 데에 있습니다. 모듈간의 의존성이 높을수록 코드를 리팩토링 하는데 어려워지고 이것은 매우 나쁜 개발 패턴들 중 하나입니다.

앞에서 설명한 것처럼 JavaScript에는 인터페이스가 없으므로 추상화에 의존하는 것은 암시적인 약속입니다. 이말인즉슨, 다른 객체나 클래스에 노출되는 메소드와 속성이 바로 암시적인 약속(추상화)가 된다는 것이죠. 아래 예제에서 암시적인 약속은 InventoryTracker 에대한 모든 요청 모듈이 requestItems 메소드를 가질 것이라는 점입니다.

안좋은 예:

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

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

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

    // 안좋은 이유: 특정 요청방법 구현에 대한 의존성을 만들었습니다.
    // requestItems는 한가지 요청방법을 필요로 합니다.
    this.requester = new InventoryRequester();
  }

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

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

좋은 예:

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

```
// 의존성을 외부에서 만들어 주입해줌으로써,
// 요청 모듈을 새롭게 만든 웹소켓 사용 모듈로 쉽게 바꿔 끼울 수 있게 되었습니다.
const inventoryTracker = new InventoryTracker(['apples', 'bananas'], new InventoryReq
inventoryTracker.requestItems();
```

[↑ 상단으로](#)

## 테스트(Testing)

테스트는 배포하는 것보다 중요합니다. 테스트 없이 배포한다는 것은 당신이 짜놓은 코드가 언제든지 오작동해도 이상하지 않다는 애기와 같습니다. 테스트에 얼마나 시간을 투자할지는 당신이 함께 일하는 팀에 달려있지만 Coverage가 100%라는 것은 개발자들에게 높은 자신감과 안도감을 줍니다. 이 말은 훌륭한 테스트 도구를 보유해야 하는 것 뿐만 아니라 **훌륭한 Coverage 도구**를 사용해야한다는 것을 의미합니다.

테스트 코드를 작성하지 않는다는 것은 그 무엇도 변명이 될 수 없습니다. 여기 **훌륭하고 많은 JavaScript 테스트 프레임워크들** 이 있습니다. 당신의 팀의 기호에 맞는 프레임워크를 고르기만 하면 됩니다. 테스트 프레임워크를 골랐다면 이제부터는 팀의 목표를 모든 새로운 기능/모듈을 짤 때 테스트 코드를 작성하는 것으로 하세요. 만약 테스트 주도 개발 방법론(Test Driven Development, TDD)이 당신에게 맞는 방법이라면 그건 훌륭한 개발 방법이 될 수 있습니다. 그러나 중요한 것은 당신이 어떠한 기능을 개발하거나 코드를 리팩토링 할 때 당신이 정한 Coverage 목표를 달성하는 것에 있습니다.

## 테스트 컨셉

안좋은 예:

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

좋은 예:

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

[↑ 상단으로](#)

## 동시성(Concurrency)

---

### Callback 대신 Promise를 사용하세요

Callback은 깔끔하지 않습니다. 그리고 엄청나게 많은 중괄호 중첩을 만들어 냅니다. ES2015/ES6에선 Promise가 내장되어 있습니다. 이것 쓰세요!

안좋은 예:

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

좋은 예:

```
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);
  });
```

[↑ 상단으로](#)

## Async/Await은 Promise보다 더욱 깔끔합니다

Promise도 Callback에 비해 정말 깔끔하지만 ES2017/ES8에선 async와 await이 있습니다. 이들은 Callback에대한 더욱 깔끔한 해결책을 줍니다. 오직 필요한 것은 함수앞에 async를 붙이는 것 뿐입니다. 그러면 함수를 논리적으로 연결하기위해 더이상 then을 쓰지 않아도 됩니다. 만약 당신이 ES2017/ES8 사용할 수 있다면 이것을 사용하세요!

안좋은 예:

```
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);
  })
```

좋은 예:

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

[↑ 상단으로](#)

## 에러 처리(Error Handling)

에러를 뱉는다는 것은 좋은 것입니다! 즉, 프로그램에서 무언가가 잘못되었을 때 런타임에서 성공적으로 확인되면 현재 스택에서 함수 실행을 중단하고 (노드에서) 프로세스를 종료하고 스택 추적으로 콘솔에서 사용자에게 그 이유를 알려줍니다.

**단순히 에러를 확인만 하지마세요**

단순히 에러를 확인하는 것만으로 그 에러가 해결되거나 대응 할 수 있게 되는 것은 아닙니다. `console.log` 를 통해 콘솔에 로그를 기록하는 것은 에러 로그를 잃어버리기 쉽기 때문에 좋은 방법이 아닙니다. 만약에 `try/catch` 로 어떤 코드를 감쌌다면 그건 당신이 그 코드에 어떤 에러가 날지도 모르기 때문에 감싼 것이므로 그에대한 계획이 있거나 어떠한 장치를 해야 합니다.

안좋은 예:

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

좋은 예:

```
try {
  functionThatMightThrow();
} catch (error) {
  // 첫번째 방법은 console.error를 이용하는 것입니다. 이건 console.log보다 조금 더 알아채:
  console.error(error);
  // 다른 방법은 유저에게 알리는 방법입니다.
  notifyUserOfError(error);
  // 또 다른 방법은 서비스 자체에 에러를 기록하는 방법입니다.
  reportErrorToService(error);
  // 혹은 그 어떤 방법이 될 수 있습니다.
}
```



[↑ 상단으로](#)

## Promise가 실패된 것을 무시하지 마세요

위의 원칙과 같은 이유입니다.

안좋은 예:

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

좋은 예:

```

getdata()
.then(data => {
  functionThatMightThrow(data);
})
.catch(error => {
  // 첫번째 방법은 console.error를 이용하는 것입니다. 이건 console.log보다 조금 더 알아채:
  console.error(error);
  // 다른 방법은 유저에게 알리는 방법입니다.
  notifyUserOfError(error);
  // 또 다른 방법은 서비스 자체에 에러를 기록하는 방법입니다.
  reportErrorToService(error);
  // 혹은 그 어떤 방법이 될 수 있습니다.
});

```

[↑ 상단으로](#)

## 포매팅(Formatting)

포매팅은 주관적입니다. 여기에 있는 많은 규칙과 마찬가지로 따르기 쉬운 규칙들이 있습니다. 여기서 알아야 할 것은 포매팅에 대해 과도하게 신경쓰는 것은 의미없다는 것입니다. 포매팅 체크를 자동으로 해주는 [많은 도구들](#)이 있기 때문입니다. 이중 하나를 골라 사용하세요. 개발자들끼리 포매팅에 대해 논쟁하는 것만큼 시간과 돈을 낭비하는 것이 없습니다.

자동으로 서식을 교정해주는 것(들여쓰기, 탭이나 스페이스나, 작은 따옴표나 큰따옴표나)에 해당하지 않는 사항에 대해서는 몇가지 지침을 따르는 것이 좋습니다.

### 일관된 대소문자를 사용하세요

JavaScript에는 정해진 타입이 없기 때문에 대소문자를 구분하는 것으로 당신의 변수나 함수명 등에서 많은 것을 알 수 있습니다. 이 규칙 또한 주관적이기 때문에 당신이 팀이 선택한 규칙들을 따르세요 중요한건 항상 일관성 있게 사용해야 한다는 것입니다.

안좋은 예:

```

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 {}

```



좋은 예:

```
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 {}
```

## ↑ 상단으로

### 함수 호출자와 함수 피호출자는 가깝게 위치시키세요

어떤 함수가 다른 함수를 호출하면 그 함수들은 소스 파일 안에서 서로 수직으로 근접해 있어야 합니다. 이상적으로는 함수 호출자를 함수 피호출자 바로 위에 위치시켜야 합니다. 우리는 코드를 읽을때 신문을 읽듯 위에서 아래로 읽기 때문에 코드를 작성 할 때도 읽을 때를 고려하여 작성 해야합니다.

안좋은 예:

```
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();

```

좋은 예:

```

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();

```

## 주석(Comments)

---

### 비즈니스 로직이 복잡한 경우에만 주석을 다세요

주석을 다는것은 사과해야할 일이며 필수적인 것이 아닙니다. 좋은 코드는 코드 자체로 말합니다.

안좋은 예:

```
function hashIt(data) {  
  // 이걸 해쉬입니다.  
  let hash = 0;  
  
  // length는 data의 길이입니다.  
  const length = data.length;  
  
  // 데이터의 문자열 개수만큼 반복문을 실행합니다.  
  for (let i = 0; i < length; i++) {  
    // 문자열 코드를 얻습니다.  
    const char = data.charCodeAt(i);  
    // 해쉬를 만듭니다.  
    hash = ((hash << 5) - hash) + char;  
    // 32-bit 정수로 바꿉니다.  
    hash &= hash;  
  }  
}
```

좋은 예:

```
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;  
  
    // 32-bit 정수로 바꿉니다.  
    hash &= hash;  
  }  
}
```

[↑ 상단으로](#)

### 주석으로 된 코드를 남기지 마세요

버전 관리 도구가 존재하기 때문에 코드를 주석으로 남길 이유가 없습니다.

안좋은 예:

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

좋은 예:

```
doStuff();
```

[↑ 상단으로](#)

## 코드 기록을 주석으로 남기지 마세요

버전 관리 도구를 이용해야하는 것을 꼭 기억하세요. 죽은 코드도 불필요한 설명도 특히 코드의 기록에 대한 주석도 필요하지 않습니다. 코드의 기록에 대해 보고 싶다면 `git log` 를 사용하세요!

안좋은 예:

```
/**  
 * 2016-12-20: 모나드 제거했음, 이해는 되지 않음 (RM)  
 * 2016-10-01: 모나드 쓰는 로직 개선 (JP)  
 * 2016-02-03: 타입체킹 하는부분 제거 (LI)  
 * 2015-03-14: 버그 수정 (JR)  
 */  
function combine(a, b) {  
  return a + b;  
}
```

좋은 예:

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

[↑ 상단으로](#)

## 코드의 위치를 설명하지 마세요

이건 정말 쓸데 없습니다. 적절한 들여쓰기와 포매팅을 하고 함수와 변수의 이름에 의미를 부여하세요.

안좋은 예:

```

////////////////////////////////////
// 스코프 모델 정의
////////////////////////////////////
$scope.model = {
  menu: 'foo',
  nav: 'bar'
};

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

```

좋은 예:

```

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

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









```

[↑ 상단으로](#)

## 번역(Translation)

---

다른 언어로도 읽을 수 있습니다:

-  French: [GavBaros/clean-code-javascript-fr](#)
-  Brazilian Portuguese: [fesnt/clean-code-javascript](#)
-  Spanish: [andersontr15/clean-code-javascript](#)
-  Spanish: [tureey/clean-code-javascript](#)
-  Simplified Chinese:
  - [alivebao/clean-code-js](#)
  - [beginor/clean-code-javascript](#)
-  Traditional Chinese: [AllJointTW/clean-code-javascript](#)
-  German: [marcbruederlin/clean-code-javascript](#)
-  Korean: [qkraudghgh/clean-code-javascript-ko](#)

-  Polish: [greg-dev/clean-code-javascript-pl](#)
-  Russian:
  - [BoryaMogila/clean-code-javascript-ru/](#)
  - [maksugr/clean-code-javascript](#)
-  Vietnamese: [hienvd/clean-code-javascript/](#)
-  Japanese: [mitsuruog/clean-code-javascript/](#)
-  Indonesia: [andirkh/clean-code-javascript/](#)
-  Italian: [frappacchio/clean-code-javascript/](#)
-  Bangla(বাংলা): [InsomniacSabbir/clean-code-javascript/](#)

[↑](#) 상단으로

---

## Releases

No releases published

---

## Packages

No packages published