Caner Karaman
Jan 20 · 4 min read

A Look at JavaScript Scopes

Understanding how scopes work in different scenarios

Deep Looking Scopes

A programmer needs to know the scope of a variable to know from where the function can be accessed. In short, the scope is where we can access a variable or function.

Contents of this blog:

- Scope Chain and Nested Scope
- Block Scope vs Function Scope
- Lifecycle of Variables

To better understand the working mechanism of Scopes, you need to know how Lexical Scope works. In the previous blog, I explained the mechanism of ***Lexical Scope*** and ***Closures.***

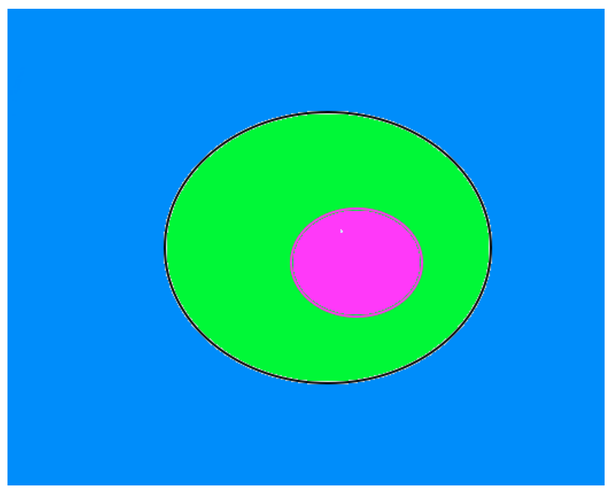
Scope Chain and Nested Scope

```
1  const numbers = [1, 2, 4, 12, 123, 55, 77, 32, 11,];
2
3  function getBiggerNumbers(no){
4    const biggerNumbers = numbers.filter((number) => {
5      return number > no
6    });
7
8    return biggerNumbers;
9  }
10
11 bigNos = getBiggerNumbers(10);
12   console.log(bigNos); // [ 12, 123, 55, 77, 32, 11 ]
13
```

scopeChain.js hosted with ♥ by GitHub view raw

There are 3 scopes in the sample code above. The 3 scopes here are intertwined with each other. Interlacing the scopes is called **nested scope**. If we count these scopes from the inside out;

1. It starts with the arrow function that takes the number parameter on the 4th line and ends with the first curly brace on line 6.
2. It starts with the getBiggerNumbers function on line 3 and covers the lines up to the curly braces on line 9.
3. All remaining fields are included in this scope. In addition, this scope is specially named global scope.



Nested Scope

It was created to better understand the above visual scopes. The area painted in pink represents the 1.scope. The green colored area represents scope 2. Blue area is global scope. Scopes that have a nested scope relationship with each other are called **scope chains**. Since there are 3 nested scopes here, this example is also an example of a scope chain.

Block Scope vs Function Scope

According to Wikipedia, Function scope can be defined as “When the scope of variables declared within a function does not extend beyond that function, this is known as **function scope**”. Block scope is defined as “The scope of a name binding is a block, which is known as **block scope**”. Let’s go through the example to understand these definitions better.

```
1  function getName(hasName) {
2    if(hasName){
3      var name = 'Caner';
4    } else {
```

```
7
8     console.log(name);
9   }
10
11  getName(true); // Caner
12  getName(false); // Admin
```

functionScope.js hosted with ❤ by GitHub view raw

The snippet you see above is an example of function scope. If you are a programmer who used to work with block scope, you probably thought that you would face a problem with this piece of code. Because the variable name in the 8th line is not defined and the error message should have been issued because it was tried to be accessed on the 8th line. But functional scope doesn't work that way. When this snippet is compiled, the lexical scope defines a variable name as undefined at the top of the function. For this reason, the variable name is accessible. In addition, the process of moving this variable to the top of the function, which the lexical scope does, is called **hoisting**. Now let's write the same piece of code with block scope and see what result we get.

```
1   function getName(hasName) {
2     if(hasName){
3       let name = 'Caner';
4     } else {
5       let name = "Admin"
6     }
7
8     console.log(name); // ReferenceError: name is not defined
9   }
10
11  getName(true);
12  getName(false);
```

blockScope.js hosted with ❤ by GitHub view raw

As you can see, this is one of the main reasons why let and const keywords are preferred over the var keyword in variable definition. Because let and const block are defined as scope, they minimize the errors that may arise.

Lifecycle of Variables

We will examine how a defined variable will behave when it is defined again with the same name in the same scope, or if its value is to be read before a variable is defined. Lexical scope provides how and when a variable can be accessed.

Function Hoisting:

```
sum(1 + 3); // 4

function sum(a, b){
        console.log(a+b);
}
```

Although the sum function in the code snippet is executed before it is defined, there is no error. Why? Lexical scope places the defined sum function at the top of the scope it was in when compiling. Therefore, when the sum function is called, it can be run without any error. This moving process is called **hoisting**.

```
sum(1 + 3); // TypeError: sum is not a function

var sum = function sum(a, b){
        console.log(a+b);
}
```

As can be seen in this code, an error has been thrown. The error is returning as TypeError: sum is not a function. Here, Javascript did not return us anything that sum function was not defined. The error tells us that the variable Sum was found but its reference is not a function. If Javascript could not access the variable name sum, it would return the error ReferenceError: sum is not defined. The sum function defined here has the value undefined.

Variable Hoisting:

```
console.log(name); // undefined
name = 'Caner';
console.log(name); // Caner
var name = "Karaman"
console.log(name); // Karaman
```

Although the variable name printed to the console in the 1st line has not been defined before, it does not throw any error while writing to the console. Likewise here, lexical scope defines variable 'name' as undefined at the top of the scope. Therefore, console.log(name) in the first line returns undefined instead of giving an error. Since the name variable was defined at the top of the scope with the help of lexical scope, a value could be assigned to the name variable in the second line without any keywords. Therefore, console.log(name) on line 3 returned 'Caner'. The same process continues on the 4th and 5th lines.

For better understanding, let's look at the same code after it has been compiled.

```
var name;
console.log(name); // undefined
name = 'Caner';
console.log(name); // Caner
name = "Karaman"
console.log(name); // Karaman
```

As you can see, lexical scope has taken the var variable and moved it to the top of the scope. Therefore, no errors are encountered.