What is Hoisting in JavaScript?

Sunil Sandhu Follow

Aug 27, 2018 · 6 min read ★

One of Javascript's many quirks is something known as hoisting.

Now if you are new to coding in Javascript, it's quite likely that you're not writing your code perfectly just yet. So because of this, it's highly likely that your hoisting isn't perfect either. 😉

Ahoy(st) there sailor!

But what is hoisting?

Basically, when Javascript compiles all of your code, all variable declarations using `var` are hoisted/lifted to the top of their functional/local scope (if declared inside a function) or to the top of their global scope (if declared outside of a function) regardless of where the actual declaration has been made. This is what we mean by "*hoisting*".

Functions declarations are also hoisted, but these go to the very top, so will sit above all of the variable declarations.

Enough talk, lets show you some basic examples of code to demonstrate the impact of hoisting.

If we were to write the following in our global scope:

```
console.log(myName);
var myName = 'Sunil';
```

Pop quiz! What do you think the console.log will output?

1. `Uncaught ReferenceError: myName is not defined`

2. `Sunil`

3. `undefined`

It turns out that this third option is actually the correct answer.

As we mentioned earlier, variables get moved to the top of their scope when your Javascript compiles at runtime (which — if we exclude the use of NodeJS — at a very basic level simply means as your webpage is loading). However, a key thing to note is that the only thing that gets moved to the top is the variable declarations , not the actual value given to the variable.

Top highlight

Just to clarify what we mean, if we had a chunk of code and let's say that on Line 10, we had `var myName = 'Sunil'`, when the Javascript gets compiled, `var myName` would get moved to the top of its scope, whilst `myName = 'Sunil'` would stay on Line 10 (or possibly now Line 11 if `var myName` were hoisted up onto Line 1).

Let's look at the same block of code from earlier, but look at how the JavaScript compiler will output the code at runtime:

```
var myName;
console.log(myName);
myName = 'Sunil';
```

This is why the console.log is able to output '**undefined**', because it recognises that the variable **myName** exists, but **myName** hasn't been given a value until the third line.

By the way...

We have named our variable `myName` instead of simply `name` as the 'window' object in the browser already has a `name` property. If we were to test this in a browser, any variables created in the global scope actually end up being part of the 'window' object. Therefore, creating `var name = 'Sunil';` is the same as doing `window.name = 'Sunil';` and therefore, creating `var name = 'Sunil';` can also be referenced by typing `window.name`.

So as window.name already exists (for your interest, window.name simply returns an empty string — at least in Chrome Dev Tools anyway), we don't really get the proper sense of how hoisting works. We, therefore, chose to use `myName` instead! Don't worry if that just went over your head, hoisting is like that!

And as we climb back out of that rabbit hole...

We also briefly mentioned earlier that functions are also hoisted to the top (right at the top, above where the variable declarations are hoisted).

So if we look at the following example:

```
function hey() {
console.log('hey ' + myName);
};
hey();
var myName = 'Sunil';
```

The **hey()** function call will return undefined still, because really the

The hey() function call will return undefined but, because really the Javascript interpreter to compiling to the following at run time:

```
function hey() {
console.log('hey ' + myName);
};
var myName;
hey();
myName = 'Sunil';
```

So by the time the function gets called, it knows that there is a variable called `myName`, but the variable has not been given a value. There are a couple of variants to this, which occur when using variable expressions of **IIFE's** (click here if you want to read an earlier article on IIFEs) but trying to get a mental grip on all of this at once is not ideal, so I'll leave you to research hoisting with respect to **function expressions** and **IIFE's** by yourself.

Having said that, everything else mentioned above should help you to get a better understanding of how hoisting works.

The concept of hoisting is the reason why you may sometimes come across other people's code where variables are declared right at the top, and then are given values later. These people are simply trying to make their code closely resemble how the interpreter will compile it in order to help them minimise any possible errors.

But what about Let and Const?

They're also hoisted — in fact, `var`, `let`, `const`, `function` and `class` declarations are hoisted — what we have to remember though is that the concept of hosting is not a literal process (ie, the declarations themselves do not move to the top of the file — it is simply a process of the JavaScript compiler reading them first in order to create space in memory for them).

The difference between `var`, `let` and `const` declarations is their **initialisation** — in plain terms this simply means the value they are given to begin with.

Instances of `var` and `let` can be initialised without a value, while `const` will throw a `Reference error` if you try to declare one without assigning it a value at the same time. So `const myName = 'Sunil'` would work, but `const myName; myName = 'Sunil';` would not. With `var` and `let`, you can try to use a `var` value before it has been assigned and it would return `undefined`. However, if you did the same with `let` you would receive a `Reference Error`.

So is there any difference between var, `let` and `const` in terms of hoisting?

Yes, if you create a `var` at the top level (global level), it would create a property on the global object — in the case of a browser, this is likely to be the window object. So creating `var myName = 'Sunil';` can also be referenced by calling `window.myName`.

However, if you wrote `let newName = 'Sunny';` this would not be accessible in the global window object — therefore, you would not be able to use `window.newName` as a reference to `'Sunny'`.

The following issue is also fundamental to your understanding of how hoisting can affect your codebase

how hoisting can affect your codebase.

Declarations made with `var` can be accessed from outside of their initial scope, whereas declarations made with `let` and `const` are not.

As we can see in the below example, declarations made with `var` return `undefined` whereas those made with `let` and `const` return errors (*credit to gvlachos for raising and writing the following*):

```
console.log('1a', myName1); // undefined
if (1) {
 console.log('1b', myName1); // undefined
 var myName1 = 'Sunil';
}

console.log('2a', myName2); // error: myName2 is not defined
if (1) {
    console.log('2b', myName2); // undefined
    let myName2 = 'Sunil';
}

console.log('3a', myName3); // error: myName3 is not defined
if (1) {
    console.log('3b', myName3); // undefined
    const myName3 = 'Sunil';
}
```

And there we have it!

If you enjoyed this article, send many claps 👏👏👏 and subscribe to Javascript In Plain English for much more of the same!

Article updated on March 28th, 2019 in order to further explain hoisting in terms of let and const and also to fix typos in the article where I had referenced 'name' instead of 'myName'. This article was updated again on September 22nd, 2019 as there was a mistake in one of the code blocks with regard to the use of let. An additional piece of text was added to explain that the concept of 'hoisting' is indeed a concept, and not a literal process.

JavaScript | Programming | Front End Development | Coding | Web Development

8.8K claps

WRITTEN BY

Sunil Sandhu

Software Engineer, Editor of JavaScript In Plain English (JSIPE)

Follow

JavaScript in Plain English

Learn the web's most important programming language.

Follow

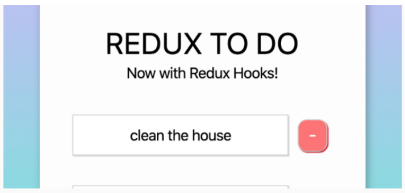
See responses (31)

More From Medium

More from JavaScript in Plain English

More from JavaScript in Plain English

More from JavaScript in Plain English



How I reduced the code in my Redux app by using Redux Hooks.

Sunil Sandhu in JavaScript in Pla...
Nov 15 · 6 min read ★
410

Upcoming new JavaScript features You should know if you use JavaScript everyday

Moon in JavaScript in Plain...
Nov 4 · 5 min read ★
1.5K

How to avoid try/catch statements nesting/chaining in JavaScript ?

Andréas Hanss in JavaScript in...
Nov 13 · 4 min read ★
321

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

Medium

About Help Legal