Hoisting in JavaScript

How hoisting plays a role in variable and function declaration scope

Even if you're already familiar with scope in JavaScript, there's another layer of how scope works with variable declarations that's important to understand in order to avoid inserting bugs in your code.

Behind the Scenes

During the compilation phase of code execution, the JavaScript engine allocates memory to save the names of declared variables and functions by *hoisting* variable and function declarations to the top of their current scope.

Hoisting does not actually move your code around; instead, it:

- Finds all of the variable and function declarations in the code
- 'Raises' the variable names and function declarations to the top of their scope before code execution
- Immediately initializes function declarations
- Postpones handling *initialization*, or assignment of values, until the code is executed

One of the advantages to this is the fact that you can call a function on a line of code that is before the line of code where you actually declare the function.

There are subtle hoisting differences between function declarations, function expressions, and variable declarations, which are important to understand to avoid errors in your code.

JavaScript hoists function declarations, including the function body, to the top of its scope. If a function is declared at the global level, then it is raised to the top of the global scope. Hoisting also occurs within nested functions, so nested functions are raised to the top of the enclosing function scope.

Overall, a function declaration can be called from anywhere within the scope as long as it is in fact declared in the code. In the code below, the `add()` function is called before it is declared, and runs as expected since the declaration of `add()` is hoisted during the compilation phase.

```
console.log(add(1,2));
// prints 3

function add(a,b) {
    return a + b
}
```

On the other hand, function expressions obey the hoisting rules for variable declarations, which have two sets of behavior–one set of rules for `var` and a shared set of rules for `let` and `const`.

`var` hoisting

`var` has its own playbook when it comes to hoisting. `var` variables are hoisted to the top of function blocks, but not other types of blocks, so you have to be careful when using `var` to declare variables to ensure you have the intended value when the variable is used. `var` variables are initialized as `undefined` when they are hoisted, which can lead to unexpected results in code if a variable is evaluated as undefined instead of an intended value.

For example, `myVarVariable` behaves the same way here:

```
console.log(myVarVariable);
var myVarVariable = 1;
```

as it does here:

```
var myVarVariable; // undefined
myVarVariable = 1;
```

`let` and `const` hoisting

Fortunately, variables defined with `let` and `const` are more predictable than `var` variables. Since the release of ES6, there are really no situations where it is preferable to use `var` variables. `let` and `const` variables are hoisted to the top of their parent block for scope, so any type of block or function can be the parent scope for those variables.

`let` and `const` differ from `var` in how they initialize as well; while the names are hoisted, they are not initialized. So calling a variable in your code before it is initialized results in a `ReferenceError`. This is one of the reasons `let` and `const` are preferable, as a `ReferenceError` will immediately alert you to the problem in your code. With `var` variables, you would need to implement unit tests to handle `undefined` values.

This is because `myLetVariable` behaves the same way here:

```
console.log(myLetVariable);
// triggers a ReferenceError
let myLetVariable = 1;
```

as it does here:

```
let myLetVariable;
console.log(myLetVariable);
myLetVariable = 1;
```

`myLetVariable` is not initialized and assigned the value 1 until the third line. So, when you use `let` or `const`, be sure to initialize the value before referencing the variables. This code snippet has the value declared and initialized before `myLetVariable` is referenced in the `console.log()` statement.

```
let myLetVariable = 1;
console.log(myLetVariable);
// prints 1
```

As mentioned, function expressions will follow the rules of variable hoisting. That said, if you are using ES6 arrow function syntax, it is vital to remember to assign the function to a variable before calling it.

```
// thisWontRun() is hoisted at the function call but not yet initialized
thisWontRun();
let thisWontRun = a => console.log(a);

// thisWillRun() gets initialized after function call below
let thisWillRun = b => console.log(b);
// thisWillRun() gets hoisted at the function call
thisWillRun();
```

Summary

In this article, we covered the fundamentals of hoisting and saw several behaviors for various variable types and function syntax. The most unpredictable variable type we looked at was `var`, which you can still see in legacy code bases, so you can't dismiss it, but you can refactor using `let` and `const` variables in your code to have more reliable block scoping.