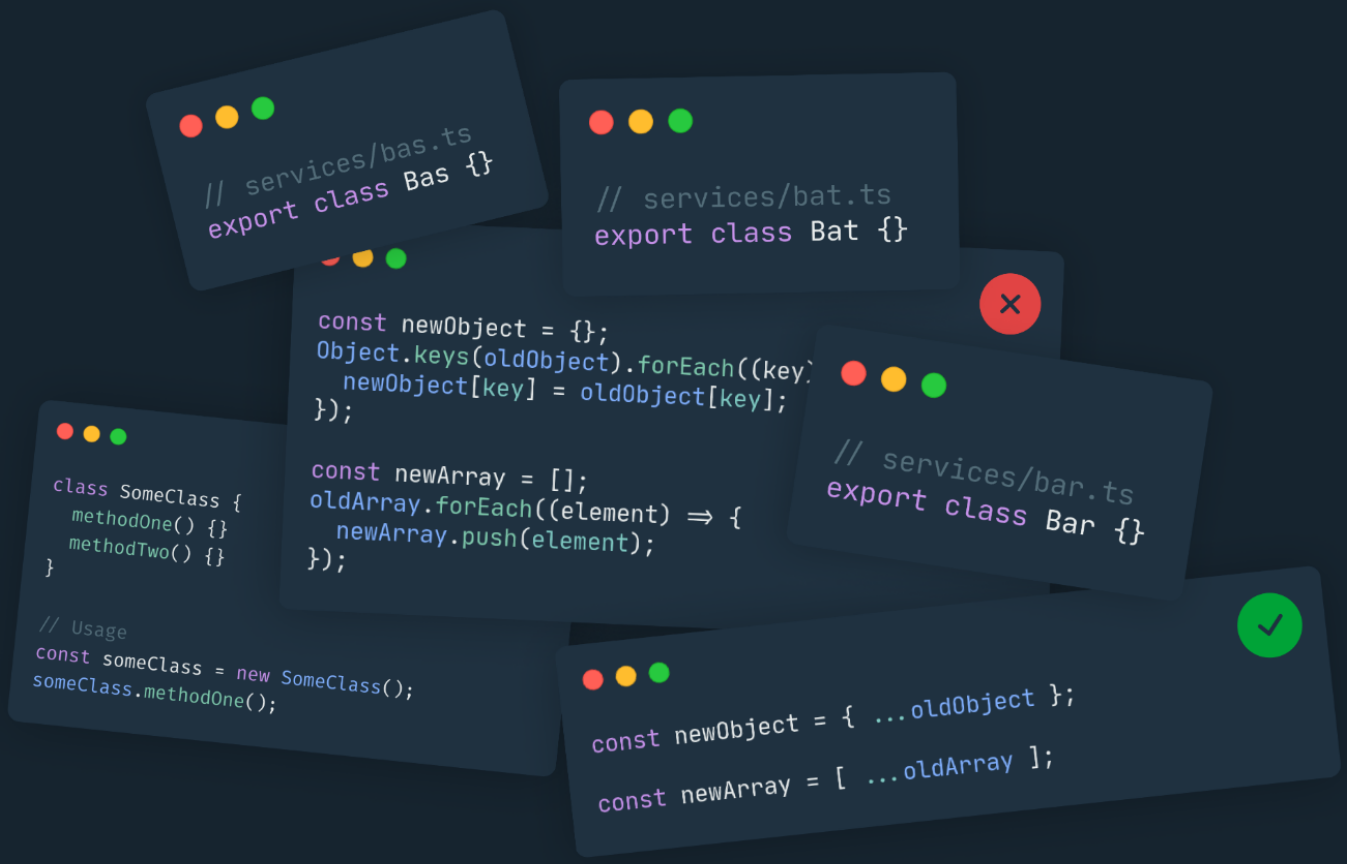
```
const includes = [
  "JavaScript",
  "TypeScript",
  "NodeJS"
];
```

65+ JAVASCRIPT CODE SNIPPETS

WITH EXPLANATIONS



eBook by Simon Høiberg

For more than a year, I've been sharing tips and tricks
in form of small code snippets for ***JavaScript***, ***TypeScript***
and ***NodeJS*** to help out the community.

This eBook contains +65 of the most popular ones.
Hand-picked, and with explanations
- ready to apply in your next project.

Enjoy!

Sincerest,
Simon Høiberg

JavaScript

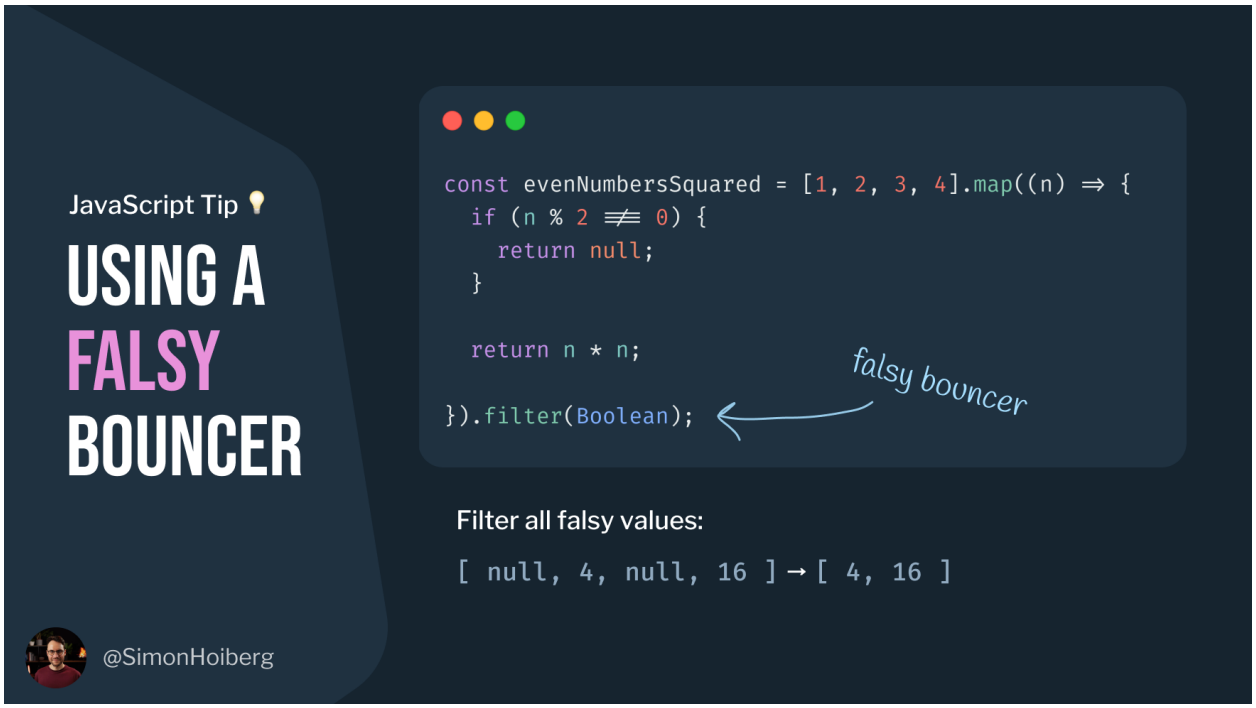
Avoid the “delete” keyword

Explanation

Don’t use *delete* to remove a property from an object.
This mutates the original object and can lead to unpredictable behavior which becomes difficult to debug.

Instead, use the rest operator to create a new copy without the given property.

Using a Falsy Bouncer



Explanation

When passing the 'Boolean' constructor directly to Array.filter as the first argument, it serves as a falsy bouncer.

This will filter all values that qualifies as falsy;
That is *false*, *null*, *undefined*, *0*, *NaN*, and "" (empty string).

Object Destructuring on arrays

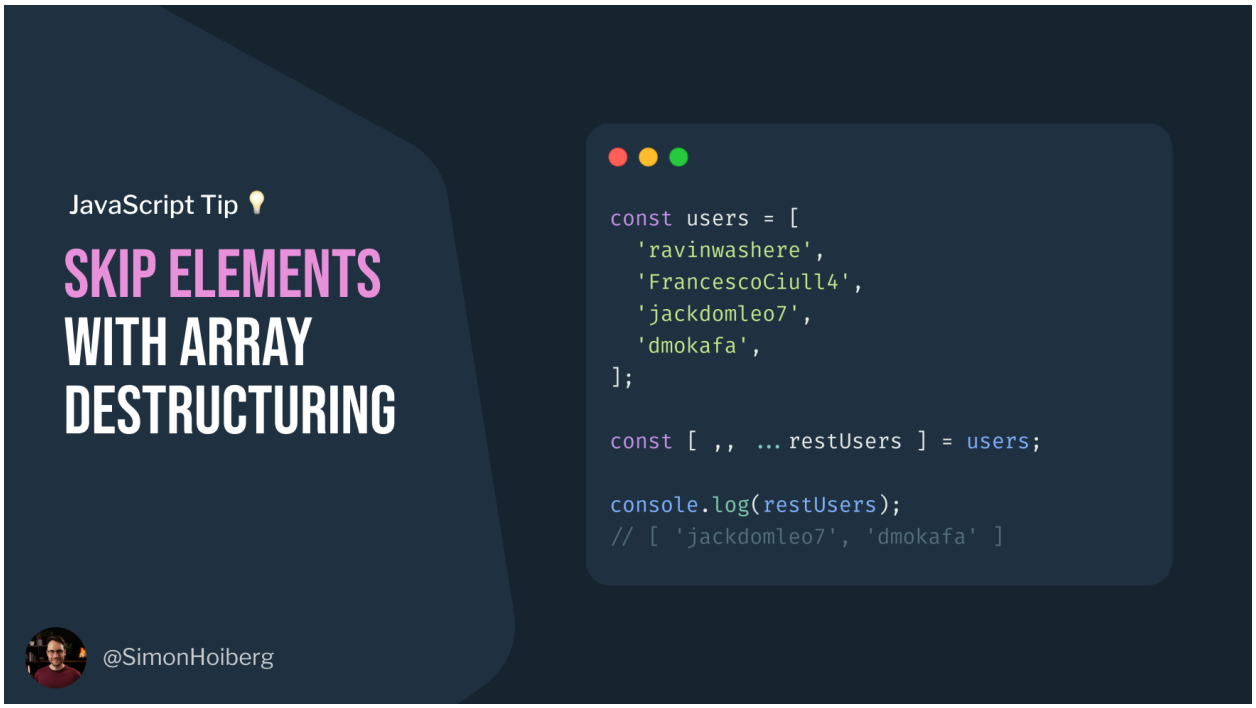
Explanation

You can destructure elements from an array using the same syntax as when destructuring for objects.

The property name corresponds to the index of the element in the array.

It’s a convenient way to pull out specific elements from an array in a single, clean line of code.

Skip elements with Array Destructuring

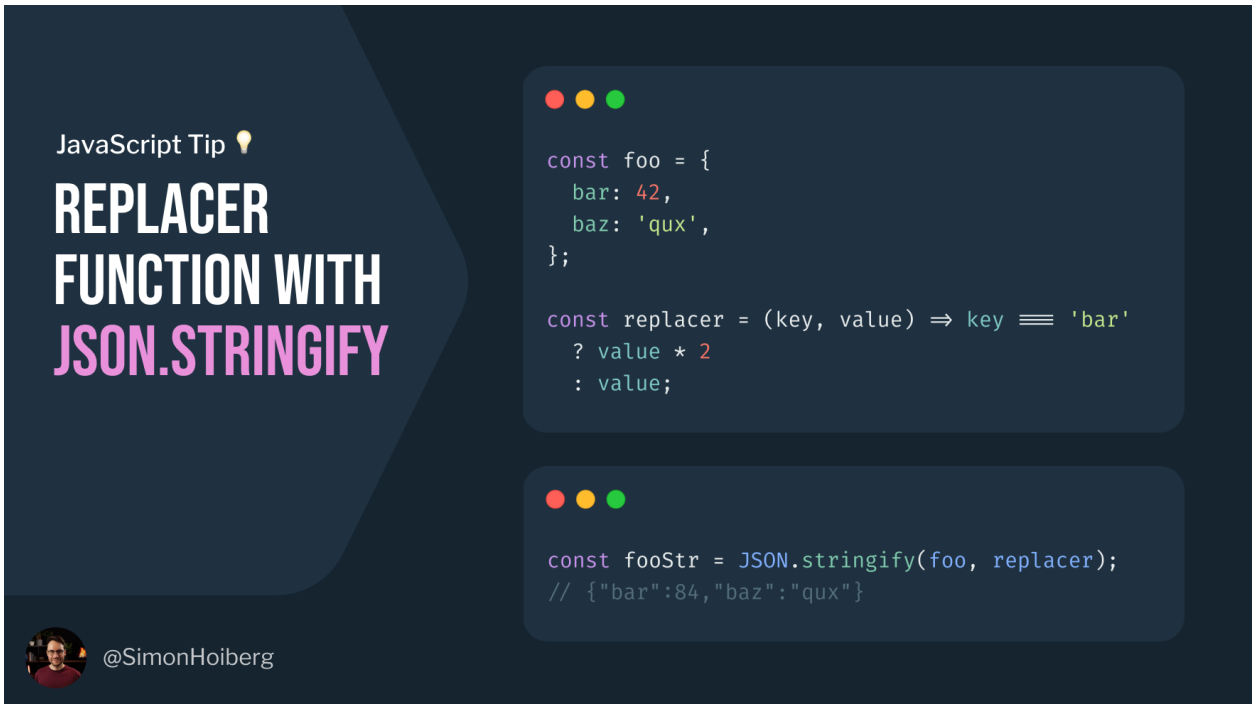


Explanation

You can use an empty 'placeholder' comma to skip elements when destructuring arrays.

If you want to access the second or third element from a list, or want to skip the first or second element (etc), this is a great and clean way to do it.

Replacer function with *JSON.stringify*



Explanation

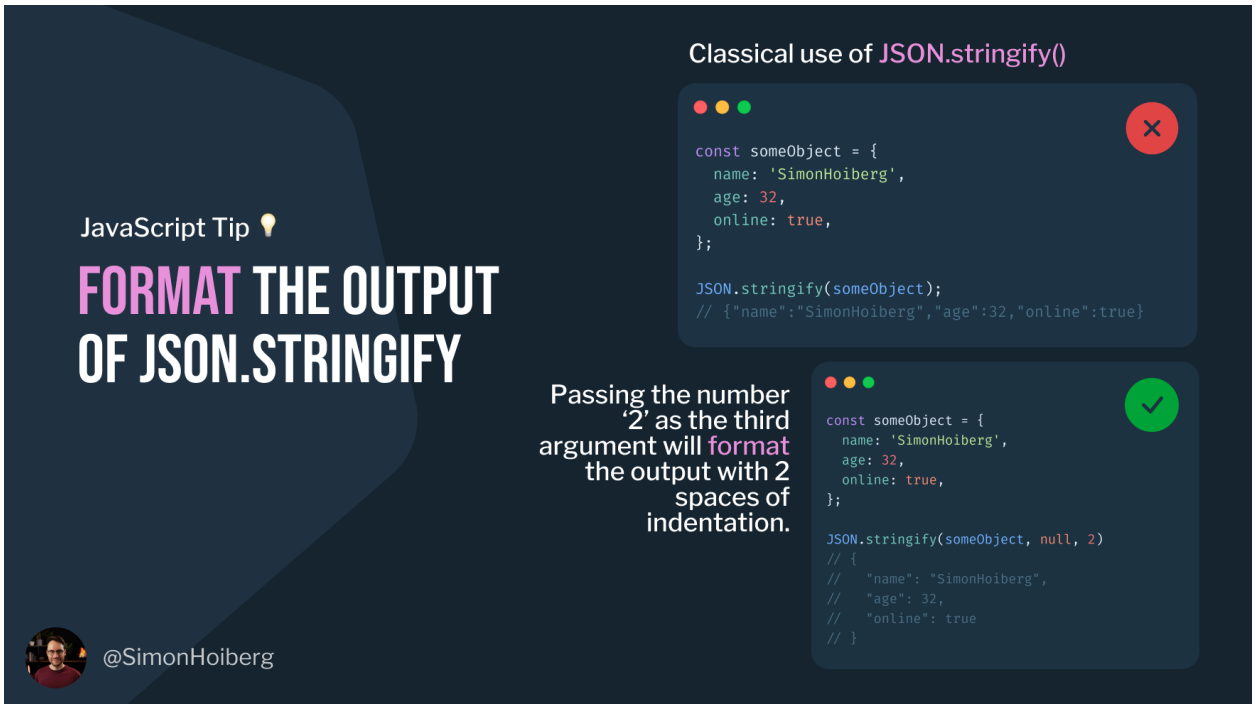
The replacer function is the second argument to JSON.stringify.

I alters the behavior of the stringification process.

The argument can also be an array or type *String* or *Number*.

This becomes an “allowlist” that filters the properties of the object with the name included in the list.

Format the output of *json.stringify*



Explanation

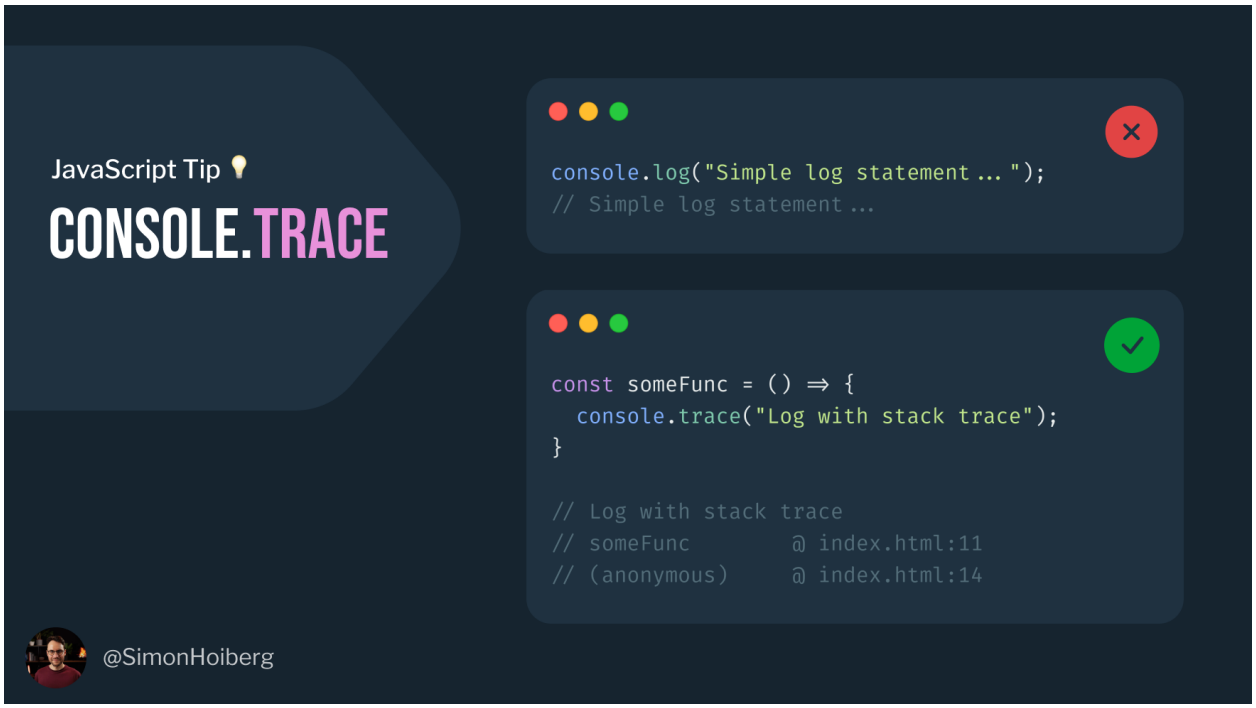
The spacer is the third argument to JSON.stringify.

You can pass a String or a Number that is used to insert white space into the output.

If you pass a Number, it will indicate the number of space characters to use.

If you pass a String, it will use it as the "space" character.

Using *console.trace*



Explanation

If you use console.trace instead of console.log, it will show you the complete call stack when debugging.

This is very convenient when you're working with larger setups with multiple files and modules.

Using *console.time*

Explanation

You can set timers using console.time.

This can be useful when debugging slow loops or function calls.

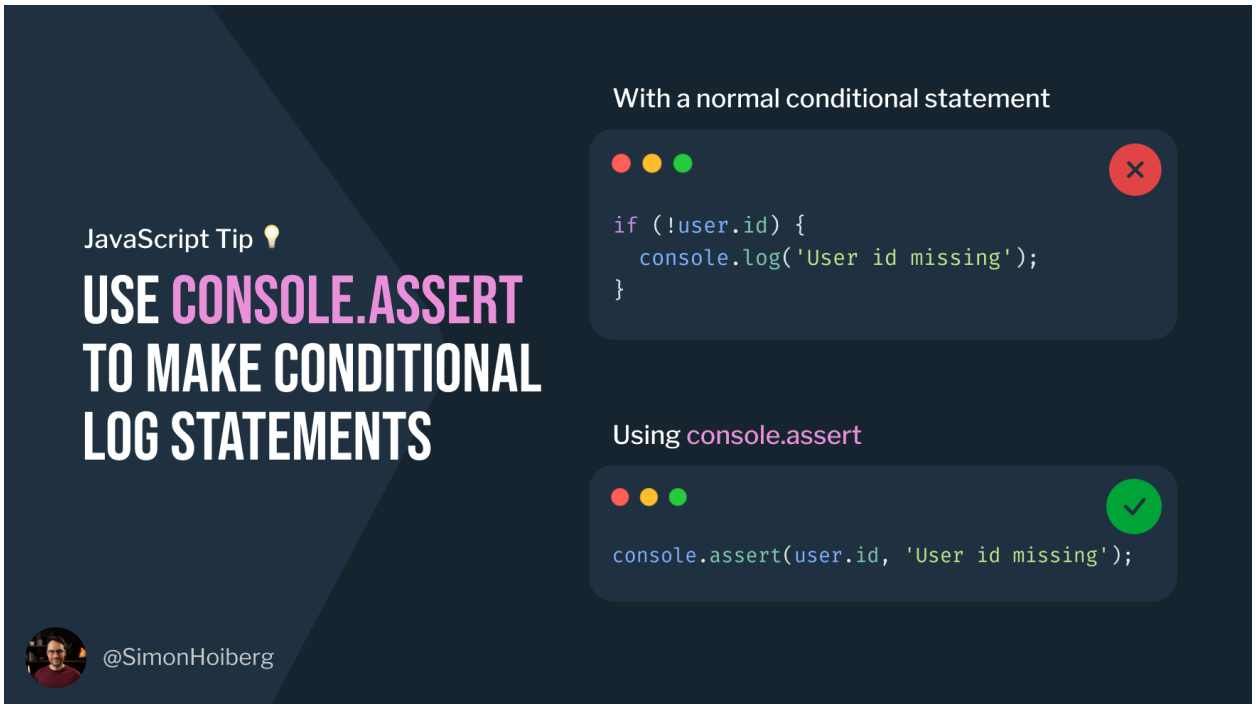
Using *console.group*

Explanation

console.group let's you use nested groups to help organize your output by visually associating related messages.

Additionally, you can use the *console.groupCollapsed* method to create a new block that is collapsed and requires clicking a disclosure button to read it.

Using *console.assert*



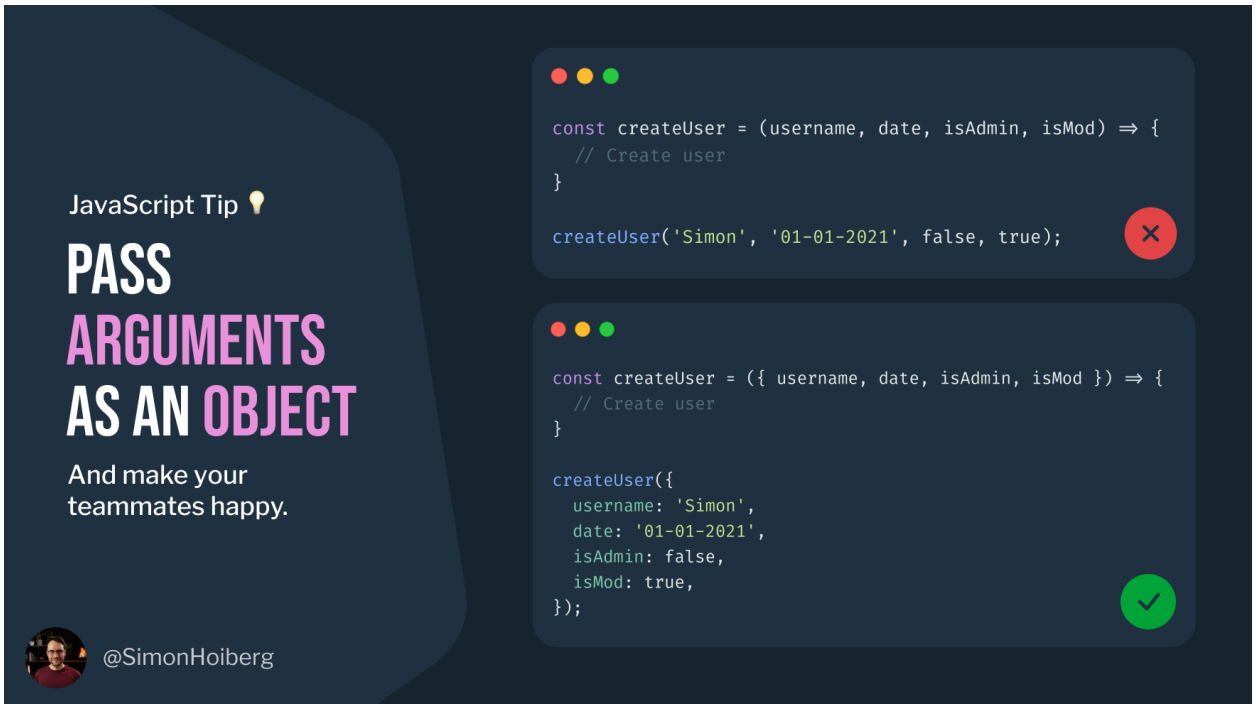
Explanation

Use *console.assert* to make conditional log statements.

The *console.assert* method writes an error message to the console if the assertion is false.

If the assertion is true, nothing happens.

Pass arguments as an object



Explanation

If more than 1 parameter is added to a function, it's very difficult to figure what these arguments mean, when the function is called.

Passing the argument contained in an object (a so-called DTO) makes it pretty clear from the names of the properties.

And also - the order doesn't matter anymore.

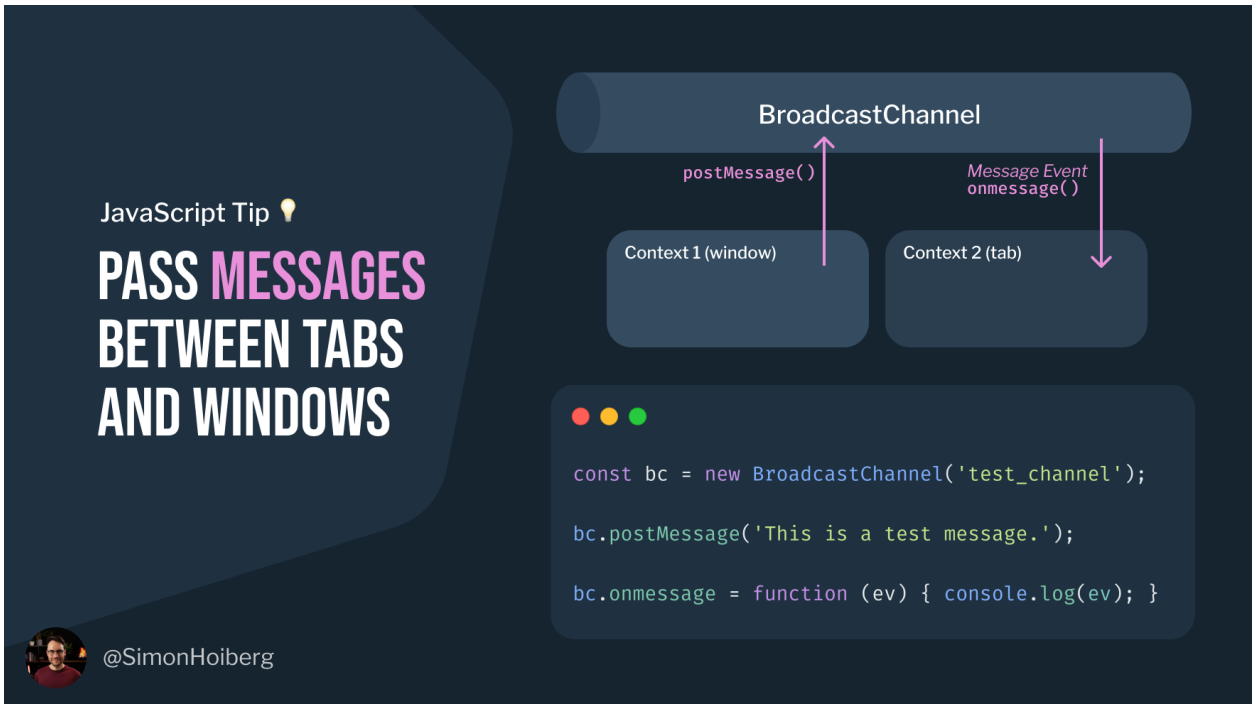
Readable numbers

Explanation

The Numeric Separators give us the ability to separate large numbers with an underscore (_) in numeric literals.

This greatly improves readability, and it will have no effect on how the JavaScript engine enteprets the number.

Pass messages between tabs and windows



Explanation

The Broadcast Channel API allows basic communication between browsing contexts (windows, tabs, frames or iframes).

Using the BroadcastChannel constructor, you can receive any messages that are posted to it without having to maintain a ferences to frames or workers.

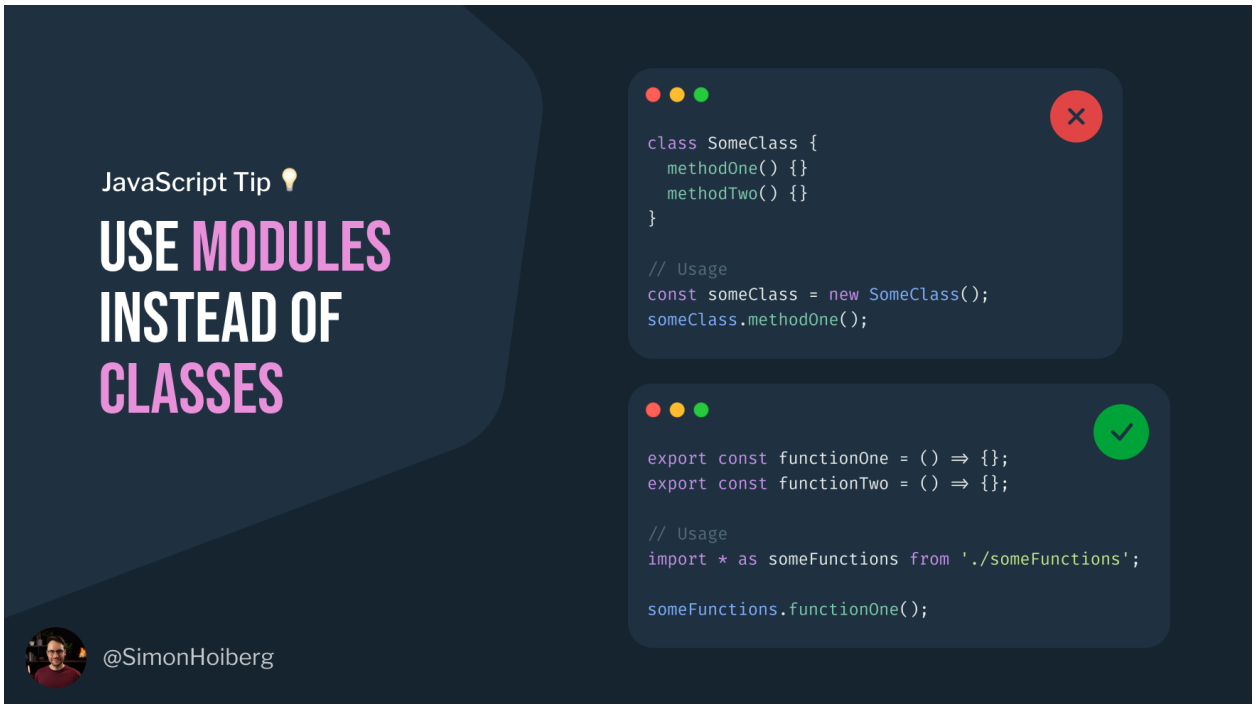
Remove duplicates from array

Explanation

The simplest way to remove duplicates from an array, is to use the Set constructor to create a new set object containing unique values (of any kind).

In other words, Set will automatically remove duplicates for us, and by spreading it into a new array, we can create a new array without duplicates.

Use modules instead of classes



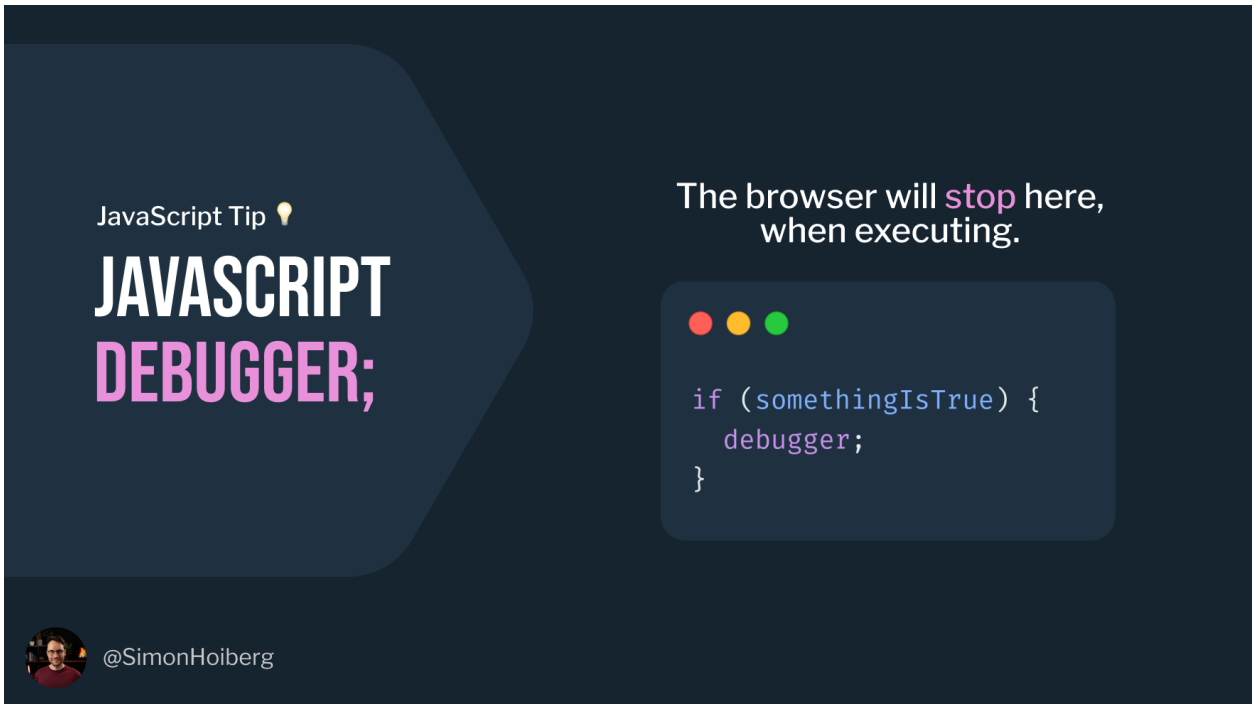
Explanation

In JavaScript, there are no classes.

It's syntactical sugar added to please developers from other languages such as Java or C#.

Most of the time, they don't serve a good purpose, and are not really useful. Instead, use modules.

Debugger;

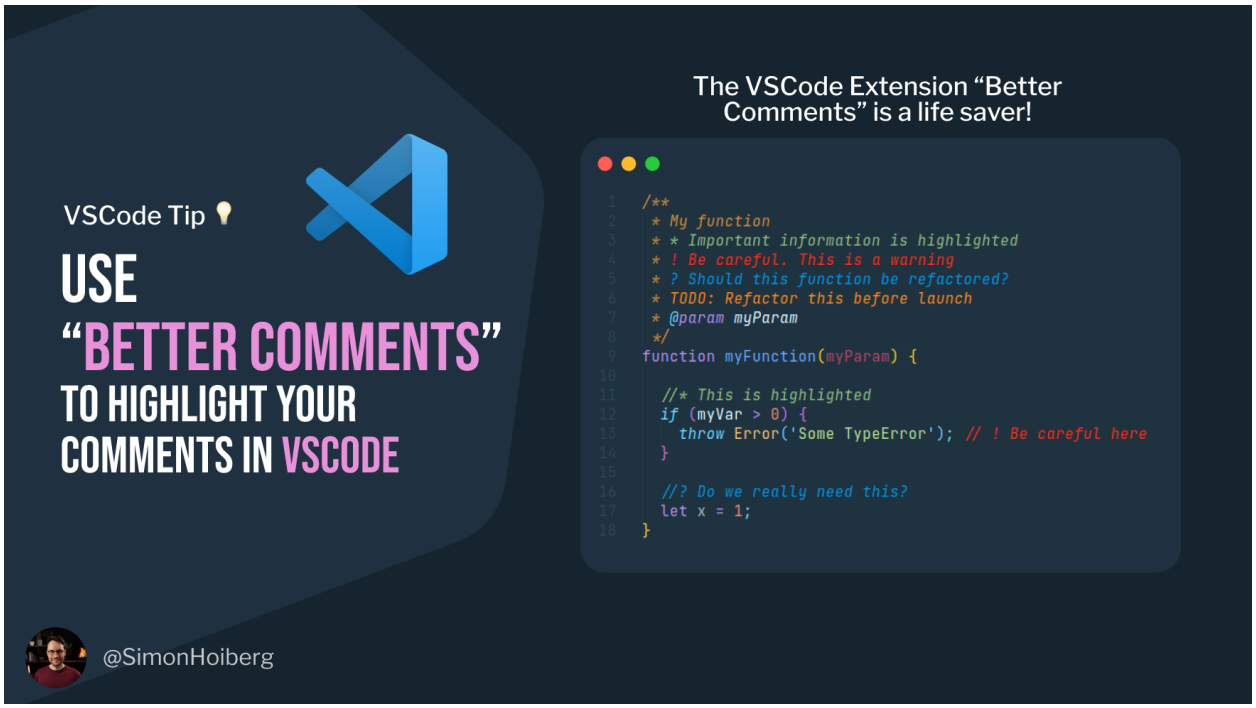


Explanation

Place a debugger; line in your code, and the browser will stop executing. This makes it easier to start investigating.

As an alternative, VSCode also ships with a great inbuilt debugger which works both with browsers and NodeJS.

VSCode - Better Comments



Explanation

The VSCode Extension "Better Comments" makes comments way more useful by applying simple color highlighting.

You can find "Better Comments" here: https://marketplace.visualstudio.com/items?itemName=aaron-bond.better-comments

VSCode - Import Cost

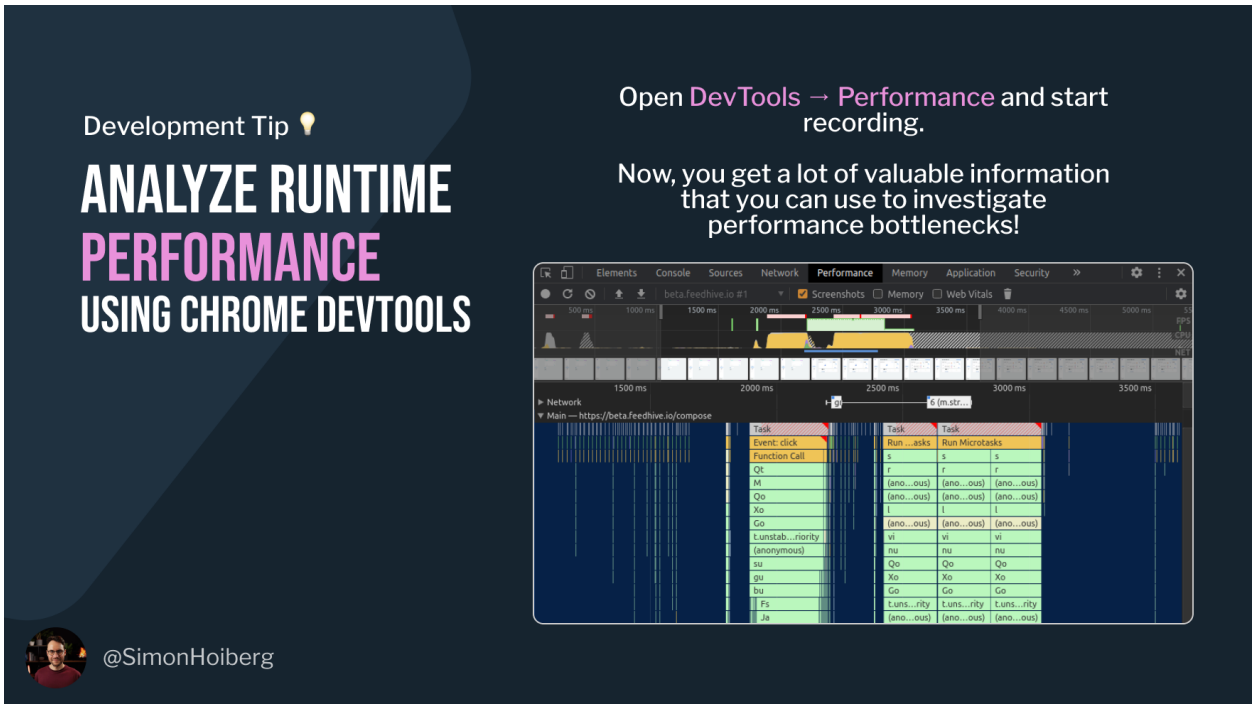
Explanation

With all the beauty of Webpack comes a typical bottleneck — that is a bundle size that quickly gets out of control.

Keeping an eye on the costs of your imports can help you improve performance.

You can find "Import Cost" here: https://marketplace.visualstudio.com/items?itemName=wix.vscode-import-cost

Analyze runtime performance using chrome devtools



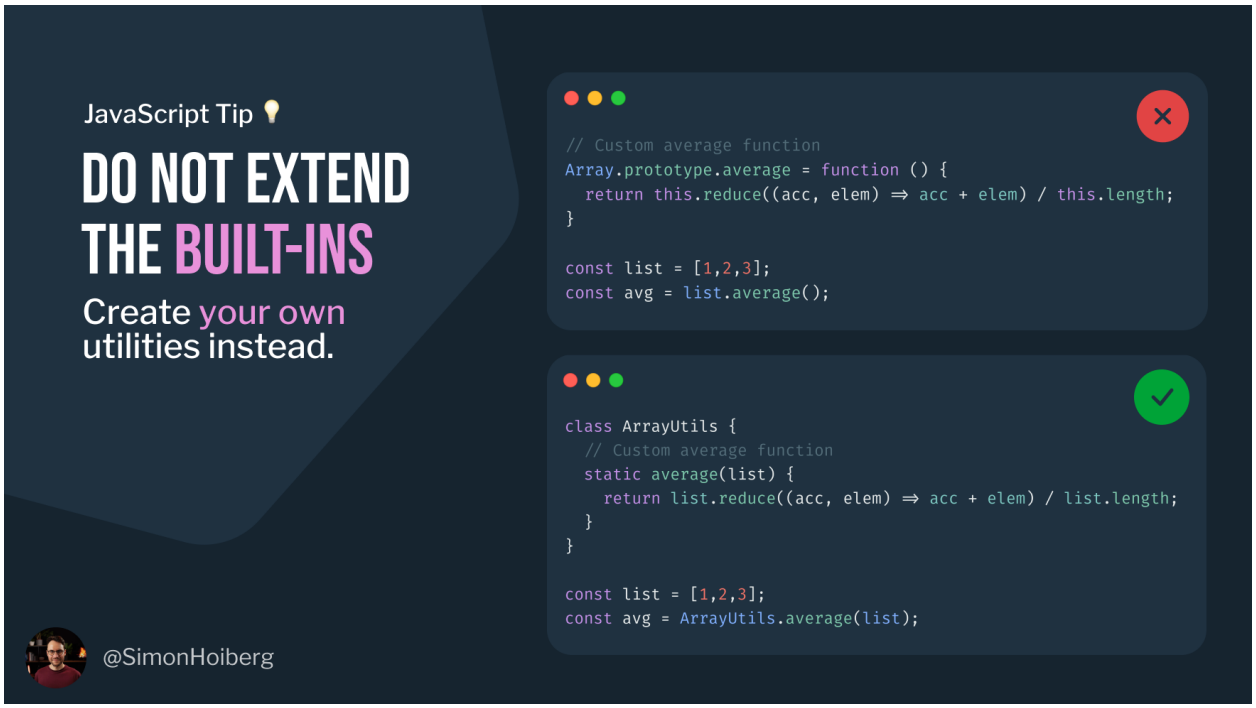
Explanation

You can use Chrome DevTools to analyze runtime performance of your web app.

Open DevTools -> Performance and start recording.

This is especially helpful when you try to identify React performance bottlenecks.

Do not extend the built-ins



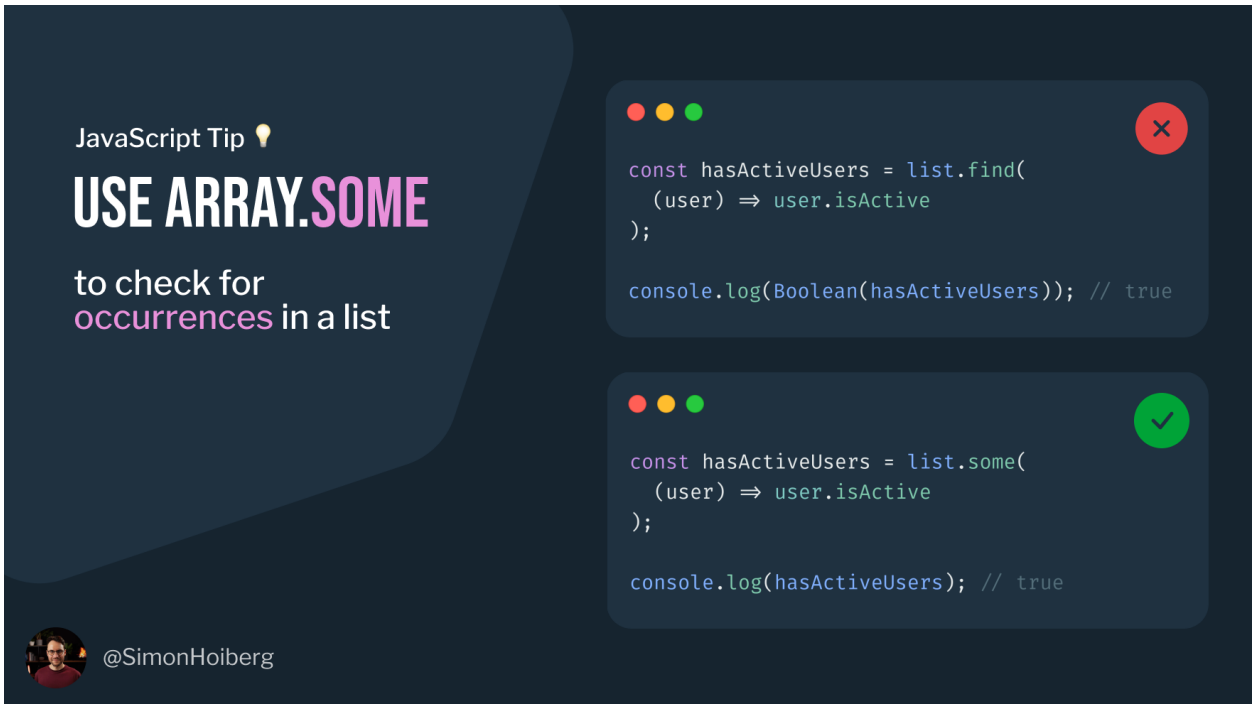
Explanation

Extending the standard built-ins is considered bad practice.

Create your own utility class instead.

(And share it on NPM, if it's useful to others).

Use *Array.some*



Explanation

Instead of using Array.find, or manually searching a list for an occurrence, use the array method Array.some instead.

It’s built for exactly that purpose.

React - Set state using currying

Explanation

By using currying, you can compose functions for different purposes.

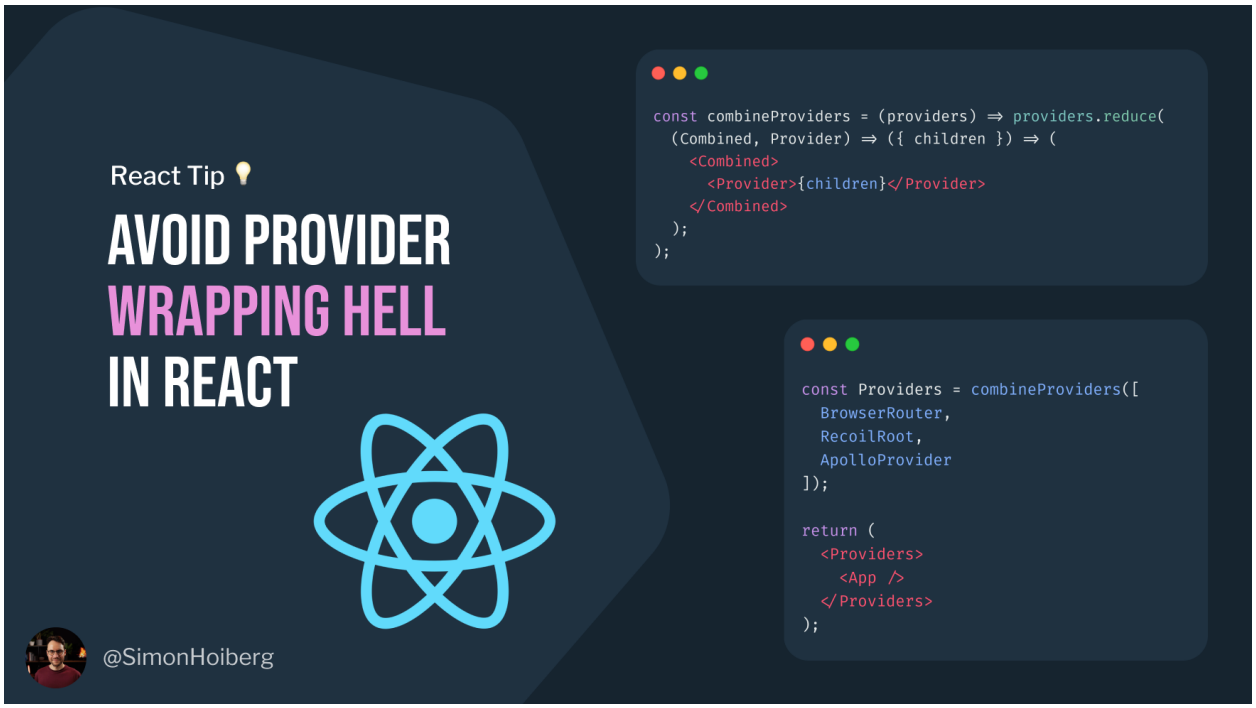
In this case, we're using one function to create different "setter" functions for updating state in React.

React - Let react assign a key to its children

Explanation

React has a built-in method that automatically assigns the keys for you when rendering a list of children.

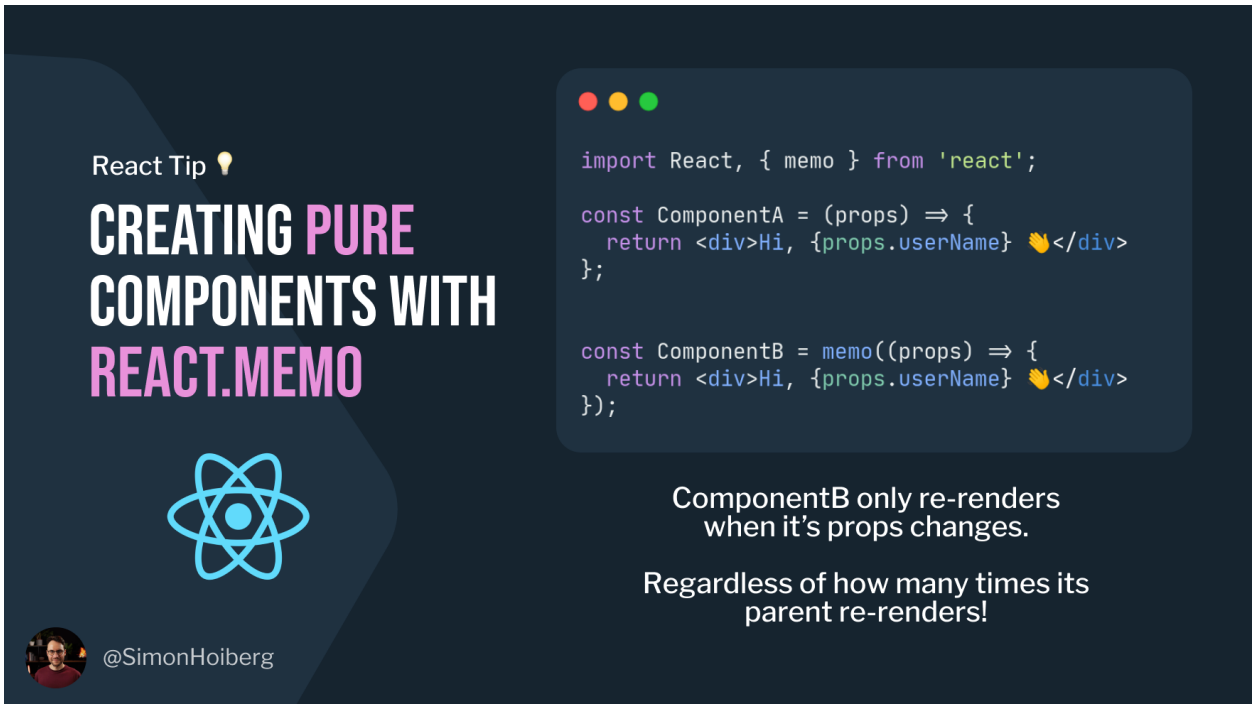
React - Avoid provider wrapping hell



Explanation

Avoid Provider wrapping hell in React by using this simple approach to combine all your Providers into a single Provider element.

React - creating pure components with *react.memo*



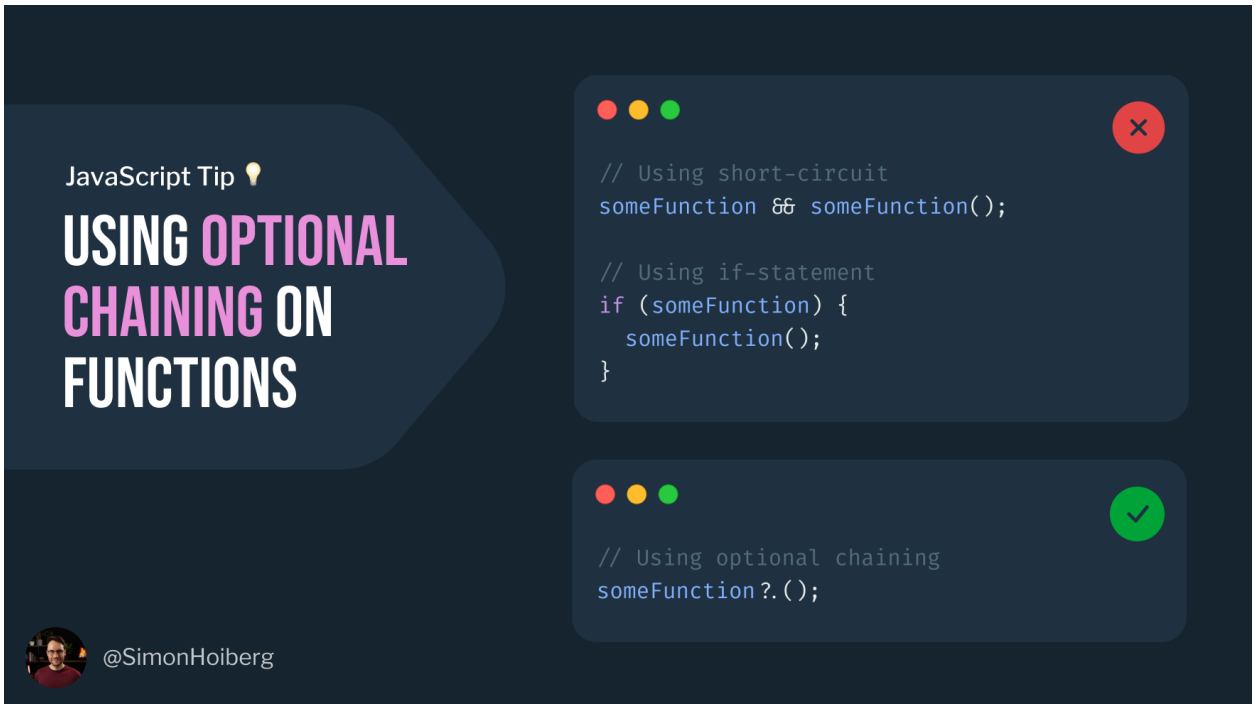
Explanation

If your component renders the same result given the same props, try wrapping it in React.memo to make them "pure".

This will prevent them from re-rendering, except when their props are changing.

It can potentially give you a great performance boost

Using optional chaining on functions



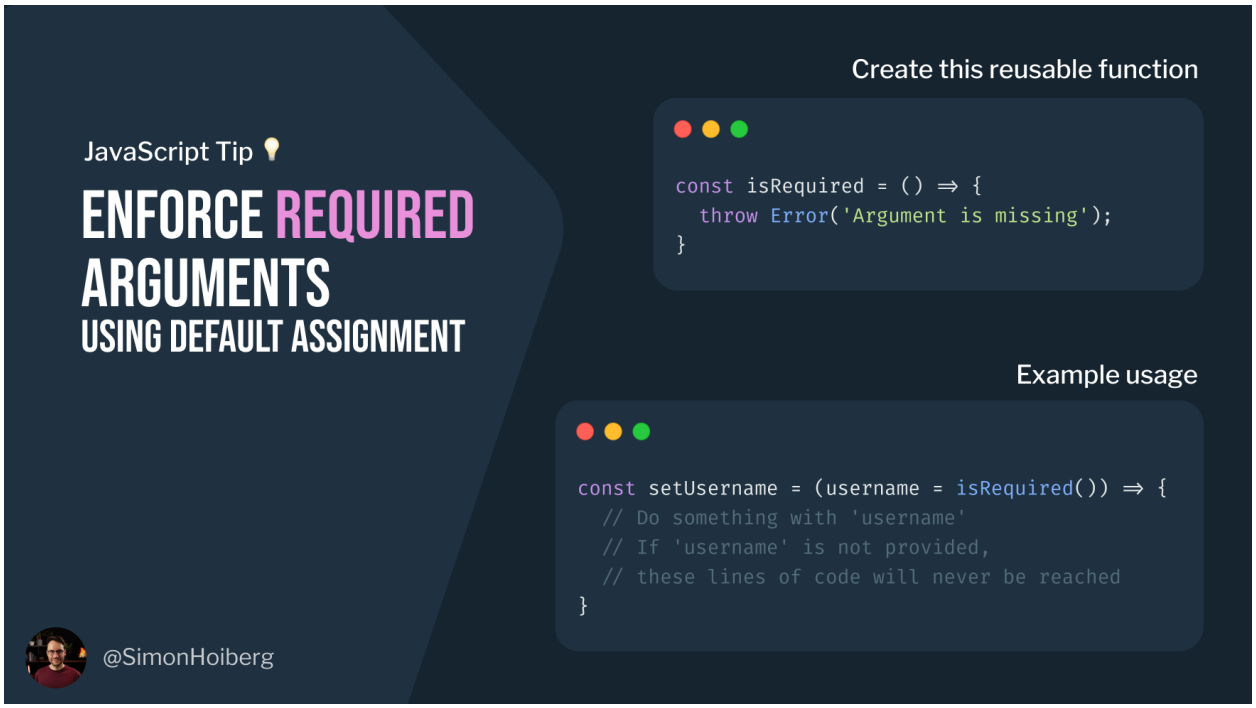
Explanation

The optional chaining operator (?.) enables you to access properties that are potentially null or undefined.

It works perfectly well on function calls as well.

A great example of usage is in React, where functions passed as props may be optional (and therefore potentially undefined).

Enforce required arguments using default assignment



Explanation

The idea here is to make a default assignment to the return-value of a function which throws an error when getting invoked.

In this way, an error will be thrown when an argument is not passed.

Be aware, that *null* is considered a value, hence passing *null* will not result in a default assignment.

Adding a leading zero

Explanation

Use this one-liner to add leading zeros to numbers.

(Very useful for dates).

Alternatively, you can use *padStart(2, '0')*.

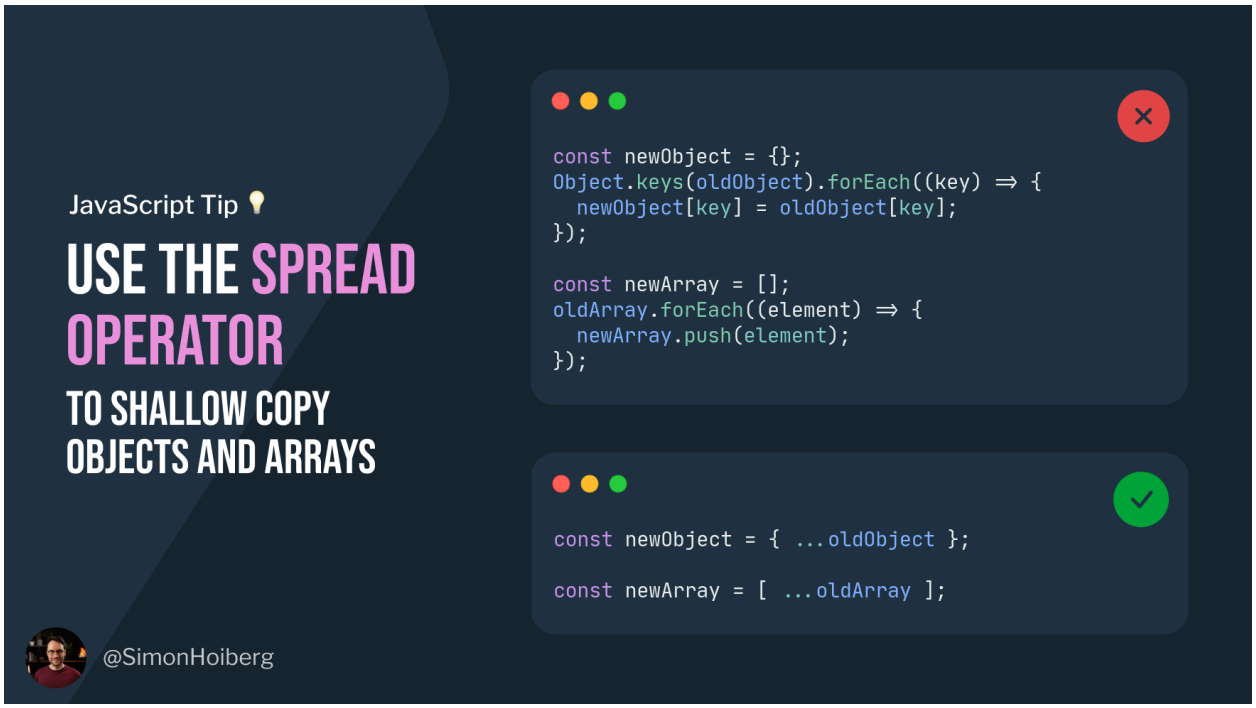
Avoid default exports

Explanation

- Poor discoverability and autocompletion.
- Horrible when using CommonJS.
- TypeScript struggles with auto-import.
- Re-exporting becomes tedious.

Generally, I always recommend simple exports + destructured import.

Use *spread* to shallow copy objects and arrays



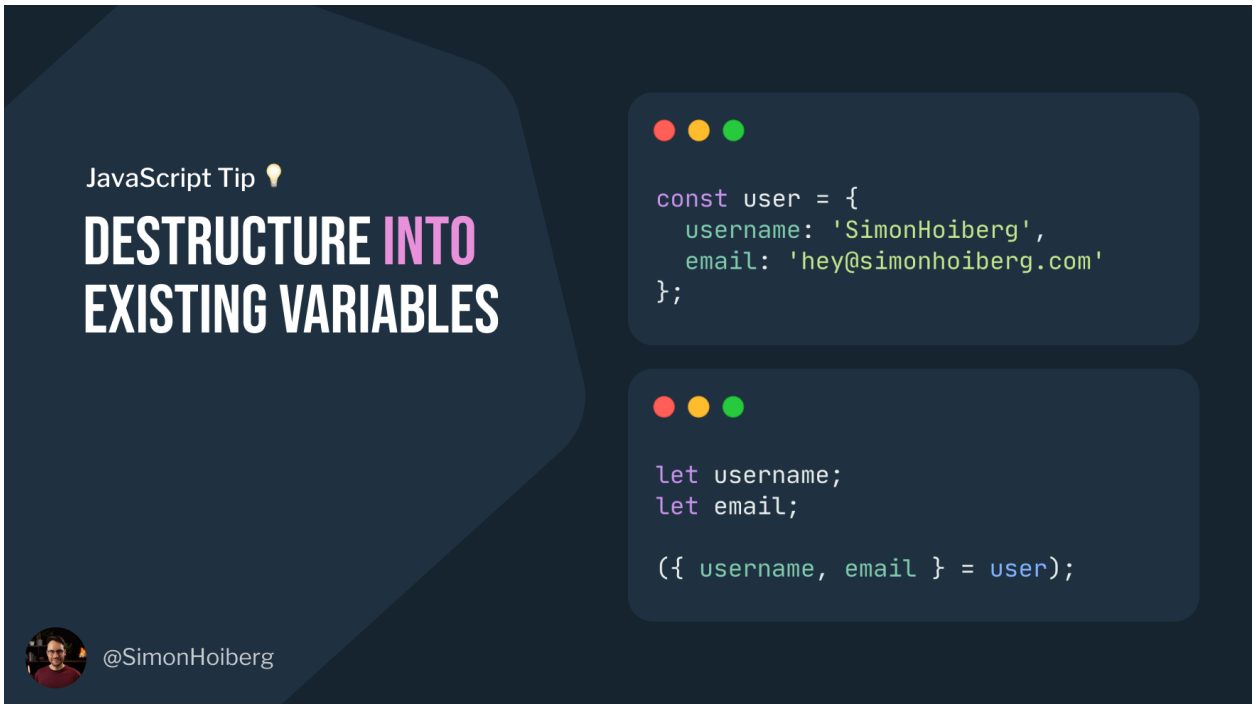
Explanation

Use the spread operator to create shallow copies of objects and arrays.

It's way cleaner than iterating and manually copying over.

The spread operator was a feature added as a part of ES6 and is supported by most major browsers.

Destructure into existing variables



Explanation

There's an easy way to destructure into existing variables.

Simply wrap the destructuring syntax in parentheses - this will result in the destructured variables being assigned as variables in the current scope.

Combining this with the use of the *let* keyword can be very useful in certain cases.

Lock an object using object freeze

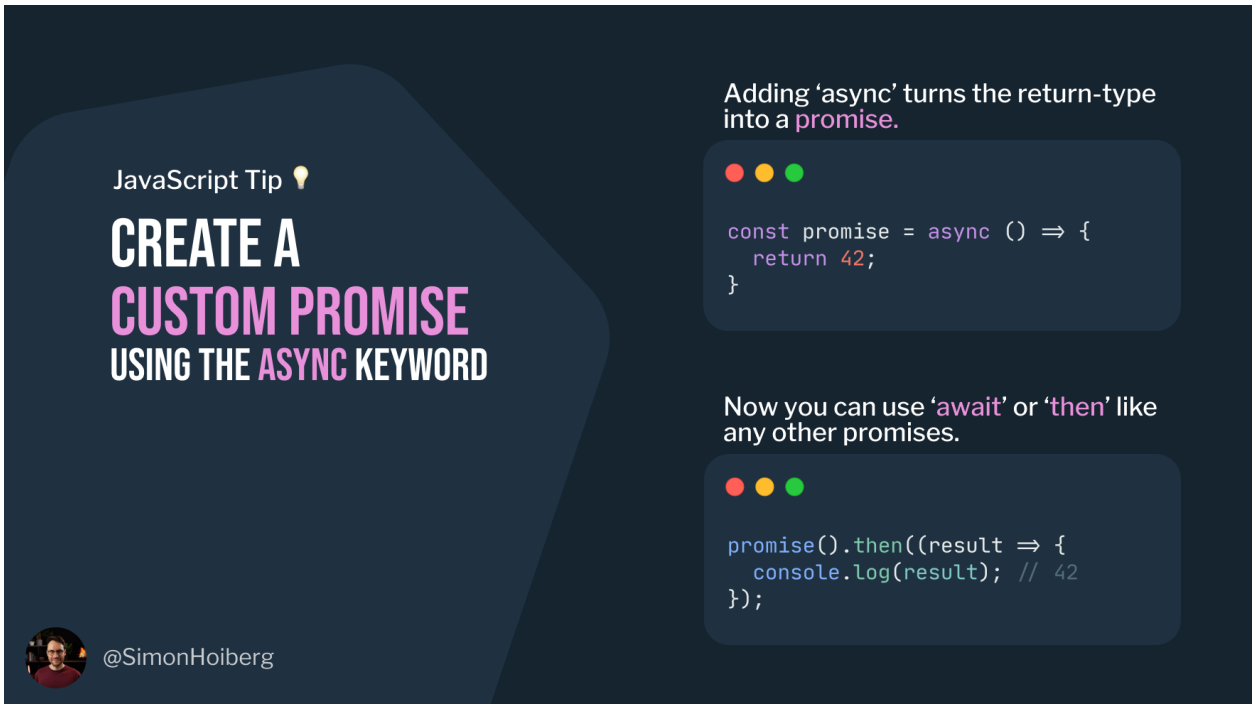
Explanation

The *Object.freeze* method freezes an object.

A frozen object can no longer be changed and will result in an error.

This means, that no properties can be reassigned or deleted from the object.

Create a custom promise using the async keyword



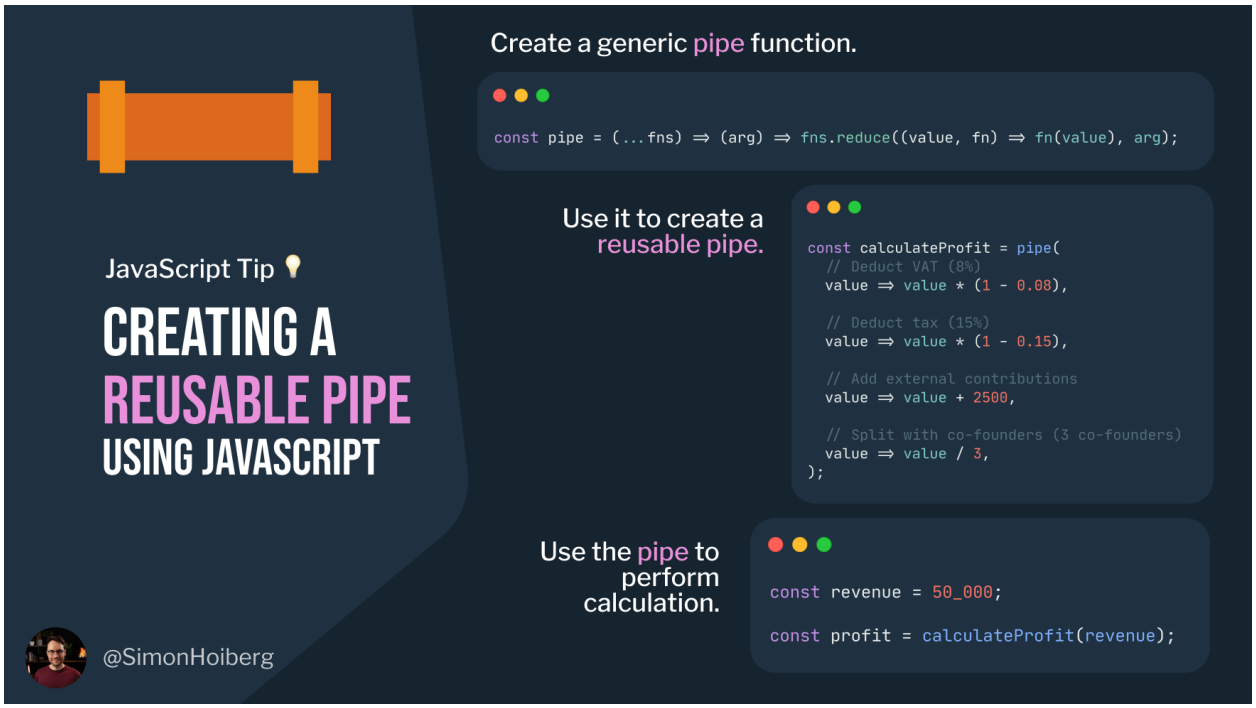
Explanation

You probably already knew that you have to put 'async' in front of the function signature in order to use 'await' inside of it.

But did you also know that 'async' turns the function into a promise?

When adding 'async' in from on the function signature, the return value automatically becomes 'awaitable' or 'thenable'.

Creating a reusable pipe



Explanation

This is an example of how you can create a reusable and composable pipe using JavaScript.

The data of the pipe flows left to right, calling each function with the output of the last one.

It's a great and clean way to simplify a flow of processing a value through multiple steps.

Stop using IIFEs

Explanation

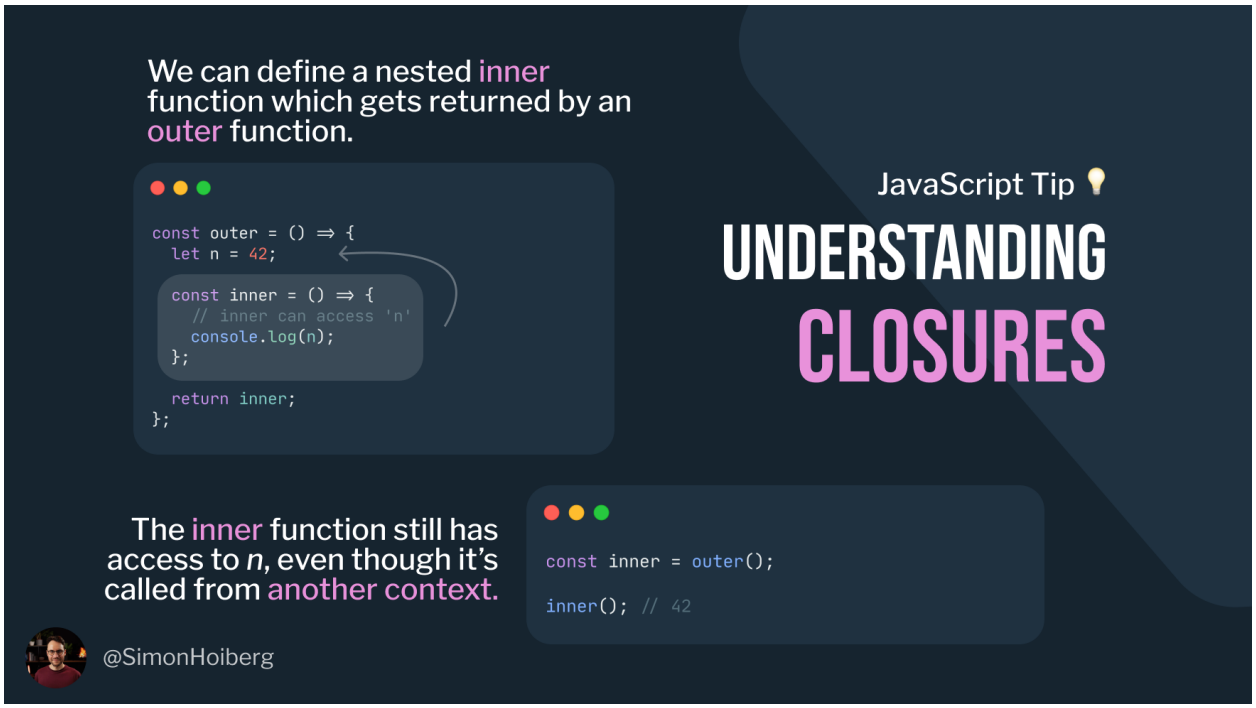
IIFEs died when modules were born.

Let them rest in peace.

You don’t need them (at least in 99% of the cases, you don’t).

In a case like this, just execute the function on a new line instead

Understanding Closures



Explanation

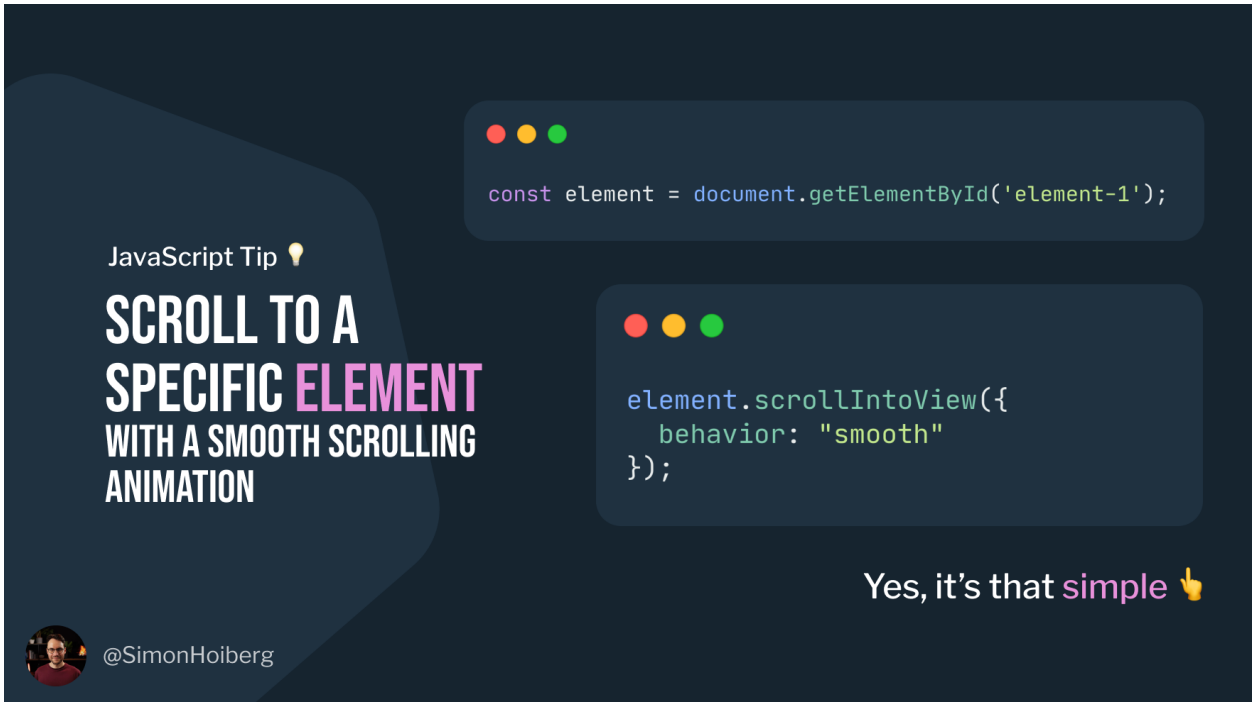
Closures is one of the fundamental building-blocks of JavaScript.

A closure gives a function access to an outer function's scope, even if the inner function is invoked from a completely different context.

This pattern is often used in combination with currying.

Scroll to a specific element

(with a smooth scrolling animation)

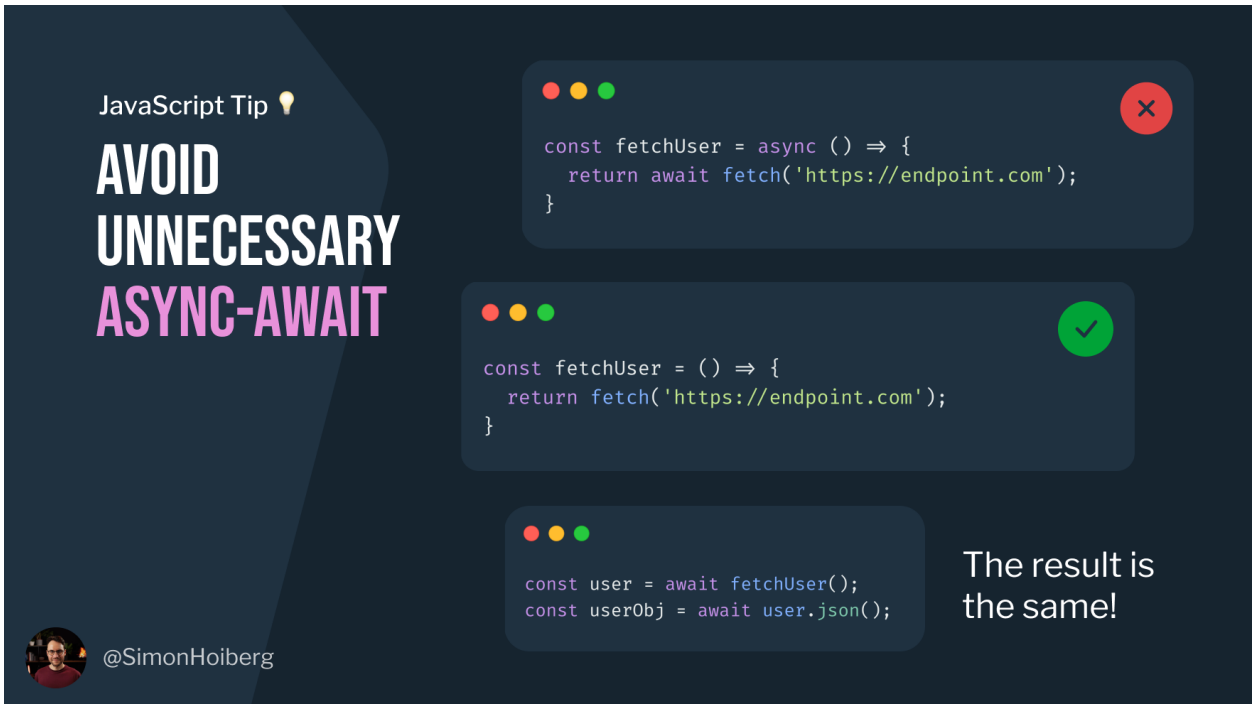


Explanation

Calling *Element.scrollIntoView* causes the elements parent container to scroll, such that the element becomes visible to the user.

It's a really easy way to invoke a smooth scrolling behavior.

Avoid unnecessary async-await



Explanation

Avoid unnecessary async-await.

If the function returns a Promise directly, there's no need to await it.

Pre- and post- hooks in package.json

Explanation

You can use pre- and post- hooks on all npm scripts.

Simply prepend "pre" or "post" to the name of your script

The result is, that these commands will be called prior to (pre) and after (post) the command that they are prepended to.

It’s a great way to avoid long and curly commands like

node prepare.js && webpack && node index.js

Quickly initialize using the -y option

Explanation

Did you know - if you add the option -y to npm init, NPM will create a starter setup for you without having to go through all the questions.

Use proper variable names - also in callbacks

Explanation

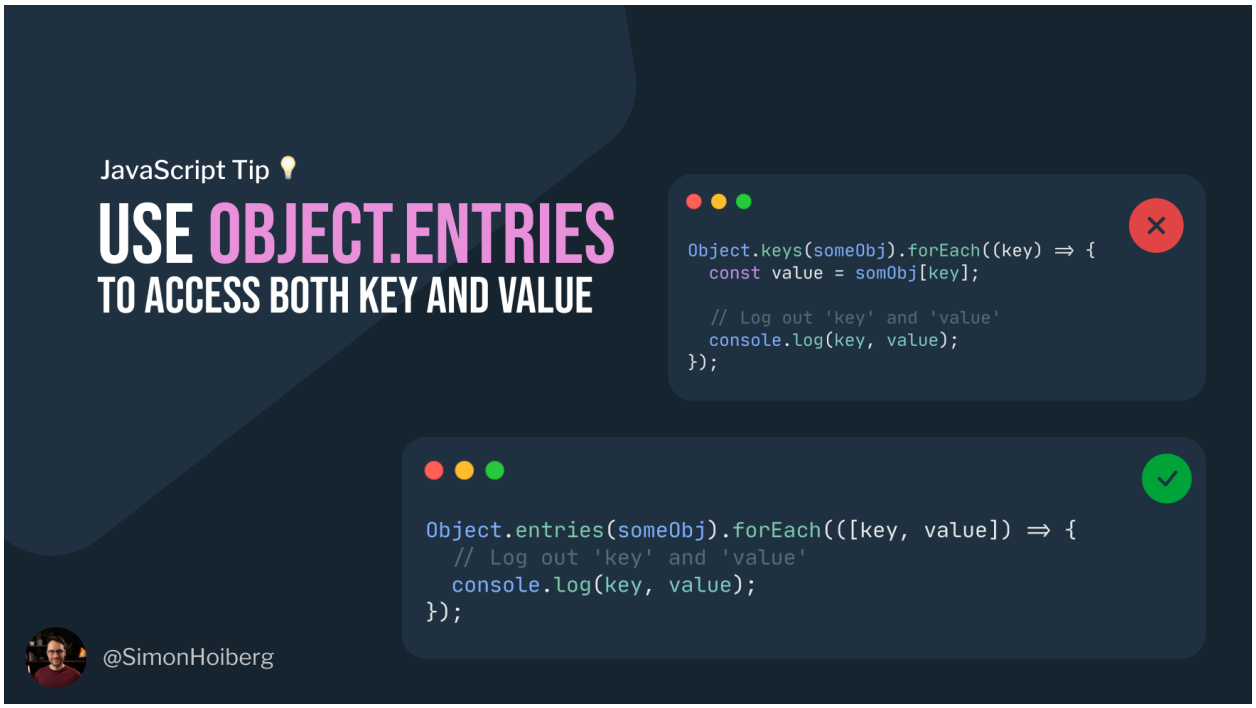
Use proper variable names - also in callbacks.

Short, concise code is not necessarily an ideal.

Clarity and readability is.

Paying with an extra line is perfectly ok.

Use object.entries to access both key and value

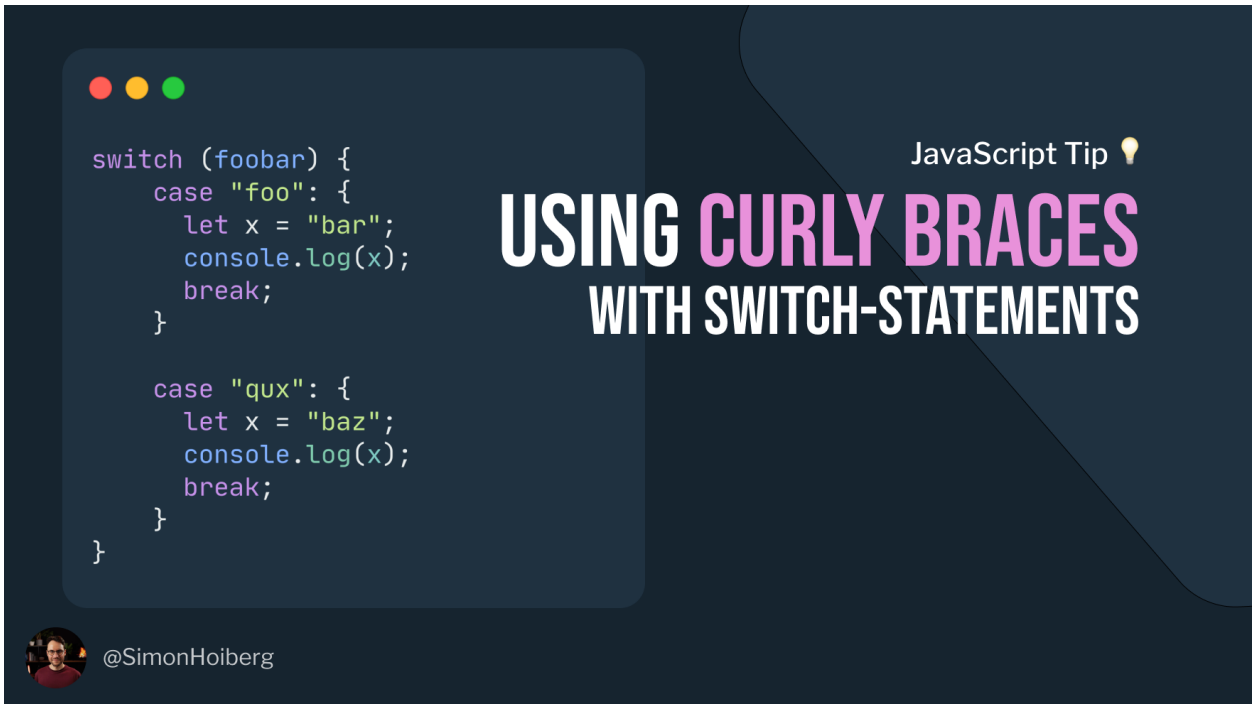


Explanation

You can use Object.entries() to iterate through the properties of an object and access both key and value.

No need to do an object lookup for each iteration.

Using curly braces with switch-statements



Explanation

Did you know that you can use curly braces with switch-statements?

Takeaways include:

- More readable
- Establishes their own block scope

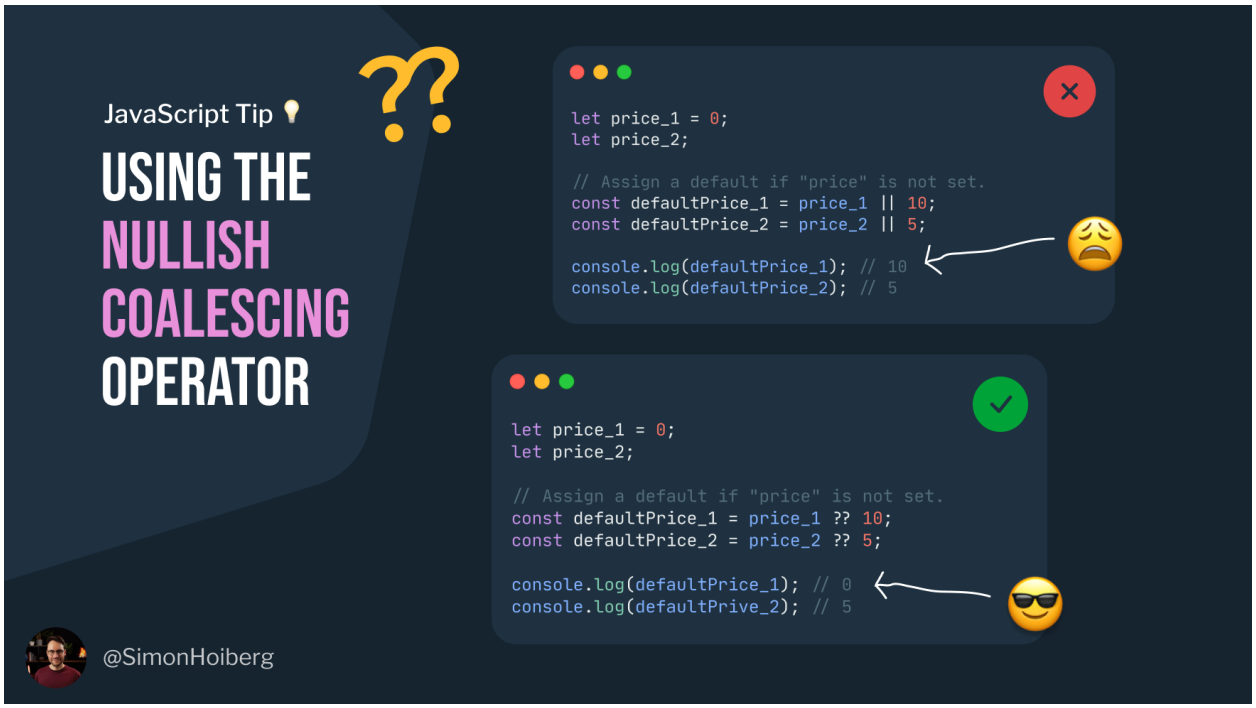
Create your own custom HTML Elements

Explanation

Did you know that you can create your own custom HTML Elements using JavaScript - and then use them in your HTML file just like any other element?

You can create some pretty powerful experiences using this technique.

Using the Nullish Coalescing Operator



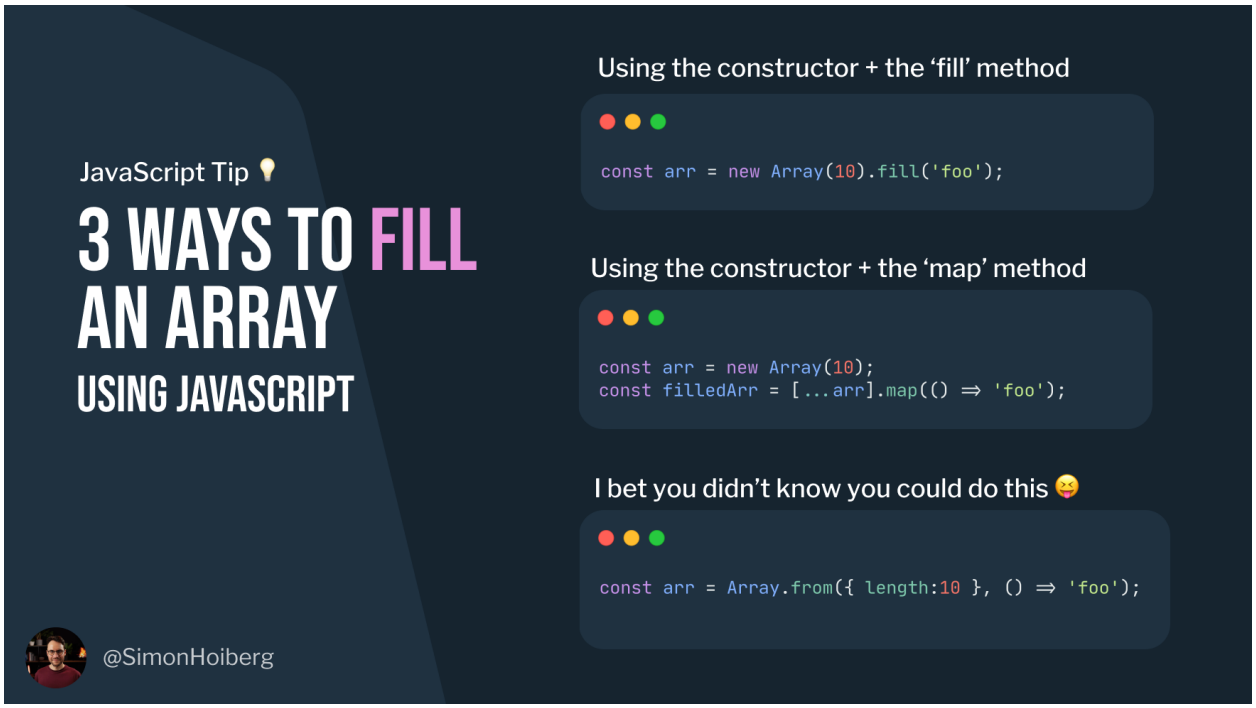
Explanation

The nullish coalescing operator will return its right-hand operand when the left side is *null* or *undefined*.

Not just falsy.

When working with numbers, this is typically very useful.

3 ways to fill an array



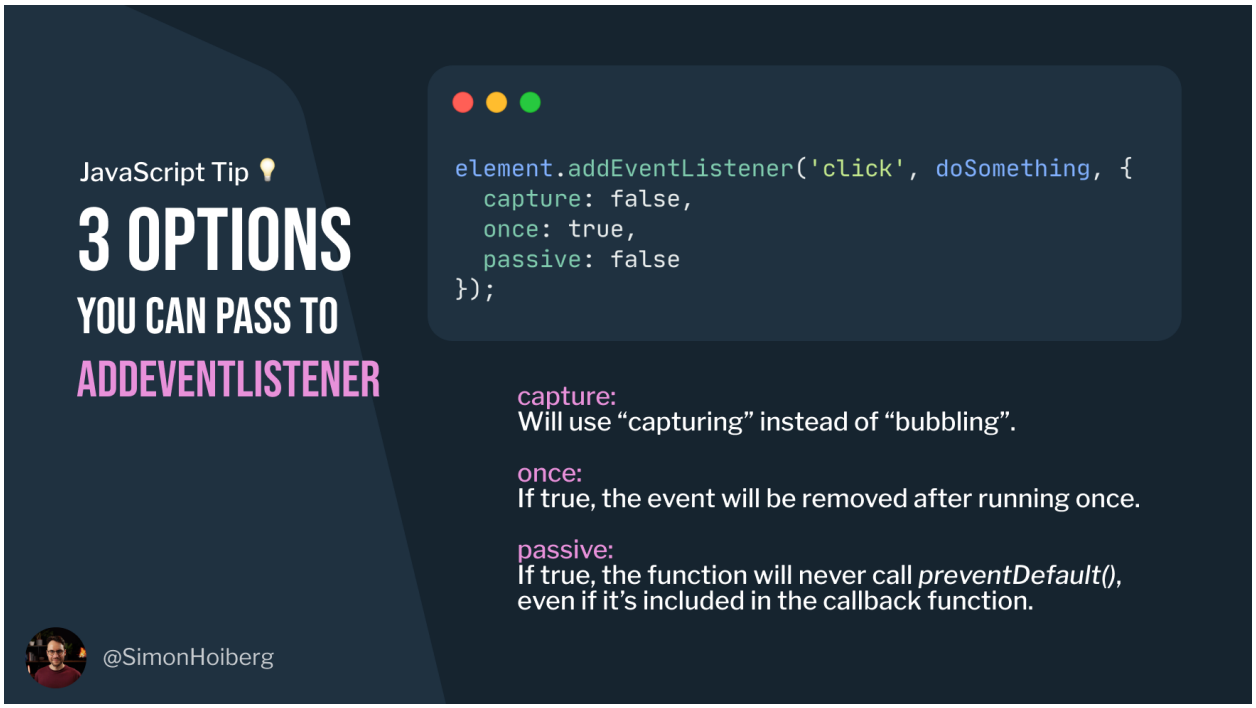
Explanation

These are 3 different ways to fill an array using JavaScript.

The most commonly used is version 1 - yet, version 2 can useful when filling the array best on business logic.

Version 3 is mostly for fun - but it can actually be done this way as well.

3 options you can pass to addEventListener



Explanation

capture:

Will use “capturing” instead of “bubbling”.

once:

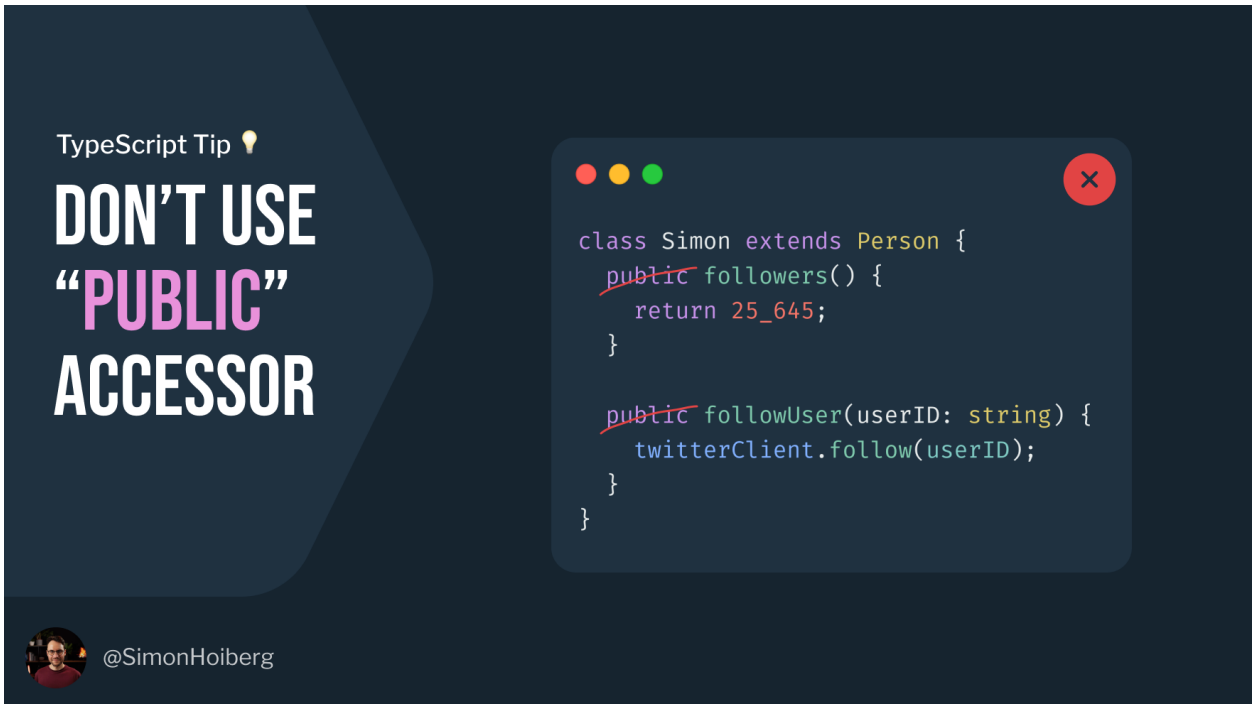
If true, the event will be removed after running once.

passive:

If true, the function will never call preventDefault(), even if it’s included in the callback function.

TypeScript

Don’t use “public” accessor



Explanation

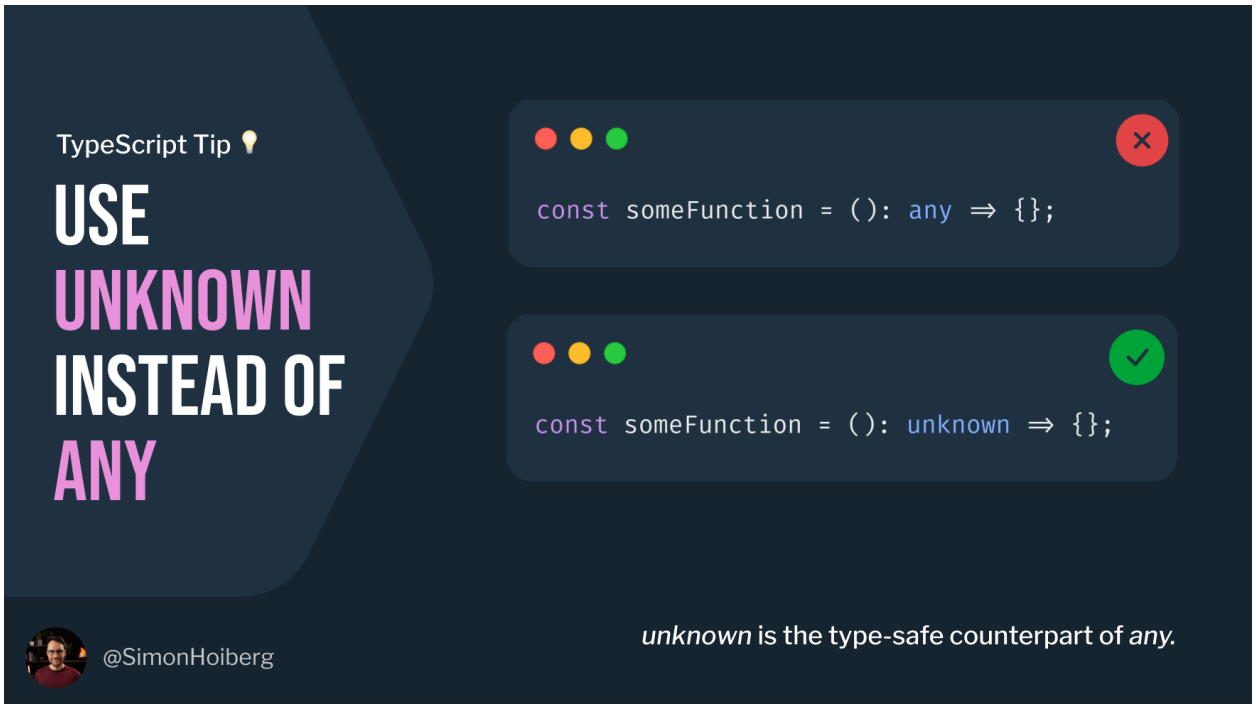
A method is public by default.

Don’t introduce unnecessary patterns to your codebase.

Also, the public accessor is not idiomatic JavaScript.

There’s no good reason to use it.

Use *unknown* instead of *any*



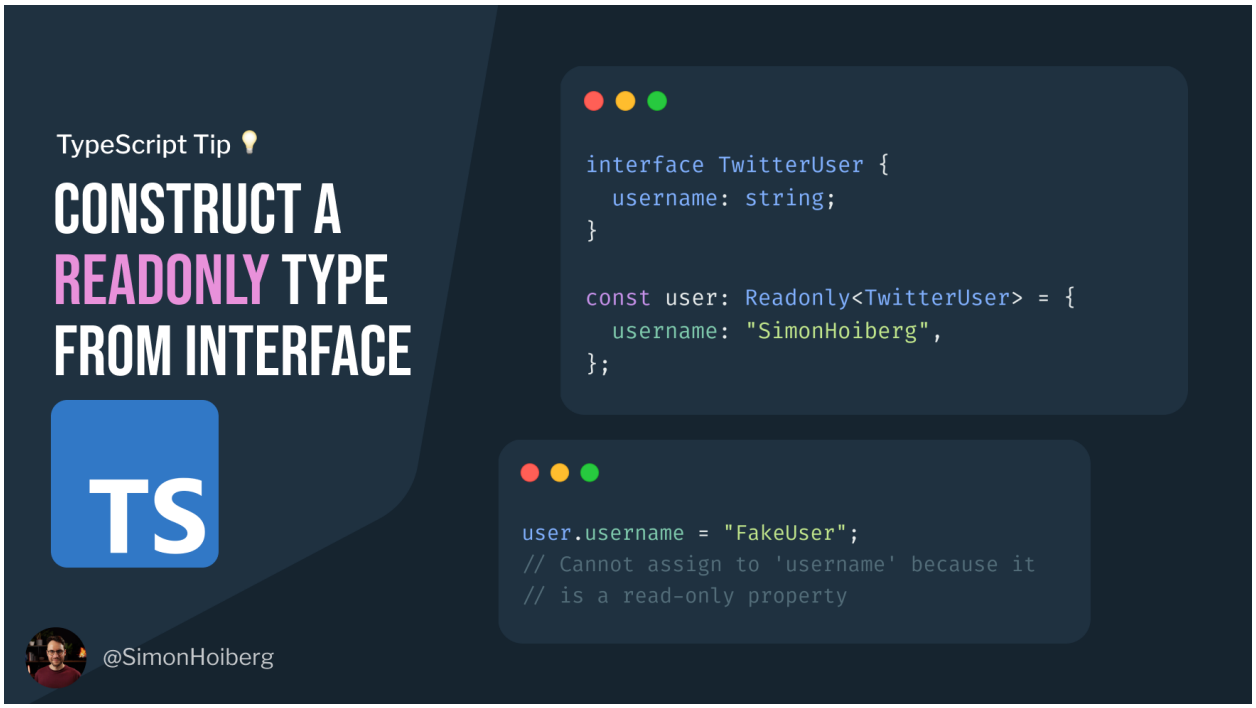
Explanation

If you cannot infer or define the type, use 'unknown' instead of 'any'.

Using the 'unknown' type forces you to null-check or narrowing the type before using it.

'unknown' is the type-safe counterpart of 'any'.

Construct a *readonly* type from interface

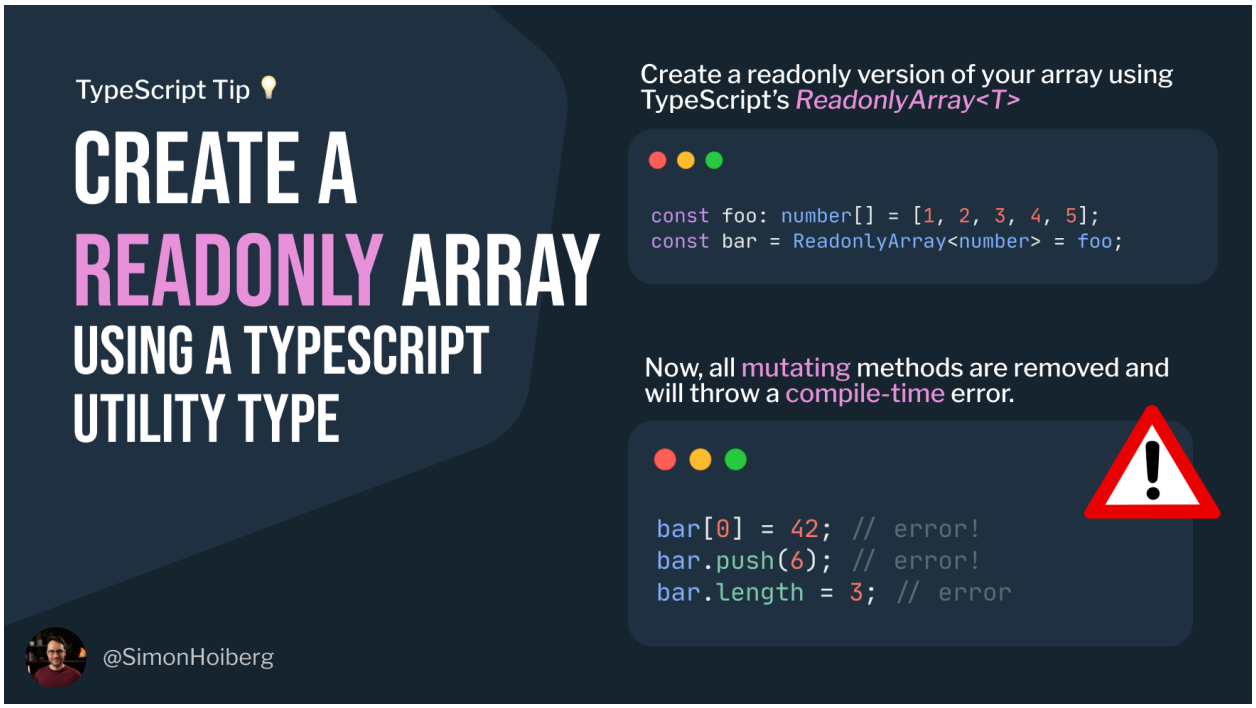


Explanation

You can use the TypeScript utility-type *Readonly* to construct a type with all properties set to readonly.

Trying to reassign a property of a *Readonly* will result in a compile-time error.

Create a readonly array using a typescript utility type



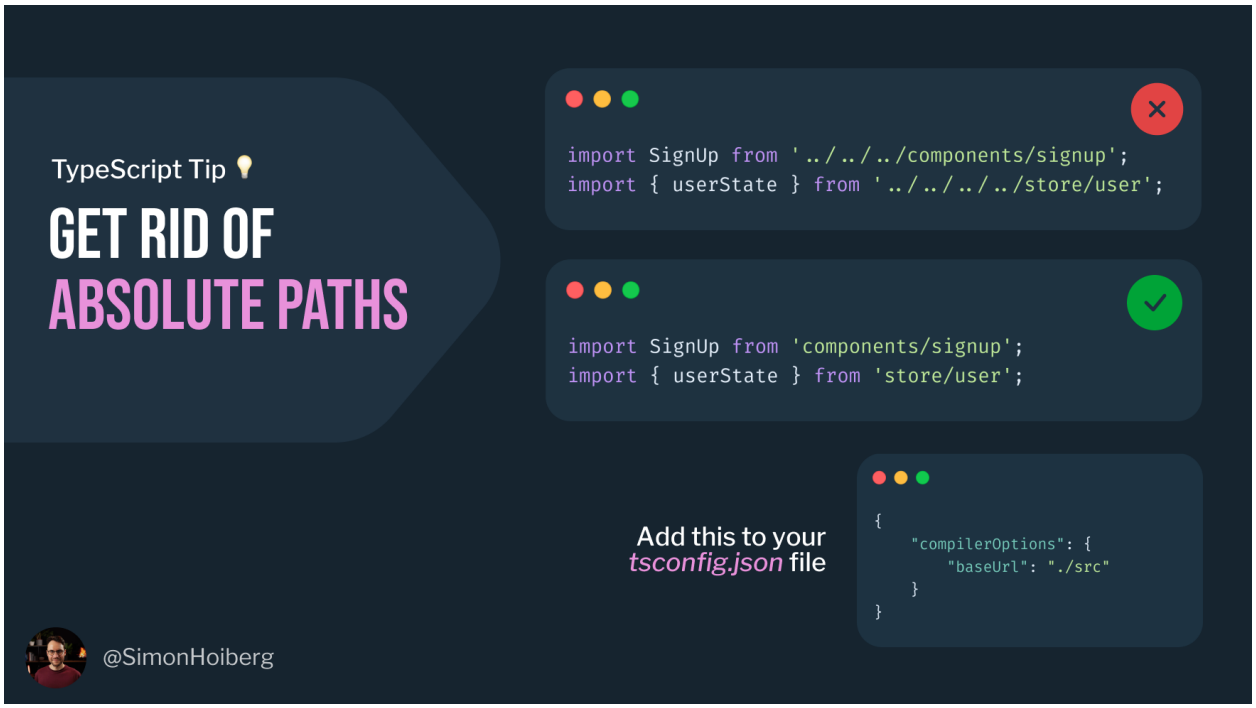
Explanation

TypeScript comes with a utility type, *ReadonlyArray<T>*.

It's equivalent to *Array<T>*, but with all mutating methods removed.

In this way, you can make sure you don't change your arrays after creation.

Get rid of absolute paths

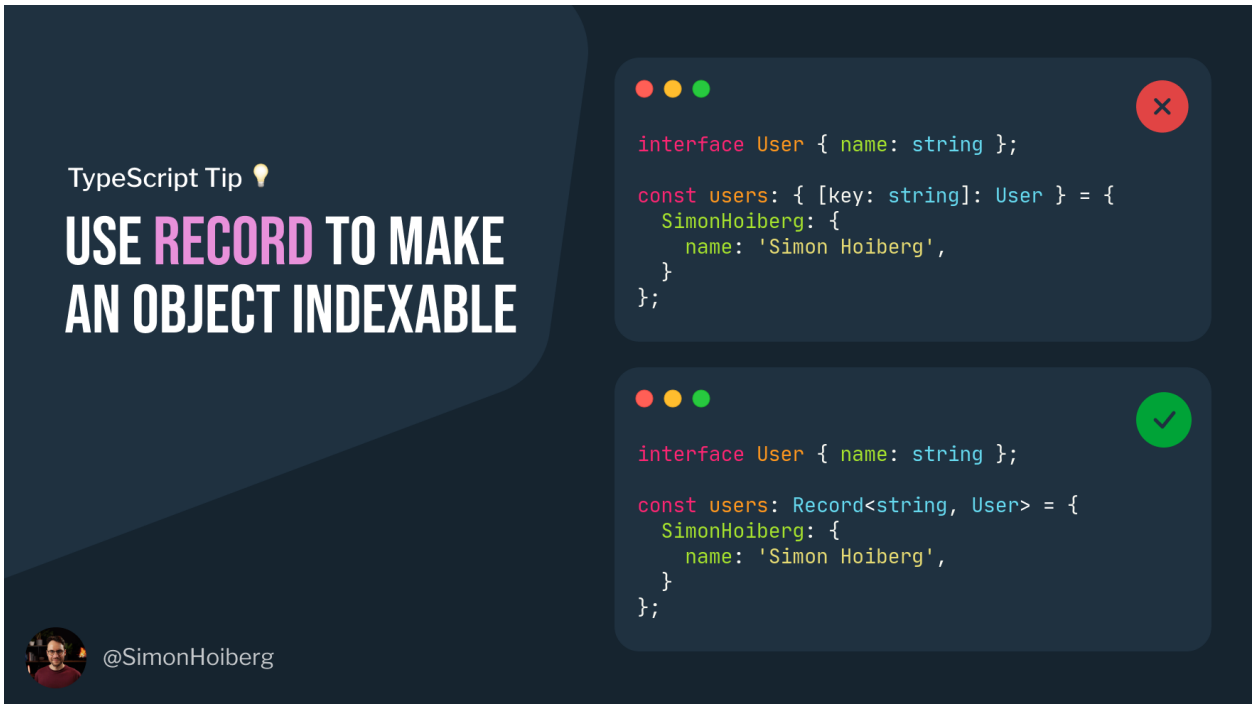


Explanation

By adding a “baseUrl” property in your *tsconfig.json*, you can specify a base directory to resolve non-absolute module names.

This is a great way to fix the leading “../../” on all your imports, and prevents you from having to change imports when you move files around.

Use record to make an object indexable

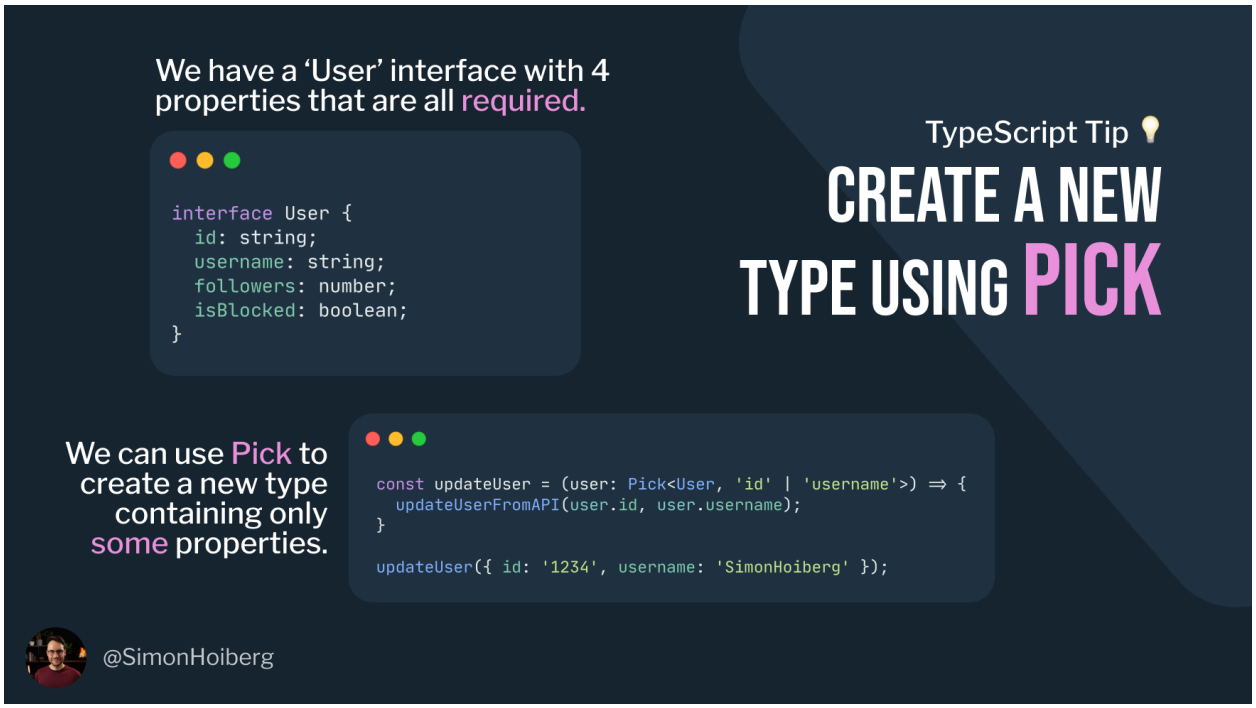


Explanation

Use the utility type 'Record' to make an object indexable, instead of typing it out manually.

It's more clean, and becomes handy if you want to map the properties of one type to another.

Create a new type using pick



Explanation

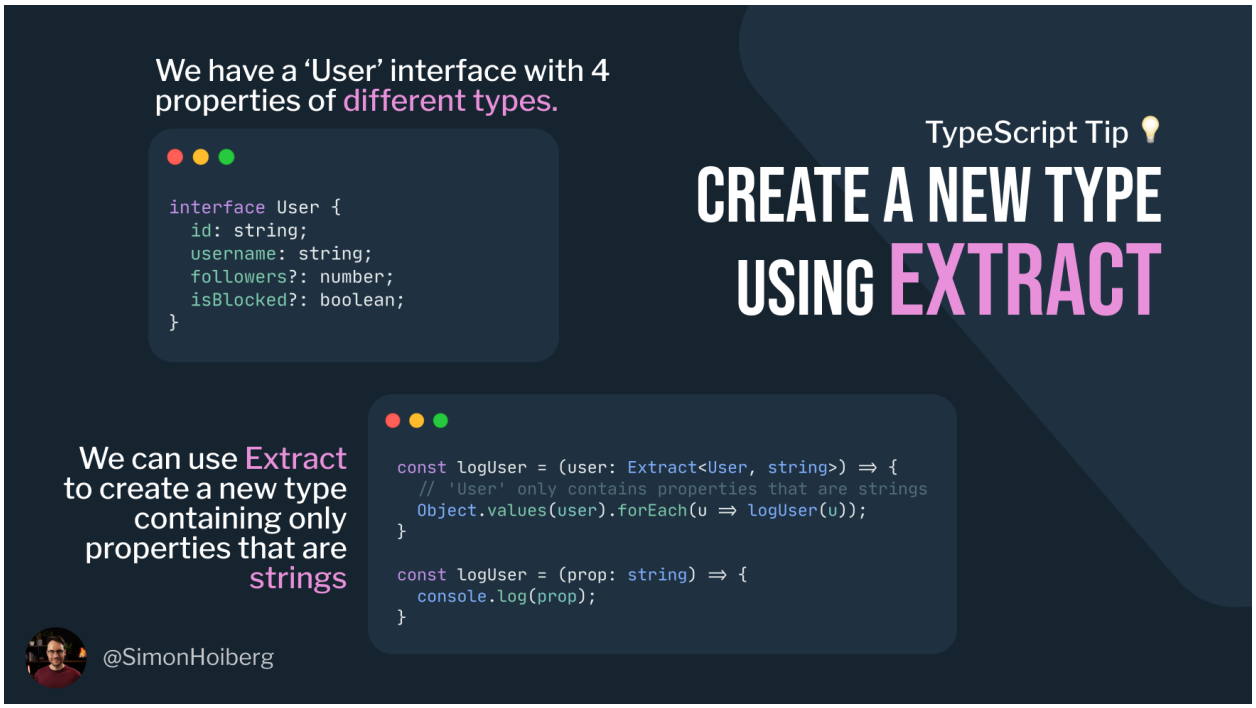
You can use *Pick* to quickly create new types.

And the cool thing is...

Pick is an inbuilt utility type.

We have that available natively in TypeScript.

Create a new type using extract



Explanation

TypeScript allows us a lot of flexibility when using their inbuilt Utility Types.

For instance, we can use *Extract* to create a new type containing only properties that are assignable to a given type.

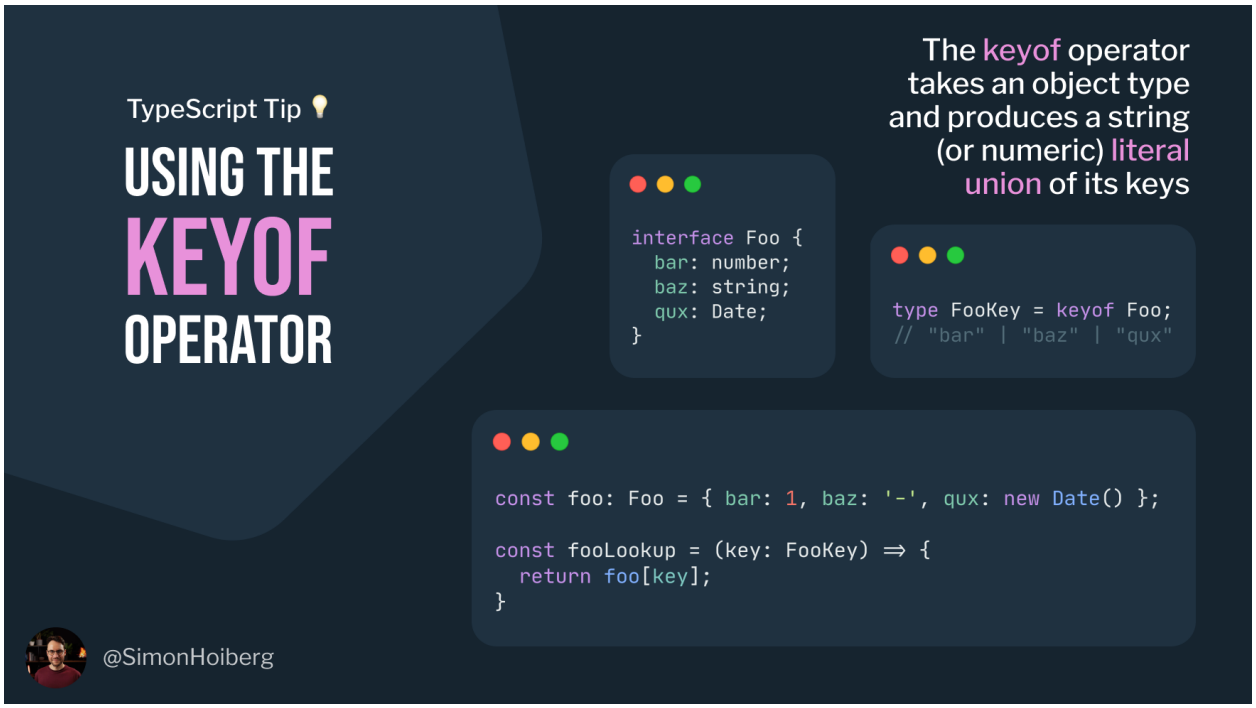
“Locking” types using const assertions

Explanation

Use const assertions to "lock down" a type to the most specific version possible.

- Prevent *Type Widening*
- Objects get *readonly* properties
- Arrays become *readonly* tuples

Using the keyof operator

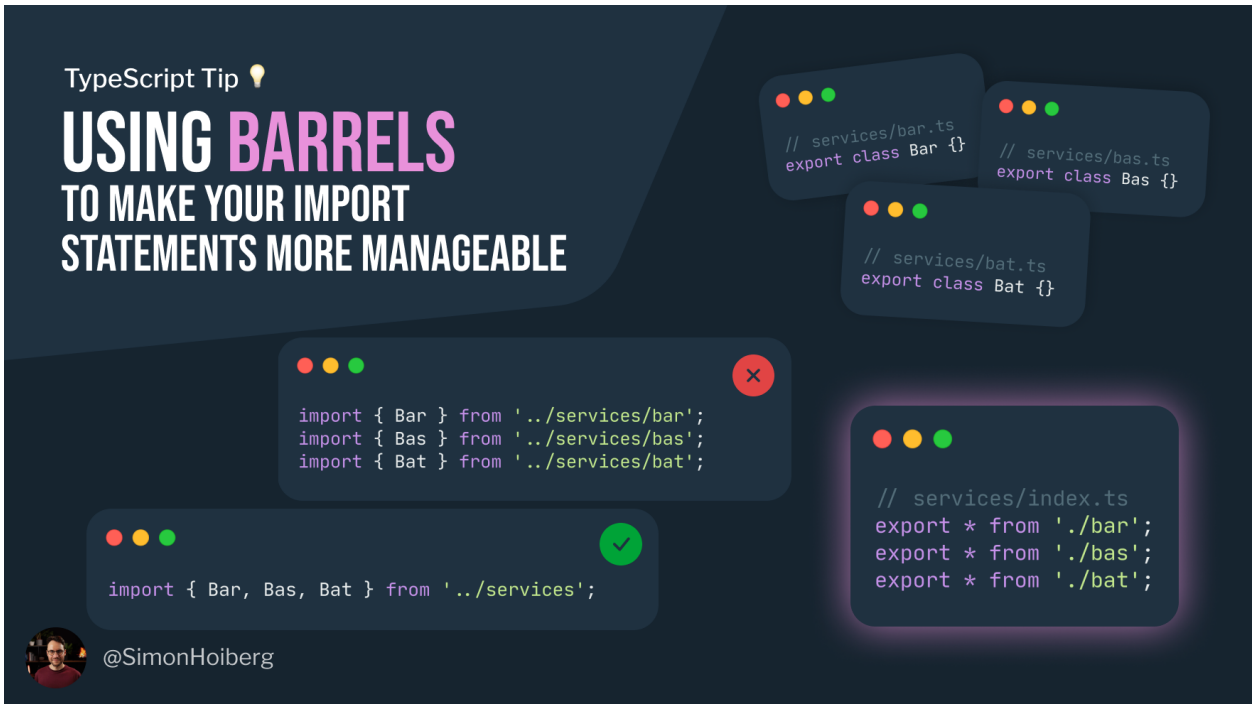


Explanation

You can use the *keyof* operator to produce a string literal union consisting of the keys of a specified object type.

The keyof operator takes an object type and produces a string (or numeric) literal union of its keys

Using barrels



Explanation

You can use barrels to make your import statements more manageable.

A barrel is a way to rollup exports from several modules into a single module.

It's a very convenient way to group modules together and import them all from one place.

NodeJS

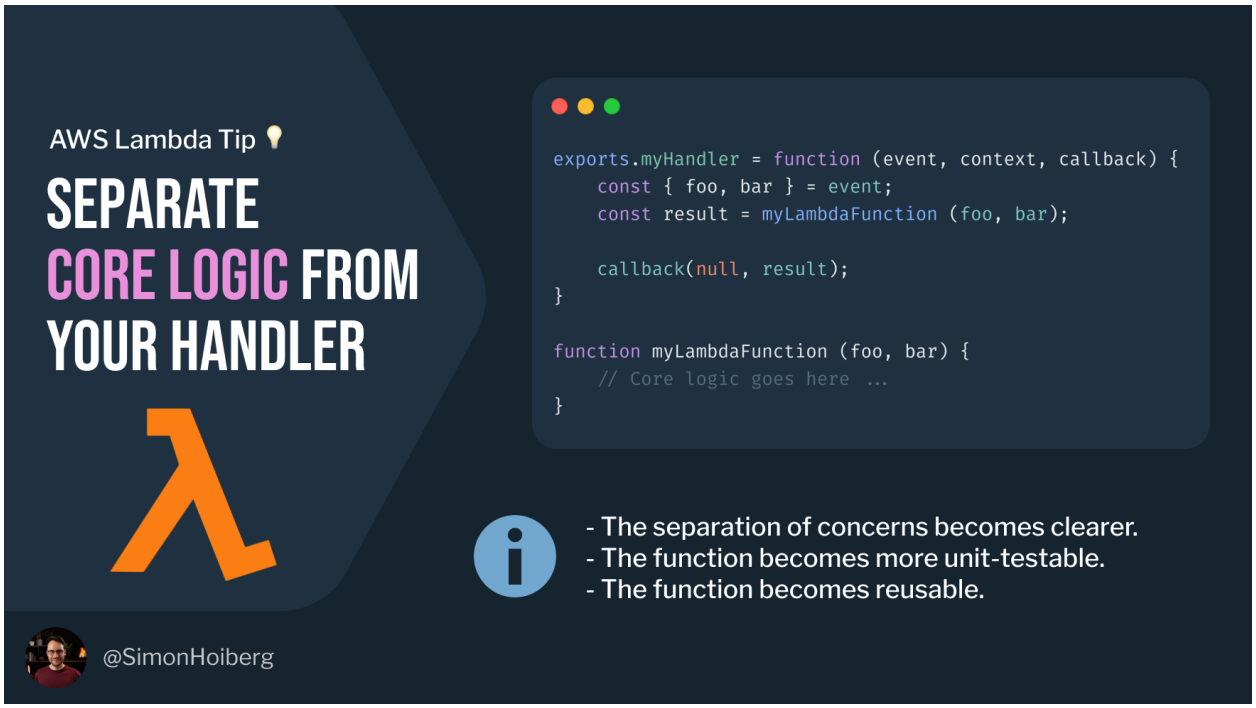
Import NodeJS builtins as promises

Explanation

Avoid callback hell while using NodeJS builtins.

You don't need promisify either - from NodeJS 10 and up, you can import the promisified versions directly from '[module]/promises'

AWS Lambda - separate core logic from your handler

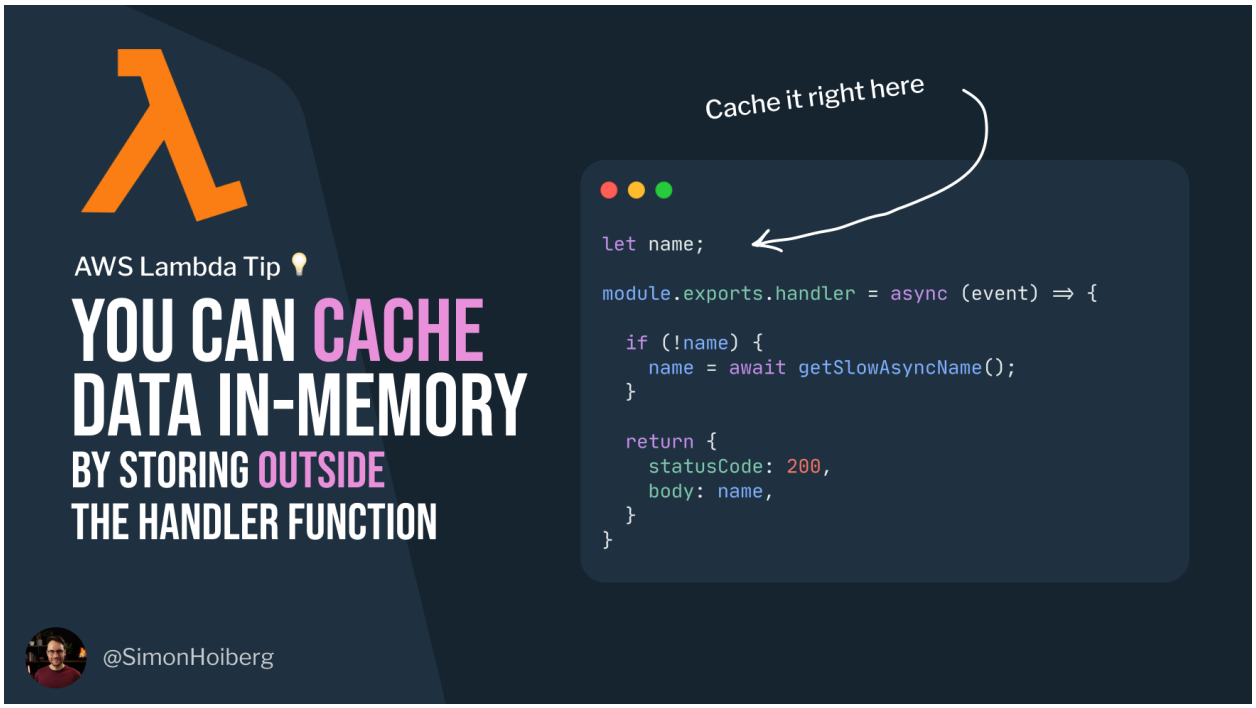


Explanation

Separate core logic from your handler.

- The separation of concerns becomes clearer.
- The function becomes more unit-testable.
- The function becomes reusable.

Cache data in-memory by storing outside the handler



Explanation

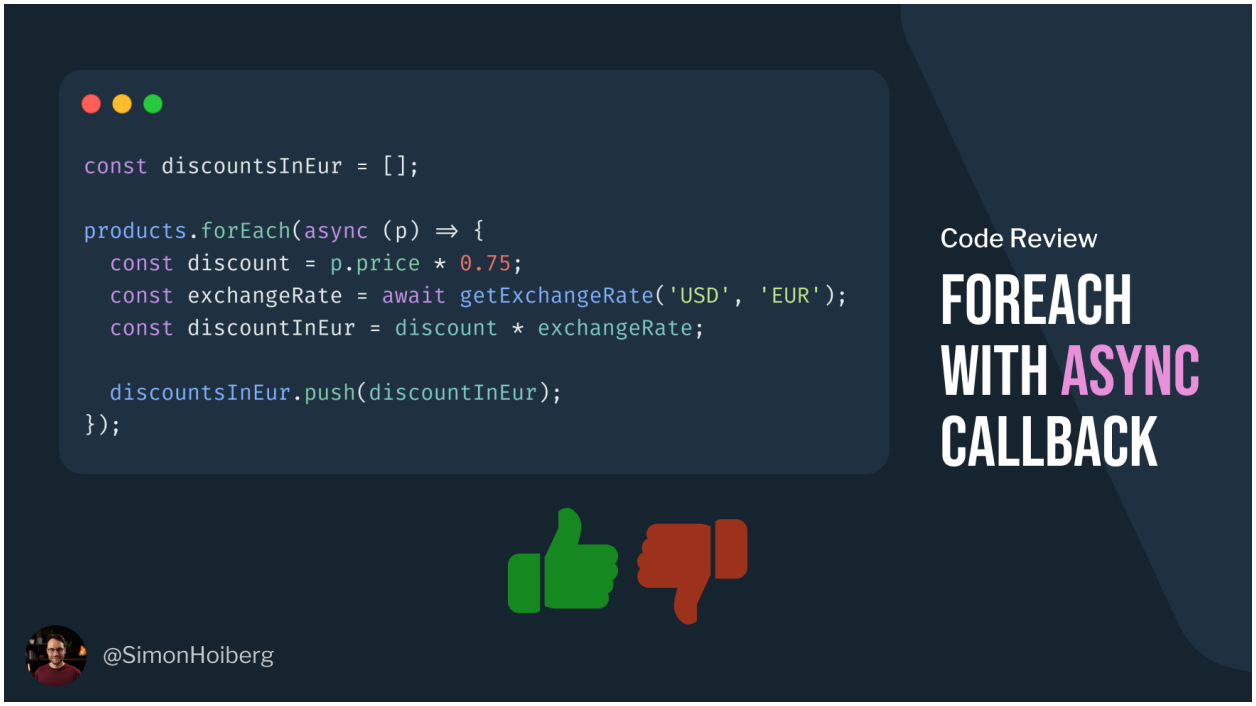
Did you know that Lambda Functions reuse the same NodeJS environment across invocations (as long as they stay awake)?

This allows you to cache data in-memory directly in the function.

Simply store it outside the handler

Code Review

Foreach with async callback?

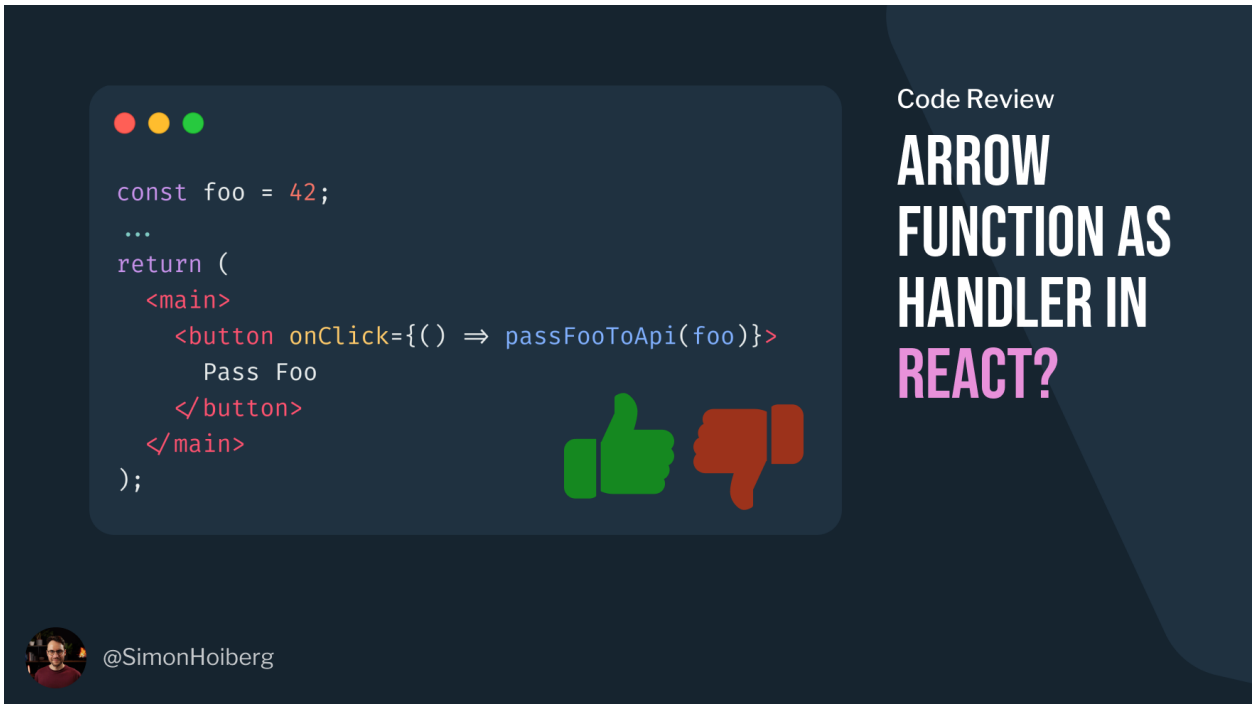


Result

Most people would reject this PR.

The most common reason is that the async call to *getExchangeRate* could have been done once, outside the callback function.

Arrow function as handler in react?



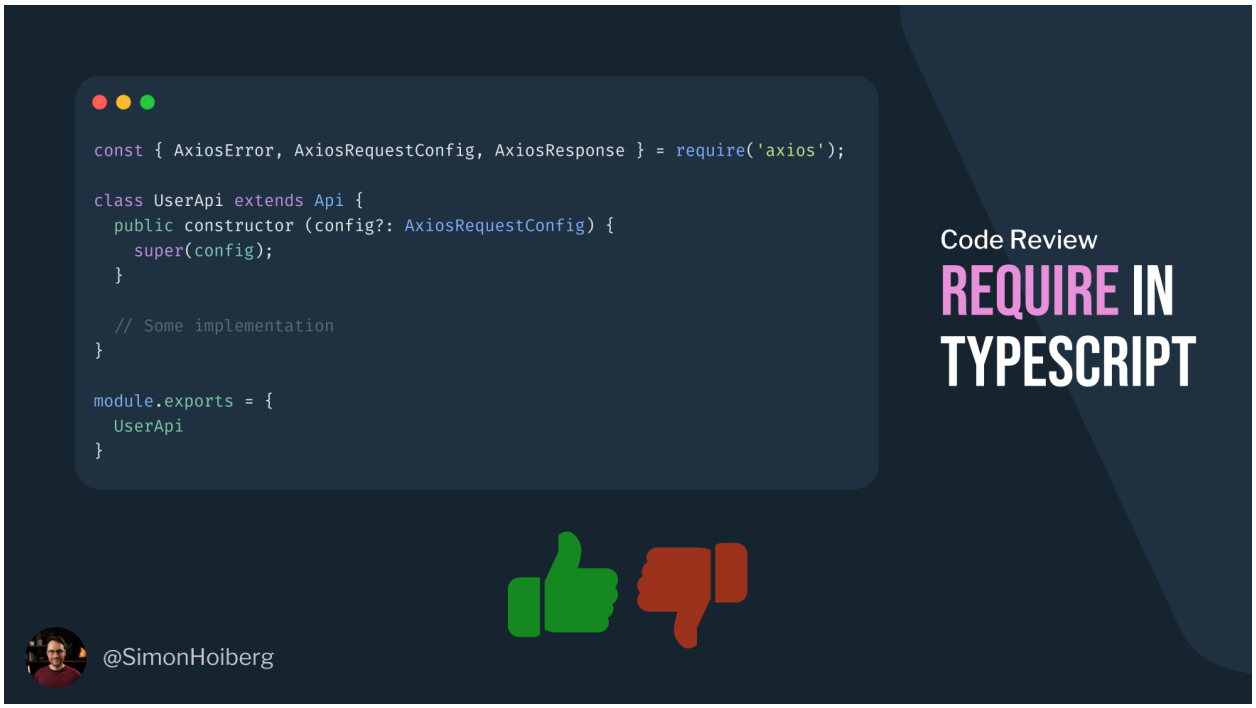
Result

Most people would approve this PR.

It's been long debated whether adding an inline arrow function as a handler in JSX causes significant performance issues.

Most people agreed, that this is not any longer an issue.
In fact, Dan Abramov himself joined this conversation on Twitter, stating that he would've approved this PR.

Require in typescript?

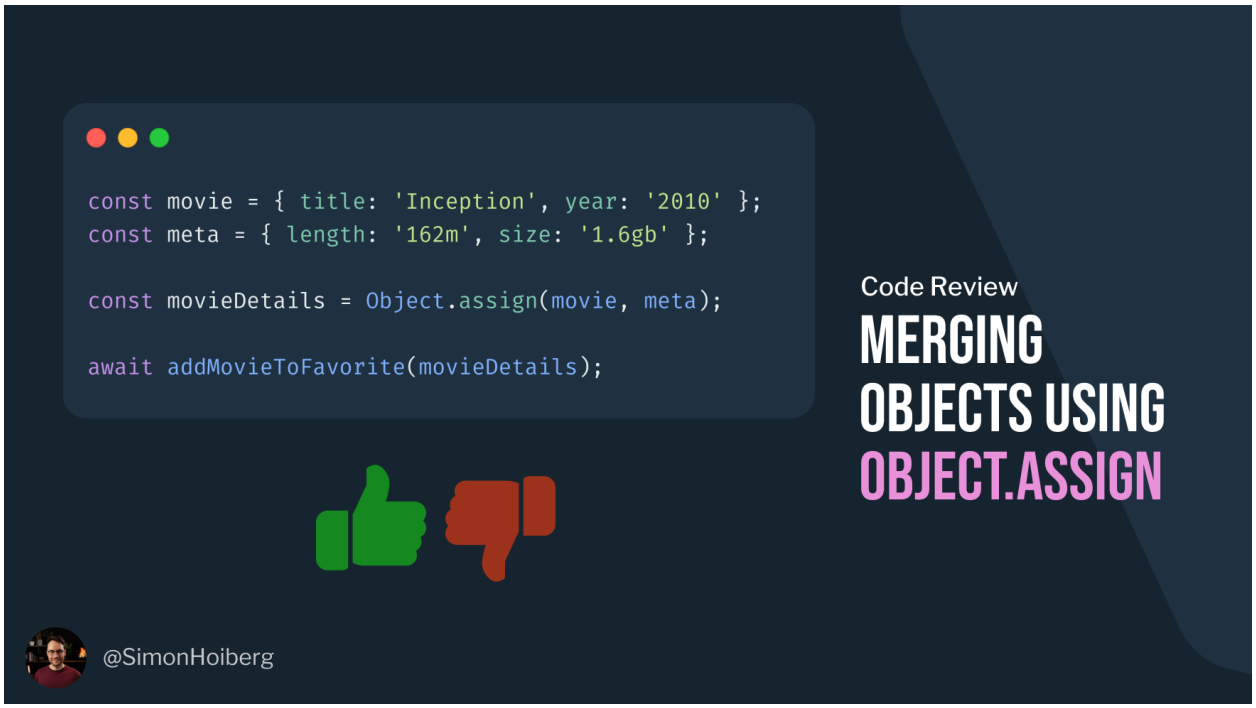


Result

Most people would reject this PR.

Most common reason is to use *import* instead of *require*.

Merging objects using object.assign?



Result

Most people would reject this PR.

Most common reason is to use the Spread Operator instead of Object.assign.

Especially, a lot pointed out, that as a bare minimum to get approved, an empty object should be passed as the first parameter to Object.assign to avoid mutating on the original object.

Nested if-statement?

Result

This one was a good mix, but most would approve this PR.

The people that would reject mostly wanted to introduce a guard clause to make an early return if *bar* is not set.

Other's argued, that a potentially falsy *bar* value should not be passed to the function - instead, it should be checked from the outside instead.