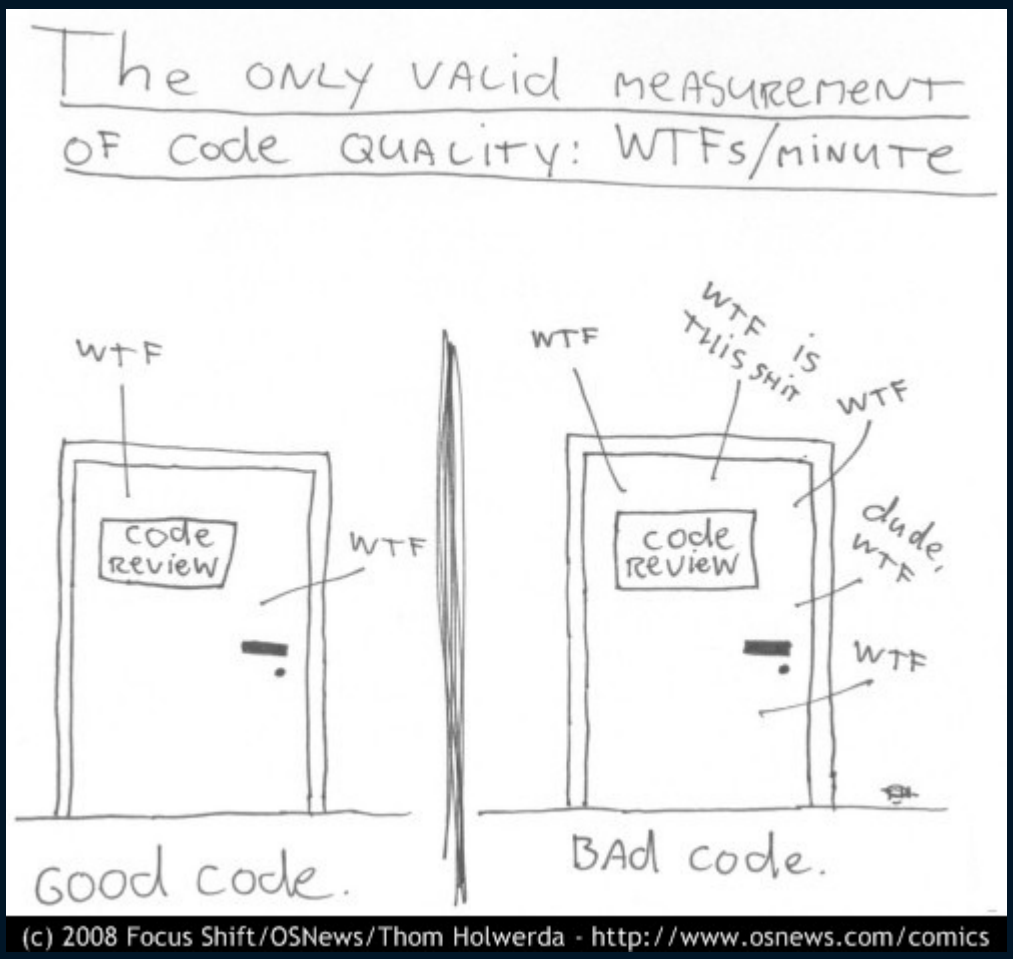
Book Cover

Table of Contents

Clean Code Typescript

Clean Code concepts adapted for TypeScript. Inspired from clean-code-javascript.

Introduction



Software engineering principles, from Robert C. Martin's book *Clean Code*, adapted for TypeScript. This is not a style guide. It's a guide to producing readable, reusable, and refactorable software in TypeScript.

Not every principle herein has to be strictly followed, and even fewer will be universally agreed upon. These are guidelines and nothing more, but they are ones codified over many years of collective experience by the authors of *Clean Code*.

Our craft of software engineering is just a bit over 50 years old, and we are still learning a lot. When software architecture is as old as architecture itself, maybe then we will have harder rules to follow. For now, let these guidelines serve as a touchstone by which to assess the quality of the TypeScript code that you and your team produce.

One more thing: knowing these won't immediately make you a better software developer, and working with them for many years doesn't mean you won't make mistakes. Every piece of code starts as a first draft, like wet clay getting shaped into its final form. Finally, we chisel away the imperfections when we review it with our peers. Don't beat yourself up for first drafts that need improvement. Beat up the code instead!

Variables

Use meaningful variable names

Distinguish names in such a way that the reader knows what the differences offer.

Bad:

```
function between<T>(a1: T, a2: T, a3: T): boolean {
  return a2 <= a1 && a1 <= a3;
}
```

Good:

```
function between<T>(value: T, left: T, right: T): boolean {
  return left <= value && value <= right;
}
```

Use pronounceable variable names

If you can’t pronounce it, you can’t discuss it without sounding like an idiot.

Bad:

```
type DtaRcrd102 = {
  genymdhms: Date;
  modymdhms: Date;
  pszqint: number;
}
```

Good:

```
type Customer = {
  generationTimestamp: Date;
  modificationTimestamp: Date;
  recordId: number;
}
```

Use the same vocabulary for the same type of variable

Bad:

```
function getUserInfo(): User;
function getUserDetails(): User;
function getUserData(): User;
```

Good:

```
function getUser(): User;
```

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 TSLint can help identify unnamed constants.

Bad:

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

Good:

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

setTimeout(restart, MILLISECONDS_IN_A_DAY);
```

Use explanatory variables

Bad:

```
declare const users: Map<string, User>;

for (const keyValue of users) {
  // iterate through users map
}
```

Good:

```
declare const users: Map<string, User>;

for (const [id, user] of users) {
  // iterate through users map
}
```

Avoid Mental Mapping

Explicit is better than implicit. *Clarity is king.*

Bad:

```
const u = getUser();
const s = getSubscription();
const t = charge(u, s);
```

Good:

```
const user = getUser();
const subscription = getSubscription();
const transaction = charge(user, subscription);
```

Don't add unneeded context

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

Bad:

```
type Car = {
  carMake: string;
  carModel: string;
  carColor: string;
}

function print(car: Car): void {
  console.log(`${car.carMake} ${car.carModel} (${car.carColor})`);
}
```

Good:

```
type Car = {
  make: string;
  model: string;
  color: string;
}

function print(car: Car): void {
  console.log(`${car.make} ${car.model} (${car.color})`);
}
```

Use default arguments instead of short circuiting or conditionals

Default arguments are often cleaner than short circuiting.

Bad:

```
function loadPages(count?: number) {
  const loadCount = count !== undefined ? count : 10;
  // ...
}
```

Good:

```
function loadPages(count: number = 10) {
  // ...
}
```

Use enum to document the intent

Enums can help you document the intent of the code. For example when we are concerned about values being different rather than the exact value of those.

Bad:

```
const GENRE = {
  ROMANTIC: 'romantic',
  DRAMA: 'drama',
  COMEDY: 'comedy',
  DOCUMENTARY: 'documentary',
}

projector.configureFilm(GENRE.COMEDY);

class Projector {
  // declaration of Projector
  configureFilm(genre) {
    switch (genre) {
      case GENRE.ROMANTIC:
        // some logic to be executed
    }
  }
}
```

Good:

```
enum GENRE {
  ROMANTIC,
  DRAMA,
  COMEDY,
  DOCUMENTARY,
}

projector.configureFilm(GENRE.COMEDY);

class Projector {
  // declaration of Projector
  configureFilm(genre) {
    switch (genre) {
      case GENRE.ROMANTIC:
        // some logic to be executed
    }
  }
}
```

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.

Consider using object literals if you are finding yourself needing a lot of arguments.

To make it obvious what properties the function expects, you can use the 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. TypeScript warns you about unused properties, which would be impossible without destructuring.

Bad:

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

createMenu('Foo', 'Bar', 'Baz', true);
```

Good:

```
function createMenu(options: { title: string, body: string, buttonText:
string, cancellable: boolean }) {
  // ...
}

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

You can further improve readability by using type aliases:

```
type MenuOptions = { title: string, body: string, buttonText: string,
cancellable: boolean };

function createMenu(options: MenuOptions) {
  // ...
}

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: Client[]) {
  clients.forEach((client) => {
    const clientRecord = database.lookup(client);
    if (clientRecord.isActive()) {
      email(client);
    }
  });
}
```

Good:

```
function emailClients(clients: Client[]) {
  clients.filter(isActiveClient).forEach(email);
}

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

Function names should say what they do

Bad:

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

const date = new Date();

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

Good:

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

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

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 parseCode(code: string) {
  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:

```
const REGEXES = [ /* ... */ ];

function parseCode(code: string) {
  const tokens = tokenize(code);
  const syntaxTree = parse(tokens);

  syntaxTree.forEach((node) => {
    // parse...
  });
}

function tokenize(code: string): Token[] {
  const statements = code.split(' ');
  const tokens: Token[] = [];

  REGEXES.forEach((regex) => {
    statements.forEach((statement) => {
      tokens.push( /* ... */ );
    });
  });

  return tokens;
}

function parse(tokens: Token[]): SyntaxTree {
  const syntaxTree: 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. 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: Developer[]) {
  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: Manager[]) {
  managers.forEach((manager) => {
    const expectedSalary = manager.calculateExpectedSalary();
    const experience = manager.getExperience();
    const portfolio = manager.getMBAProjects();

    const data = {
      expectedSalary,
      experience,
      portfolio
    };

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

Good:

```
class Developer {
  // ...
  getExtraDetails() {
    return {
      githubLink: this.githubLink,
    }
  }
}

class Manager {
  // ...
  getExtraDetails() {
    return {
      portfolio: this.portfolio,
    }
  }
}

function showEmployeeList(employee: Developer | Manager) {
  employee.forEach((employee) => {
    const expectedSalary = employee.calculateExpectedSalary();
    const experience = employee.getExperience();
    const extra = employee.getExtraDetails();

    const data = {
      expectedSalary,
      experience,
      extra,
    };

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

You should be critical about code duplication. Sometimes there is a tradeoff between duplicated code and increased complexity by introducing unnecessary abstraction. When two implementations from two different modules look similar but live in different domains, duplication might be acceptable and preferred over extracting the common code. The extracted common code in this case introduces an indirect dependency between the two modules.

Set default objects with Object.assign or destructuring

Bad:

```
type MenuConfig = { title?: string, body?: string, buttonText?: string,
cancellable?: boolean };

function createMenu(config: MenuConfig) {
  config.title = config.title || 'Foo';
  config.body = config.body || 'Bar';
  config.buttonText = config.buttonText || 'Baz';
  config.cancellable = config.cancellable !== undefined ?
config.cancellable : true;

  // ...
}

createMenu({ body: 'Bar' });
```

Good:

```
type MenuConfig = { title?: string, body?: string, buttonText?: string,
cancellable?: boolean };

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

  // ...
}

createMenu({ body: 'Bar' });
```

Alternatively, you can use destructuring with default values:

```
type MenuConfig = { title?: string, body?: string, buttonText?: string,
cancellable?: boolean };

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

createMenu({ body: 'Bar' });
```

To avoid any side effects and unexpected behavior by passing in explicitly the `undefined` or `null` value, you can tell the TypeScript compiler to not allow it. See `--strictNullChecks` option in TypeScript.

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: string, temp: boolean) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
}
```

Good:

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

function createFile(name: string) {
  fs.create(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.
let name = 'Robert C. Martin';

function toBase64() {
  name = btoa(name);
}

toBase64();
// If we had another function that used this name, now it'd be a Base64
value

console.log(name); // expected to print 'Robert C. Martin' but instead
'Um9iZXJ0IEMuIE1hcnRpbg=='
```

Good:

```
const name = 'Robert C. Martin';

function toBase64(text: string): string {
  return btoa(text);
}

const encodedName = toBase64(name);
console.log(name);
```

Avoid Side Effects (part 2)

In JavaScript, primitives are passed by value and objects/arrays are passed by reference. In the case of objects and arrays, if your function makes a change in a shopping cart array, for example, by adding an item to purchase, then any other function that uses that `cart` array will be affected by this addition. That may be great, however it can be bad too. 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 "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 it has a reference to a shopping cart array that the `addItemToCart` function modified by adding an unwanted item.

A great solution would be for the `addItemToCart` to always clone the `cart`, edit it, and return the clone. This ensures that no other functions that are holding onto a reference of the shopping cart will be affected by any 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! (see pure function)

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:

```
function addItemToCart(cart: CartItem[], item: Item): void {
  cart.push({ item, date: Date.now() });
};
```

Good:

```
function addItemToCart(cart: CartItem[], item: Item): CartItem[] {
  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 classes and simply extend the `Array` global.

Bad:

```
declare global {
  interface Array<T> {
    diff(other: T[]): Array<T>;
  }
}

if (!Array.prototype.diff) {
  Array.prototype.diff = function <T>(other: T[]): T[] {
    const hash = new Set(other);
    return this.filter(elem => !hash.has(elem));
  };
}
```

Good:

```
class MyArray<T> extends Array<T> {
  diff(other: T[]): T[] {
    const hash = new Set(other);
    return this.filter(elem => !hash.has(elem));
  };
}
```

Favor functional programming over imperative programming

Favor this style of programming when you can.

Bad:

```
const contributions = [
  {
    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 < contributions.length; i++) {
  totalOutput += contributions[i].linesOfCode;
}
```

Good:

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

const totalOutput = contributions
  .reduce((totalLines, output) => totalLines + output.linesOfCode, 0);
```

Encapsulate conditionals

Bad:

```
if (subscription.isTrial || account.balance > 0) {
  // ...
}
```

Good:

```
function canActivateService(subscription: Subscription, account:
Account) {
  return subscription.isTrial || account.balance > 0;
}

if (canActivateService(subscription, account)) {
  // ...
}
```

Avoid negative conditionals

Bad:

```
function isEmailNotUsed(email: string): boolean {
  // ...
}

if (isEmailNotUsed(email)) {
  // ...
}
```

Good:

```
function isEmailUsed(email: string): boolean {
  // ...
}

if (!isEmailUsed(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 {
  private type: string;
  // ...

  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();
      default:
        throw new Error('Unknown airplane type.');
    }
  }

  private getMaxAltitude(): number {
    // ...
  }
}
```

Good:

```
abstract class Airplane {
  protected getMaxAltitude(): number {
    // shared logic with subclasses ...
  }

  // ...
}

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

TypeScript is a strict syntactical superset of JavaScript and adds optional static type checking to the language. Always prefer to specify types of variables, parameters and return values to leverage the full power of TypeScript features. It makes refactoring more easier.

Bad:

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

Good:

```
type Vehicle = Bicycle | Car;

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

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: string) {
  // ...
}

function requestModule(url: string) {
  // ...
}

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

Good:

```
function requestModule(url: string) {
  // ...
}

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

Use iterators and generators

Use generators and iterables when working with collections of data used like a stream.
There are some good reasons:

- decouples the callee from the generator implementation in a sense that callee decides how many items to access

- lazy execution, items are streamed on demand
- built-in support for iterating items using the `for-of` syntax
- iterables allow to implement optimized iterator patterns

Bad:

```
function fibonacci(n: number): number[] {
  if (n === 1) return [0];
  if (n === 2) return [0, 1];

  const items: number[] = [0, 1];
  while (items.length < n) {
    items.push(items[items.length - 2] + items[items.length - 1]);
  }

  return items;
}

function print(n: number) {
  fibonacci(n).forEach(fib => console.log(fib));
}

// Print first 10 Fibonacci numbers.
print(10);
```

Good:

```
// Generates an infinite stream of Fibonacci numbers.
// The generator doesn't keep the array of all numbers.
function* fibonacci(): IterableIterator<number> {
  let [a, b] = [0, 1];

  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

function print(n: number) {
  let i = 0;
  for (const fib of fibonacci()) {
    if (i++ === n) break;
    console.log(fib);
  }
}

// Print first 10 Fibonacci numbers.
print(10);
```

There are libraries that allow working with iterables in a similar way as with native arrays, by chaining methods like `map`, `slice`, `forEach` etc. See itiriri for an example of advanced manipulation with iterables (or itiriri-async for manipulation of async iterables).

```
import itiriri from 'itiriri';

function* fibonacci(): IterableIterator<number> {
  let [a, b] = [0, 1];

  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

itiriri(fibonacci())
  .take(10)
  .forEach(fib => console.log(fib));
```

Objects and Data Structures

Use getters and setters

TypeScript supports getter/setter syntax. Using getters and setters to access data from objects that encapsulate behavior could be better than simply looking for a property on an object. "Why?" you might ask. Well, here's a list of reasons:

- 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:

```
type BankAccount = {
  balance: number;
  // ...
}

const value = 100;
const account: BankAccount = {
  balance: 0,
  // ...
};

if (value < 0) {
  throw new Error('Cannot set negative balance.');
}

account.balance = value;
```

Good:

```
class BankAccount {
  private accountBalance: number = 0;

  get balance(): number {
    return this.accountBalance;
  }

  set balance(value: number) {
    if (value < 0) {
      throw new Error('Cannot set negative balance.');
    }

    this.accountBalance = value;
  }

  // ...
}

// Now `BankAccount` encapsulates the validation logic.
// If one day the specifications change, and we need extra validation
rule,
// we would have to alter only the `setter` implementation,
// leaving all dependent code unchanged.
const account = new BankAccount();
account.balance = 100;
```

Make objects have private/protected members

TypeScript supports `public` *(default)*, `protected` and `private` accessors on class members.

Bad:

```
class Circle {
  radius: number;

  constructor(radius: number) {
    this.radius = radius;
  }

  perimeter() {
    return 2 * Math.PI * this.radius;
  }

  surface() {
    return Math.PI * this.radius * this.radius;
  }
}
```

Good:

```
class Circle {
  constructor(private readonly radius: number) {
  }

  perimeter() {
    return 2 * Math.PI * this.radius;
  }

  surface() {
    return Math.PI * this.radius * this.radius;
  }
}
```

Prefer immutability

TypeScript's type system allows you to mark individual properties on an interface / class as *readonly*. This allows you to work in a functional way (unexpected mutation is bad). For more advanced scenarios there is a built-in type `Readonly` that takes a type `T` and marks all of its properties as readonly using mapped types (see mapped types).

Bad:

```
interface Config {
  host: string;
  port: string;
  db: string;
}
```

Good:

```
interface Config {
  readonly host: string;
  readonly port: string;
  readonly db: string;
}
```

Case of Array, you can create a read-only array by using `ReadonlyArray<T>`. do not allow changes such as `push()` and `fill()`, but can use features such as `concat()` and `slice()` that do not change the value.

Bad:

```
const array: number[] = [ 1, 3, 5 ];
array = []; // error
array.push(100); // array will updated
```

Good:

```
const array: ReadonlyArray<number> = [ 1, 3, 5 ];
array = []; // error
array.push(100); // error
```

Declaring read-only arguments in TypeScript 3.4 is a bit easier.

```
function hoge(args: readonly string[]) {
  args.push(1); // error
}
```

Prefer const assertions for literal values.

Bad:

```
const config = {
  hello: 'world'
};
config.hello = 'world'; // value is changed

const array  = [ 1, 3, 5 ];
array[0] = 10; // value is changed

// writable objects is returned
function readonlyData(value: number) {
  return { value };
}

const result = readonlyData(100);
result.value = 200; // value is changed
```

Good:

```
// read-only object
const config = {
  hello: 'world'
} as const;
config.hello = 'world'; // error

// read-only array
const array  = [ 1, 3, 5 ] as const;
array[0] = 10; // error

// You can return read-only objects
function readonlyData(value: number) {
  return { value } as const;
}

const result = readonlyData(100);
result.value = 200; // error
```

type vs. interface

Use type when you might need a union or intersection. Use interface when you want `extends` or `implements`. There is no strict rule however, use the one that works for you. For a more detailed explanation refer to this answer about the differences between `type` and `interface` in TypeScript.

Bad:

```
interface EmailConfig {
  // ...
}

interface DbConfig {
  // ...
}

interface Config {
  // ...
}

//...

type Shape = {
  // ...
}
```

Good:

```ts
type EmailConfig = {
  // ...
}

type DbConfig = {
  // ...
}

type Config  = EmailConfig | DbConfig;

// ...

interface Shape {
  // ...
}

class Circle implements Shape {
  // ...
}

class Square implements Shape {
  // ...
}
```

Classes

Classes should be small

The class' size is measured by its responsibility. Following the *Single Responsibility principle* a class should be small.

Bad:

```
class Dashboard {
  getLanguage(): string { /* ... */ }
  setLanguage(language: string): void { /* ... */ }
  showProgress(): void { /* ... */ }
  hideProgress(): void { /* ... */ }
  isDirty(): boolean { /* ... */ }
  disable(): void { /* ... */ }
  enable(): void { /* ... */ }
  addSubscription(subscription: Subscription): void { /* ... */ }
  removeSubscription(subscription: Subscription): void { /* ... */ }
  addUser(user: User): void { /* ... */ }
  removeUser(user: User): void { /* ... */ }
  goToHomePage(): void { /* ... */ }
  updateProfile(details: UserDetails): void { /* ... */ }
  getVersion(): string { /* ... */ }
  // ...
}
```

Good:

```
class Dashboard {
  disable(): void { /* ... */ }
  enable(): void { /* ... */ }
  getVersion(): string { /* ... */ }
}

// split the responsibilities by moving the remaining methods to other
classes
// ...
```

High cohesion and low coupling

Cohesion defines the degree to which class members are related to each other. Ideally, all fields within a class should be used by each method. We then say that the class is *maximally cohesive*. In practice, this however is not always possible, nor even advisable. You should however prefer cohesion to be high.

Coupling refers to how related or dependent are two classes toward each other. Classes are said to be low coupled if changes in one of them doesn't affect the other one.

Good software design has **high cohesion** and **low coupling**.

Bad:

```
class UserManager {
  // Bad: each private variable is used by one or another group of
methods.
  // It makes clear evidence that the class is holding more than a
single responsibility.
  // If I need only to create the service to get the transactions for a
user,
  // I'm still forced to pass and instance of `emailSender`.
  constructor(
    private readonly db: Database,
    private readonly emailSender: EmailSender) {
  }

  async getUser(id: number): Promise<User> {
    return await db.users.findOne({ id });
  }

  async getTransactions(userId: number): Promise<Transaction[]> {
    return await db.transactions.find({ userId });
  }

  async sendGreeting(): Promise<void> {
    await emailSender.send('Welcome!');
  }

  async sendNotification(text: string): Promise<void> {
    await emailSender.send(text);
  }

  async sendNewsletter(): Promise<void> {
    // ...
  }
}
```

Good:

```
class UserService {
  constructor(private readonly db: Database) {
  }

  async getUser(id: number): Promise<User> {
    return await this.db.users.findOne({ id });
  }

  async getTransactions(userId: number): Promise<Transaction[]> {
    return await this.db.transactions.find({ userId });
  }
}

class UserNotifier {
  constructor(private readonly emailSender: EmailSender) {
  }

  async sendGreeting(): Promise<void> {
    await this.emailSender.send('Welcome!');
  }

  async sendNotification(text: string): Promise<void> {
    await this.emailSender.send(text);
  }

  async sendNewsletter(): Promise<void> {
    // ...
  }
}
```

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(
    private readonly name: string,
    private readonly email: string) {
  }

  // ...
}

// Bad because Employees "have" tax data. EmployeeTaxData is not a type
of Employee
class EmployeeTaxData extends Employee {
  constructor(
    name: string,
    email: string,
    private readonly ssn: string,
    private readonly salary: number) {
    super(name, email);
  }

  // ...
}
```

Good:

```
class Employee {
  private taxData: EmployeeTaxData;

  constructor(
    private readonly name: string,
    private readonly email: string) {
  }

  setTaxData(ssn: string, salary: number): Employee {
    this.taxData = new EmployeeTaxData(ssn, salary);
    return this;
  }

  // ...
}

class EmployeeTaxData {
  constructor(
    public readonly ssn: string,
    public readonly salary: number) {
  }

  // ...
}
```

Use method chaining

This pattern is very useful and commonly used in many libraries. It allows your code to be expressive, and less verbose. For that reason, use method chaining and take a look at how clean your code will be.

Bad:

```
class QueryBuilder {
  private collection: string;
  private pageNumber: number = 1;
  private itemsPerPage: number = 100;
  private orderByFields: string[] = [];

  from(collection: string): void {
    this.collection = collection;
  }

  page(number: number, itemsPerPage: number = 100): void {
    this.pageNumber = number;
    this.itemsPerPage = itemsPerPage;
  }

  orderBy(...fields: string[]): void {
    this.orderByFields = fields;
  }

  build(): Query {
    // ...
  }
}

// ...

const queryBuilder = new QueryBuilder();
queryBuilder.from('users');
queryBuilder.page(1, 100);
queryBuilder.orderBy('firstName', 'lastName');

const query = queryBuilder.build();
```

Good:

```
class QueryBuilder {
  private collection: string;
  private pageNumber: number = 1;
  private itemsPerPage: number = 100;
  private orderByFields: string[] = [];

  from(collection: string): this {
    this.collection = collection;
    return this;
  }

  page(number: number, itemsPerPage: number = 100): this {
    this.pageNumber = number;
    this.itemsPerPage = itemsPerPage;
    return this;
  }

  orderBy(...fields: string[]): this {
    this.orderByFields = fields;
    return this;
  }

  build(): Query {
    // ...
  }
}

// ...

const query = new QueryBuilder()
  .from('users')
  .page(1, 100)
  .orderBy('firstName', 'lastName')
  .build();
```

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(private readonly user: User) {
  }

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

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

Good:

```
class UserAuth {
  constructor(private readonly user: User) {
  }

  verifyCredentials() {
    // ...
  }
}


class UserSettings {
  private readonly auth: UserAuth;

  constructor(private readonly user: User) {
    this.auth = new UserAuth(user);
  }

  changeSettings(settings: UserSettings) {
    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();
  }

  // ...
}

class NodeAdapter extends Adapter {
  constructor() {
    super();
  }

  // ...
}

class HttpRequester {
  constructor(private readonly adapter: Adapter) {
  }

  async fetch<T>(url: string): Promise<T> {
    if (this.adapter instanceof AjaxAdapter) {
      const response = await makeAjaxCall<T>(url);
      // transform response and return
    } else if (this.adapter instanceof NodeAdapter) {
      const response = await makeHttpCall<T>(url);
      // transform response and return
    }
  }
}

function makeAjaxCall<T>(url: string): Promise<T> {
  // request and return promise
}

function makeHttpCall<T>(url: string): Promise<T> {
  // request and return promise
}
```

Good:

```
abstract class Adapter {
  abstract async request<T>(url: string): Promise<T>;

  // code shared to subclasses ...
}

class AjaxAdapter extends Adapter {
  constructor() {
    super();
  }

  async request<T>(url: string): Promise<T>{
    // request and return promise
  }

  // ...
}

class NodeAdapter extends Adapter {
  constructor() {
    super();
  }

  async request<T>(url: string): Promise<T>{
    // request and return promise
  }

  // ...
}

class HttpRequester {
  constructor(private readonly adapter: Adapter) {
  }

  async fetch<T>(url: string): Promise<T> {
    const response = await this.adapter.request<T>(url);
    // 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 parent 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(
    protected width: number = 0,
    protected height: number = 0) {

  }

  setColor(color: string): this {
    // ...
  }

  render(area: number) {
    // ...
  }

  setWidth(width: number): this {
    this.width = width;
    return this;
  }

  setHeight(height: number): this {
    this.height = height;
    return this;
  }

  getArea(): number {
    return this.width * this.height;
  }
}

class Square extends Rectangle {
  setWidth(width: number): this {
    this.width = width;
```

```
    this.height = width;
    return this;
  }

  setHeight(height: number): this {
    this.width = height;
    this.height = height;
    return this;
  }
}

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

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

Good:

```
abstract class Shape {
  setColor(color: string): this {
    // ...
  }

  render(area: number) {
    // ...
  }

  abstract getArea(): number;
}

class Rectangle extends Shape {
  constructor(
    private readonly width = 0,
    private readonly height = 0) {
    super();
  }
```

```
    getArea(): number {
      return this.width * this.height;
    }
  }

  class Square extends Shape {
    constructor(private readonly length: number) {
      super();
    }

    getArea(): number {
      return this.length * this.length;
    }
  }

  function renderLargeShapes(shapes: Shape[]) {
    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)

ISP states that "Clients should not be forced to depend upon interfaces that they do not use.". This principle is very much related to the Single Responsibility Principle. What it really means is that you should always design your abstractions in a way that the clients that are using the exposed methods do not get the whole pie instead. That also include imposing the clients with the burden of implementing methods that they don’t actually need.

Bad:

```
interface SmartPrinter {
  print();
  fax();
  scan();
}

class AllInOnePrinter implements SmartPrinter {
  print() {
    // ...
  }

  fax() {
    // ...
  }

  scan() {
    // ...
  }
}

class EconomicPrinter implements SmartPrinter {
  print() {
    // ...
  }

  fax() {
    throw new Error('Fax not supported.');
  }

  scan() {
    throw new Error('Scan not supported.');
  }
}
```

Good:

```
interface Printer {
  print();
}

interface Fax {
  fax();
}

interface Scanner {
  scan();
}

class AllInOnePrinter implements Printer, Fax, Scanner {
  print() {
    // ...
  }

  fax() {
    // ...
  }

  scan() {
    // ...
  }
}

class EconomicPrinter implements Printer {
  print() {
    // ...
  }
}
```

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 Angular, 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.

DIP is usually achieved by a using an inversion of control (IoC) container. An example of a powerful IoC container for TypeScript is InversifyJs

Bad:

```
import { readFile as readFileCb } from 'fs';
import { promisify } from 'util';

const readFile = promisify(readFileCb);

type ReportData = {
  // ..
}

class XmlFormatter {
  parse<T>(content: string): T {
    // Converts an XML string to an object T
  }
}

class ReportReader {

  // BAD: We have created a dependency on a specific request
implementation.
  // We should just have ReportReader depend on a parse method: `parse`
  private readonly formatter = new XmlFormatter();

  async read(path: string): Promise<ReportData> {
    const text = await readFile(path, 'UTF8');
    return this.formatter.parse<ReportData>(text);
  }
}

// ...
const reader = new ReportReader();
await report = await reader.read('report.xml');
```

Good:

```
import { readFile as readFileCb } from 'fs';
import { promisify } from 'util';

const readFile = promisify(readFileCb);

type ReportData = {
  // ..
}

interface Formatter {
  parse<T>(content: string): T;
}

class XmlFormatter implements Formatter {
  parse<T>(content: string): T {
    // Converts an XML string to an object T
  }
}


class JsonFormatter implements Formatter {
  parse<T>(content: string): T {
    // Converts a JSON string to an object T
  }
}

class ReportReader {
  constructor(private readonly formatter: Formatter) {
  }

  async read(path: string): Promise<ReportData> {
    const text = await readFile(path, 'UTF8');
    return this.formatter.parse<ReportData>(text);
  }
}

// ...
const reader = new ReportReader(new XmlFormatter());
await report = await reader.read('report.xml');

// or if we had to read a json report
const reader = new ReportReader(new JsonFormatter());
await report = await reader.read('report.json');
```

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 with typings support for TypeScript, 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.

The three laws of TDD

1. You are not allowed to write any production code unless it is to make a failing unit test pass.

2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.

3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

F.I.R.S.T. rules

Clean tests should follow the rules:

- **Fast** tests should be fast because we want to run them frequently.

- **Independent** tests should not depend on each other. They should provide same output whether run independently or all together in any order.

- **Repeatable** tests should be repeatable in any environment and there should be no

excuse for why they fail.

- **Self-Validating** a test should answer with either *Passed* or *Failed*. You don't need to compare log files to answer if a test passed.

- **Timely** unit tests should be written before the production code. If you write tests after the production code, you might find writing tests too hard.

Single concept per test

Tests should also follow the *Single Responsibility Principle*. Make only one assert per unit test.

Bad:

```
import { assert } from 'chai';

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

    date = new AwesomeDate('1/1/2015');
    assert.equal('1/31/2015', date.addDays(30));

    date = new AwesomeDate('2/1/2016');
    assert.equal('2/29/2016', date.addDays(28));

    date = new AwesomeDate('2/1/2015');
    assert.equal('3/1/2015', date.addDays(28));
  });
});
```

Good:

```
import { assert } from 'chai';

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

  it('handles leap year', () => {
    const date = new AwesomeDate('2/1/2016');
    assert.equal('2/29/2016', date.addDays(28));
  });

  it('handles non-leap year', () => {
    const date = new AwesomeDate('2/1/2015');
    assert.equal('3/1/2015', date.addDays(28));
  });
});
```

The name of the test should reveal its intention

When a test fail, its name is the first indication of what may have gone wrong.

Bad:

```
describe('Calendar', () => {
  it('2/29/2020', () => {
    // ...
  });

  it('throws', () => {
    // ...
  });
});
```

Good:

```
describe('Calendar', () => {
  it('should handle leap year', () => {
    // ...
  });

  it('should throw when format is invalid', () => {
    // ...
  });
});
```

Concurrency

Prefer promises vs callbacks

Callbacks aren't clean, and they cause excessive amounts of nesting *(the callback hell)*. There are utilities that transform existing functions using the callback style to a version that returns promises (for Node.js see `util.promisify`, for general purpose see pify, es6-promisify)

Bad:

```
import { get } from 'request';
import { writeFile } from 'fs';

function downloadPage(url: string, saveTo: string, callback: (error:
Error, content?: string) => void) {
  get(url, (error, response) => {
    if (error) {
      callback(error);
    } else {
      writeFile(saveTo, response.body, (error) => {
        if (error) {
          callback(error);
        } else {
          callback(null, response.body);
        }
      });
    }
  });
}

downloadPage('https://en.wikipedia.org/wiki/Robert_Cecil_Martin',
'article.html', (error, content) => {
  if (error) {
    console.error(error);
  } else {
    console.log(content);
  }
});
```

Good:

```
import { get } from 'request';
import { writeFile } from 'fs';
import { promisify } from 'util';

const write = promisify(writeFile);

function downloadPage(url: string, saveTo: string): Promise<string> {
  return get(url)
    .then(response => write(saveTo, response));
}

downloadPage('https://en.wikipedia.org/wiki/Robert_Cecil_Martin',
'article.html')
  .then(content => console.log(content))
  .catch(error => console.error(error));
```

Promises supports a few helper methods that help make code more concise:

Pattern	Description
`Promise.resolve(value)`	Convert a value into a resolved promise.
`Promise.reject(error)`	Convert an error into a rejected promise.
`Promise.all(promises)`	Returns a new promise which is fulfilled with an array of fulfillment values for the passed promises or rejects with the reason of the first promise that rejects.
`Promise.race(promises)`	Returns a new promise which is fulfilled/rejected with the result/error of the first settled promise from the array of passed promises.

`Promise.all` is especially useful when there is a need to run tasks in parallel.
`Promise.race` makes it easier to implement things like timeouts for promises.

Async/Await are even cleaner than Promises

With `async`/`await` syntax you can write code that is far cleaner and more understandable than chained promises. Within a function prefixed with `async` keyword you have a way to tell the JavaScript runtime to pause the execution of code on the `await` keyword (when used on a promise).

Bad:

```
import { get } from 'request';
import { writeFile } from 'fs';
import { promisify } from 'util';

const write = util.promisify(writeFile);

function downloadPage(url: string, saveTo: string): Promise<string> {
  return get(url).then(response => write(saveTo, response));
}

downloadPage('https://en.wikipedia.org/wiki/Robert_Cecil_Martin',
'article.html')
  .then(content => console.log(content))
  .catch(error => console.error(error));
```

Good:

```
import { get } from 'request';
import { writeFile } from 'fs';
import { promisify } from 'util';

const write = promisify(writeFile);

async function downloadPage(url: string, saveTo: string):
Promise<string> {
  const response = await get(url);
  await write(saveTo, response);
  return response;
}

// somewhere in an async function
try {
  const content = await
downloadPage('https://en.wikipedia.org/wiki/Robert_Cecil_Martin',
'article.html');
  console.log(content);
} catch (error) {
  console.error(error);
}
```

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.

Always use Error for throwing or rejecting

JavaScript as well as TypeScript allow you to `throw` any object. A Promise can also be rejected with any reason object. It is advisable to use the `throw` syntax with an `Error` type. This is because your error might be caught in higher level code with a `catch` syntax. It would be very confusing to catch a string message there and would make debugging more painful. For the same reason you should reject promises with `Error` types.

Bad:

```
function calculateTotal(items: Item[]): number {
  throw 'Not implemented.';
}

function get(): Promise<Item[]> {
  return Promise.reject('Not implemented.');
}
```

Good:

```
function calculateTotal(items: Item[]): number {
  throw new Error('Not implemented.');
}

function get(): Promise<Item[]> {
  return Promise.reject(new Error('Not implemented.'));
}

// or equivalent to:

async function get(): Promise<Item[]> {
  throw new Error('Not implemented.');
}
```

The benefit of using `Error` types is that it is supported by the syntax `try/catch/finally` and implicitly all errors have the `stack` property which is very powerful for debugging. There are also another alternatives, not to use the `throw` syntax and instead always return custom error objects. TypeScript makes this even easier. Consider following example:

```
type Result<R> = { isError: false, value: R };
type Failure<E> = { isError: true, error: E };
type Failable<R, E> = Result<R> | Failure<E>;

function calculateTotal(items: Item[]): Failable<number, 'empty'> {
  if (items.length === 0) {
    return { isError: true, error: 'empty' };
  }

  // ...
  return { isError: false, value: 42 };
}
```

For the detailed explanation of this idea refer to the original post.

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);
}

// or even worse

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

Good:

```
import { logger } from './logging'

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

Don't ignore rejected promises

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

Bad:

```
getUser()
  .then((user: User) => {
    return sendEmail(user.email, 'Welcome!');
  })
  .catch((error) => {
    console.log(error);
  });
```

Good:

```
import { logger } from './logging'

getUser()
  .then((user: User) => {
    return sendEmail(user.email, 'Welcome!');
  })
  .catch((error) => {
    logger.log(error);
  });

// or using the async/await syntax:

try {
  const user = await getUser();
  await sendEmail(user.email, 'Welcome!');
} catch (error) {
  logger.log(error);
}
```

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. The general rule to follow is *keep consistent formatting rules*.

For TypeScript there is a powerful tool called TSLint. It's a static analysis tool that can help you improve dramatically the readability and maintainability of your code. There are ready to use TSLint configurations that you can reference in your projects:

- TSLint Config Standard - standard style rules
- TSLint Config Airbnb - Airbnb style guide
- TSLint Clean Code - TSLint rules inspired by the Clean Code: A Handbook of Agile Software Craftsmanship
- TSLint react - lint rules related to React & JSX
- TSLint + Prettier - lint rules for Prettier code formatter
- ESLint rules for TSLint - ESLint rules for TypeScript
- Immutable - rules to disable mutation in TypeScript

Refer also to this great TypeScript StyleGuide and Coding Conventions source.

Use consistent capitalization

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() {}

type animal = { /* ... */ }
type Container = { /* ... */ }
```

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() {}

type Animal = { /* ... */ }
type Container = { /* ... */ }
```

Prefer using `PascalCase` for class, interface, type and namespace names. Prefer using `camelCase` for variables, functions and class members.

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(private readonly employee: Employee) {
  }

  private lookupPeers() {
    return db.lookup(this.employee.id, 'peers');
  }

  private lookupManager() {
    return db.lookup(this.employee, 'manager');
  }

  private getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }

  review() {
    this.getPeerReviews();
    this.getManagerReview();
    this.getSelfReview();

    // ...
  }

  private getManagerReview() {
    const manager = this.lookupManager();
  }

  private getSelfReview() {
    // ...
  }
}

const review = new PerformanceReview(employee);
review.review();
```

Good:

```
class PerformanceReview {
  constructor(private readonly employee: Employee) {
  }

  review() {
    this.getPeerReviews();
    this.getManagerReview();
    this.getSelfReview();

    // ...
  }

  private getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }

  private lookupPeers() {
    return db.lookup(this.employee.id, 'peers');
  }

  private getManagerReview() {
    const manager = this.lookupManager();
  }

  private lookupManager() {
    return db.lookup(this.employee, 'manager');
  }

  private getSelfReview() {
    // ...
  }
}

const review = new PerformanceReview(employee);
review.review();
```

Organize imports

With clean and easy to read import statements you can quickly see the dependencies of current code. Make sure you apply following good practices for `import` statements:

- Import statements should be alphabetized and grouped.

- Unused imports should be removed.
- Named imports must be alphabetized (i.e. `import {A, B, C} from 'foo';`)
- Import sources must be alphabetized within groups, i.e.: `import * as foo from 'a'; import * as bar from 'b';`
- Groups of imports are delineated by blank lines.
- Groups must respect following order:
    - Polyfills (i.e. `import 'reflect-metadata';`)
    - Node builtin modules (i.e. `import fs from 'fs';`)
    - external modules (i.e. `import { query } from 'itiriri';`)
    - internal modules (i.e `import { UserService } from 'src/services/userService';`)
    - modules from a parent directory (i.e. `import foo from '../foo'; import qux from '../../foo/qux';`)
    - modules from the same or a sibling's directory (i.e. `import bar from './bar'; import baz from './bar/baz';`)

Bad:

```
import { TypeDefinition } from '../types/typeDefinition';
import { AttributeTypes } from '../model/attribute';
import { ApiCredentials, Adapters } from './common/api/authorization';
import fs from 'fs';
import { ConfigPlugin } from './plugins/config/configPlugin';
import { BindingScopeEnum, Container } from 'inversify';
import 'reflect-metadata';
```

Good:

```
import 'reflect-metadata';

import fs from 'fs';
import { BindingScopeEnum, Container } from 'inversify';

import { AttributeTypes } from '../model/attribute';
import { TypeDefinition } from '../types/typeDefinition';

import { ApiCredentials, Adapters } from './common/api/authorization';
import { ConfigPlugin } from './plugins/config/configPlugin';
```

Use typescript aliases

Create prettier imports by defining the paths and baseUrl properties in the compilerOptions section in the `tsconfig.json`

This will avoid long relative paths when doing imports.

Bad:

```
import { UserService } from '../../../services/UserService';
```

Good:

```
import { UserService } from '@services/UserService';
```

```
// tsconfig.json
...
  "compilerOptions": {
    ...
    "baseUrl": "src",
    "paths": {
      "@services": ["services/*"]
    }
    ...
  }
...
```

Comments

The use of a comments is an indication of failure to express without them. Code should be the only source of truth.

Don’t comment bad code—rewrite it. — *Brian W. Kernighan and P. J. Plaugher*

Prefer self explanatory code instead of comments

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

Bad:

```
// Check if subscription is active.
if (subscription.endDate > Date.now) {  }
```

Good:

```
const isSubscriptionActive = subscription.endDate > Date.now;
if (isSubscriptionActive) { /* ... */ }
```

Don't leave commented out code in your codebase

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

Bad:

```
type User = {
  name: string;
  email: string;
  // age: number;
  // jobPosition: string;
}
```

Good:

```
type User = {
  name: string;
  email: string;
}
```

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: Added type-checking (LI)
 * 2015-03-14: Implemented combine (JR)
 */
function combine(a: number, b: number): number {
  return a + b;
}
```

Good:

```
function combine(a: number, b: number): number {
  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. Most IDE support code folding feature that allows you to collapse/expand blocks of code (see Visual Studio Code folding regions).

Bad:

```
////////////////////////////////////////////////////////////////////////////
////////
// Client class
////////////////////////////////////////////////////////////////////////////
////////
class Client {
  id: number;
  name: string;
  address: Address;
  contact: Contact;

////////////////////////////////////////////////////////////////////////////
////////
  // public methods
////////////////////////////////////////////////////////////////////////////
////////
  public describe(): string {
    // ...
  }

////////////////////////////////////////////////////////////////////////////
////////
  // private methods
////////////////////////////////////////////////////////////////////////////
////////
  private describeAddress(): string {
    // ...
  }

  private describeContact(): string {
    // ...
  }
};
```

Good:

```
class Client {
  id: number;
  name: string;
  address: Address;
  contact: Contact;

  public describe(): string {
    // ...
  }

  private describeAddress(): string {
    // ...
  }

  private describeContact(): string {
    // ...
  }
};
```

TODO comments

When you find yourself that you need to leave notes in the code for some later improvements, do that using `// TODO` comments. Most IDE have special support for those kind of comments so that you can quickly go over the entire list of todos.

Keep in mind however that a *TODO* comment is not an excuse for bad code.

Bad:

```
function getActiveSubscriptions(): Promise<Subscription[]> {
  // ensure `dueDate` is indexed.
  return db.subscriptions.find({ dueDate: { $lte: new Date() } });
}
```

Good:

```
function getActiveSubscriptions(): Promise<Subscription[]> {
  // TODO: ensure `dueDate` is indexed.
  return db.subscriptions.find({ dueDate: { $lte: new Date() } });
}
```