# **“use strict”;**

**For a long time, JavaScript evolved without compatibility issues.**

* New features were added to the language while old functionality didn’t change.
* That had the benefit of never breaking existing code. But the downside was that any mistake or an imperfect decision made by JavaScript’s creators got stuck in the language forever.

**This was the case until 2009 when ECMAScript 5 (ES5) appeared.**

* It added new features to the **language** and modified some of the existing ones.
* To keep the old code working, most such modifications are off by default.
* You need to explicitly enable them with a special directive: "use strict".

**Strict mode makes several changes to normal JavaScript semantics:**

1. Eliminates some JavaScript silent errors by changing them to throw errors.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
3. Prohibits some syntax likely to be defined in future versions of ECMAScript.

## **Ensure that “use strict” is at the top**

**Please make sure that "use strict" is at the top of your scripts, otherwise strict mode may not be enabled.**

alert("some code");

// "use strict" below is ignored--it must be at the top

"use strict";

// strict mode is not activated

## **There’s no way to cancel *use strict***

**There is no directive like "no use strict" that reverts the engine to old behavior.**

* **Once we enter strict mode, there’s no going back.**

## **Browser console**

**When you use a developer console to run code, please note that it doesn’t use strict by default.**

* **Sometimes, when use strict makes a difference, you’ll get incorrect results.**
* **So, how to actually use strict in the console?**
* **First, you can try to press Shift+Enter to input multiple lines, and put use strict on top, like this.**

'use strict'; //<Shift+Enter for a newline>

//  ...your code

<Enter to run>

## Should we “use strict”?

**One could recommend to start scripts with "use strict"… But you know what’s cool?**

* Modern JavaScript supports “classes” and “modules” – advanced language structures (we’ll surely get to them), that enable use strict automatically.
* So we don’t need to add the "use strict" directive, if we use them.

**So, for now "use strict"; is a welcome guest at the top of your scripts. Later, when your code is all in classes and modules, you may omit it.**

* As of now, we’ve got to know about use strict in general.

## **Invoking strict mode**

**When it is located at the top of a script, the whole script works the “modern” way.**

**Strict mode applies to entire scripts or to individual functions.**

* It doesn't apply to block statements enclosed in {} braces; attempting to apply it to such contexts does nothing. eval code, Function code, event handler attributes, strings passed to setTimeout(), and related functions are either function bodies or entire scripts, and invoking strict mode in them works as expected.

### **Strict mode for scripts**

**// Whole-script strict mode syntax**

**"use strict";**

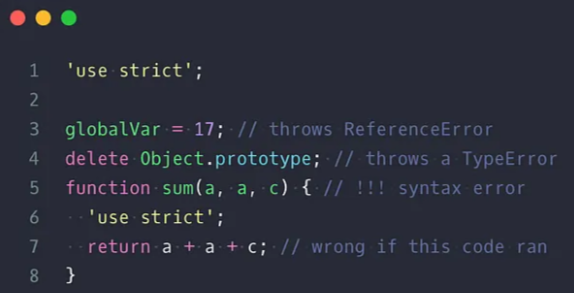
**const v = "Hi! I'm a strict mode script!";**

### **Strict mode for functions**

**// Whole-script strict mode syntax**

**"use strict";**

**const v = "Hi! I'm a strict mode script!";**



# **Use Function expressions instead of Function Declarations**

**Unless you want to take advantage of Function behavior and properties, prefer function expressions.**

***Function declarations*** are hoisted and although it can be useful sometimes, avoid them as they introduce weird behavior to the code and it is not always obvious what's happening.

****Try to make it clear where the function you are using comes from and they come before you use them to avoid weird access.

# Stop using “var”

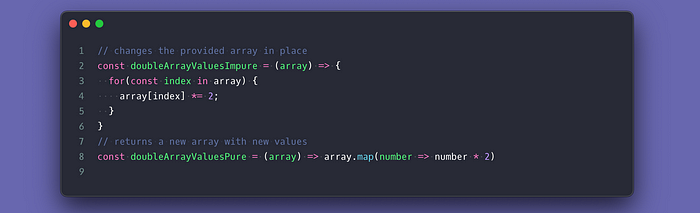
Declarations with “var” are also hoisted which makes var declarations accessible before where the declaration happened which is weird, non-obvious behavior.

# **Use “const” and immutability as much as possible**

Prefer immutability as much as possible. Constantly changing data and passing it around can make it hard to track bugs and the changes themselves. Work on data copies and avoid side effects.

## Prefer Pure Functions

Continuing on the side effect note, ensure your functions are not changing data they are called with or data in the scope where they are created.



## Prefer Class over Constructor Functions

Although the constructor function allows you to do some very nice stuff, if you find yourself reaching out for its prototype is a sign you need to use “class” which is supported pretty much anywhere. It is cleaner and something people are more likely to understand.

## Use “destructuring”

Destructuring is elegant and makes it more obvious what you need from arrays and objects and also gives you the opportunity to rename things to help give more sense to your code.



## Only work with the data you need

Like the above examples, destructuring is a good way to extract the data you need to do the job, also makes it a habit to only call methods and functions with the things they need. This also goes to the data coming from the API. Extract and clean up only the data you need before storing or doing anything to it.

## Always use “===”

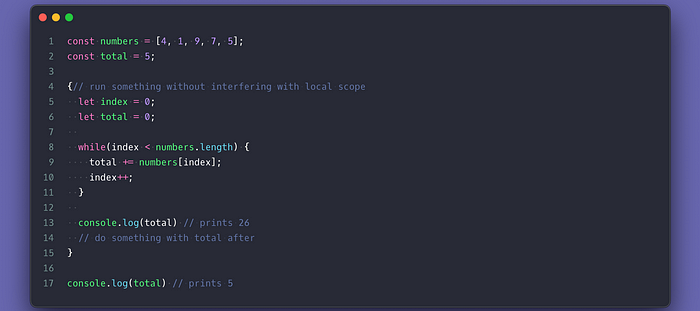
The triple equal checks for value and type and it is something you always want to do. Make it a habit to always triple-check and avoid undesirable effects.

## Avoid Global Variables

Avoiding creating things in global objects unless you are creating a library/framework. Global property names may collide with third parties or something a colleague also introduced and are hard to debug.

## Wrap loose declarations in blocks

You can avoid name clash and lose temporary declarations access by wrapping a quick logic in its own scope.



## Organize your declarations

Be consistent with the way you declare things. Put all your declarations on top starting with the constants down to the variables. Make constants all uppercase to indicate they are constants which will prevent devs from trying to change them.

## Don't initialize things with “undefined”

Something is “undefined” when it lacks value. Let’s agree that assigning “no value” as a “value” for something is a pretty weird concept right? Since JavaScript already makes things “undefined” how can you tell whether something is undefined because of you or JavaScript? It makes it hard to debug why things are “undefined” so prefer setting things to “null” instead.

## Always initialize your declarations

For the same reason, you should not give “undefined” as a value to declarations, you should not leave them without a value because they are “undefined” by default.

## Lint your code and have a consistent style

Linting your code is the best way to ensure a consistent look and feel of your code and make sure people don't do weird things to it as well. It puts everyone on the same page.

## Use TypeScript

TypeScript can help you a lot in delivering better code. It will need some getting used to if you never tried a type system but it pays off in the long run.

## Functions and methods should do one thing only

It is easy to get carried away with adding extra stuff to a function while you are at it and the best way to find out whether a function is doing too much is by looking at its name. The name should tell what the function does and anything unrelated should go.



## Don’t be lazy when naming things

Always put some effort into naming things. If it is hard to name you probably gave it extra responsibility or do not understand what it is. Give it at least a 3 letter name with meaning.

## Avoid unnecessary declarations

Some declarations can be avoided altogether so only declare when it is strictly necessary. Too many declarations may hint at a lack of proper code design or declaration consideration

## Use default values when possible

Having defaults is more elegant than throwing errors because something was not provided. If you really want to catch not provided values you can check my article on [25 JavaScript solution](https://medium.com/javascript-in-plain-english/25-javascript-code-solutions-utility-tricks-you-need-to-know-about-3023f7ed993e) where I share a way to make things required that throw an error if no value is provided.

## Always have a default case for switch statements

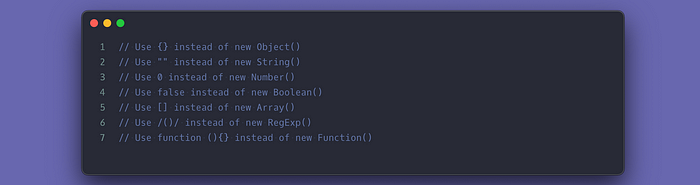
Don't leave your switch statements without a default case because something can go wrong and you want to make sure you catch it.

## Never use “eval”

Never! It is not necessary.

## Avoid the “new” keyword

Except for class and constructor functions instancing, you should never use the “new” keyword for anything else. They can slow compilers down.



## Add meaningful comments for nonobvious things

Only add comments when you did something not common, weird, or require context to be understood. Also, add comments to things that are a hack or may require improvements/fixing later on so the next person knows why. Add comments in your third parties' modules and modules in your codebase to explain the architecture and the intention behind things.

## Keep ternaries simple

Worst case scenario you have two nested ternaries. Anything longer should be an if statement or switch for readability and easy to debug reasons.

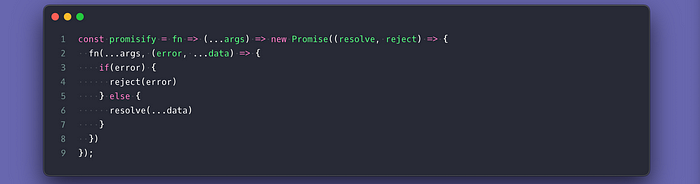
## Simplify with optional chaining

Get rid of those nested checks and use the “?” Operator.



## Prefer promises over callbacks

Promises are easy to use and anything with a callback can be “promisified”. Callbacks are the functions to call once something is done whether synchronous or not and with promises and async…await you get to do things asynchronous which may speed up your code, especially because JavaScript is single-threaded. You can't get away with promises only sometimes but promises make it easy to read code as well.



## For loops > .forEach sometimes

Don't change things into an array just so you can “.forEach” it. You are adding an extra process to a slow alternative. Loops are faster and allow you to use the “continue” and “break” keywords to control the looping.

## “for…in” and “for…of”

The for-in and for-of loops are very powerful ways to loop. The “for-of” loop lets you go over the values of the array, strings, Map, Set, etc. No need to change something into an array to use .forEach. I would avoid the “for-in” for looping as it is the slowest one and iterates over prototype keys.

## Optimize for loops?

For loops are already optimized by the compiler so no need for that kinda of optimization.



## Always “try…catch” JSON methods

Don't trust things passed to JSON methods “.stringify” and “.parse”. Try to catch them to make sure they don't fail and break your code.

## Prefer template strings

It is that simple. Template strings allow you to inject values into the string and they keep the format that can come in handy.

## Avoid nesting or chaining loops

When you chain iteration method or nest loops you are increasing the complexity of the code which may slow things down later on or as your data grows. Even though some operations may require it, always assess your looping strategy to ensure you don’t have unnecessary loops or loops that can be combined together.

## Avoid Weird Unreadable hacks

They are all over the internet because people find them “cool”. They are usually weird, non-conventional, and non-obvious when you look at them. It is always best to follow the guidelines of the tool you are using to ensure proper performance. Hacking should be that last alternative.

## Prefer the [“rest” operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) over [“arguments”](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments)

The “rest” operator will work with arrow functions where “arguments” are not available. Stick to one way to access your function arguments.

## Prefer “[globalThis](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis" \t "_blank)” for global access

Let the JavaScript handle the rest and make sure that your code will work whether it is inside a Web Worker or Backend Node.

## Understand JavaScript but Build with Libraries and Frameworks

I recommend investing time in understanding the JavaScript language itself but build with powerful tools like React and Angular to avoid common mistakes. Make sure you follow their guidelines since these tools already guard against common mistakes and employ best practices.

## Add semicolons, always!

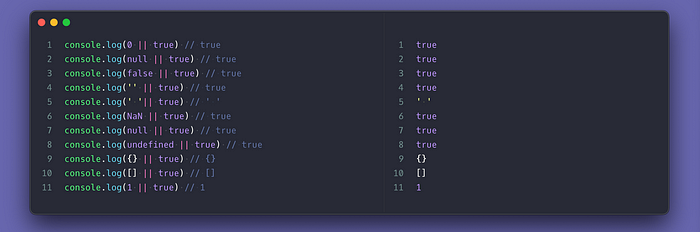
You may be surprised to find out that you can get away with not putting a semicolon in the JavaScript code. Know that the compiler adds them and tools like Babel may easily misread your code and cause a bug to make to production. Always add semicolons!

## Readable > Performance unless you need Performance

There are ways to get more performance by doing things that are often hard to read but unless you are desperate for performance at the code level (which is rare), make it readable.

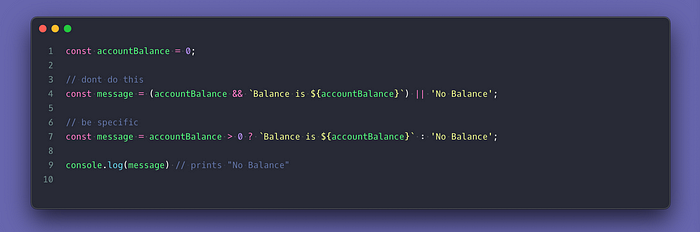
## Be careful with “Truthy” and “Falsy” checks

Don’t rely on the [“truthy”](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) and [“falsy”](https://developer.mozilla.org/en-US/docs/Glossary/Falsy)checks since you can easily introduce bugs to your code. Try to be specific in your checks as unexpected things may pass as a truthy check.



## Prefer Ternary over logical “||” and “&&” checks

The “or” and “and” operators evaluate the “true” and “false” of a value which may result in undesired results. Also, don’t rely on it to do weird logical condition checks as they are not readable and easy to understand.

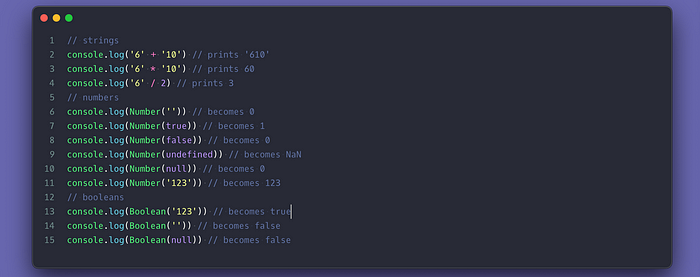


## Watch out for “undefined” and “null” with the “??” operator

The [nullish coalescing operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing_operator" \t "_blank) makes sure that null and undefined values are not picked and it is perfect for cases where you want to ensure that there is a value or fallback to a default value.

## Be careful with automatic type conversions

This is probably another reason to try TypeScript as JavaScript does an automatic type conversion on the fly which may not be what you are expecting. “Truthy” values become “true” and “Falsy” values become “false”. Doing math between numbers and strings may actually work or result in a string concatenation. Numbers almost always turn “Falsy” values into “zero” and “Truthy” into “one”.



## Never trust data you don't create

Whenever you are dealing with data coming from a user or from an API you don't know, make sure it is of the right type and in a format that you can work with before you do any operation on it.

## Use regex when extracting and looking for things in Strings

Regex is super powerful and fun. Avoid weird manipulation like looking for indexes and grabbing things. Regex allows you to look for complex patterns and

## [IIFE](https://developer.mozilla.org/en-US/docs/Glossary/IIFE) and small utility libraries

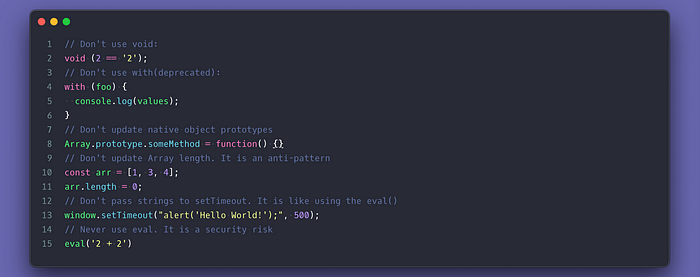
IIFE is an excellent way to execute things as early as possible which you can take advantage of to set up some stuff before the rest of the code starts running. You can also use it to initialize small libraries with a simple API that allows you to encapsulate some complex logic and expose an object you can use to interact with similar to how [jQuery](https://code.jquery.com/jquery-3.5.1.js" \t "_blank) is built.

## Avoid repeating yourself with utilities

Always turn things you do repeatedly into small generic functions that you can reuse later on. As a developer, you should not be repeating things and small functions make them easy to test and reuse.

## Don’t take advantage of weird JavaScript “features”

Things like updating [array length property](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length), using the [“with” keyword](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with), [void keyword](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void), and updating native Object prototypes like Date, Array, Object, etc. Others like passing a string to [setTimeout and setInterval](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout" \t "_blank). Just because the language allows you to, does not mean you should.



## Add Unit Tests

As a developer, I often found bugs when I started adding unit tests. Tests are the ultimate way to ensure that your code as error-free as possible. [Jest](https://jestjs.io/) is an excellent option to start with but there are others out there that are also as simple to use.