Scopes , Closures And Hoisting

Created	@June 1, 2022 4:50 PM
Class	
Type	
Materials	
Reviewed	☐

What the hell is a scope ?

In programming, scope is a term related to vision. Now vision of what ? Scope talks about the vision of functions that what all variables they can see. Scope if where the function will look for variables and other functions. So we are going to look for identifiers. The variables which are visible to functions are the only variables they can access.

Now every variables inside your code is present in one of the two states:

- Either it will be getting some value assigned to it.
- Some value will be retrieved from it.

```
let x = 10; // here we are assigning the value 10 to x
console.log(x); // here the value of x is retrived
```

How JS handles the scope of variables ?

```
let x = 10;
console.log(x);
console.log("hi");
```

Now, a lot of people think that , if we will run the above code, then JS starts reading line 1, and immediately executes it, then reads line 2 and immediately executes it. `But that's not how JS will run the code`.

Why ?

So, actually JS reads your whole code twice.

The first phase of code reading is called as `Lexical parsing`. The second phase is called as `Code execution`.

Block scope and function Scope

There are three kind of scopes in JS:

- `Blocks`
- `Function`
- `Global`

Block Scope

In JS and many other languages, we define the term Blocks as a set of statements enclosed between a pair of curly braces.

So any variable is said to have block scope if it is only accessible inside the given block. That means any variable declared inside a block is only accessible in that block. This is the meaning of block scope. So anytime if somebody says you that the variable has a block scope that means there is block in which the variable was declared and it is only accessible there.

So in the code below `x` has block scope of if.

Note: It is not because `y` is declared inside if that is the reason of `y` having block scope. Because if you change the statement with `var y=20;` it's scope will become function scope. So it's not about just the place of declaration.

```
function fun () {
  let x = 10;
  if (x > 5) {
    let y = 20;
    console.log("inside blocks", x);
```

```
    console.log("inside blocks", y);
  }

  console.log("outside blocks", x);
  console.log("outside blocks", y);
}
fun();
```

Function scope

Now if a block is declared for a function (as to initialise a function also we need a pair of curly brace) the scope given by that block is `Function scope`.

A variable having function scope is accessible everywhere in the function. For example: `x` in the above code. So because `x` has a function scope, it is accessible everywhere in the function.

Global Scope

If a variable is accessible everywhere in the code, then it is said to have global scope.

Let and Var

`Let` : Whenever we declare a variable using let, then the variable will get the scope of the nearest block. If the nearest block is a for loop then it will be given scope of for, otherwise let's say if the nearest block is a function, then we will get the whole function's scope.

So in the above example: the nearest block of `x` is function so it is given scope of the whole function i.e. it will be visible in the whole function whereas `y` is declared inside the if block so it is given the scope of if, i.e. it is only visible inside if not the whole function.

`var` : Whenever we declare a variable using var, then the variable will get scope of the nearest function, that means it will be accessible everywhere in the nearest function.

So that is why if we declare `y` by `var y=20;` we get y with function scope.

Note: There is a slight diff in the function scope that var gives and function scope that a variable gets it the nearest block is function with let.

Note: For `const` scoping works exactly like `let`.

```
function gun () {
    function fun () {
      let x = 10;
      if (x > 5) {
        var y = 20;
        console.log("inside blocks", x);
        console.log("inside blocks", y);
      }

      console.log("outside blocks", x);
      console.log("outside blocks", y);
    }
  fun();
  console.log("inside gun", x);
  console.log("inside gun", y);
}

gun();
```

In the above code, the scope of x and y is inside fun only, that is why it is not accessible outside fun, and throws error outside fun.

```
Note: Not just the position of declaration or let/var defines the scope of variables. It i
s decided by multiple factors in which one of the factors are let/var and position.
```

Lexical parsing

What is lexical parsing ? We give this name lexical parsing to the first phase because JS supports lexical scopes. (NOTE: There are two ways to handle scopes, - Lexical scope - Dynamic scope)

In the first phase of lexical parsing, every variable’s scope get assigned. But how ?

The lexical parsing strategy of JS will look through the code line by line without executing it and every time it sees an identifier, it will ask the current block that do you already know about this variable (asking about a variable means that do u declare this variable ? or was this in some outside scope in which the current block is present)? if not then this variable might belong to you if we are assigning the value not for the case when we are using the value.

If we are using a value inside a scope without declaring it then it will throw error.

```
var teacher = "Sanket"; // global
function fun () { // global
  teacher = "pulkit"; // global
  console.log("relevel");
}
fun(); // global
console.log(teacher); // global
```

After lexical parsing is done, every variable has the corresponding scope, but not values.

When we go to phase 2, that is execution then we assign value.

```
var teacher = "Sanket"; // global
function fun () { // global
  let teacher = "pulkit"; // fun
  console.log("relevel");
}
fun(); // global
console.log(teacher); // global
```

Due to the fact that fun's scope already knew about teacher, it didn't checked outside fun for the scope of teacher and got fun's scope. That why this time the value "sanket" didn't get updated.

Same thing happens if we do `var teacher = "pulkit"` because var also gives scope of function fun.

```
var teacher = "Sanket"; // global
function fun () { // global
  teacher = "pulkit"; // global
  function gun () {
    let teacher = "Vishwa"; // local
  }
  gun();
  console.log("relevel");
}
fun(); // global
console.log(teacher); // global
```

Here the final teacher value will be pulkit as it teacher inside fun is a global scoped variable and teacher inside gun is local to gun.