

| Clean Coding Practices

Table of Contents

- [Table of Contents](#)
 1. [Variables](#)
 2. [Functions](#)
 3. [Objects and Data Structures](#)
 4. [Classes](#)
 5. [SOLID](#)
 6. [Testing](#)
 7. [Concurrency](#)
 8. [Error Handling](#)
 9. [Formatting](#)
 10. [Comments](#)
 11. [Translation](#)

Variables

Use meaningful and pronounceable variable names

Bad:

```
const yyyymmddstr = moment().format("YYYY/MM/DD");
```

Good:

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

Use the same vocabulary for the same type of variable

Bad:

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

Good:

```
getUser();
```

Use searchable names

We will read more code than we will ever write. It's important that the code we do write is readable and searchable. By *not* naming variables that end up being meaningful for understanding our program, we hurt our readers.

Make your names searchable. Tools like

[buddy.js](#) and

[ESLint](#)

can help identify unnamed constants.

Bad:

```
// What the heck is 86400000 for?
setTimeout(blastOff, 86400000);
```

Good:

```
// Declare them as capitalized named constants.
const MILLISECONDS_PER_DAY = 60 * 60 * 24 * 1000; //86400000;

setTimeout(blastOff, MILLISECONDS_PER_DAY);
```

Use explanatory variables

Bad:

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

Good:

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

Avoid Mental Mapping

Explicit is better than implicit.

Bad:

```
const locations = ["Austin", "New York", "San Francisco"];
locations.forEach(l => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  // Wait, what is `l` for again?
  dispatch(l);
});
```

Good:

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

Don't add unneeded context

If your class/object name tells you something, don't repeat that in your variable name.

Bad:

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

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

Good:

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

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

Use default parameters instead of short circuiting or conditionals

Default parameters are often cleaner than short circuiting. Be aware that if you use them, your function will only provide default values for `undefined` arguments. Other "falsy" values such as `''`, `''`, `false`, `null`, `0`, and `NaN`, will not be replaced by a default value.

Bad:

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

Good:

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

Functions

Function arguments (2 or fewer ideally)

Limiting the amount of function parameters is incredibly important because it makes testing your function easier. Having more than three leads to a combinatorial explosion where you have to test tons of different cases with each separate argument.

One or two arguments is the ideal case, and three should be avoided if possible. Anything more than that should be consolidated. Usually, if you have

more than two arguments then your function is trying to do too much. In cases where it's not, most of the time a higher-level object will suffice as an argument.

Since JavaScript allows you to make objects on the fly, without a lot of class boilerplate, you can use an object if you are finding yourself needing a lot of arguments.

To make it obvious what properties the function expects, you can use the ES2015/ES6 destructuring syntax. This has a few advantages:

1. When someone looks at the function signature, it's immediately clear what properties are being used.
2. It can be used to simulate named parameters.
3. Destructuring also clones the specified primitive values of the argument object passed into the function. This can help prevent side effects. Note: objects and arrays that are destructured from the argument object are NOT cloned.
4. Linters can warn you about unused properties, which would be impossible without destructuring.

Bad:

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

Good:

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

Functions should do one thing

This is by far the most important rule in software engineering. When functions do more than one thing, they are harder to compose, test, and reason about. When you can isolate a function to just one action, it can be refactored easily and your code will read much cleaner. If you take nothing else away from this guide other than this, you'll be ahead of many developers.

Bad:

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

Good:

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

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

Function names should say what they do

Bad:

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

const date = new Date();

// It's hard to tell from the function name what is added
addToDate(date, 1);
```

Good:

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

```
}

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

Functions should only be one level of abstraction

When you have more than one level of abstraction your function is usually doing too much. Splitting up functions leads to reusability and easier testing.

Bad:

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

Good:

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

```

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

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

  return tokens;
}

function parse(tokens) {
  const syntaxTree = [];
  tokens.forEach(token => {
    syntaxTree.push(`/ * ... */`);
  });

  return syntaxTree;
}

```

Remove duplicate code

Do your absolute best to avoid duplicate code. Duplicate code is bad because it means that there's more than one place to alter something if you need to change some logic.

Imagine if you run a restaurant and you keep track of your inventory: all your tomatoes, onions, garlic, spices, etc. If you have multiple lists that you keep this on, then all have to be updated when you serve a dish with tomatoes in them. If you only have one list, there's only one place to update!

Oftentimes you have duplicate code because you have two or more slightly different things, that share a lot in common, but their differences force you to have two or more separate functions that do much of the same things. Removing duplicate code means creating an abstraction that can handle this set of different things with just one function/module/class.

Getting the abstraction right is critical, that's why you should follow the SOLID principles laid out in the [Classes](#) section. Bad abstractions can be worse than duplicate code, so be careful! Having said this, if you can make

a good abstraction, do it! Don't repeat yourself, otherwise you'll find yourself updating multiple places anytime you want to change one thing.

Bad:

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

Good:

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

    const data = {
      expectedSalary,
      experience
    };

    switch (employee.type) {
```

```

        case "manager":
            data.portfolio = employee.getMBAProjects();
            break;
        case "developer":
            data.githubLink = employee.getGithubLink();
            break;
    }

    render(data);
});
}

```

Set default objects with Object.assign

Bad:

```

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);

```

Good:

```

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

function createMenu(config) {
  let finalConfig = Object.assign(
    {
      title: "Foo",
      body: "Bar",

```

```

        buttonText: "Baz",
        cancellable: true
    },
    config
);
return finalConfig
// config now equals: {title: "Order", body: "Bar", buttonText: "Send",
cancellable: true}
// ...
}

createMenu(menuConfig);

```

Don't use flags as function parameters

Flags tell your user that this function does more than one thing. Functions should do one thing. Split out your functions if they are following different code paths based on a boolean.

Bad:

```

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

```

Good:

```

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

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

```

Avoid Side Effects (part 1)

A function produces a side effect if it does anything other than take a value in and return another value or values. A side effect could be writing to a file, modifying some global variable, or accidentally wiring all your money to a stranger.

Now, you do need to have side effects in a program on occasion. Like the previous example, you might need to write to a file. What you want to do is to centralize where you are doing this. Don't have several functions and classes that write to a particular file. Have one service that does it. One and only one.

The main point is to avoid common pitfalls like sharing state between objects without any structure, using mutable data types that can be written to by anything, and not centralizing where your side effects occur. If you can do this, you will be happier than the vast majority of other programmers.

Bad:

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

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

splitIntoFirstAndLastName();

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

Good:

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

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

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

[↑ back to top](#)

Avoid Side Effects (part 2)

In JavaScript, some values are unchangeable (immutable) and some are changeable (mutable). Objects and arrays are two kinds of mutable values so it's important to handle them carefully when they're passed as parameters to a function. A

JavaScript function can change an object's properties or alter the contents of an array which could easily cause bugs elsewhere.

Suppose there's a function that accepts an array parameter representing a shopping cart. If the function makes a change in that shopping cart array - by adding an item to purchase, for example - then any other function that uses that same `cart` array will be affected by this addition. That may be great, however it could also be bad. Let's imagine a bad situation:

The user clicks the "Purchase" button which calls a `purchase` function that spawns a network request and sends the `cart` array to the server. Because of a bad network connection, the `purchase` function has to keep retrying the request. Now, what if in the meantime the user accidentally clicks an "Add to Cart" button on an item they don't actually want before the network request begins? If that happens and the network request begins, then that purchase function will send the accidentally added item because the `cart` array was modified.

A great solution would be for the `addItemToCart` function to always clone the `cart`, edit it, and return the clone. This would ensure that functions that are still using the old shopping cart wouldn't be affected by the changes.

Two caveats to mention to this approach:

1. There might be cases where you actually want to modify the input object, but when you adopt this programming practice you will find that those cases are pretty rare. Most things can be refactored to have no side effects!
2. Cloning big objects can be very expensive in terms of performance. Luckily, this isn't a big issue in practice because there are [great libraries](#) that allow this kind of programming approach to be fast and not as memory intensive as it would be for you to manually clone objects and arrays.

Bad:

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

Good:

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

Don't write to global functions

Polluting globals is a bad practice in JavaScript because you could clash with another library and the user of your API would be none-the-wiser until they get an exception in production. Let's think about an example: what if you wanted to extend JavaScript's native Array method to have a `diff` method that could show the difference between two arrays? You could write your new function to the `Array.prototype`, but it could clash with another library that tried to do the same thing. What if that other library was just using `diff` to find the difference between the first and last elements of an array? This is why it would be much better to just use ES2015/ES6 classes and simply extend the `Array` global.

Bad:

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

Good:

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

Favor functional programming over imperative programming

JavaScript isn't a functional language in the way that Haskell is, but it has a functional flavor to it. Functional languages can be cleaner and easier to test. Favor this style of programming when you can.

Bad:

```
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;
  }

```

Good:

```

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.reduce(
  (totalLines, output) => totalLines + output.linesOfCode,
  0
);

```

Encapsulate conditionals

Bad:

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

Good:

```
function shouldShowSpinner(fsm, listNode) {  
  return fsm.state === "fetching" && isEmpty(listNode);  
}  
  
if (shouldShowSpinner(fsmInstance, listNodeInstance)) {  
  // ...  
}
```

Avoid negative conditionals

Bad:

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

Good:

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

Avoid conditionals

This seems like an impossible task. Upon first hearing this, most people say, "how am I supposed to do anything without an `if` statement?" The answer is that you can use polymorphism to achieve the same task in many cases. The second question is usually, "well that's great but why would I want to do that?" The answer is a previous clean code concept we learned: a function should only do

one thing. When you have classes and functions that have `if` statements, you are telling your user that your function does more than one thing. Remember, just do one thing.

Bad:

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

Good:

```
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();
    }
}
```

Avoid type-checking (part 1)

JavaScript is untyped, which means your functions can take any type of argument. Sometimes you are bitten by this freedom and it becomes tempting to do type-checking in your functions. There are many ways to avoid having to do this. The first thing to consider is consistent APIs.

Bad:

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

Good:

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

Avoid type-checking (part 2)

If you are working with basic primitive values like strings and integers, and you can't use polymorphism but you still feel the need to type-check, you should consider using TypeScript. It is an excellent alternative to normal JavaScript, as it provides you with static typing on top of standard JavaScript syntax. The problem with manually type-checking normal JavaScript is that doing it well requires so much extra verbiage that the faux "type-safety" you get doesn't make up for the lost readability. Keep your JavaScript clean, write good tests, and have good code reviews. Otherwise, do all of that but with TypeScript (which, like I said, is a great alternative!).

Bad:

```
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");  
}
```

Good:

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

Don't over-optimize

Modern browsers do a lot of optimization under-the-hood at runtime. A lot of times, if you are optimizing then you are just wasting your time. [There are good resources](#)

for seeing where optimization is lacking. Target those in the meantime, until they are fixed if they can be.

Bad:

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

Good:

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

Remove dead code

Dead code is just as bad as duplicate code. There's no reason to keep it in your codebase. If it's not being called, get rid of it! It will still be safe in your version history if you still need it.

Bad:

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

```

}

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

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

```

Good:

```

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

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

```

Objects and Data Structures

Use getters and setters

Using getters and setters to access data on objects could be better than simply looking for a property on an object. "Why?" you might ask. Well, here's an unorganized list of reasons why:

- When you want to do more beyond getting an object property, you don't have to look up and change every accessor in your codebase.
- Makes adding validation simple when doing a `set`.
- Encapsulates the internal representation.
- Easy to add logging and error handling when getting and setting.
- You can lazy load your object's properties, let's say getting it from a server.

Bad:

```

function makeBankAccount() {
  // ...

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

```

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

Good:

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

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

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

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

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

Make objects have private members

This can be accomplished through closures (for ES5 and below).

Bad:

```
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
```

Good:

```
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
```

Classes

Prefer ES2015/ES6 classes over ES5 plain functions

It's very difficult to get readable class inheritance, construction, and method definitions for classical ES5 classes. If you need inheritance (and be aware that you might not), then prefer ES2015/ES6 classes. However, prefer small functions over classes until you find yourself needing larger and more complex objects.

Bad:

```
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() {};

```

Good:

```

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

Use method chaining

This pattern is very useful in JavaScript and you see it in many libraries such as jQuery and Lodash. It allows your code to be expressive, and less verbose. For that reason, I say, use method chaining and take a look at how clean your code will be. In your class functions, simply return `this` at the end of every function, and you can chain further class methods onto it.

Bad:

```
class Car {  
    constructor(make, model, color) {  
        this.make = make;  
        this.model = model;  
        this.color = color;  
    }  
  
    setMake(make) {  
        this.make = make;  
    }  
  
    setModel(model) {  
        this.model = model;  
    }  
  
    setColor(color) {  
        this.color = color;  
    }  
  
    save() {  
        console.log(this.make, this.model, this.color);  
    }  
}  
  
const car = new Car("Ford", "F-150", "red");  
car.setColor("pink");  
car.save();
```

Good:


```

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

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

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

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

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

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

```

Prefer composition over inheritance

As stated famously in [Design Patterns](#) by the Gang of Four, you should prefer composition over inheritance where you can. There are lots of good reasons to use inheritance and lots of good reasons to use composition. The main point for this maxim is that if your mind instinctively goes for inheritance, try to think if composition could model your problem better. In some cases it can.

You might be wondering then, "when should I use inheritance?" It depends on your problem at hand, but this is a decent list of when inheritance makes more sense than composition:

1. Your inheritance represents an "is-a" relationship and not a "has-a" relationship (Human->Animal vs. User->UserDetails).
2. You can reuse code from the base classes (Humans can move like all animals).
3. You want to make global changes to derived classes by changing a base class. (Change the caloric expenditure of all animals when they move).

Bad:

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

    // ...
}

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

    // ...
}
```

Good:

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

    // ...
}

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

    setTaxData(ssn, salary) {
```

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

SOLID

Single Responsibility Principle (SRP)

As stated in Clean Code, "There should never be more than one reason for a class to change". It's tempting to jam-pack a class with a lot of functionality, like when you can only take one suitcase on your flight. The issue with this is that your class won't be conceptually cohesive and it will give it many reasons to change. Minimizing the amount of times you need to change a class is important. It's important because if too much functionality is in one class and you modify a piece of it, it can be difficult to understand how that will affect other dependent modules in your codebase.

Bad:

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

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

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

Good:

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

```

Open/Closed Principle (OCP)

As stated by Bertrand Meyer, "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." What does that mean though? This principle basically states that you should allow users to add new functionalities without changing existing code.

Bad:

```

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 === "nodeAdapter") {
    return makeHttpCall(url).then(response => {
      // transform response and return
    });
  }
}

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

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

```

Good:

```

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
    });
  }
}

```

```
});  
}  
}
```

Liskov Substitution Principle (LSP)

This is a scary term for a very simple concept. It's formally defined as "If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)." That's an even scarier definition.

The best explanation for this is if you have a parent class and a child class, then the base class and child class can be used interchangeably without getting incorrect results. This might still be confusing, so let's take a look at the classic Square-Rectangle example. Mathematically, a square is a rectangle, but if you model it using the "is-a" relationship via inheritance, you quickly get into trouble.

Bad:

```
class Rectangle {  
  constructor() {  
    this.width = 0;  
    this.height = 0;  
  }  
  
  setColor(color) {  
    // ...  
  }  
  
  render(area) {  
    // ...  
  }  
  
  setWidth(width) {  
    this.width = width;  
  }  
  
  setHeight(height) {  
    this.height = height;  
  }  
  
  getArea() {  
    return this.width * this.height;  
  }  
}
```

```

}

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

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

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

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

```

Good:

```

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);

```

Interface Segregation Principle (ISP)

JavaScript doesn't have interfaces so this principle doesn't apply as strictly as others. However, it's important and relevant even with JavaScript's lack of type system.

ISP states that "Clients should not be forced to depend upon interfaces that they do not use." Interfaces are implicit contracts in JavaScript because of duck typing.

A good example to look at that demonstrates this principle in JavaScript is for classes that require large settings objects. Not requiring clients to setup huge amounts of options is beneficial, because most of the time they won't need all of the settings. Making them optional helps prevent having a "fat interface".

Bad:

```

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

```



```

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

    traverse() {
        // ...
    }
}

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

```

Good:

```

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() {}
    }
});

```

Dependency Inversion Principle (DIP)

This principle states two essential things:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend on abstractions.

This can be hard to understand at first, but if you've worked with AngularJS, you've seen an implementation of this principle in the form of Dependency Injection (DI). While they are not identical concepts, DIP keeps high-level modules from knowing the details of its low-level modules and setting them up. It can accomplish this through DI. A huge benefit of this is that it reduces the coupling between modules. Coupling is a very bad development pattern because it makes your code hard to refactor.

As stated previously, JavaScript doesn't have interfaces so the abstractions that are depended upon are implicit contracts. That is to say, the methods and properties that an object/class exposes to another object/class. In the example below, the implicit contract is that any Request module for an `InventoryTracker` will have a `requestItems` method.

Bad:

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

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

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

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

  requestItems() {
```

```

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

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

```

Good:

```

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

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

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

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

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

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

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

```

```
new InventoryRequesterV2()  
);  
inventoryTracker.requestItems();
```

Testing

Testing is more important than shipping. If you have no tests or an inadequate amount, then every time you ship code you won't be sure that you didn't break anything. Deciding on what constitutes an adequate amount is up to your team, but having 100% coverage (all statements and branches) is how you achieve very high confidence and developer peace of mind. This means that in addition to having a great testing framework, you also need to use a [good coverage tool](#).

There's no excuse to not write tests. There are [plenty of good JS test frameworks](#), so find one that your team prefers.

When you find one that works for your team, then aim to always write tests for every new feature/module you introduce. If your preferred method is Test Driven Development (TDD), that is great, but the main point is to just make sure you are reaching your coverage goals before launching any feature, or refactoring an existing one.

Single concept per test

Bad:

```
import assert from "assert";  
  
describe("MomentJS", () => {  
  it("handles date boundaries", () => {  
    let date;  
  
    date = new MomentJS("1/1/2015");  
    date.addDays(30);  
    assert.equal("1/31/2015", date);  
  
    date = new MomentJS("2/1/2016");  
    date.addDays(28);  
    assert.equal("02/29/2016", date);  
  
    date = new MomentJS("2/1/2015");  
    date.addDays(28);  
    assert.equal("03/01/2015", date);
```

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

Good:

```
import assert from "assert";  
  
describe("MomentJS", () => {  
  it("handles 30-day months", () => {  
    const date = new MomentJS("1/1/2015");  
    date.addDays(30);  
    assert.equal("1/31/2015", date);  
  });  
  
  it("handles leap year", () => {  
    const date = new MomentJS("2/1/2016");  
    date.addDays(28);  
    assert.equal("02/29/2016", date);  
  });  
  
  it("handles non-leap year", () => {  
    const date = new MomentJS("2/1/2015");  
    date.addDays(28);  
    assert.equal("03/01/2015", date);  
  });  
});
```

Concurrency

Use Promises, not callbacks

Callbacks aren't clean, and they cause excessive amounts of nesting. With ES2015/ES6, Promises are a built-in global type. Use them!

Bad:

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

```

        console.error(writeErr);
    } else {
        console.log("File written");
    }
});
}
}
);

```

Good:

```

import { get } from "request-promise";
import { writeFile } from "fs-extra";

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

```

Async/Await are even cleaner than Promises

Promises are a very clean alternative to callbacks, but ES2017/ES8 brings `async` and `await` which offer an even cleaner solution. All you need is a function that is prefixed in an `async` keyword, and then you can write your logic imperatively without a `then` chain of functions. Use this if you can take advantage of ES2017/ES8 features today!

Bad:

```

import { get } from "request-promise";
import { writeFile } from "fs-extra";

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

```

```
    console.error(err);  
  });
```

Good:

```
import { get } from "request-promise";  
import { writeFile } from "fs-extra";  
  
async function getCleanCodeArticle() {  
  try {  
    const body = await get(  
      "https://en.wikipedia.org/wiki/Robert_Cecil_Martin"  
    );  
    await writeFile("article.html", body);  
    console.log("File written");  
  } catch (err) {  
    console.error(err);  
  }  
}  
  
getCleanCodeArticle()
```

Error Handling

Thrown errors are a good thing! They mean the runtime has successfully identified when something in your program has gone wrong and it's letting you know by stopping function execution on the current stack, killing the process (in Node), and notifying you in the console with a stack trace.

Don't ignore caught errors

Doing nothing with a caught error doesn't give you the ability to ever fix or react to said error. Logging the error to the console (`console.log`) isn't much better as often times it can get lost in a sea of things printed to the console. If you wrap any bit of code in a `try/catch` it means you think an error may occur there and therefore you should have a plan, or create a code path, for when it occurs.

Bad:

```
try {  
  functionThatMightThrow();  
} catch (error) {
```

```
    console.log(error);  
  }
```

Good:

```
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!  
}
```

Don't ignore rejected promises

For the same reason you shouldn't ignore caught errors from `try/catch`.

Bad:

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

Good:

```
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!  
});
```

Formatting

Formatting is subjective. Like many rules herein, there is no hard and fast rule that you must follow. The main point is DO NOT ARGUE over formatting.

There are [tons of tools](#) to automate this.

Use one! It's a waste of time and money for engineers to argue over formatting.

For things that don't fall under the purview of automatic formatting (indentation, tabs vs. spaces, double vs. single quotes, etc.) look here for some guidance.

Use consistent capitalization

JavaScript is untyped, so capitalization tells you a lot about your variables, functions, etc. These rules are subjective, so your team can choose whatever they want. The point is, no matter what you all choose, just be consistent.

Bad:

```
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 {}
```

Good:

```
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 {}
```

Function callers and callees should be close

If a function calls another, keep those functions vertically close in the source file. Ideally, keep the caller right above the callee. We tend to read code from top-to-bottom, like a newspaper. Because of this, make your code read that way.

Bad:

```
class PerformanceReview {  
  constructor(employee) {  
    this.employee = employee;  
  }  
  
  lookupPeers() {  
    return db.lookup(this.employee, "peers");  
  }  
  
  lookupManager() {  
    return db.lookup(this.employee, "manager");  
  }  
  
  getPeerReviews() {  
    const peers = this.lookupPeers();  
    // ...  
  }  
  
  perfReview() {  
    this.getPeerReviews();  
    this.getManagerReview();  
    this.getSelfReview();  
  }  
  
  getManagerReview() {  
    const manager = this.lookupManager();  
  }  
  
  getSelfReview() {  
    // ...  
  }  
}  
  
const review = new PerformanceReview(employee);  
review.perfReview();
```

Good:

```
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();
```

Comments

Only comment things that have business logic complexity.

Comments are an apology, not a requirement. Good code *mostly* documents itself.

Bad:

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

  // Length of string
  const length = data.length;

  // Loop through every character in data
  for (let i = 0; i < length; i++) {
    // Get character code.
    const char = data.charCodeAt(i);
    // Make the hash
    hash = (hash << 5) - hash + char;
    // Convert to 32-bit integer
    hash &= hash;
  }
}
```

Good:

```
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;

    // Convert to 32-bit integer
    hash &= hash;
  }
}
```

Don't leave commented out code in your codebase

Version control exists for a reason. Leave old code in your history.

Bad:

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

Good:

```
doStuff();
```

Don't have journal comments

Remember, use version control! There's no need for dead code, commented code, and especially journal comments. Use `git log` to get history!

Bad:

```
/**
 * 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;
}
```

Good:

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

Avoid positional markers

They usually just add noise. Let the functions and variable names along with the proper indentation and formatting give the visual structure to your code.

Bad:

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

////////////////////////////////////
/////
```

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

Good:

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

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