

JAVASCRIPT IN THE INDUSTRY

HOW PROFESSIONAL TEAMS WANT YOU TO WRITE CODE



Simon Høiberg

Introduction	4
Functional style	5
Imperative vs. declarative	5
Working with arrays	10
Avoid using classes	18
Working with instances when you really don't have to	19
Classes vs Hooks in React	23
Create your own abstractions	25
Control flows	28
Guard clauses	28
Skip the 'else' part	31
Using a switch-statement	35
Using functions and partial applications	36
Code hygiene & good practices	42
Pass arguments as an object	42
Return associated data as tuples	46
Do not extend builtins	47
Avoid default exports	48
Poor Discoverability	49
Annoying when using CommonJS	49
Annoying when using Dynamic Imports	49
Name Protection	49
Re-exporting	49
Drop the single-letter variable names	50
Arrow function vs. function declaration	51

TypeScript	52
Don't overdo it	52
Let TypeScript infer the type	54
Avoid using any	56
React	57
Use React hooks	57
Don't worry about arrow functions in JSX	57
Don't return JSX from inner functions	59
Ternary inside the JSX	60
Split into a separate component	60
Don't wrap useEffect in an async IIFE	61
Don't overuse the inbuilt hooks	62
Stop overusing useMemo	62
Stop overusing useCallback	64
NodeJS & AWS Lambda	65
Use promises instead of callbacks	65
Using promisify	66
Using module/promises	67
Async handlers in AWS Lambda	67
Final words	69
Which conventions should I use?	69
Additional resources	70

Introduction

I have had a front-row seat while JavaScript has taken over the industry.

In the past ten years of working as a freelance consultant, I've had the joy of working with many different teams. I've participated in countless PR code reviews and many heated discussions about JavaScript, best practices, clean code, patterns, and coding styles.

I have taught JavaScript, React, and NodeJS on various talks and workshops, and furthermore, I've spent the last year notoriously presenting my Twitter audience with small bits of code, asking them what they preferred and if they would have approved it in a PR.

Here are my findings, impressions, and personal experience.

All put together in an easy-to-digest e-book including examples and explanations.

This book is opinionated.

For each and every example in this book, you will find developers in the industry who disagree. This book teaches industry preferences and standards. But it is, by definition, opinionated.

This is not a JavaScript fundamentals book.

Do note, that this book assumes fundamental understanding of the JavaScript language.

Functional style

JavaScript is a multi-paradigm programming language.

This means that the language is open for programming in different styles, including object-oriented, procedural, prototypal, and functional.

By far, the most common styles you see are **object-oriented** and **functional**. Even though object-oriented languages such as Java and C# have had an enormous impact on the industry, the functional style JavaScript is the most popular and accepted in 2021. This is where you should be heading.

Imperative vs. declarative

Object-oriented programming follows an **imperative** paradigm.

Functional programming follows a **declarative** paradigm.

Let's look at the difference.

Imperative programming focuses on **how** you want something done.

You alter the state of your program step by step.

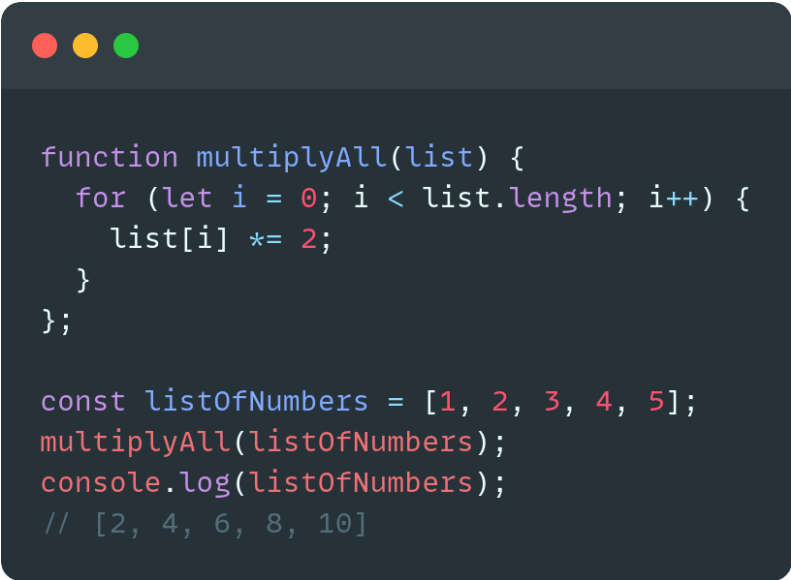
Declarative programming focuses on **what** the end result will be.

How to obtain that result is less interesting, as long as the process of obtaining it does **not** alter the state of your program.

When a function carries out a task in a deterministic way without changing its environment (changing state in its surroundings), we call it **pure**.

Let's say we want to iterate through a list of numbers and multiply them all by 2.

An **imperative** style would look something like this.



```
function multiplyAll(list) {  
  for (let i = 0; i < list.length; i++) {  
    list[i] *= 2;  
  }  
};  
  
const listOfNumbers = [1, 2, 3, 4, 5];  
multiplyAll(listOfNumbers);  
console.log(listOfNumbers);  
// [2, 4, 6, 8, 10]
```

Simple and straightforward.

But there are two key observations we need to make here:

1. The function *multiplyAll* does **not** have a return value.

Since it's not returning anything, we cannot determine the result based on the input. Hence, it's **not pure**.

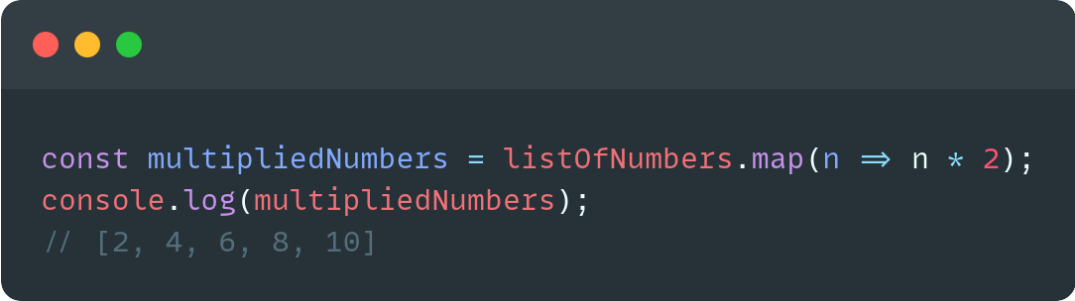
2. The original *listOfNumbers* array gets **mutated**.

The function alters (changes the state) of the input given as an argument. Hence, it's **not immutable**.

These two simple behaviors are likely to result in bugs, code that is hard to debug, hard to test, and is likely to break your application at scale!

On a lot of professional teams, this code would be immediately rejected in a Pull Request.

Let's rewrite it to follow a **declarative** style instead.



```
const multipliedNumbers = listOfNumbers.map(n => n * 2);  
console.log(multipliedNumbers);  
// [2, 4, 6, 8, 10]
```

Well, isn't this our lucky day?

JavaScript has a method on the Array prototype built in that does exactly what we need. It's called *map*.

Let's do a few observations again:

1. The function *map* has a return value. It's a list with each element altered as specified in the *callback function* we provide.
Map doesn't change its surroundings and will return the same result every time, given the same callback and input. Hence, *map* is **pure**.
2. The original *listOfNumbers* stays untouched. *Map* will create a new copy of the array for us, and alter that one instead. Hence, *map* is **immutable**.

As it turns out, all the Array-methods (forEach, Map, Reduce, Filter, ...) are following a **declarative** programming style.

The code becomes way cleaner, easier to test, less prone to introduce bugs, and much more likely to be approved in a Pull Request on most professional teams today.

Be careful about the pitfalls, though.

The Array-methods are only **pure** and **immutable** to the extent that you use them as intended.



```
const names = ["albert", "ronnie", "julie"];
const people = {};

const peopleWithAges = names.reduce((newPeople, name) => {
  newPeople[name] = Math.floor(Math.random() * 100);
  return newPeople;
}, people);

console.log(people);
// { albert: 18, ronnie: 94, julie: 31 }
```

This is an example I've seen very often.

The *people* object is still mutated on, and the surroundings are still changed.

Doing it as part of *reduce* isn't changing this.



```
let max = 0;

numbers.forEach((n) => {
  if (n > max) max = n;
});
```

Or how about this?

It's as imperative and mutable as it can possibly be!

Why not just use a regular for-loop then? It has better performance.

Similarly, utility libraries like lodash and date-fns, and UI libraries like React - that are all built with a functional and declarative style in mind - are only going to offer all its benefits if it's used the way it was intended.

In my career as a consultant, I've often witnessed chaotic, careless - and sometimes tragically funny - mixes of styles piled on top of each other, which totally defeats the purpose of keeping consistent with conventions and coding styles throughout the application.

Of course, software development often goes fast, and there are many deadlines, demands from product owners, bugs to fix fast, and so on. So the key is to get a good habit of writing clean and declarative code under your skin and understand when to be stubborn about it and when to simply let go.

In the rest of this chapter, I will provide some of the most common examples of **imperative** code that could (and should) be refactored into **declarative** code and how to do it.

Working with arrays


In JavaScript (and in programming, generally), you'll find yourself working with arrays a lot.

This is why the famous array methods introduced in ES6 became so popular. They really make our lives easier.

And recall: They are following a **declarative** programming style.

In this example, we want to filter a list of candidates to only include candidates who are also active members (exists in the *activeMembers* array).

Below is an example of doing this **imperatively**.



```
const activeMembers = ["alex", "stephanie"];
const candidates = ["alex", "martin", "anna"];

const activeCandidates = [];

for (const candidate of candidates) {
  if (activeMembers.includes(candidate)) {
    activeCandidates.push(candidate);
  }
}

console.log(activeCandidates);
// ["alex"]
```

There are two red flags that should pop up in your mind here.

1. You are declaring an array *outside* the body of the loop.
2. You are using the *push* method to populate an array.

As a rule of thumb: Whenever you're about to call the *push* method, ask yourself if this can be expressed in a **declarative** way instead.

The *push* method is mutating on the array.

Let's refactor it.



```
const activeMembers = ["alex", "stephanie"];
const candidates = ["alex", "martin", "anna"];

const activeCandidates = candidates.filter(
  (candidate) => activeMembers.includes(candidate)
);

console.log(activeCandidates);
// ["alex"]
```

Ahh, much better!

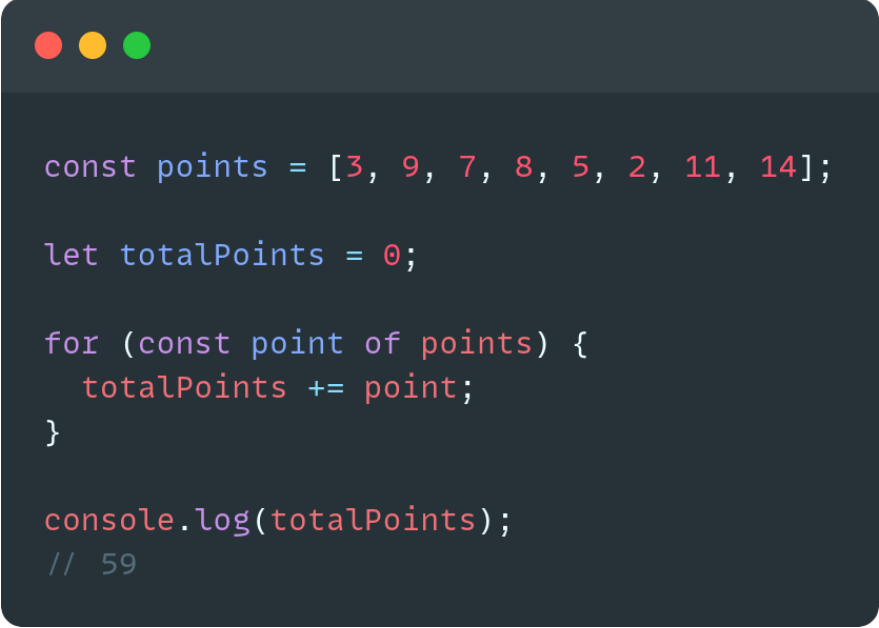
No surrounding states are changed. Nothing is mutated on.

The `activeCandidates` is declared and directly assigned the result of *filter*, which takes care of the job for us.

Let's take another example.

Here, we want to iterate through a list of numbers and find the sum.

Below is an example of doing this **imperatively**.



```
const points = [3, 9, 7, 8, 5, 2, 11, 14];

let totalPoints = 0;

for (const point of points) {
  totalPoints += point;
}

console.log(totalPoints);
// 59
```

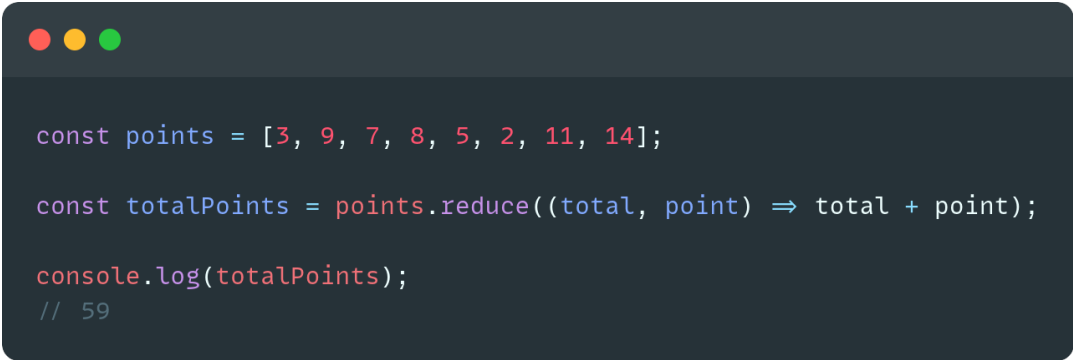
There are two red flags that should pop up in your mind here.

1. You are declaring a variable *outside* the body of the loop.
2. You are using the *let* keyword to initiate the variable.

As a rule of thumb: Whenever you find yourself using the *let* keyword, ask yourself if you can achieve the desired result **without** reassigning the variable.

In most cases, you don't want to use *let*. You want to initiate and assign variables immediately using *const*.

Let's refactor it.



```
const points = [3, 9, 7, 8, 5, 2, 11, 14];  
  
const totalPoints = points.reduce((total, point) => total + point);  
  
console.log(totalPoints);  
// 59
```

Perfect!

One line of code. No surrounding states are changed. No variables are being reassigned.

I sometimes hear people propose this argument:

“Reduce is difficult to understand. If you have juniors on your team, you may want to go with the imperative approach”.

I personally couldn't disagree more.


It all depends on its use - but when *reduce* is kept simple, as in the example above, it is perfectly within most junior developers' capability to comprehend.

This is often a bad excuse from mid-levels and seniors who still haven't adapted to this way of writing code but lack clear arguments for why they would stick in their old imperative ways.

Let's take another example. Sorting!

This one gets a bit messier.

Take a look at the code below.



```
const grades = ["C", "A", "I", "B", "F"];

const sortedGrades = grades.sort();

console.log(grades);
// ["A", "B", "C", "F", "I"]
```

At first glance, this looks good. We're not using *let*, and we're not mutating on the original array using something like *push*. We use an Array-method that **declaratively** returns the desired result.

But take a look at the *console.log* statement.

Yes. JavaScript can be awful like this. It turns out that the *sort* method is really old, and **does not** create a copy of the original array.

Instead, it both **mutates** on the original array and returns its own array from the method call.

Weird? Yes.

But there are a handful of old Array-methods that do this.

Be careful with *push*, *shift*, *unshift*, *pop*, *reverse*, *splice*, *sort*, and *fill*.

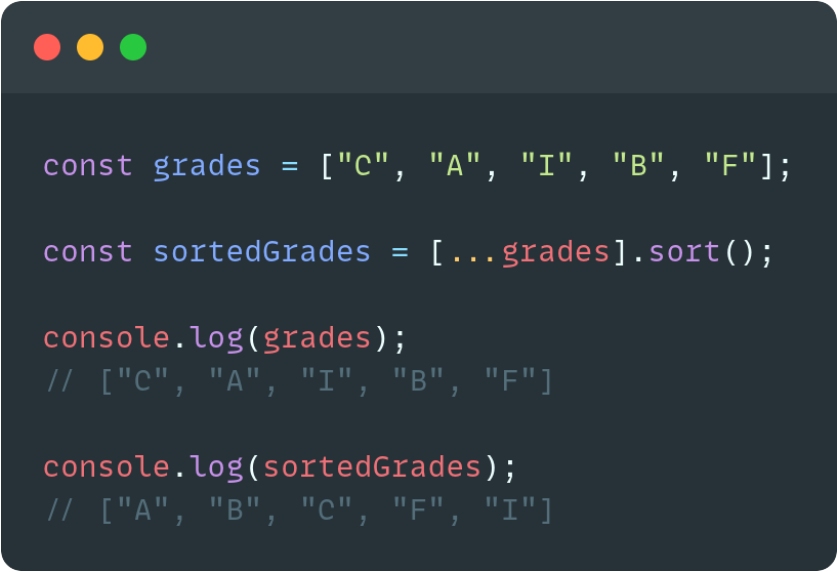
Fortunately, as we saw in the example from before, most often we can simply avoid calling these methods at all, to stay out of trouble.

However, there are cases, like using *sort*, where we have to use a method that mutates the original array, in lack of better options.

Array destructuring to the rescue!

Whenever these occasions arise, make sure to manually copy the array first, before performing an operation on it.

It's as simple as this.



```
const grades = ["C", "A", "I", "B", "F"];

const sortedGrades = [...grades].sort();

console.log(grades);
// ["C", "A", "I", "B", "F"]

console.log(sortedGrades);
// ["A", "B", "C", "F", "I"]
```

That *[...grades]* makes the entire difference.

Get this habit into your fingers.

Let's look at an example using asynchronous operations.

Iterative operations including asynchronous behavior can be a little more tricky to do using a **declarative** style.

Let's say we have a list of names.

For each name, we want to make an API call and get some information, and then keep a new list with this collection of information.

A typical **imperative** approach may look like this.



```
const names = ['jack', 'annie', 'peter'];

async function getInformationFromName(name) {
  const request = await fetch('https://some.api?name=' + name);
  const result = await request.json();
  return result;
}

const namesWithInformation = [];

for (const name of names) {
  const information = await getInformationFromName(name);
  namesWithInformation.push(information);
}
```

Using an **imperative** approach makes a bit more sense in this case.

Refactoring to using *map* or *forEach* is not that straightforward here.

The *callbacks* provided to *map* or *forEach* (or any of the array-methods) are not awaited, so we have no way of knowing when the full collection of information is done and ready to use.

However, there is still a way we can write this in a nice, **declarative** way - and it even includes a bonus takeaway!

Using the *Promise.all* method.



```
const names = ["jack", "annie", "peter"];

async function getInformationFromName(name) {
  const request = await fetch('https://some.api?name=' + name);
  const result = await request.json();
  return result;
}

const namesWithInformation = await Promise.all(
  names.map(getInformationFromName)
);
```

Awesome! Now, *map* returns a list of promises, and the *Promise.all* method takes a list of promises and resolves them in parallel.

Not only did the code become much more nice and clean - but we also benefit from the fact that the promises are not resolved sequentially, which speeds up the process and increases performance.

Avoid using classes

Ahh, classes. At the very root of object-oriented programming lies the concept of classes.

Classes are blueprints for creating objects. They encapsulate data along with code to mutate on that data.

In JavaScript, classes are built on prototypes and are used as a template for creating objects.

They are - as in any other programming language - **inherently imperative** in their nature.

The use of classes has its place. It does make sense on some occasions. But most often, you can avoid using them - and if you can, **you should**.

So generally, whenever you are about to use a class declaration, ask yourself if it's really necessary. Can you express the logic you need in a **declarative** way instead, using only functions? In 9/10 cases, I bet you can!

Let's take a look at some of the most common uses of classes, which can easily be refactored into using a **declarative** programming style instead.

Working with instances when you really don't have to

Let's look at this class which will format a name and creates a greeting.

```
class NameBuilder {
  constructor(name) {
    this.name = name;
  }

  capitalize() {
    this.name = this.name
      .split(' ')
      .map(s => s.charAt(0).toUpperCase() + s.substring(1))
      .join(' ');
  }

  createGreeting() {
    this.name = `Hello, ${this.name}`;
  }

  get modifiedName() {
    return this.name;
  }
}
```

```
const nameBuilder = new NameBuilder('simon hoiberg');
nameBuilder.capitalize();
nameBuilder.createGreeting();
const newName = nameBuilder.modifiedName;

console.log(newName);
// Hello, Simon Hoiberg
```

The idea here is to create a class that encapsulates the logic of building our name, so the consumer doesn't have to worry about implementation details and each step in the process.

And that's great!

But the thing is - there is no need to work with instances in this case. Having to use the *new* keyword to create a new *NameBuilder* instance doesn't serve any kind of purpose here.

Yet, this is the most common use of classes I've seen in the industry. The intention is good - but the implementation is pointless.

Now, when pointing this out in a code review, most programmers will get the point. And a common response is: Let's just make the methods static. Then there's no need to create instances anymore.

```
class NameBuilder {
  static capitalize(name) {
    return name
      .split(' ')
      .map(s => s.charAt(0).toUpperCase() + s.substring(1))
      .join(' ');
  }

  static createGreeting(name) {
    return `Hello, ${name}`;
  }
}
```

```
const capitalized = NameBuilder.capitalize('simon hoiberg');
const greeting = NameBuilder.createGreeting(capitalized);

console.log(greeting);
// Hello, Simon Hoiberg
```

Great! This is definitely an improvement.

We got rid of the instance variable, so no mutation anymore.

We also have methods that now return values deterministically, based on its argument. They are pure.

But, now that we're so close - why don't we just get rid of the class entirely?
It seems like there's literally no need for it anymore.

```
JS namebuilder.js

export function capitalize(name) {
  return name
    .split(" ")
    .map((s) => s.charAt(0).toUpperCase() + s.substring(1))
    .join(" ");
}

export function createGreeting(name) {
  return `Hello, ${name}`;
}
```

```
import { capitalize, createGreeting } from "./namebuilder";

const capitalized = capitalize("simon hoiberg");
const greeting = createGreeting(capitalized);

console.log(greeting);
// Hello, Simon Hoiberg
```

Awesome! We did it - we got rid of the class entirely!

Now, the code is **declarative**, easier to test, easier to debug, and **treeshakable**. This means that modern build tools like Webpack and Rollup are able to remove the *dead code* of functions that may never be used in a specific case. It will result in a smaller bundle size of your code.

Classes vs Hooks in React

In 2018, React introduced *hooks*.

They did this to get rid of classes and the confusing *this* keyword.

Why? Because React is an inherently **declarative** UI library, and classes just don't really fit in.

If you were working with React prior to 2018, I bet you've been writing a lot of code like this.


```
import React from "react";

export class ShowCount extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  componentDidMount() {
    this.setState({ count: this.props.count });
  }

  render() {
    return (
      <div>
        <p> Count: {this.state.count} </p>
      </div>
    );
  }
}
```

Fortunately, after *hooks* were introduced, we can now benefit from writing clean and **declarative** code in React.



```
import React, { useEffect, useState } from "react";

export function ShowCount(props) {
  const [count, setCount] = useState();

  useEffect(() => {
    setCount(props.count);
  }, [props.count]);

  return (
    <div>
      <p> Count: {count} </p>
    </div>
  );
}
```

Functional components were already a thing before hooks, but we often needed to refactor it into a class component when the logic grew.

In this way, we don't have to anymore.

We also don't have to worry about *this*, binding methods, not being able to share logic, and a lot of other things.

I'll be covering more React best practices later in this book.

Create your own abstractions

So, we've covered some good rules of thumbs to follow.

But often enough, the code we write in our software is more complex than the examples we've looked at here.

There are cases where writing **declaratively**, avoiding the *let* keyword, not reassigning variables, not altering data in the surrounding scope of a loop, and so on, is simply going to produce tedious, verbose, and hard-to-read code.

And obviously, we don't want to force any style or pattern to an extent where it makes our code worse, "just because.... functional programming".

But instead of turning it all around, we can abstract that code away in a function, and then use that function **declaratively** in the context where it's needed.

Even if the code is not necessarily reusable, it's still a great practice to package it away and keep it in its own function scope where it can't do any harm. Hence, you want to keep that function **pure**.

I've seen this very often, and it's a great in-between solution.

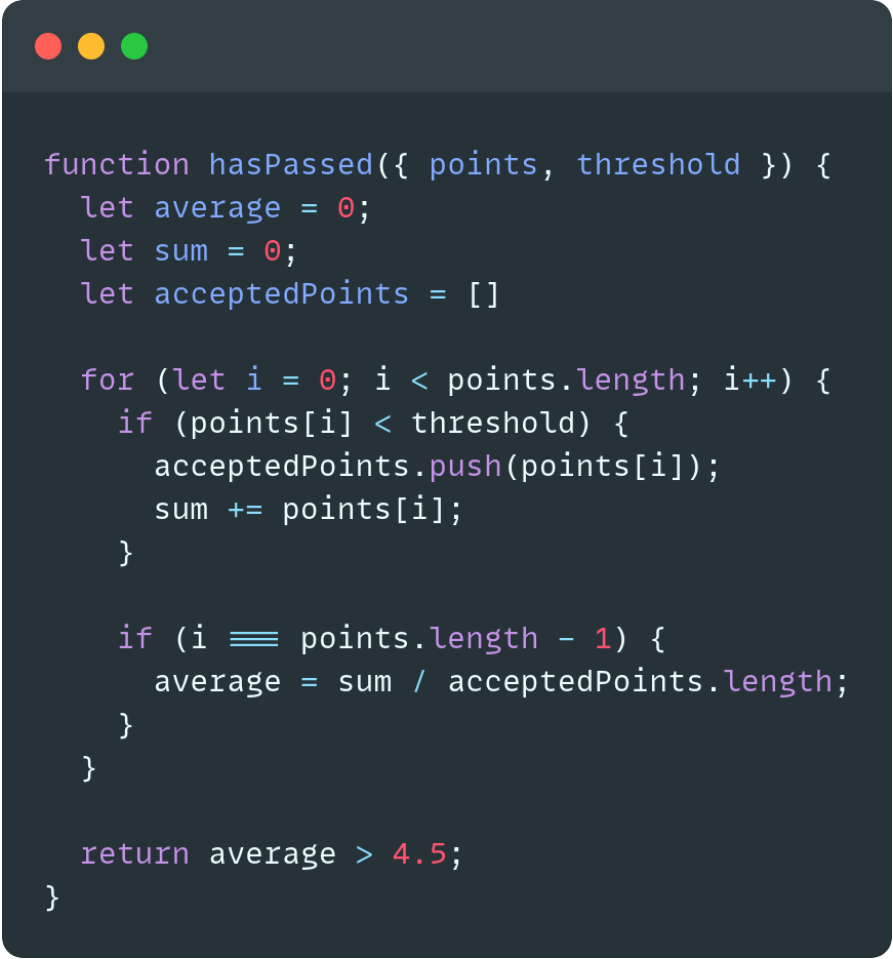
Keep the code **procedural**, and then let the function return the final result.

Procedural code is still **imperative** code.

You can reassign variables, alter data outside of loops, and mutate on arrays. But you still cannot use classes, methods, instance variables, and so on.

Let's take an example.

Say we need a function, *hasPassed* which, given a list of points and a threshold, does some computation to evaluate if the average of the points is enough to pass. A developer on the team pushes the following code in a PR.



```
function hasPassed({ points, threshold }) {  
  let average = 0;  
  let sum = 0;  
  let acceptedPoints = []  
  
  for (let i = 0; i < points.length; i++) {  
    if (points[i] < threshold) {  
      acceptedPoints.push(points[i]);  
      sum += points[i];  
    }  
  
    if (i === points.length - 1) {  
      average = sum / acceptedPoints.length;  
    }  
  }  
  
  return average > 4.5;  
}
```

Sure, as a reviewer, you could jump on this code and demand to refactor everything to use a strictly **declarative** pattern.

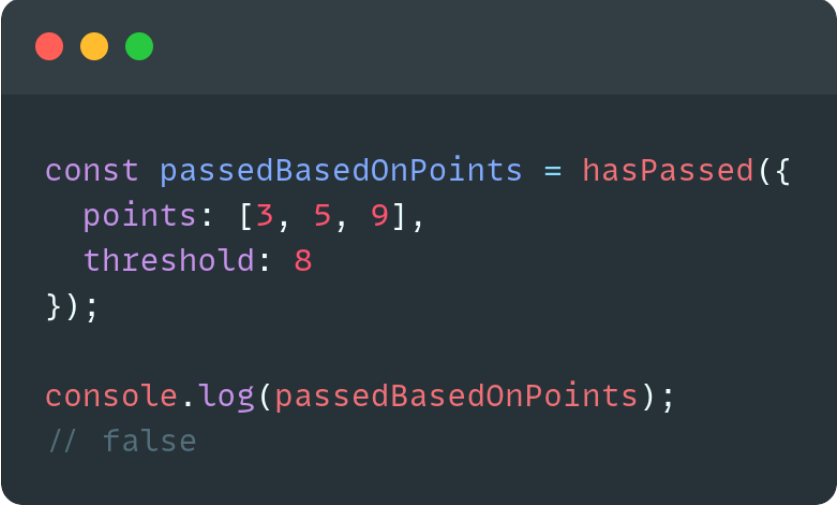
But there's actually no point. The code does what it's supposed to do - and most importantly, the function *hasPassed* is still **pure**.

The use of **imperative** code inside of the function cannot do any harm.

And now I'm able to use the function like this.

Nice, clean, and in a **declarative** way. That's all I could ever ask for.

PR approved.



```
const passedBasedOnPoints = hasPassed({  
  points: [3, 5, 9],  
  threshold: 8  
});  
  
console.log(passedBasedOnPoints);  
// false
```

The observant reader may have also noticed how I'm passing arguments wrapped in an object here.

That's another good practice that'll definitely make your team members happy. More on that later in the book.

As it is with everything - whether to use **declarative** or **imperative** code depends on the situation.

The world is not binary, and the same goes for computers! (...oh, wait?)

Joking aside, I recommend using a **functional** style as much as possible and based on my experience, this is also the most popular choice in the industry. Use **imperative** code with a bit of skepticism, and strive to avoid it when possible and when it makes sense.

Control flows

In programming, a **control flow** is how the interpreter runs your code from top to bottom.

There are a handful of statements and operators that change the **control flow**. They are common for mostly all programming languages; *loops, if-else, switch, function calls, ternary, etc.*

Hopefully, you know all of these already, and understand how they work. (If not, it's time to revisit fundamentals).

In this chapter, I'll discuss **how** these statements and operators are used, and what is widely accepted as best practice in the industry.

Let's start with one of my favorites.

Guard clauses

"In computer programming, a guard is a boolean expression that must evaluate to true if the program execution is to continue in the branch in question. Regardless of which programming language is used, guard code or a guard clause is a check of integrity preconditions used to avoid errors during execution."

— [Wikipedia](#)

Let's take a look at an example.



```
function getValidCandidate(candidate, members) {  
  if (candidate) {  
    const member = members.find(  
      (member) => member.name === candidate  
    );  
  
    if (member) {  
      const hasLegalAge = member.age > 18;  
  
      if (hasLegalAge) {  
        return member;  
      }  
    }  
  }  
}
```

We have a function, *getValidCandidate*, which checks if a candidate is valid, provided a list of members and returns the member if the candidate is valid, or *undefined* otherwise.

The code is sound.

But look how nested it is? *If*s wrapping other *ifs*, nested 3 times.

Uhf. Let's rewrite this and use **guard clauses** instead.



Much better! We *flattened* all the nested levels.

And we're now using **guard clauses**.

Notice how we have these breakpoints in the function every time we check if something is *falsy*?

These are the guards.

They prevent the function from continuing what it's doing and instead returning early if the *guarding condition* is not met.

Naturally, we also know that the end result is the *last* return of the function.

Skip the 'else' part

The if-else statement. It's one of the first things we learn when starting to program.

Now, let's move on by unlearning how to use *else*.

Generally speaking, if your function includes an *else*, it's a sign that your function is doing more than one thing.

It's violating the well-known *Single Responsibility Principle*, and often results in error-prone branching logic that gets hard to wrap your head around at scale.

So as a rule of thumb, whenever you're about to write *else*, stop and reconsider what you're doing and search for an alternative way to express the logic.

Let's cover a few ways that we can avoid using *else*.

One of them is **guard clauses**, which we just covered above.

Another approach is using *default values*.

Let's take an example.

Let's say we have a function, *negateOdd*, which takes a number and negates it if the number is odd.

```
function negateOdd(n) {  
  let result;  
  
  if (n % 2 !== 0) {  
    result = n;  
  } else {  
    result = n * -1;  
  }  
  
  return result;  
}
```

The function does what it's supposed to. But it's unnecessarily using an *else*. Let's refactor it, and instead, use a *default value*.

```
function negateOdd(n) {  
  let result = n;  
  
  if (n % 2 !== 0) {  
    result = n * -1;  
  }  
  
  return result;  
}
```

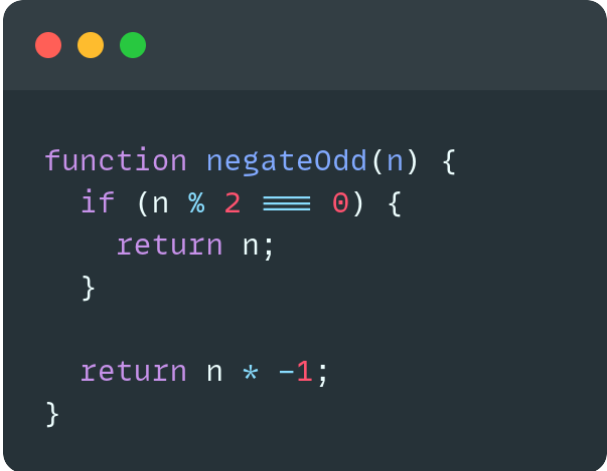

Better already! We now assign *result* with a *default value*, and the variable will only be changed if the condition in the *if-statement* is met.

Now, we also wrap the special case of the function in an *if*.

You could argue that when reading this function later it might help us quickly pinpoint where the essential part of the logic is going on.

But let's do even better. We know from the previous chapter that we're supposed to question the use of *let* and **imperative** code.

Let's see if we can make this function even more readable and concise.

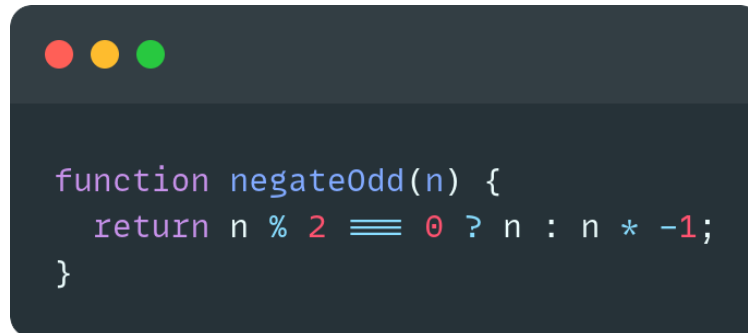


```
function negateOdd(n) {  
  if (n % 2 === 0) {  
    return n;  
  }  
  
  return n * -1;  
}
```

There we go! No variable assignment at all. Only different return values based on the given condition.

Notice how we also moved that essential part of the logic to the end of the function body. A lot of developers prefer having the essential part of a function's job put at the end of the function body - just like we did with the **guard clause** before.

There is a version of *if-else* that is generally accepted.
It's the *ternary operator*.



```
function negateOdd(n) {  
  return n % 2 === 0 ? n : n * -1;  
}
```

A *ternary operator* is essentially an *if-else*.

So, why is this accepted if we're supposed to avoid using *else*?

Because it's used to assign or return a value directly.

There's no body. No opening to writing multiple lines of logic.

That's also why the *ternary operator* should be restricted to only such usage.

I have seen developers misuse the *ternary operator* to branch the **control flow** in very creative and mysterious ways, including the use of *nested ternaries*, which is - by the vast majority of professional teams - considered filthy code that should be strictly avoided.

You could argue how readable and clean the solution using *ternary operator* above is, but in most cases, a function like this would pass a code review.

Using a switch-statement

Most likely, you're not going to need it too often, but there are cases where a *switch-statement* makes sense.

```
function pointsFromGrade(grade) {  
  switch (grade) {  
    case "A":  
      return 10;  
    case "B":  
      return 8;  
    case "C":  
      return 5;  
    case "I":  
      return 3;  
    case "F":  
      return -1;  
    default:  
      return NaN;  
  }  
}
```

If you're about to write *if-else-if-else-if...* a series of times, you probably want to consider using a *switch-statement* instead.

There are certain libraries, for instance, Redux for React, that have been using *switch-statements* as part of their reducers, per convention.

Yet, for a case like this, you could use an object to store the values instead.

If the values need logic to infer, you could store functions in the object, and call it after the object lookup.

```
const pointsFromGrade = {  
  A: 10,  
  B: 8,  
  C: 5,  
  I: 3,  
  F: -1  
};  
  
const p = pointsFromGrade.B;
```

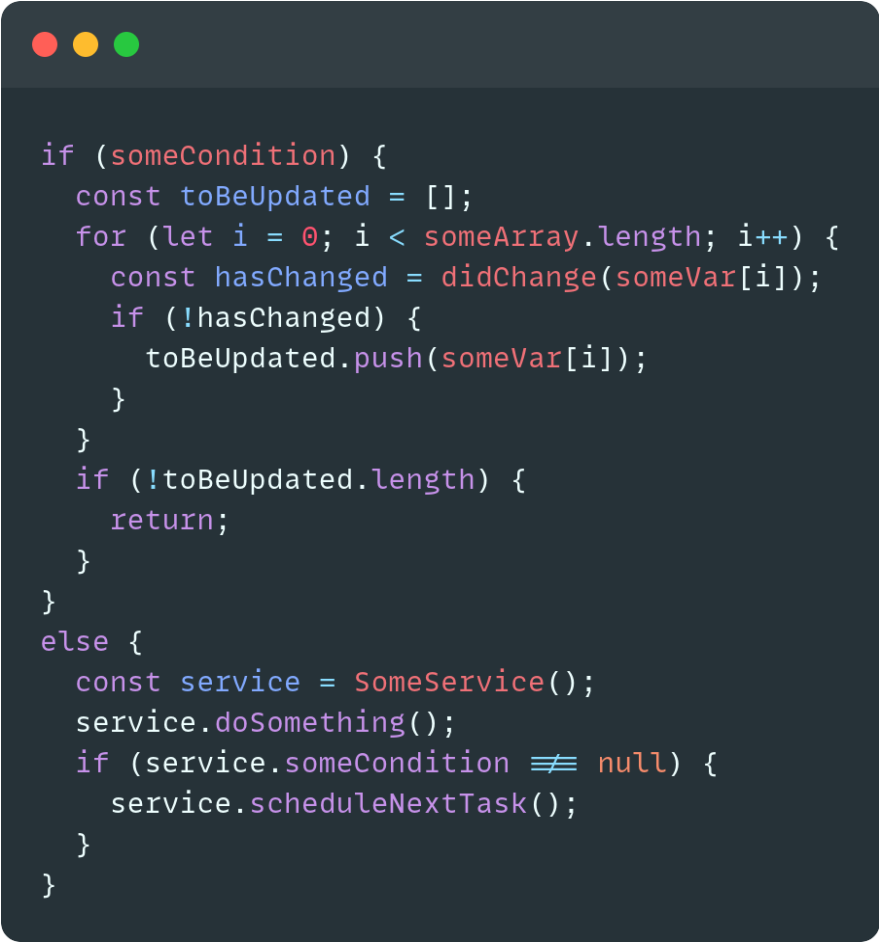
Using functions and partial applications

If you find yourself overusing the statements and operators that change the **control flow**, it's typically a symptom of your code doing way too much.

You get what's commonly known as spaghetti-code.

Take a look at this made-up example below. Do you recognize it?

I bet you've been writing code like this more often than not.



```
if (someCondition) {
  const toBeUpdated = [];
  for (let i = 0; i < someArray.length; i++) {
    const hasChanged = didChange(someVar[i]);
    if (!hasChanged) {
      toBeUpdated.push(someVar[i]);
    }
  }
  if (!toBeUpdated.length) {
    return;
  }
}
else {
  const service = SomeService();
  service.doSomething();
  if (service.someCondition !== null) {
    service.scheduleNextTask();
  }
}
```

In particular, notice how the code in the *if* and *else* seems completely unrelated.

I think we can agree that it's hard to follow the logic of this kind of code, and that it's really easy to sneak in a bug here, and can take hours to spot.

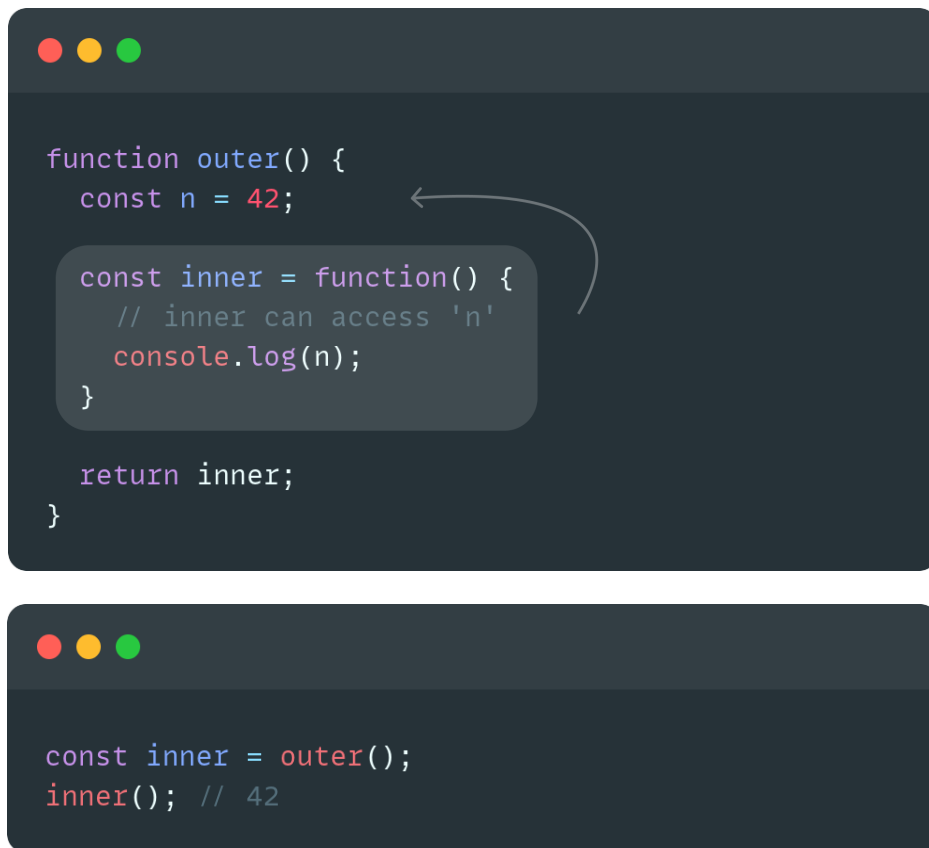
When you find yourself writing code like this, back up, and start splitting it up into individual functions.

You'll end up with a nice, flat list of function calls instead, each with a name representing what they do.

To include conditionals, you can consider using *partial applications*.

Without going into too much detail, a *partial application* in JavaScript, is when a function returns another function, and when that inner function gets invoked, it'll have access to the variables from the scope of the outer function. This is also known as a closure.

Take a look at the illustration below as a quick reference.

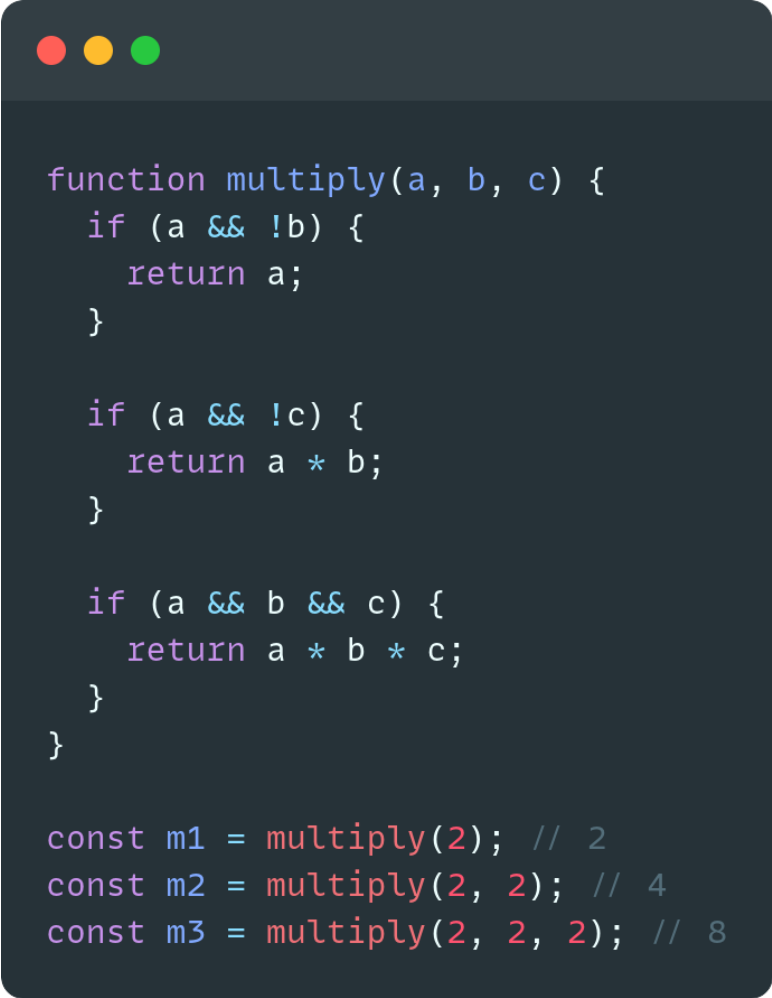


```
function outer() {  
  const n = 42;  
  
  const inner = function() {  
    // inner can access 'n'  
    console.log(n);  
  }  
  
  return inner;  
}  
  
const inner = outer();  
inner(); // 42
```

I will have to admit that it's not something I have seen used too often. Granted, this is a more advanced technique from functional programming. Yet, in certain cases, you can use it to create some quite concise and useful constructs.

Let's take an example.

Here, we have a function, *multiply*, which takes three numbers as arguments and multiplies them. What makes this case complicated, is that only the first argument is mandatory. That means we have to check for the arguments *b* and *c* before performing the multiplication.

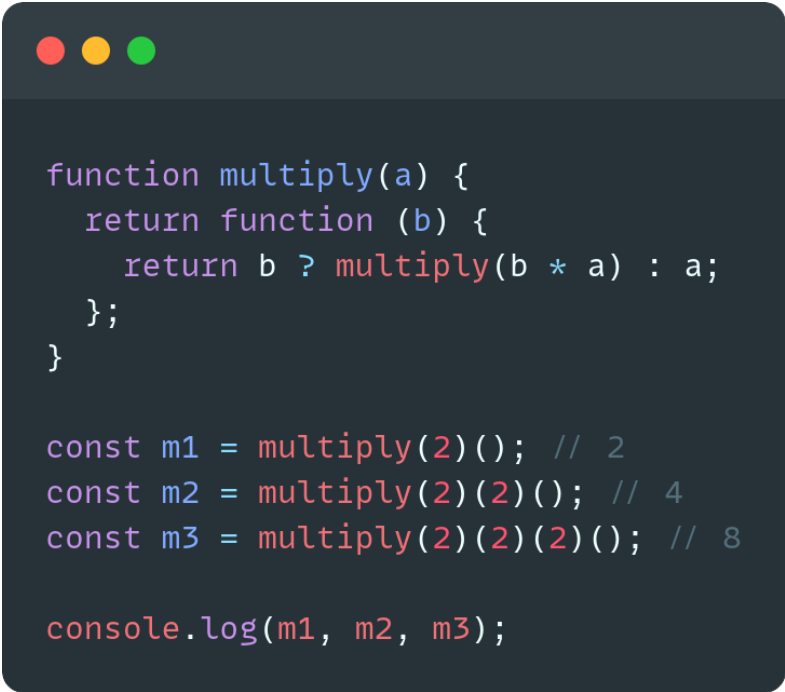


```
function multiply(a, b, c) {  
  if (a && !b) {  
    return a;  
  }  
  
  if (a && !c) {  
    return a * b;  
  }  
  
  if (a && b && c) {  
    return a * b * c;  
  }  
}  
  
const m1 = multiply(2); // 2  
const m2 = multiply(2, 2); // 4  
const m3 = multiply(2, 2, 2); // 8
```

As you can see, this quickly becomes confusing.

A lot of if statements are needed to verify the validity of the input, and it doesn't even cover all cases - certain assumptions are still made here.

Let's try to refactor it using the construct of a *partial application* instead.



```
function multiply(a) {  
  return function (b) {  
    return b ? multiply(b * a) : a;  
  };  
}  
  
const m1 = multiply(2)(); // 2  
const m2 = multiply(2)(2)(); // 4  
const m3 = multiply(2)(2)(2)(); // 8  
  
console.log(m1, m2, m3);
```

As you see, we got the *multiply* function slimmed down a lot.

And not only that - now the function can actually multiply an indefinite amount of numbers, provided one *part* at a time.

Let's break down what's happening here.

The function *multiply* is returning another function that, if given an argument *b*, calls itself recursively and applies *a* from the outer function multiplied with *b*. If *b* is not provided, it simply returns *a*.

Don't worry if this seems confusing at first.

Let's take a more lightweight, practical example from React instead.

We're updating a form using controlled input fields.

```
function App() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const setInput = (setter) => (event) => {
    setter(event.currentTarget.value);
  }

  return (
    <form>
      <input
        value={username}
        onChange={setInput(setUsername)}
      />
      <input
        value={password}
        onChange={setInput(setPassword)}
      />
    </form>
  );
}
```

Instead of using multiple function calls, *arrow function expressions* inside of the JSX, or *if-statements* to determine which state to update, we use a *partial application* (in this case by currying), to take a setter function as an argument and then return the handler that is provided to the *onChange* prop.

Code hygiene & good practices

We've covered two fundamental areas of programming and which styles and preferences are most commonly accepted in the industry.

In this chapter, I'll provide examples of what many professional teams consider good code hygiene in JavaScript, without them being tied to particular programming areas or concepts.

Let's start with an example we saw in the previous chapter.

Pass arguments as an object

Say we have a function, *createUser*, which requires four arguments in order to create a new user (by calling some API).

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a JavaScript function definition for `createUser`.

```
function createUser(username, date, isAdmin, isMod) {  
  return fetch('https://some.api', {  
    method: 'POST',  
    body: JSON.stringify({  
      username, date, isAdmin, isMod  
    })  
  });  
}
```

When looking at the function signature itself, things seem to make pretty good sense.

But how about when we call it?

```
createUser('Simon', '2021-01-01', false, true);
```

It's pretty unclear what the arguments mean, right?

Especially the last two booleans. I would have to go to the implementation to look it up.

Instead, we can wrap the arguments in an object.

```
function createUser({ username, date, isAdmin, isMod }) {  
  return fetch('https://some.api', {  
    method: 'POST',  
    body: JSON.stringify({  
      username, date, isAdmin, isMod  
    })  
  });  
}
```

Thanks to [ES6 object destructuring](#), we can do this easily by simply adding curly brackets around the arguments.

Now, whenever we call *createUser*, we pass an object as a single argument with the required values as properties instead.

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. The editor contains a single line of JavaScript code: `createUser({` on the first line, `username: 'Simon',` on the second, `date: '2021-01-01',` on the third, `asAdmin: false,` on the fourth, `isMod: true,` on the fifth, and `});` on the sixth. The code is color-coded: `createUser` is red, `{` is blue, `username` is purple, `'Simon'` is green, `date` is purple, `'2021-01-01'` is green, `asAdmin` is purple, `false` is orange, `isMod` is purple, `true` is orange, and `});` is blue.

See how nice that reads out now.

We're no longer in doubt what those booleans mean.

Also, we get a few more benefits from this approach:

- It's easier to handle optional properties
- We don't depend on the order of the arguments

From my experience in the industry, this is a very popular approach whenever your function requires more than 1-2 arguments.

There's another version of this that I've seen very often:

Passing optional arguments as an *options* object.

This is adopted in NodeJS and a range of well-known Open-Source libraries.

The idea is to pass 1-2 essential arguments and then pass the remaining arguments as an *options* object.

```
function createUser(username, date, options) {  
  return fetch('https://some.api', {  
    method: 'POST',  
    body: JSON.stringify({  
      username,  
      date,  
      isAdmin: options ? !!options.isAdmin : false,  
      isMod: options ? !!options.isMod : false  
    })  
  });  
}
```

Now we need to check if the *options* object is set before accessing its values and provide proper fallbacks.

On the other hand, calling the *createUser* function now looks very clean.

```
createUser('Simon', '2021-01-01', {  
  isMod: true,  
});
```

The first two arguments are pretty obvious, and we can now optionally provide options when needed.

Return associated data as tuples

In JavaScript, we can return a single value from a function.

But often enough, we need to return multiple values that are associated together.

A common approach is to return a [tuple](#) containing the values.

In short, a tuple is an ordered list of elements.

Let's say we have a function, *getElementPosition*, that retrieves a DOM element's position given an id.

In this case, *x* and *y* are associated values. Thus, it makes perfect sense to return them both as a tuple.

```
function getElementPosition(elementID) {
  const element = document.getElementById(elementID);

  if (!element) {
    return [NaN, NaN];
  }

  const rect = element.getBoundingClientRect();
  return [rect.x, rect.y];
}

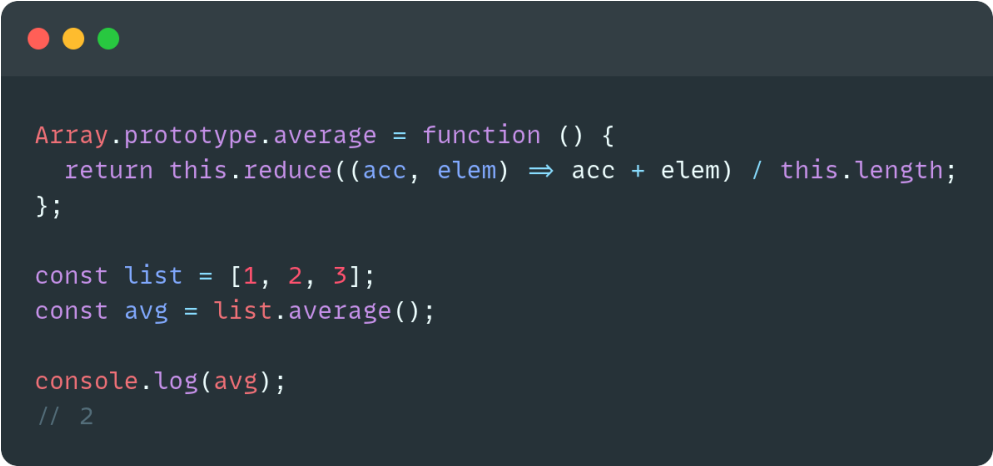
const [x, y] = getElementPosition('my-element');
console.log(x, y);
// 278.5, 122.3
```

Do not extend builtins

When you first understand JavaScript's prototype chain and understand how you can extend any prototype, including the JavaScript builtins, you may experience a sensation of overwhelming superpowers.

Now, you can extend the Array prototype to include your own custom array method along with *map*, *filter*, and *reduce*.

I mean - is this just cool or what?



```
Array.prototype.average = function () {  
  return this.reduce((acc, elem) => acc + elem) / this.length;  
};  
  
const list = [1, 2, 3];  
const avg = list.average();  
  
console.log(avg);  
// 2
```

It's not! In fact, this is deeply problematic and can lead to a range of rather serious issues. Extending the builtins should be avoided at all costs.

In fact, there are horror stories of entire applications being brought to their knees because of this pattern.

Unfortunately, some third-party libraries in the past had a bad habit of doing this, and you may accidentally override it, causing a world of headaches!

Instead, keep things separated and pure!

Build your own utility pack instead, and keep it in its own module.

```
JS array-utils.js

export function average(list) {
  return list.reduce((acc, elem) => acc + elem) / list.length;
}
```

```
import { average } from "../array-utils";

const list = [1, 2, 3];
const avg = average(list);

console.log(avg);
// 2
```

Much better!

As this collection grows, it's also easy to extract it into an NPM package so it can be reused across projects.

Avoid default exports

This brings us to the next point.

As you may have noticed, I haven't used *export default* in any of my examples.

That's deliberate.

This is especially important if you're using TypeScript, but I strongly recommend getting rid of the habits of using *default exports* entirely.

Why?

Poor Discoverability

It makes it hard for IntelliSense in modern code editors like VSCode and Webstorm to suggest exports from a module when you start typing.

Annoying when using CommonJS

I don't know if you've ever had to put that *.default* at the end of a *require* statement? Pretty annoying and ugly, isn't it?

Annoying when using Dynamic Imports

Similar to *CommonJS* and *require*, using a *Dynamic Import* also forces you to dot into a *.default*.

Name Protection

When you import from a *default export*, you can essentially name the import whatever you want. It's hard to refactor later, and it may result in naming inconsistencies and typos.

Re-exporting

Using *barrels* to re-export modules from a centralized place is common when creating packages for NPM, or when dealing with larger code-bases with lots of modules. The *default export* forces you to rename your export, which again may lead to inconsistencies in naming.


Some would argue that it's a matter of use-case, though using *named exports* instead of *default exports* doesn't really come with any tradeoffs, so I suggest simply sticking to that.

Drop the single-letter variable names

I've seen this a lot. Single-letter variable names, especially in callbacks.

Short, concise code is not necessarily ideal.


Take a look at this code right here.



```
const images = users.filter((u) => {  
  return u.entities.find((e = e.type === 'profile_image'));  
});
```

u and *e*? No thanks, clarity, and readability is ideal.

Even if it cost a few extra lines.



```
const images = users.filter((user) => {  
  return user.entities.find(  
    (entity = entity.type === 'profile_image')  
  );  
});
```

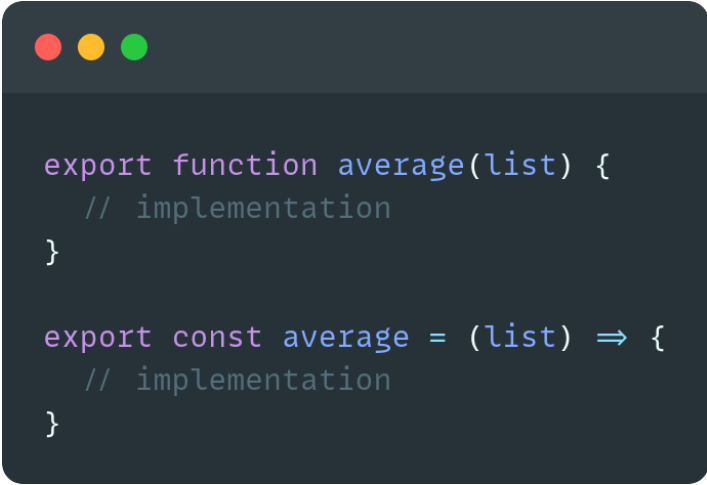
There are a few places where the single-letter variable name is conventionally accepted. For instance, in for-loops, using the *i* variable. Other than that, get a good habit of naming the arguments of the callback. Typically, you want to be using the singular version of the word of the list. Like I do here with *users* and *user*.

Arrow function vs. function declaration

You may have noticed that I'm using the *function* keyword in my examples, rather than using *arrow function expressions* using *const*.

I'm going to mention it here, even though I cannot give you a clear direction on this one.

In my experience from the industry, this is strictly down to preference.



```
export function average(list) {  
  // implementation  
}  
  
export const average = (list) => {  
  // implementation  
}
```

After ES6, the *arrow function expression* seemed to rise in popularity, and indeed, they do behave differently than classical function declarations.

In the later years, I've noticed that the *function* keyword has found its way back, and as long as you're using it without depending on its lexical context (using it **purely**, avoid using *this*, and avoid using it as a constructor), then you're fine using either one.

If you're joining a team and the codebase already follows the convention of using either one, stick with that.

TypeScript

TypeScript is a language for application-scale JavaScript development.

It's a typed superset of JavaScript used to add static typing and describe the shapes of objects while working with JavaScript.

It's an extremely popular choice on professional software teams building complex web, mobile, and desktop applications at scale.

Yet, it's definitely not all teams I've been working with that have been equally excited about it.

In fact, some teams downright hate it.

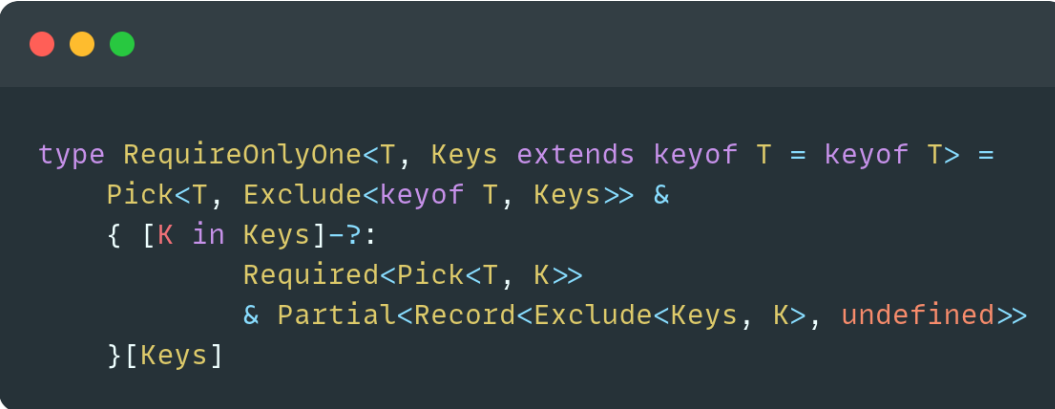
In this chapter, I'll try to describe the "sweet spot" that both routinized JavaScript developers and TypeScript enthusiasts seem to be on board with, as well as describing some do's and don'ts that I've seen from the industry.

Don't overdo it

If you're working with a team of experienced JavaScript developers, who are likely to appreciate some of the flexibility you get from the loosely-typed JavaScript, you are going to get a lot of pushback if you're trying to introduce (or even enforce) TypeScript used to its fullest capacity.

So from my experience with TypeScript in the industry, it stands out very clear: don't overdo it!

It's great if you're enthusiastic about TypeScript. It's an awesome language. But there's not a lot of developers who want to see something like this.



```
type RequireOnlyOne<T, Keys extends keyof T = keyof T> =  
  Pick<T, Exclude<keyof T, Keys>> &  
  { [K in Keys]-?:  
    Required<Pick<T, K>>  
    & Partial<Record<Exclude<Keys, K>, undefined>>  
  }[Keys]
```

(Example code from an actual project I've worked on)

From what I've seen, the most effective use of TypeScript really boils down to the following:

- Describing the shape of objects with *interfaces*
- Typing the *arguments* and *return values* of functions
- Using *generics* to use more general-purpose function signatures
- Using the [TypeScript Utility Types](#) for more flexible type descriptions

In particular, it's popular to describe the JSON response from an API using interfaces. The IntelliSense enhances productivity quite radically, and with modern solutions like GraphQL, you can get tools that auto-generate these interfaces based on the schema.

This part of TypeScript that's not annoying, but simply saves time, seems to be quite popular with everyone.

Let TypeScript infer the type

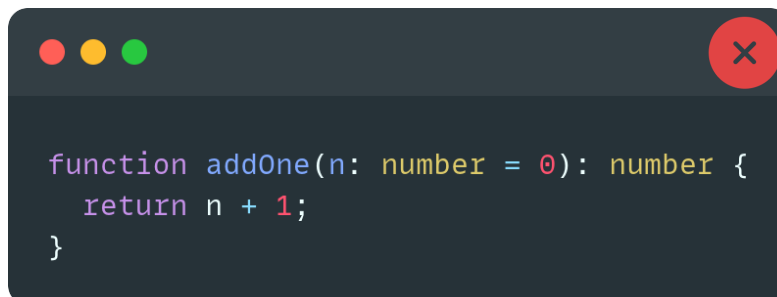
One thing I've noticed from new TypeScript converts is that they tend to insist on typing *everything*.

All functions *must* have a return type, all arguments *must* be typed, all variable declarations *must* be followed immediately by their type, and so on.

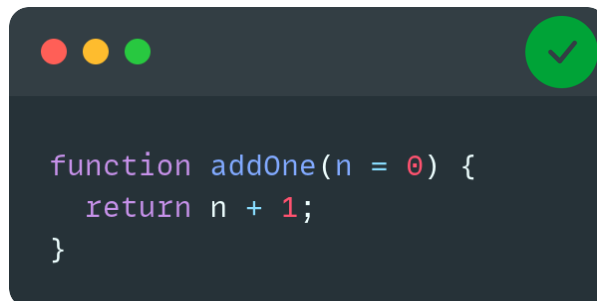
This is the part that's going to feel very tedious and bureaucratic to the rest of the team who may not be flying high on TypeScript.

And more importantly: it's not necessary.

TypeScript is **very** good at inferring the types themselves. You don't have to trivially type out every single thing.

A code editor window with a dark background. It has three colored window control buttons (red, yellow, green) on the top left and a red circular close button with a white 'X' on the top right. The code inside is:

```
function addOne(n: number = 0): number {  
    return n + 1;  
}
```

A code editor window with a dark background. It has three colored window control buttons (red, yellow, green) on the top left and a green circular button with a white checkmark on the top right. The code inside is:

```
function addOne(n = 0) {  
    return n + 1;  
}
```

You can always hover the mouse over an argument or a function to see if TypeScript infers the type (correctly) by itself.



You can do the same with variables. If you hover the mouse over the *name* variable in the last example, you'll notice that TypeScript figured it out from the assignment. No need to type it.

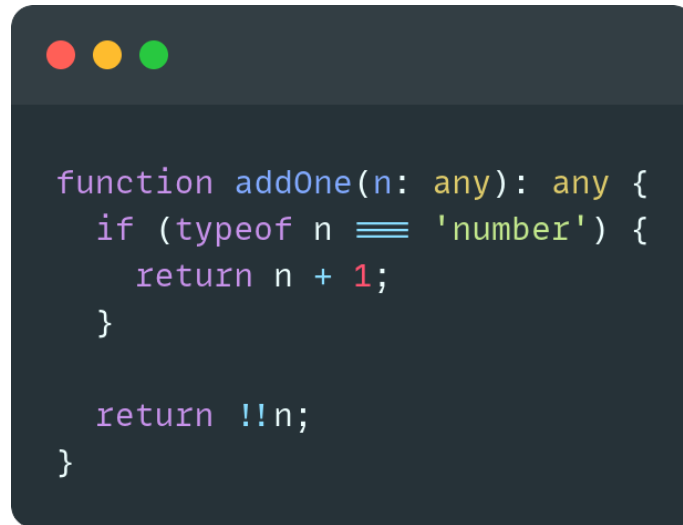


In React, the *useState* hook is also smart enough to let TypeScript infer the type based on the default value. No need to type it.

Avoid using *any*

Going from one extreme to another: I've also witnessed quite a few teams changing to TypeScript because of the hype.

But without properly using it, mainly due to writing *any* all over the place.



```
function addOne(n: any): any {  
  if (typeof n === 'number') {  
    return n + 1;  
  }  
  
  return !!n;  
}
```

Obviously, TypeScript is adding zero value here.

I totally get it - if you're new to TypeScript, some of the rather obscure compile-time errors can be confusing and feel like it's slowing you down. But if you're too busy to learn the basics of TypeScript, then I strongly suggest not using TypeScript. Stick to JavaScript, in that case.

There's no point in using a strictly typed language if you're going to enforce loose typing everywhere.

Avoid using *any*. Or avoid using TypeScript altogether.

React

React is a huge subject, and there are a lot of different opinions out there. In fact, React is one of the areas of work where I have encountered the most disagreement on different teams.

In this chapter, I'll touch upon some of the most common discussions I've witnessed and provide my general impression of how most teams prefer their React code.

Use React hooks

I'll keep this one short. If you're still stuck writing class components, get rid of that habit right away.

It is my clear impression that by far most teams have now effectively moved to use React Hooks, so unless you've been dealing with legacy projects where class components are carried over, you can safely expect that the next team you're going to join will be using React hooks.

Don't worry about arrow functions in JSX

If you've been learning React for a while, I'm sure you've been taught that you shouldn't use arrow functions in JSX.

An extra function will be created on each render, and it's bad for performance.

```

export function App() {
  return (
    <div>
      <button onClick={() => console.log("Click")}>Click</button>
    </div>
  );
}

```

Not really. The potential performance impact this has is negligible.

It is, on the other hand, convenient and, in a lot of cases, more readable.

It's generally accepted that the benefits outweigh the potential performance issues that may be.



Simon Hoiberg @SimonHoiberg · Sep 22, 2020
React Code Review ✓

A member of your team has introduced this code where an arrow function is passed directly to `onClick` .

Are you going to approve this PR, or will you kindly ask for a change? 🤔



121 66 556



Dan
@dan_abramov

Replying to @SimonHoiberg

lgdm

7:21 PM · Sep 22, 2020 · Twitter Web App

9 Retweets 3 Quote Tweets 462 Likes

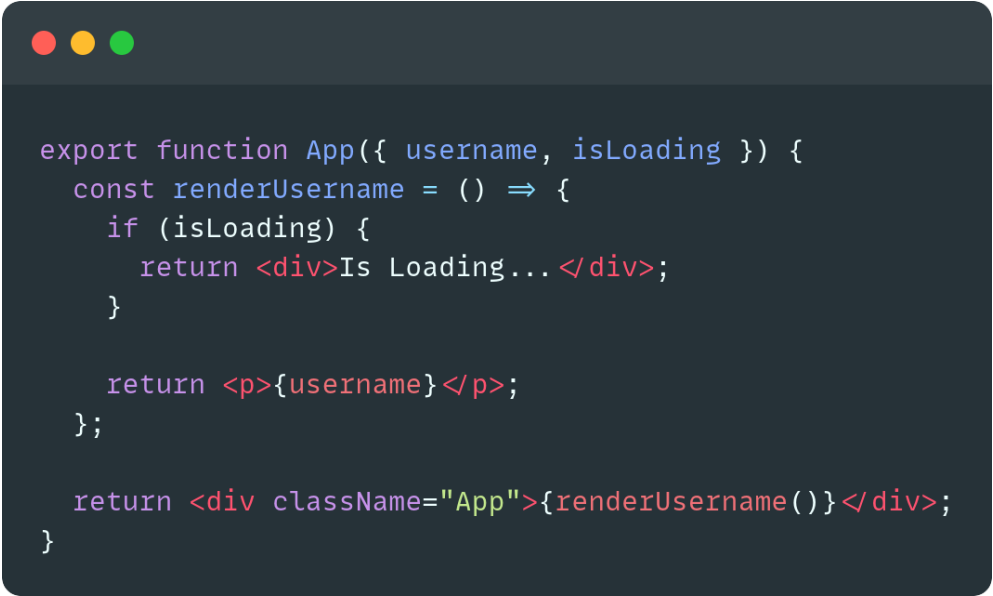
As a matter of fact, when I asked this question [on Twitter](#) last year, Dan Abramov from the React Core team commented “lgdm” (looks good to me), implying that he would approve a PR containing this code without reservations.

Don't return JSX from inner functions

I've seen this pattern a lot, and I've been involved in a few discussions on this pattern.

Yet, I've found that most teams dislike this.

If you're about to declare an inner function in a component (perhaps with the prefix *render*) that renders JSX conditionally, you should reconsider.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following JavaScript code:

```
export function App({ username, isLoading }) {  
  const renderUsername = () => {  
    if (isLoading) {  
      return <div>Is Loading... </div>;  
    }  
  
    return <p>{username}</p>;  
  };  
  
  return <div className="App">{renderUsername()}</div>;  
}
```

If you find the need to return JSX conditionally, you can use these two popular solutions:

- Ternary inside the JSX
- Split into a separate component

Ternary inside the JSX

I've been involved in countless discussions on the use of ternary inside the JSX, and there seem to be quite heated opinions about it.

From my experience, most developers are good with a solution like this, as long as the logic is kept simple.

```
export function App({ username, isLoading }) {
  return (
    <div className="App">
      {isLoading ? <div>Is Loading...</div> : <p>{username}</p>}
    </div>
  );
}
```

Split into a separate component

```
export function App({ username, isLoading }) {
  return (
    <div className="App">
      <LoadingUsername username={username} isLoading={isLoading} />
    </div>
  );
}

function LoadingUsername({ username, isLoading }) {
  if (isLoading) {
    return <div>Is Loading...</div>;
  }

  return <p>{username}</p>;
}
```

This is the most “correct” way to do this in React.

It comes with the price of having to pass down the *username* and *isLoading* props an extra level, and if callbacks are needed, these will have to be passed as well.

Don't wrap `useEffect` in an `async IIFE`

I've seen this pattern around a few times, and it's never a super popular solution.

You probably recognize the use case. You need to do something *asynchronous* in *useEffect*, but you cannot use the *await* keyword.



```
export function App() {  
  useEffect(() => {  
    (async () => {  
      await fetch("https://some.api");  
    })();  
  }, []);  
  
  return <div className="App"></div>;  
}
```

Some would argue that you should be careful about doing something *asynchronous* in *useEffect*, to begin with.

In any circumstances, this is not pretty!

If you need to do this, either declare the function inside *useEffect* and call it immediately after, or - better - declare it in the component scope instead.



```
export function App() {  
  const callSomeApi = async () => {  
    await fetch("https://some.api");  
  }  
  
  useEffect(() => {  
    callSomeApi();  
  }, []);  
  
  return <div className="App"></div>;  
}
```

Much better! Most likely, as logic grows, we'd have to do that at some point anyway. We might as well just do it now and keep the code clean.

Don't overuse the inbuilt hooks

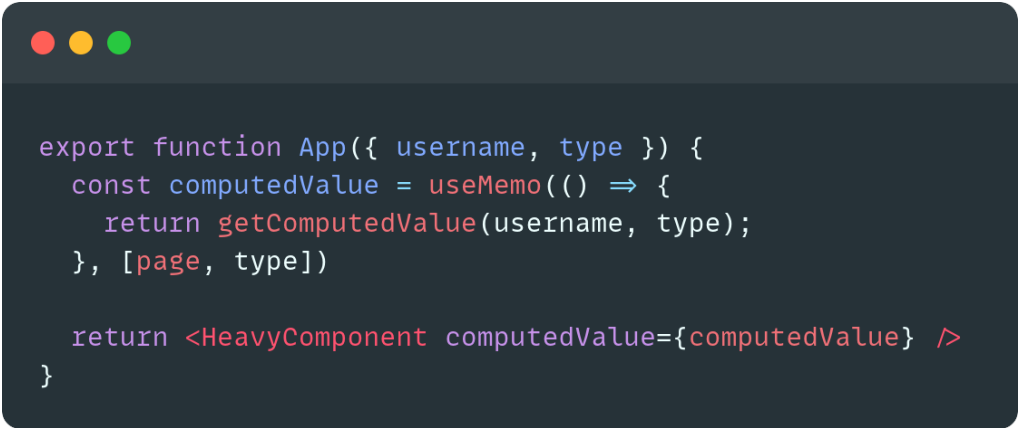
Yes, hooks are super cool! And when you learn how to use them, you may feel the urge to put them everywhere.

Please don't.

Stop overusing *useMemo*

I've encountered this a lot. The fear of assigning a variable from a function call directly in the component leads developers to use *useMemo*.

Let's take an example.



```
export function App({ username, type }) {  
  const computedValue = useMemo(() => {  
    return getComputedValue(username, type);  
  }, [page, type])  
  
  return <HeavyComponent computedValue={computedValue} />  
}
```

It's easy to see how the developer has been reasoning here.

The *HeavyComponent* shouldn't rerender unnecessarily, so the *computedValue* is *memoized*.

The *useMemo* hook is meant to prevent extensive computation every time a component re-renders, so the first thing we need to ask ourselves here is; is the *getComputedValue* actually an extensive operation?

In most cases, it's **not**.

And the cost of having React manage a *memoized* value actually outweighs the performance benefits you get from using *useMemo*.

Instead, don't be afraid of calling *getComputedValue* directly from the root of the component.

```
export function App({ username, type }) {  
  const computedValue = getComputedValue(username, type);  
  
  return <HeavyComponent computedValue={computedValue} />  
}
```

If the *computedValue* resolves to a primitive (a string, number, boolean, etc), we don't have to worry about *HeavyComponent* re-rendering either.

If the value doesn't change, it won't re-render.

Stop overusing *useCallback*

Similarly, I've seen the *useCallback* hook being overused as well.

It mostly boils down to two situations:

Either *useCallback* is used to set an initial state while preventing an infinite render-loop. But in this case, *useEffect* is really the hook for the job.

Or, it's used in the belief that the result will be *memoized*.

In that case, *useMemo* is the hook for the job, but most likely, you simply want to compute whatever you need in the component itself.

In fact, improper use of *useCallback* actually worsens performance.

If you're in doubt when to use *useCallback*, remember that as a rule of thumb, you're not going to need it.

NodeJS & AWS Lambda

NodeJS is a backend runtime environment that runs on the V8 engine and enables us to write and execute server-side JavaScript.

Since its initial release 12 years ago, it has gotten immensely popular.

It also happens to be the most popular choice when writing serverless functions on AWS (Lambda).

It's a huge topic, and I'll keep this rather short.

There are a few things I'd like to highlight.

Use promises instead of callbacks

NodeJS was originally built using a callback pattern for asynchronous calls.

All of NodeJS's builtins are structured this way: You provide the main arguments along with a callback function that is applied when the asynchronous operation is done.



```
const fs = require("fs");

fs.writeFile("./file.txt", "some-text", (err, data) => {
  if (err) {
    console.error();
  }

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

Remember, NodeJS was introduced in 2009.


In 2021, we use promises.

Fortunately, it's quite easy to convert these methods to using promises instead. Let's look at two different ways.

Using *promisify*

You can use a utility function, *promisify*, from the *utils* module to wrap the function using a callback in a promise.

It works for all functions that follow the NodeJS callback convention, which means that it works for a range of old third-party libraries for NodeJS as well.



```
const fs = require("fs");
const { promisify } = require("util");
const writeFilePromise = promisify(fs.writeFile);

try {
  const data = await writeFilePromise("./file.txt", "some-text");
  console.log(data);
} catch (error) {
  console.error(error);
}
```

Fortunately, in NodeJS version 12 and up, it became even easier for us.

Using *module/promises*

Instead of handling the promise-wrapping yourself, all NodeJS builtins come with a promisified version of their functions, straight from the module itself.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following JavaScript code:

```
const fs = require("fs/promises");

try {
  const data = await fs.writeFile("./file.txt", "some-text");
  console.log(data);
} catch (error) {
  console.error(error);
}
```

This is, by far, the easiest way to use promises in NodeJS in 2021.

Please, stick to this pattern anywhere you can.

Async handlers in AWS Lambda

The same goes for AWS Lambda.

You don't have to use the *callback* argument of your AWS Lambda functions anymore.

If your handler is *async*, you can simply return the result instead of applying it to the callback argument.

This is how an AWS Lambda function was expressed prior to the support of NodeJS version 8.

```
module.exports.handler = (event, context, callback) => {  
  const response = {  
    statusCode: 200,  
    body: JSON.stringify({ message: 'hello world' }),  
  };  
  
  callback(null, response);  
};
```

Today, we don't have to use the *callback* argument anymore.
Instead, declare the handler as *async*, and return the result instead.

```
module.exports.handler = async (event, context) => {  
  const response = {  
    statusCode: 200,  
    body: JSON.stringify({ message: 'hello world' }),  
  };  
  
  return response;  
};
```

If you need to fulfill promises during the function call, you simply apply the *await* keyword like you normally would.

Final words

Which conventions should I use?

So, after reading this book, you may be left with the question: Which conventions should I use? Should I blindly follow all the rules, styles, and conventions from this book?

No!

All teams are different. All codebases are different.

The two most important things to consider first, are keeping a consistent code base and playing well on a team.

So if you're invited to a team where both the team and code-base are already +6 months old, try your best to adopt the styles, preferences, and conventions you already see.

Listen to your team - if everyone else prefers it in a specific way, go with that. If the codebase uses a certain pattern consistently, stick with that.

That said, suggestions are always good, and - in my experience - welcomed on most professional teams.

Remember, being a software developer in the real world is mostly a people's game. It's much less about programming than you might think.

— Simon Høiberg

Additional resources

If you're new to JavaScript and looking for good resources to get started, I recommend these:

You Don't Know JS Book Series by Kyle Simpson:

<https://www.amazon.com/gp/bookseries/B01N9EBP9V>

(Fantastic books!)

Free Code Camp's YouTube Channel:

<https://www.youtube.com/c/Freecodecamp>

(A gold mine of FREE resources)

NodeJS Design Patterns:

<https://www.nodejsdesignpatterns.com/>

(This is for you that know JavaScript, but want to get better at NodeJS)

Finally, I want to mention Snappify, which I used to create all the cool code snippets for this book:

<https://snappify.io/>

(It's built by some awesome friends of mine, you should really check it out)

If you haven't already, you can find me on both YouTube, Twitter, LinkedIn, and Instagram, where I'm very active in sharing knowledge about SaaS and online business, software development, and a lot of JavaScript.

YouTube:

<https://www.youtube.com/SimonHoiberg>

Twitter:

<https://twitter.com/SimonHoiberg>

LinkedIn:

<https://www.linkedin.com/in/simonhoiberg/>

Instagram:

<https://www.instagram.com/simonhoiberg/>