Hoisting

Introduction

When we send JavaScript to the browser to run, the interpreter breaks that down into two steps. First, it *parses* the code, and ensures it's all viable JavaScript friendly syntax. Second, it runs the code. You might think your code is running line by line, but with JavaScript, this isn't always the case. It is common for even experienced Devs to not completely understand the hoisting rules, but it is important for writing clean JS.

Variable Hoisting with var

What happens when we use a variable we haven't created?

console.log(magicalUnicorns);

copy

We get *ReferenceError: magicalUnicorns is not defined*. This is an example of a parsing error: Our code never ran, it stopped right there as soon as the parser noticed we were trying to use an uninitialized variable. Let's see how the var keyword affects this:

console.log(magicalUnicorns);

var magicalUnicorns = "awesome";

copy

If you ran the above code, you might have noticed something odd. *There was no ReferenceError.* Instead our console.log output *undefined*. Something else must be going on here. What's really happening is our var variable is being **hoisted**.

In JavaScript, **variable declarations are hoisted to the top of their scope.** Here's how the interpreter actually reads the previous example:

var magicalUnicorns; // the declaration gets hoisted to the top of the scope by itself

console.log(magicalUnicorns); // we log it as undefined

magicalUnicorns = "awesome"; // the assignment stays exactly where it was

copy

On the flip side, let and const do not allow us to do this.

console.log(magicalUnicorns);

let magicalUnicorns = "awesome!";

copy

In the above example, let will produce ReferenceErrors if we try to call the variable this way.

Hoisting in Functions

We discuss earlier that **function calls create their own scope**. How do you think the next snippet will run?

var foo = "bar";

function magic(){

foo = "hello world";

console.log(foo);

var foo;

}

magic();

console.log(foo);

copy

When you run this code, what are your console logs and in what order? The answer might be a surprise. Let's break it down.

var foo; // 'foo' is a declaration, it's global and gets hoisted

function magic(){ // 'magic()' also gets hoisted to the top

var foo; // here 'foo' is declared within 'magic()' and gets hoisted to the top of its scope

foo = "hello world"; // we assign a value to our function scoped 'foo'

console.log(foo); // we log it as 'hello world'

}

foo = "bar"; // here, we assign a value to the global 'foo'

magic(); // magic is called, the first console.log runs

console.log(foo); // finally we log the global foo

copy

There are two takeaways here: **Functions act like a cage, preventing declarations from being hoisted out of them.** Another thing to remember is that **standalone functions also get hoisted.** Let's see if you can predict the output of this next example without running it:

magicalUnicorns();

var magicalUnicorns = function(){

console.log("Will it blend?");

}

console.log("Don't breathe this!");

Which console log fires first? Neither! We get the error 'magicalUnicorns is not a function'. Why? The variable magicalUnicorns got hoisted to the top, and we tried to invoke it before we assigned the function to it.

Key Rules of Hoisting

• Variable declarations (var) rise to the top of their scope like hot air balloons.

• Functions create their own scope and act like cages, preventing declarations from rising out.

• Assignments, or = signs, act like anchors. They stay put, no matter how the code is rearranged.

• let and const do not get hoisted,  and will throw an error if called before they are declared. const will even throw a syntax error if it is called before it is assigned.