

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

# 65+ JAVASCRIPT CODE SNIPPETS WITH EXPLANATIONS

The image shows a dark blue background with several floating code snippets in white text, each enclosed in a semi-transparent rounded rectangle. Each snippet has three colored dots (red, yellow, green) above it. Some snippets have a red 'X' or a green checkmark in a circle to the right.

- // services/bas.ts  
export class Bas {}
- // services/bat.ts  
export class Bat {}
- const newObject = {};  
Object.keys(oldObject).forEach((key)  
 newObject[key] = oldObject[key];  
)
- const newArray = [];  
oldArray.forEach((element) => {  
 newArray.push(element);  
});
- // services/bar.ts  
export class Bar {}
- const newObject = { ...oldObject };  
const newArray = [ ...oldArray ];

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

<b>JavaScript</b>	<b>6</b>
Avoid the “delete” keyword	6
Using a Falsy Bouncer	7
Object Destructuring on arrays	8
Skip elements with Array Destructuring	9
Replacer function with JSON.stringify	10
Format the output of json.stringify	11
Using console.trace	12
Using console.time	13
Using console.group	14
Using console.assert	15
Pass arguments as an object	16
Readable numbers	17
Pass messages between tabs and windows	18
Remove duplicates from array	19
Use modules instead of classes	20
Debugger;	21
VSCode - Better Comments	22
VSCode - Import Cost	23
Analyze runtime performance using chrome devtools	24
Do not extend the built-ins	25
Use Array.some	26
React - Set state using currying	27
React - Let react assign a key to its children	28
React - Avoid provider wrapping hell	29
React - creating pure components with react.memo	30
Using optional chaining on functions	31
Enforce required arguments using default assignment	32
Adding a leading zero	33
Avoid default exports	34
Use spread to shallow copy objects and arrays	35

Destructure into existing variables	36
Lock an object using object freeze	37
Create a custom promise using the async keyword	38
Creating a reusable pipe	39
Stop using IIFEs	40
Understanding Closures	41
Scroll to a specific element (with a smooth scrolling animation)	42
Avoid unnecessary async-await	43
Pre- and post- hooks in package.json	44
Quickly initialize using the -y option	45
Use proper variable names - also in callbacks	46
Use object.entries to access both key and value	47
Using curly braces with switch-statements	48
Create your own custom HTML Elements	49
Using the Nullish Coalescing Operator	50
3 ways to fill an array	51
3 options you can pass to addEventListener	52
 <b>TypeScript</b>	53
Don't use "public" accessor	53
Use unknown instead of any	54
Construct a readonly type from interface	55
Create a readonly array using a typescript utility type	56
Get rid of absolute paths	57
Use record to make an object indexable	58
Create a new type using pick	59
Create a new type using extract	60
"Locking" types using const assertions	61
Using the keyof operator	62
Using barrels	63

<b>NodeJS</b>	<b>63</b>
Import NodeJS builtins as promises	64
AWS Lambda - separate core logic from your handler	65
Cache data in-memory by storing outside the handler	66
 <b>Code Review</b>	 <b>67</b>
Foreach with async callback?	67
Arrow function as handler in react?	68
Require in typescript?	69
Merging objects using object.assign?	70
Nested if-statement?	71

# JavaScript

## Avoid the “delete” keyword

JavaScript Tip 💡

## AVOID THE DELETE KEYWORD

Don't use 'delete' to remove a property from an object.

```
const simon = {  
    age: 32,  
    handle: 'SimonHoiberg',  
};  
  
delete simon.age;  
  
console.log(simon);  
// { handle: 'SimonHoiberg' }
```

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

```
const simon = {  
    age: 32,  
    handle: 'SimonHoiberg',  
};  
  
const { age, ...newSimon } = simon;  
  
console.log(newSimon);  
// { handle: 'SimonHoiberg' }
```

@SimonHoiberg

## 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

JavaScript Tip 🎉

## USING A FALSY BOUNCER

const evenNumbersSquared = [1, 2, 3, 4].map((n) => {  
 if (n % 2 === 0) {  
 return null;  
 }  
  
 return n \* n;  
}).filter(Boolean);

*falsy bouncer*

Filter all falsy values:  
[ null, 4, null, 16 ] → [ 4, 16 ]

 @SimonHoiberg

## 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



## OBJECT DESTRUCTURING ON ARRAYS

```
const countries = [  
  'Denmark',  
  'Switzerland',  
  'Norway',  
  'USA',  
  'UK'  
];  
  
const { 0: dk, 3: usa } = countries;  
  
console.log(dk) // Denmark  
console.log(usa) // USA
```



@SimonHoiberg

## 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

The image shows a dark-themed mobile application interface. On the left, there's a sidebar with a circular profile picture of a man and the handle '@SimonHoiberg'. The main content area has a title 'JavaScript Tip' with a lightbulb icon, followed by the text 'SKIP ELEMENTS WITH ARRAY DESTRUCTURING'. Below this is a code snippet in a code editor-like window:

```
const users = [
  'ravinwashere',
  'FrancescoCiulli4',
  'jackdomleo7',
  'dmokafa',
];

const [ , , ...restUsers ] = users;

console.log(restUsers);
// [ 'jackdomleo7', 'dmokafa' ]
```

## 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



JavaScript Tip 🌟

## REPLACER FUNCTION WITH JSON.STRINGIFY

```
const foo = {  
    bar: 42,  
    baz: 'qux',  
};  
  
const replacer = (key, value) => key === 'bar'  
    ? value * 2  
    : value;
```

```
const fooStr = JSON.stringify(foo, replacer);  
// {"bar":84,"baz":"qux"}
```

 @SimonHoiberg

## Explanation

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

JavaScript Tip 

## FORMAT THE OUTPUT OF JSON.STRINGIFY



@SimonHoiberg

### Classical use of `JSON.stringify()`

```
const someObject = {  
  name: 'SimonHoiberg',  
  age: 32,  
  online: true,  
};  
  
JSON.stringify(someObject);  
// {"name": "SimonHoiberg", "age": 32, "online": true}
```

Passing the number  
'2' as the third  
argument will format  
the output with 2  
spaces of  
indentation.

```
const someObject = {  
  name: 'SimonHoiberg',  
  age: 32,  
  online: true,  
};  
  
JSON.stringify(someObject, null, 2)  
// {  
//   "name": "SimonHoiberg",  
//   "age": 32,  
//   "online": true  
// }
```

## 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`

JavaScript Tip 💡

# CONSOLE TRACE

```
console.log("Simple log statement ...");
// Simple log statement ...
```

```
const someFunc = () => {
  console.trace("Log with stack trace");
}

// Log with stack trace
// someFunc      @ index.html:11
// (anonymous)   @ index.html:14
```

 @SimonHoiberg

### 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`

The screenshot shows a browser developer tools console window. On the left, there's a dark sidebar with a 'JavaScript Tip' icon (a lightbulb) and the text 'CONSOLE.TIME' in large white letters. On the right, the main console area has three colored status dots (red, yellow, green) at the top. Below them is the following code:

```
console.time('timer-1');

// time consuming operation
const items = [];
for (let i = 0; i < 1000000; i++) {
  items.push({ i });
}

console.timeEnd('timer-1');
```

At the bottom of the console, the output is shown in a box:

```
timer-1: 152.140869140625 ms
↳ undefined
↳ |
```

On the far left of the main window, there's a small circular profile picture of a man and the handle '@SimonHoiberg'.

## Explanation

You can set timers using `console.time`.

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

## Using `console.group`

The screenshot shows a browser developer tools console window. At the top, there are three colored status indicators: red, yellow, and green. Below them, the following JavaScript code is displayed:

```
console.group('User');
console.log(user.name);
console.log(access[user.name]);
console.groupEnd();
```

When this code is run, it creates a collapsed group named "User". By clicking the disclosure arrow next to "User", the following expanded output is shown:

▼ User
SimonHoiberg
admin

On the left side of the screenshot, there is a small circular profile picture of a man and the text "@SimonHoiberg".

### 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`

JavaScript Tip 🌟

### USE CONSOLE ASSERT TO MAKE CONDITIONAL LOG STATEMENTS

With a normal conditional statement

```
if (!user.id) {  
  console.log('User id missing');  
}
```

Using `console.assert`

```
console.assert(user.id, 'User id missing');
```

 @SimonHoiberg

## 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

JavaScript Tip 🌟

## PASS ARGUMENTS AS AN OBJECT

And make your teammates happy.

 @SimonHoiberg

```
const createUser = (username, date, isAdmin, isMod) => {
  // Create user
}

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

```
const createUser = ({ username, date, isAdmin, isMod }) => {
  // Create user
}

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

## 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

JavaScript Tip 💡

## READABLE NUMBERS

```
const largeNumber = 1000000000000;
```

```
const largeNumber = 1_000_000_000_000;
```

```
const largeNumber = 1e12;
```

@SimonHoiberg

## 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 interprets the number.

## Pass messages between tabs and windows

The diagram shows a central **BroadcastChannel** object at the top. Below it are two boxes: **Context 1 (window)** on the left and **Context 2 (tab)** on the right. An upward-pointing arrow from **Context 1** to the **BroadcastChannel** is labeled `postMessage()`. A downward-pointing arrow from the **BroadcastChannel** to **Context 2** is labeled `Message Event onmessage()`.

**JavaScript Tip**💡

## PASS MESSAGES BETWEEN TABS AND WINDOWS

```
const bc = new BroadcastChannel('test_channel');

bc.postMessage('This is a test message.');

bc.onmessage = function (ev) { console.log(ev); }
```

 @SimonHoiberg

### 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 references to frames or workers.

## Remove duplicates from array



JavaScript Tip 🎉

# REMOVE DUPLICATES FROM ARRAY

 @SimonHoiberg

```
new Set();
```

```
const numbers = [1, 2, 2, 4, 5, 6, 7, 7, 7, 8, 2, 1];
```

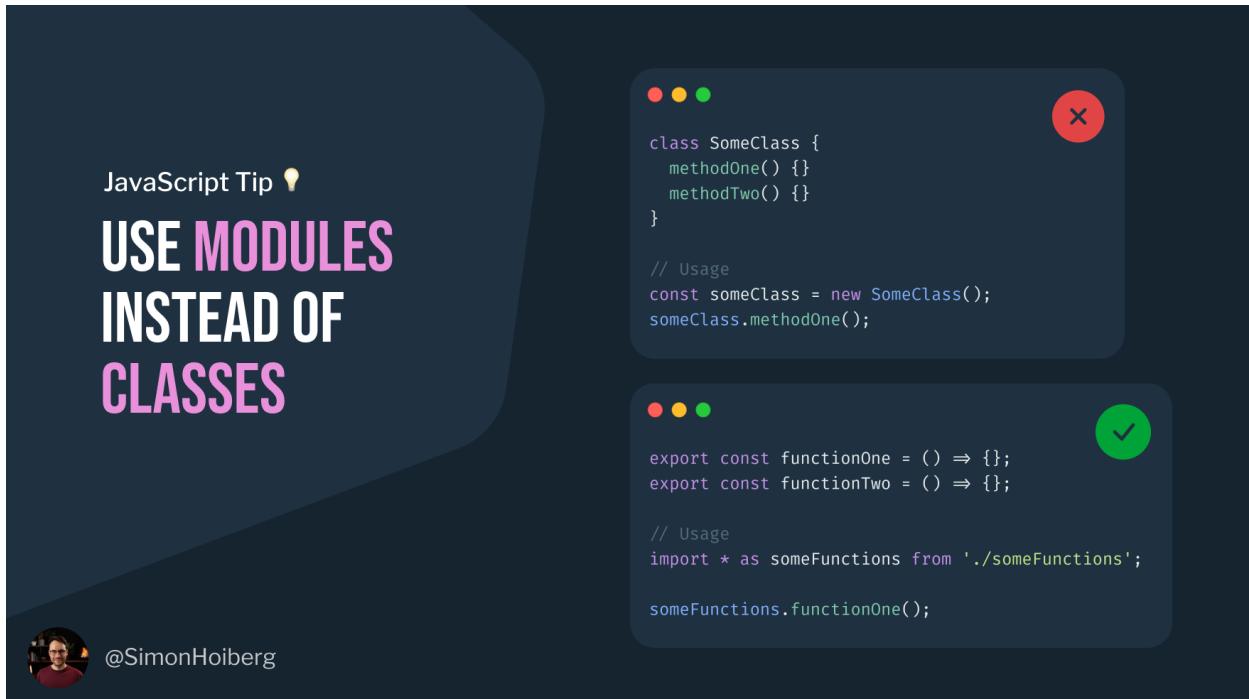
```
const uniqueNumbers = [... new Set(numbers)];  
// [ 1, 2, 4, 5, 6, 7, 8 ]
```

### 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



JavaScript Tip 💡

# USE MODULES INSTEAD OF CLASSES

 @SimonHoiberg

class SomeClass {  
 methodOne() {}  
 methodTwo() {}  
}  
  
// Usage  
const someClass = new SomeClass();  
someClass.methodOne();

export const functionOne = () => {};  
export const functionTwo = () => {};  
  
// Usage  
import \* as someFunctions from './someFunctions';  
  
someFunctions.functionOne();

### 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;

JavaScript Tip 🎉

# JAVASCRIPT DEBUGGER;

The browser will **stop** here,  
when executing.

● ● ●

```
if (somethingIsTrue) {  
  debugger;  
}
```

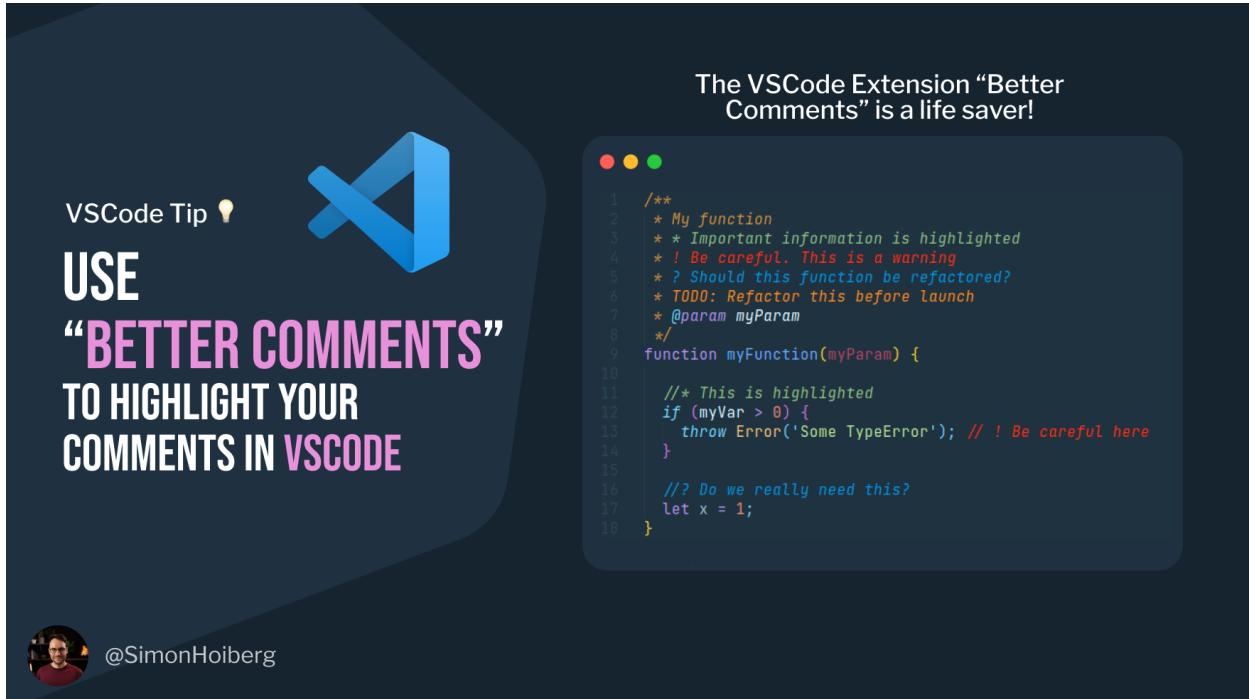
 @SimonHoiberg

## 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



VSCode Tip 💡

### USE “IMPORT COST” TO ANALYSE THE BUNDLE SIZE OF YOUR IMPORTS

The VSCode logo is shown in the top right corner.

This will help you identify potential issues with your bundle size.

```
import * as React from 'react';  8K (gzipped: 3.3K)
import { Provider } from 'react-redux';  12K (gzipped: 4.4K)
import {
  BrowserRouter,
  Switch,
  Route,
  Redirect
} from 'react-router-dom';  21.7K (gzipped: 6.4K)
```

 @SimonHoiberg

## 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.

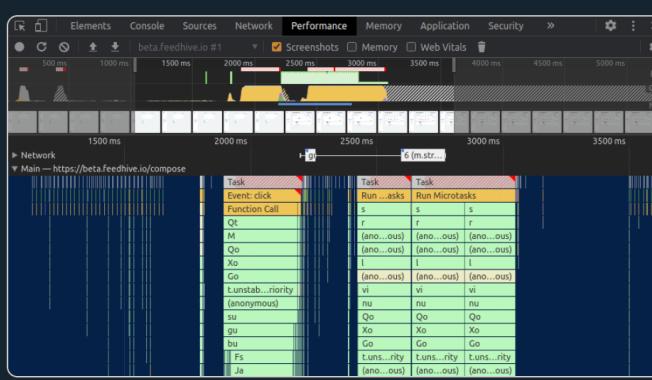
You can find "Import Cost" here:

<https://marketplace.visualstudio.com/items?itemName=wix.vscode-import-cost>

# Analyze runtime performance using chrome devtools

Development Tip💡

## ANALYZE RUNTIME PERFORMANCE USING CHROME DEVTOOLS



Open DevTools → Performance and start recording.

Now, you get a lot of valuable information that you can use to investigate performance bottlenecks!

 @SimonHoiberg

## 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

JavaScript Tip 🎉

### DO NOT EXTEND THE BUILT-INS

Create your own  
utilities instead.



@SimonHoiberg

```
// Custom average function
Array.prototype.average = function () {
  return this.reduce((acc, elem) => acc + elem) / this.length;
}

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

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

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

## 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

JavaScript Tip 💡

### USE ARRAY.SOME

to check for occurrences in a list

```
const hasActiveUsers = list.find(  
  (user) => user.isActive  
);  
  
console.log(Boolean(hasActiveUsers)); // true
```

const hasActiveUsers = list.some(  
 (user) => user.isActive  
);  
  
console.log(hasActiveUsers); // true

@SimonHoiberg

## 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

The image shows a presentation slide with a dark blue background. On the left, there's a large white rounded rectangle containing the title "SET STATE USING CURRYING" in bold black and pink letters. To the left of the title is a small circular profile picture of a man and the text "@SimonHoiberg". Above the title, it says "React Tip 🌟". On the right, there's a code editor window showing a snippet of React code. The code defines a function component "App" that uses the useState hook to manage "username" and "password" state. It then creates a "setInput" function using currying, which takes a "setter" function and returns a closure that updates the state when an input changes. Finally, it renders a form with two inputs: one for "username" and one for "password", both using the "setInput" function as their onChange handler.

```
const 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>
  );
}
```

## 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

React Tip 💡

## LET REACT ASSIGN A KEY TO ITS CHILDREN



@SimonHoiberg

Manually assigning a key

```
•••  
{someData.map(item => (  
  <div key={item.id}>Hello!</div>  
))}
```

Letting React handle it 😊

```
•••  
{React.Children.toArray(  
  someData.map(item => <div>Hello!</div>)  
)}
```

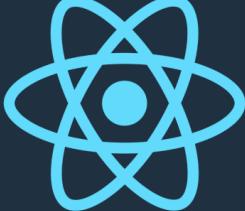
## Explanation

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

# React - Avoid provider wrapping hell

React Tip💡

## AVOID PROVIDER WRAPPING HELL IN REACT



 @SimonHoiberg

```
const combineProviders = (providers) => providers.reduce((Combined, Provider) => ({ children }) => (
  <Combined>
    <Provider>{children}</Provider>
  </Combined>
));

```

```
const Providers = combineProviders([
  BrowserRouter,
  RecoilRoot,
  ApolloProvider
]);

return (
  <Providers>
    <App />
  </Providers>
);

```

## 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`

React Tip💡

## CREATING PURE COMPONENTS WITH REACT.MEMO



 @SimonHoiberg

```
import React, { memo } from 'react';

const ComponentA = (props) => {
  return <div>Hi, {props.userName} 👋</div>
};

const ComponentB = memo((props) => {
  return <div>Hi, {props.userName} 👋</div>
});
```

ComponentB only re-renders when its props changes.  
Regardless of how many times its parent re-renders!

## 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

JavaScript Tip 💡

## USING OPTIONAL CHAINING ON FUNCTIONS

`// Using short-circuit  
someFunction && someFunction();`

`// Using if-statement  
if (someFunction) {  
 someFunction();  
}`

`// Using optional chaining  
someFunction?.();`

 @SimonHoiberg

## 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

JavaScript Tip 🎉

## ENFORCE REQUIRED ARGUMENTS USING DEFAULT ASSIGNMENT

Create this reusable function

```
constisRequired = () => {
  throw Error('Argument is missing');
}
```

Example usage

```
const setUsername = (username =isRequired()) => {
  // Do something with 'username'
  // If 'username' is not provided,
  // these lines of code will never be reached
}
```

@SimonHoiberg

## 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

Adding leading zeros to dates is often super tedious.

Here's a one-liner you can use.

```
const d = new Date();

const day = `0${d.getDate()}`.slice(-2);
// 04 or 13
```



@SimonHoiberg

JavaScript Tip 

# 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

JavaScript Tip 💡

## AVOID DEFAULT EXPORTS

Poor discoverability and autocompletion.

Horrible when using CommonJS.

TypeScript struggles with auto-import.

Re-exporting becomes tedious.

 @SimonHoiberg

✗

```
// Export
class Foo {}
export default Foo;

// Import
import Foo from './Foo';
```

✓

```
// Export
class Foo {}
export { Foo };

// Import
import { Foo } from './Foo';
```

## 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

JavaScript Tip 🎉

# USE THE SPREAD OPERATOR TO SHALLOW COPY OBJECTS AND ARRAYS

 @SimonHoiberg

```
const newObject = {};
Object.keys(oldObject).forEach((key) => {
  newObject[key] = oldObject[key];
});

const newArray = [];
oldArray.forEach((element) => {
  newArray.push(element);
});
```

```
const newObject = { ...oldObject };

const newArray = [ ...oldArray ];
```

### 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

JavaScript Tip !

## DESTRUCTURE INTO EXISTING VARIABLES



@SimonHoiberg

```
const user = {  
    username: 'SimonHoiberg',  
    email: 'hey@simonhoiberg.com'  
};
```

```
let username;  
let email;  
  
({ username, email } = user);
```

## 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

JavaScript Tip 💡

# LOCK AN OBJECT USING OBJECT FREEZE

```
const person = {  
    username: 'SimonHoiberg'  
};  
  
Object.freeze(person);
```

```
person.username = 'FakeUser';  
// Throws an error in strict mode
```



### 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

JavaScript Tip 🌟

## CREATE A CUSTOM PROMISE USING THE ASYNC KEYWORD



@SimonHoiberg

Adding ‘`async`’ turns the return-type into a `promise`.

```
const promise = async () => {
  return 42;
}
```

Now you can use ‘`await`’ or ‘`then`’ like any other promises.

```
promise().then((result => {
  console.log(result); // 42
});
```

## 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 front of the function signature, the return value automatically becomes ‘`awaitable`’ or ‘`thenable`’.

# Creating a reusable pipe



JavaScript Tip 🌟

## CREATING A REUSABLE PIPE USING JAVASCRIPT

Create a generic `pipe` function.

```
const pipe = (...fns) => (arg) => fns.reduce((value, fn) => fn(value), arg);
```

Use it to create a reusable pipe.

```
const calculateProfit = pipe(  
  // Deduct VAT (8%)  
  value => value * (1 - 0.08),  
  
  // Deduct tax (15%)  
  value => value * (1 - 0.15),  
  
  // Add external contributions  
  value => value + 2500,  
  
  // Split with co-founders (3 co-founders)  
  value => value / 3,  
);
```

Use the `pipe` to perform calculation.

```
const revenue = 50_000;  
  
const profit = calculateProfit(revenue);
```

## 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

JavaScript Tip 💡

# STOP USING IIFE<sup>s</sup>



 @SimonHoiberg

```
(async function doSomethingAsync() {  
  const foo = await bar();  
  const baz = foo.qux;  
  
  return baz;  
})();
```

```
async function doSomethingAsync() {  
  const foo = await bar();  
  const baz = foo.qux;  
  
  return baz;  
}  
  
doSomethingAsync();
```

## 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

We can define a nested **inner** function which gets returned by an **outer** function.

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

JavaScript Tip 

## UNDERSTANDING CLOSURES

The **inner** function still has access to **n**, even though it's called from **another context**.



@SimonHoiberg

```
const inner = outer();
inner(); // 42
```

## 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)

The image shows a dark-themed mobile application interface. On the left, there's a large, semi-transparent dark blue overlay containing white text: "JavaScript Tip 💡", "SCROLL TO A", "SPECIFIC ELEMENT", "WITH A SMOOTH SCROLLING", and "ANIMATION". In the top right corner, there's a code snippet in a light blue rounded rectangle with three colored dots (red, yellow, green) at the top: 

```
const element = document.getElementById('element-1');
```

 Below it is another code snippet in a similar light blue rounded rectangle with three colored dots: 

```
element.scrollIntoView({  
  behavior: "smooth"  
});
```

 At the bottom right, the text "Yes, it's that simple 🤘" is displayed in white. In the bottom left corner, there's a small circular profile picture of a man and the handle "@SimonHoiberg".

## Explanation

Calling `Element.scrollIntoView` causes the element's 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

JavaScript Tip 🌟

### AVOID UNNECESSARY ASYNC-AWAIT

`const fetchUser = async () => {  
 return await fetch('https://endpoint.com');  
}`

`const fetchUser = () => {  
 return fetch('https://endpoint.com');  
}`

The result is the same!

@SimonHoiberg

## 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

NPM Tip💡

# PRE- AND POST-HOOKS IN PACKAGE.JSON

```
...  
"scripts": {  
  "preinstall": "node prepare.js",  
  "postintall": "node clean.js",  
  "build": "webpack",  
  "postbuild": "node index.js",  
  "postversion": "npm publish",  
  "customscript": "node custom.js",  
  "precustomscript": "node custom.js -pre"  
}
```

 @SimonHoiberg

### 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

JavaScript Tip 💡

# USE PROPER VARIABLE NAMES ALSO IN CALLBACKS

 @SimonHoiberg

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

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

## 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



JavaScript Tip 🌟

# USE OBJECT.ENTRIES TO ACCESS BOTH KEY AND VALUE

```
Object.keys(someObj).forEach((key) => {
  const value = somObj[key];
  // Log out 'key' and 'value'
  console.log(key, value);
});
```

```
Object.entries(someObj).forEach(([key, value]) => {
  // Log out 'key' and 'value'
  console.log(key, value);
});
```

 @SimonHoiberg

## 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

switch (foobar) {  
 case "foo": {  
 let x = "bar";  
 console.log(x);  
 break;  
 }  
  
 case "qux": {  
 let x = "baz";  
 console.log(x);  
 break;  
 }  
}

JavaScript Tip 

## USING CURLY BRACES WITH SWITCH-STATEMENTS

 @SimonHoiberg

## 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

JavaScript Tip 🌟

## CREATE YOUR OWN CUSTOM HTML ELEMENTS



@SimonHoiberg

Define you Custom HTML Element

```
class FooElement extends HTMLElement {  
    connectedCallback() {  
        this.innerHTML = "This is a custom element";  
    }  
}  
  
customElements.define("foo-element", FooElement);
```

Use it like any other HTML Element

```
<body>  
    <foo-element />  
</body>
```

## 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

The screenshot shows a mobile application interface. On the left, there's a dark blue sidebar with a yellow question mark icon and the text "JavaScript Tip". Below that is a profile picture of a man and the handle "@SimonHoiberg". The main content area has two cards. The top card, marked with a red "X", contains code that uses the OR operator (||) to assign a default value of 10 to `defaultPrice\_1` if `price\_1` is not set. It logs 10 and 5 respectively. The bottom card, marked with a green checkmark, shows the same code but uses the nullish coalescing operator (??) instead. It logs 0 and 5 respectively. A hand-drawn style arrow points from the text "Not just falsy." to the second card.

```
let price_1 = 0;
let price_2;

// Assign a default if "price" is not set.
const defaultPrice_1 = price_1 || 10;
const defaultPrice_2 = price_2 || 5;

console.log(defaultPrice_1); // 10
console.log(defaultPrice_2); // 5
```

```
let price_1 = 0;
let price_2;

// Assign a default if "price" is not set.
const defaultPrice_1 = price_1 ?? 10;
const defaultPrice_2 = price_2 ?? 5;

console.log(defaultPrice_1); // 0
console.log(defaultPrice_2); // 5
```

## 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

JavaScript Tip 💡

# 3 WAYS TO FILL AN ARRAY USING JAVASCRIPT

 @SimonHoiberg

Using the constructor + the 'fill' method

```
const arr = new Array(10).fill('foo');
```

Using the constructor + the 'map' method

```
const arr = new Array(10);
const filledArr = [...arr].map(() => 'foo');
```

I bet you didn't know you could do this 😊

```
const arr = Array.from({ length:10 }, () => 'foo');
```

## Explanation

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

The most commonly used is version 1 - yet, version 2 can be 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



JavaScript Tip 🌟

# 3 OPTIONS YOU CAN PASS TO ADDEVENTLISTENER

```
element.addEventListener('click', doSomething, {  
  capture: false,  
  once: true,  
  passive: false  
});
```

**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.

 @SimonHoiberg

### 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

TypeScript Tip 🎉

## DON'T USE “PUBLIC” ACESSOR

```
class Simon extends Person {  
    public followers() {  
        return 25_645;  
    }  
  
    public followUser(userID: string) {  
        twitterClient.follow(userID);  
    }  
}
```

@SimonHoiberg

## 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`

TypeScript Tip 💡

# USE UNKNOWN INSTEAD OF ANY

`const someFunction = (): any => {};`

`const someFunction = (): unknown => {};`

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

@SimonHoiberg

### 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

TypeScript Tip 💡

## CONSTRUCT A READONLY TYPE FROM INTERFACE

```
interface TwitterUser {  
    username: string;  
}  
  
const user: Readonly<TwitterUser> = {  
    username: "SimonHoiberg",  
};
```

```
user.username = "FakeUser";  
// Cannot assign to 'username' because it  
// is a read-only property
```

 @SimonHoiberg

## 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

TypeScript Tip !

# CREATE A READONLY ARRAY USING A TYPESCRIPT UTILITY TYPE

 @SimonHoiberg

Create a readonly version of your array using TypeScript's `ReadonlyArray<T>`

● ● ●

```
const foo: number[] = [1, 2, 3, 4, 5];
const bar = ReadonlyArray<number> = foo;
```

Now, all mutating methods are removed and will throw a compile-time error.

● ● ●

```
bar[0] = 42; // error!
bar.push(6); // error!
bar.length = 3; // error
```



### 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

TypeScript Tip 💡

## GET RID OF ABSOLUTE PATHS

`import SignUp from '../../../../../components/signup';  
import { userState } from '../../../../../store/user';`

`import SignUp from 'components/signup';  
import { userState } from 'store/user';`

Add this to your `tsconfig.json` file

```
{  
  "compilerOptions": {  
    "baseUrl": "./src"  
  }  
}
```

@SimonHoiberg

## 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

TypeScript Tip 💡

## USE RECORD TO MAKE AN OBJECT INDEXABLE



@SimonHoiberg

```
interface User { name: string };

const users: { [key: string]: User } = {
  SimonHoiberg: {
    name: 'Simon Hoiberg',
  }
};
```

```
interface User { name: string };

const users: Record<string, User> = {
  SimonHoiberg: {
    name: 'Simon Hoiberg',
  }
};
```

## 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

We have a 'User' interface with 4 properties that are all required.

```
interface User {  
  id: string;  
  username: string;  
  followers: number;  
  isBlocked: boolean;  
}
```

We can use **Pick** to create a new type containing only some properties.

```
const updateUser = (user: Pick<User, 'id' | 'username'>) => {  
  updateUserFromAPI(user.id, user.username);  
}  
  
updateUser({ id: '1234', username: 'SimonHoiberg' });
```



@SimonHoiberg

TypeScript Tip 

## 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

We have a 'User' interface with 4 properties of different types.

```
interface User {  
  id: string;  
  username: string;  
  followers?: number;  
  isBlocked?: boolean;  
}
```

We can use **Extract** to create a new type containing only properties that are strings

```
const logUser = (user: Extract<User, string>) => {  
  // 'User' only contains properties that are strings  
  Object.values(user).forEach(u => logUser(u));  
}  
  
const logUser = (prop: string) => {  
  console.log(prop);  
}
```

 @SimonHoiberg

TypeScript Tip 

# 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

Use the assertion “`as const`” upon assignment

```
const foo = { bar: 'baz' } as const;
```

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

TypeScript Tip

# “LOCKING” TYPES USING CONST ASSERTIONS



```
foo.bar = 'qux';
```

// Error: Cannot assign to 'bar' because it is a read-only property.

 @SimonHoiberg

## 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

TypeScript Tip 🌟

## USING THE KEYOF OPERATOR

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

```
interface Foo {  
    bar: number;  
    baz: string;  
    qux: Date;  
}  
  
type FooKey = keyof Foo;  
// "bar" | "baz" | "qux"
```

```
const foo: Foo = { bar: 1, baz: '-', qux: new Date() };  
  
const fooLookup = (key: FooKey) => {  
    return foo[key];  
}
```

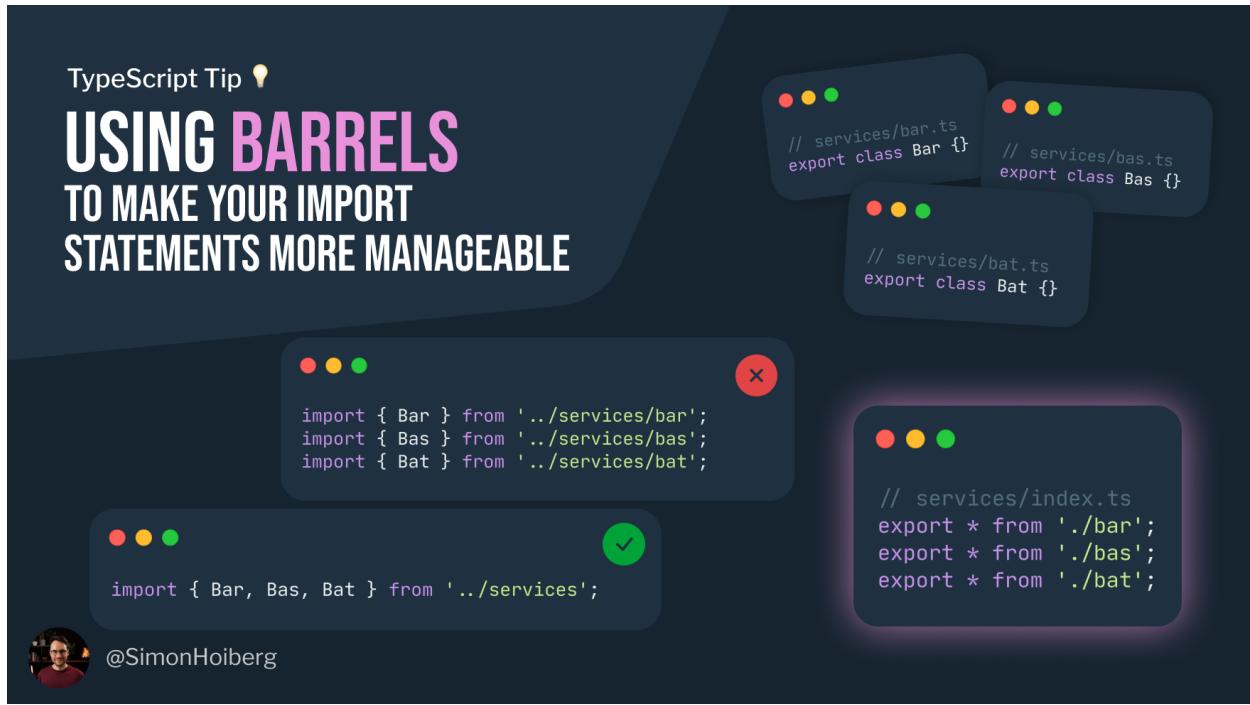
 @SimonHoiberg

## 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

The 'original' way with **callback hell**.

```
const fs = require("fs");
fs.writeFile("/path/file.json", stringToWrite, (error) => {
  if (error) {
    console.error(error);
    return;
  }
  console.log("file created successfully!");
});
```

**Import as promise directly from 'fs/promises'**

```
const fs = require("fs/promises");
try {
  await fs.writeFile("/path/file.json", stringToWrite);
  console.log("file created successfully!");
} catch (error) {
  console.error(error);
}
```

NodeJS Tip

**IMPORT NODEJS  
BUILTINS AS  
PROMISES**

@SimonHoiberg

## 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

AWS Lambda Tip !

## SEPARATE CORE LOGIC FROM YOUR HANDLER



 @SimonHoiberg

```
exports.myHandler = function (event, context, callback) {
  const { foo, bar } = event;
  const result = myLambdaFunction (foo, bar);

  callback(null, result);
}

function myLambdaFunction (foo, bar) {
  // Core logic goes here ...
}
```



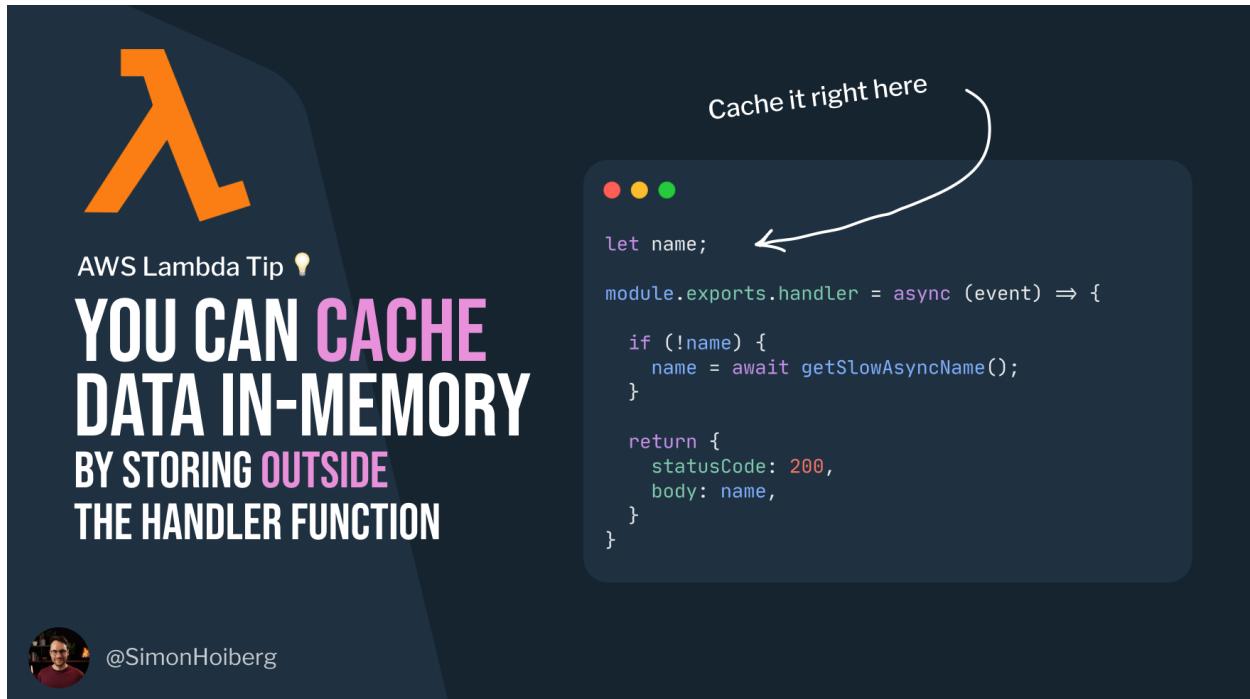
- The separation of concerns becomes clearer.
- The function becomes more unit-testable.
- The function becomes reusable.

## 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?

Code Review  
**FOREACH  
WITH ASYNC  
CALLBACK**

```
const discountsInEur = [];

products.forEach(async (p) => {
  const discount = p.price * 0.75;
  const exchangeRate = await getExchangeRate('USD', 'EUR');
  const discountInEur = discount * exchangeRate;

  discountsInEur.push(discountInEur);
});
```

@SimonHoiberg

## 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?

The screenshot shows a GitHub pull request interface. On the left, there's a dark-themed code editor window with three status dots at the top (red, yellow, green). The code inside is:

```
const foo = 42;
...
return (
  <main>
    <button onClick={() => passFooToApi(foo)}>
      Pass Foo
    </button>
  </main>
);
```

On the right, the title of the pull request is displayed in large white text: "ARROW FUNCTION AS HANDLER IN REACT?". Below the title are two large, semi-transparent icons: a green thumbs-up icon and a red thumbs-down icon.

At the bottom left, there's a small circular profile picture of a person and the handle "@SimonHoiberg".

## 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?

The image shows a screenshot of a code review interface. On the left, there is a dark-themed code editor window containing the following TypeScript code:

```
const { AxiosError, AxiosRequestConfig, AxiosResponse } = require('axios');

class UserApi extends Api {
  public constructor (config?: AxiosRequestConfig) {
    super(config);
  }

  // Some implementation
}

module.exports = {
  UserApi
}
```

On the right side of the interface, the text "Code Review" is visible above a large, bold title "REQUIRE IN TYPESCRIPT". Below the title is a large, stylized thumbs-down icon. At the bottom left, there is a small circular profile picture of a person and the handle "@SimonHoiberg".

## Result

Most people would reject this PR.

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

# Merging objects using object.assign?

The screenshot shows a code review interface. At the top, there are three colored dots (red, yellow, green). Below them is the code snippet:

```
const movie = { title: 'Inception', year: '2010' };
const meta = { length: '162m', size: '1.6gb' };

const movieDetails = Object.assign(movie, meta);

await addMovieToFavorite(movieDetails);
```

On the right side of the interface, the text "Code Review" is visible above the title "MERGING OBJECTS USING OBJECT.ASSIGN" in large, bold, white and pink letters.

In the center, there are two large, semi-transparent icons: a green thumbs-up icon on the left and a red thumbs-down icon on the right.

At the bottom left, there is a small circular profile picture of a person and the handle "@SimonHoiberg".

## 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?

```
function foo(bar) {
  if (bar) {
    /*
     * A bunch of code ...
     * Calculate some 'result'
    */

    return result;
  }
}
```

Code Review  
**NESTED  
IF-STATEMENT**

@SimonHoiberg

## 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.

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