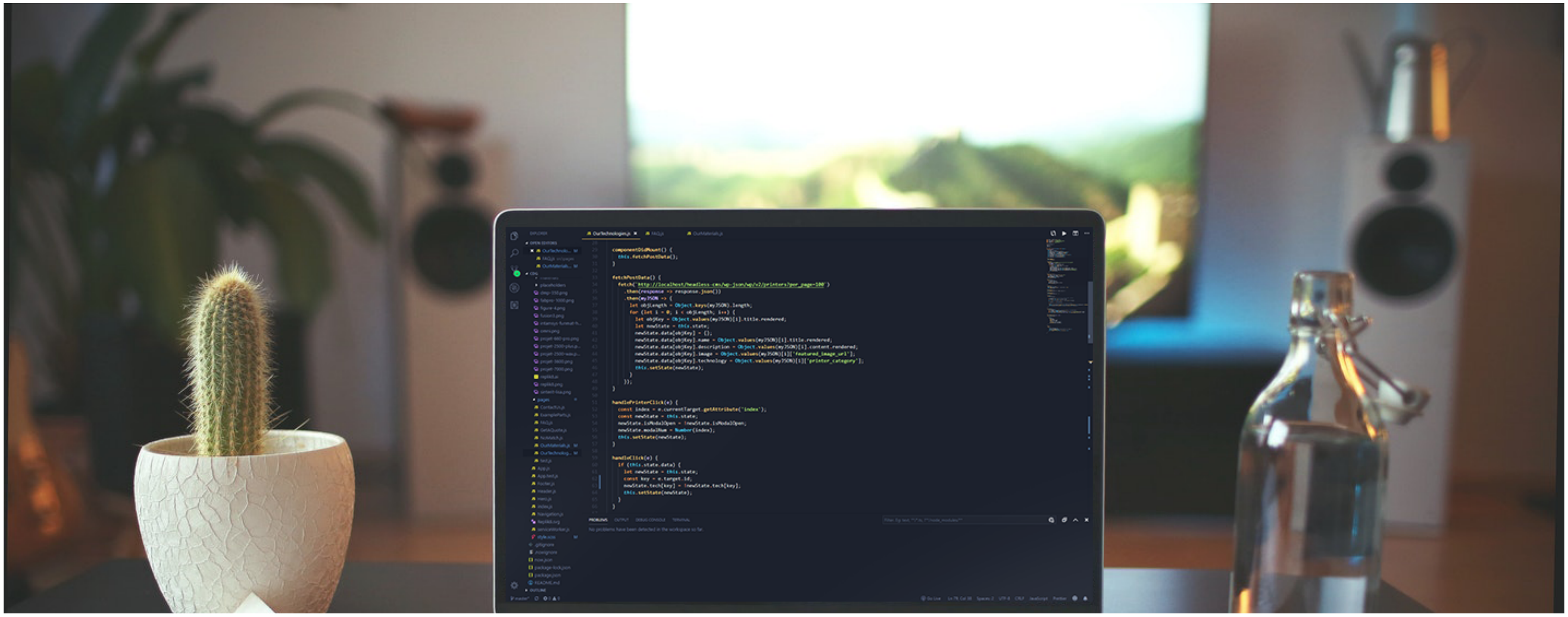
11 JavaScript Tricks You Won't Find in Most Tutorials

Useful tips for writing more concise and performant JavaScript

Bret Cameron Follow
Mar 26 · 9 min read ★



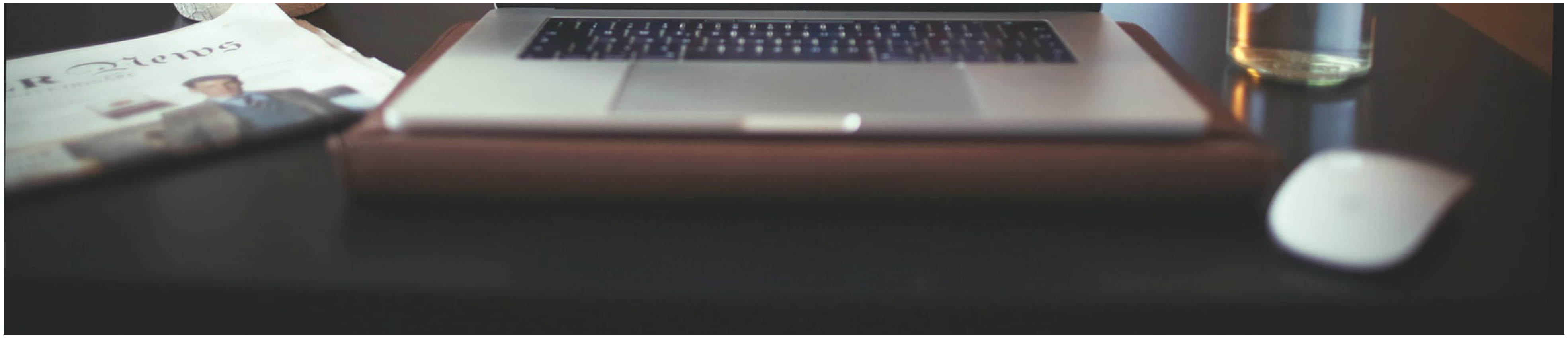


Image Credit: Igor Miske / Unsplash

When I began learning JavaScript, I made a list of every time-saving trick that I found in other people's code, on code challenge websites, and anywhere *other than* the tutorials I was using.

I have been contributing to this list since then, and in this article, I'll share 11 hand-picked tips that strike me as particularly clever or useful. This post is intended to be useful for beginners, but I hope even intermediate JavaScript developers will find something new in this list.

While many of these tricks are handy in any context, a few of them may be better suited for code golf than production-level code, where clarity is often more important than concision; I'll let you be the judge of that!

So, in no particular order, here are 11 neat ways to write more concise and more performant code.

1. Filter Unique Values

ARRAYS

The `Set` object type was introduced in ES6, and along with `...`, the ‘spread’ operator, we can use it to create a new array with *only* the unique values.

```
const array = [1, 1, 2, 3, 5, 5, 1]
const uniqueArray = [...new Set(array)];

console.log(uniqueArray); // Result: [1, 2, 3, 5]
```

Before ES6, isolating unique values would involve a lot more code than that!

This trick works for arrays containing primitive types: `undefined`, `null`, `boolean`, `string` and `number`. (If you had an array containing objects, functions or additional arrays, you’d need a different approach!)

2. Short-Circuit Evaluation

CONDITIONALS

The ternary operator is a quick way to write simple (and sometimes not-so-simple) conditional statements, like these:

```
x > 100 ? 'Above 100' : 'Below 100';
x > 100 ? (x > 200 ? 'Above 200' : 'Between 100-200') : 'Below 100';
```

But sometimes even the ternary operator is more complicated than necessary. Instead, we can use the 'and' `&&` and 'or' `||` logical operators to evaluate certain expressions in an even more concise way. This is often called 'short-circuiting' or 'short-circuit evaluation'.

How It Works

Let's say we want to return just one of two or more options.

Using `&&` will return the first `false` or 'falsy' value. If every operand evaluates to `true` , the last evaluated expression will be returned.

```
let one = 1, two = 2, three = 3;
console.log(one && two && three); // Result: 3

console.log(0 && null); // Result: 0
```

Using `||` will return the first `true` or 'truthy' value. If every operand evaluates to `false` , the last evaluated expression will be returned.

```
let one = 1, two = 2, three = 3;
console.log(one || two || three); // Result: 1

console.log(0 || null); // Result: null
```

Example 1

Let's say we want to return the `length` of a variable, but we don't know the variable type.

We could use an `if/else` statement to check that `foo` is an acceptable type, but this could get pretty longwinded. Short circuit evaluation allows us to do this instead:

```
return (foo || []).length;
```

If the variable `foo` is truthy, it will be returned. Otherwise, the `length` of the empty array will be returned: `0` .

Example 2

Have you ever had problems accessing a nested object property? You might not know if the object or one of the sub-properties exists, and this can cause frustrating errors.

Let's say we wanted to access a property called `data` within `this.state` , but `data` is undefined until our program has successfully returned a fetch request.

Depending on where we use it, calling `this.state.data` could prevent our app from running. To get around this, we could wrap it in a conditional:

```
if (this.state.data) {
  return this.state.data;
} else {
  return 'Fetching Data';
}
```

But that seems pretty repetitive. The 'or' operator provides a more concise solution:

```
return (this.state.data || 'Fetching Data');
```

We can't refactor the code above to use `&&` . The statement `'Fetching Data' && this.state.data` will return `this.state.data` whether it is `undefined` or not. This is because `'Fetching Data'` is 'truthy', and so the `&&` will always pass over it when it is listed first.

A New Proposed Feature: Optional Chaining

There is currently a proposal to allow 'optional chaining' when attempting to return a property deep in a tree-like structure. Under the proposal, the question mark symbol `?` could be used to extract a property *only* if it is not `null` .

For example, we could refactor our example above to `this.state.data?.()` , thus only returning `data` if it is not `null` .

Or, if we were mainly concerned about whether `state` was defined or not, we could return `this.state?.data` .

The proposal is currently at Stage 1, as an experimental feature. You can read about it here, and you can use in your JavaScript now via Babel, by adding @babel/plugin-proposal-optional-chaining to your `.babelrc` file.

3. Convert to Boolean

TYPE CONVERSION

Besides the regular boolean values `true` and `false` , JavaScript also treats all other values as either 'truthy' or 'falsy'.

Unless otherwise defined, all values in JavaScript are 'truthy' with the exception of `0` , `""` , `null` , `undefined` , `NaN` and of course `false` , which are 'falsy'.

We can easily switch between true and false by using the negative operator `!` , which will also convert the type to `"boolean"` .

```
const isTrue  = !0;
const isFalse = !1;
const alsoFalse = !!0;

console.log(isTrue); // Result: true
console.log(typeof true); // Result: "boolean"
```

This kind of type conversion can be handy in conditional statements, although the only reason you'd choose to define `false` as `!1` is if you were playing code golf!

4. Convert to String

TYPE CONVERSION

To quickly convert a number to a string, we can use the concatenation operator `+` followed by an empty set of quotation marks `""` .

```
const val = 1 + "";

console.log(val); // Result: "1"
console.log(typeof val); // Result: "string"
```

5. Convert to Number

TYPE CONVERSION

The opposite can be quickly achieved using the addition operator `+` .

```
let int = "15";
int = +int;

console.log(int); // Result: 15
console.log(typeof int); Result: "number"
```

This may also be used to convert booleans to numbers, as below:

```
console.log(+true);  // Return: 1
console.log(+false); // Return: 0
```

There may be contexts where the `+` will be interpreted as the concatenation operator rather than the addition operator. When that happens (and you want to return an integer, not a float) you can instead use two tildes: `~~` .

A tilde, known as the 'bitwise NOT operator', is an operator equivalent to `-n — 1` . So, for example, `~15` is equal to `-16` .

Using two tildes in a row effectively negates the operation, because `— ( — n — 1) — 1 = n + 1 — 1 = n` . In other words, `~ — 16` equals `15` .

```
const int = ~~"15"

console.log(int); // Result: 15
console.log(typeof int); Result: "number"
```

Though I can't think of many use-cases, the bitwise NOT operator can also be used on booleans: `~true = -2` and `~false = -1` .

6. Quick Powers

OPERATIONS

Since ES7, it has been possible to use the exponentiation operator `**` as a shorthand for powers, which is faster than writing `Math.pow(2, 3)` . This is straightforward stuff, but it makes the list because not many tutorials have been updated to include this operator!

```
console.log(2 ** 3); // Result: 8
```

This shouldn't be confused with the `^` symbol, commonly used to represent exponents, but which in JavaScript is the bitwise XOR operator.

Before ES7, shorthand existed only for powers with base 2, using the bitwise left shift operator `<<` :

```
// The following expressions are equivalent:

Math.pow(2, n);
2 << (n - 1);
2**n;
```

For example, `2 << 3 = 16` is equivalent to `2 ** 4 = 16` .

7. Quick Float to Integer

OPERATIONS / TYPE CONVERSION

If you want to convert a float to an integer, you can use `Math.floor()` , `Math.ceil()` or `Math.round()` . But there is also a faster way to truncate a float to an integer using `|`, the bitwise OR operator.

```
console.log(23.9 | 0);  // Result: 23
console.log(-23.9 | 0); // Result: -23
```

The behaviour of `|` varies depending on whether you're dealing with positive or negative numbers, so it's best only to use this shortcut if you're sure.

If `n` is positive, `n | 0` effectively rounds down. If `n` is negative, it effectively rounds up. To put it more accurately, this operation removes whatever comes after the decimal point, truncating a float to an integer.

You can get the same rounding effect by using `~~`, as above, and in fact *any* bitwise operator would force a float to an integer. The reasons these particular operations work is that — once forced to an integer — the value is left unchanged.

Remove Final Digits

The bitwise OR operator can also be used to remove any amount of digits from the end of an integer. This means we don't have to use code like this to convert between types:

```
let str = "1553";
Number(str.substring(0, str.length - 1));
```

Instead, the bitwise OR operator allows us to write:

```
console.log(1553 / 10   | 0)  // Result: 155
console.log(1553 / 100  | 0)  // Result: 15
console.log(1553 / 1000 | 0)  // Result: 1
```

8. Automatic Binding in Classes

CLASSES

We can use ES6 arrow notation in class methods, and by doing so binding is implied. This will often save several lines of code in our class constructor, and we can happily say goodbye to repetitive expressions such as `this.myMethod = this.myMethod.bind(this)` !

```
import React, { Component } from React;

export default class App extends Compononent {
  constructor(props) {
  super(props);
  this.state = {};
  }

myMethod = () => {
    // This method is bound implicitly!
  }

render() {
    return (
      <>
        <div>
          {this.myMethod()}
        </div>
      </>
    )
  }
};
```

9. Truncate an Array

ARRAYS

If you want to remove values from the end of an array destructively, there's are faster alternatives than using `splice()` .

For example, if you know the size of your original array, you can re-define its length property, like so:

```
let array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
array.length = 4;

console.log(array); // Result: [0, 1, 2, 3]
```

This is a particularly concise solution. However, I have found the run-time of the `slice()` method to be even faster. If speed is your main goal, consider using something like this:

```
let array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
array = array.slice(0, 4);

console.log(array); // Result: [0, 1, 2, 3]
```

10. Get the Last Item(s) in an Array

ARRAYS

The array method `slice()` can take negative integers, and if provided it will take values from the end of the array rather than the beginning.

```
let array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

console.log(array.slice(-1)); // Result: [9]
console.log(array.slice(-2)); // Result: [8, 9]
console.log(array.slice(-3)); // Result: [7, 8, 9]
```

11. Format JSON Code

JSON

Lastly, you may have used `JSON.stringify` before, but did you realise it can also help indent your JSON for you?

The `stringify()` method takes two optional parameters: a `replacer` function, which you can use to filter the JSON that is displayed, and a `space` value.

The `space` value takes an integer for the number of spaces you want or a string (such as `'\t'` to insert tabs), and it can make it a lot easier to read fetched JSON data.

```
console.log(JSON.stringify({ alpha: 'A', beta: 'B' }, null, '\t'));

// Result:
// '{
//     "alpha": A,
//     "beta": B
// }'
```

. . .

Overall, I hope you found these tips as useful as I did when I first discovered them.

Got any JavaScript tricks of your own? I'd love to read them in the comments below!

. . .

12. [Deprecated] Cache Array Length in Loops

LOOPS

In the original version of this article, I shared a tip to cache array length in `for` loops. However, if it is a read-only loop, modern JavaScript engines deal with this at the point of compilation. It is no longer necessary unless the length of the array changes (and, if that is the case, you'll probably want it to be recalculated with every iteration anyway).

Thanks to several commenters who pointed this out. If you'd like to find more, check out this question on StackOverflow.

For those who are interested, there used to be some performance incentive to writing `for (let i = 0, len = array.length; i < len; i++)` rather than `for (let i = 0; i < array.length; i++)` . This is no longer the case!

Get More Web Development Tips, Tricks and Learning Resources

Get solid gold sent to your inbox. Every week!

Email

Sign up

☐ I agree to leave Medium.com and submit this information, which will be collected and used according to Upscribe's privacy policy.

JavaScript Web Development Programming Coding Code

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we’ll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you’re at it. Just $5/month. Upgrade

About Help Legal