CLOSURES

let keyword

let allows you to **declare variables that are limited in scope to the block,** statement, or expression on which it is used. Unlike the var keyword, this defines a variable globally or locally to an entire function regardless of block scope.

Scoping Rules

Variables declared by let have their scope in the block for which they are defined, as well as in any contained sub-blocks. In this way, let works very much like var. The main difference is that the scope of a var variable is the entire enclosing function.

```
function varTest( ) {
        var x = 1;
        if (true) {
            var x = 2; // same variable!
            console.log(x); // 2
  }
  console.log(x); // 2
}
```

```
function varTest( ) {
        let x = 1;
        if (true) {
            let x = 2; // same variable!
            console.log(x); // 2
  }
  console.log(x); // 1
}
```

Scoping in for loop

```
for(var a = 1; a < 5; a++){
   setTimeout(function( ){
       console.log(a)
    }, 1000);
 }
```
Output : 5 5 5 5

```
for(let a = 1; a < 5; a++){
   setTimeout(function( ){
       console.log(a)
    }, 1000);
 }
```
Output : 1 2 3 4

- Desired Output was **1 2 3 4**, which came out to when variable declared was using let

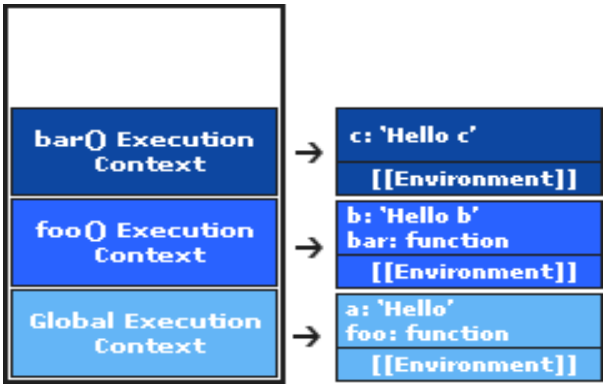
The loop with **var prints 5 5 5 5** instead of 1 2 3 4. The short answer for this phenomenon is **that the for loop executes first, then it looks for the a value, which is 5**, and then outputs four times, one for each loop iteration.

The loop with **let prints 1 2 3 4**. Every round of let creates a new variable and bounds it with the closure. Let a gets a new binding for every iteration of the loop. This means that **every closure, if the function is created in the loop, captures a different instance.**

Lexical Environment

```
var a = 'Hello';
function foo(){
  var b = 'Hello b';
  function bar() {
    var c = 'Hello c';
    console.log(c); // You can access me here.
    console.log(b); // You can access me too..
    console.log(a); // You can also access me..
    }
    bar();
 }
foo();
```

When this code runs initially, a **Global Environment was created, and a and foo( ) was stored in that**. When we invoked the function foo below, a new environment was created and stored the variable b in foo environment, which is only visible to bar function because bar is an inner function in the foo environment. When invoking the foo( ), we also called the bar() function, creating a new environment for their definition.

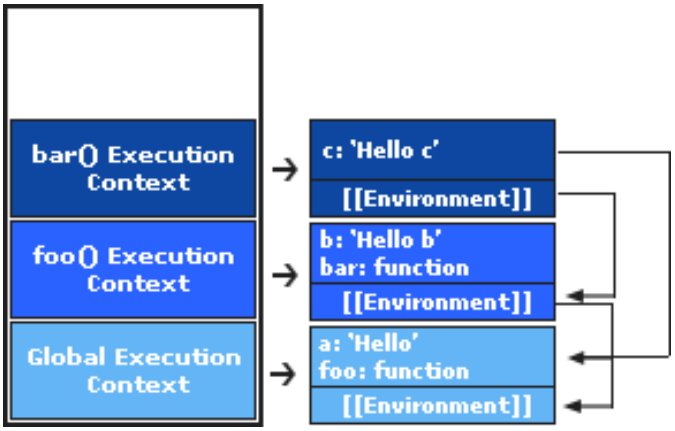


- ❖ Whenever we call a function, a new function execution context is created and pushed into the execution context stack with a new associated lexical environment.

Inside the bar function, we do logging to check if the variables we create are visible and can access them in their environment.

However, when logging was done, **variables "b" and "a"** were successfully displayed.
How did this happen ?

This is because, In the second logging, we call the variable b, which is not in the bar function scope. So **javascript does this internally to search it in other Outer Environment until they found that variable.** In our case, they found the variable b at foo function. Since the bar function references the foo function, the foo function environment does not pop out in the Execution Context! It was also successfully displayed when variable a was logged because variable a is stored in the Global Execution Context. Everyone can access the Scope in the Global Execution Context.



CLOSURES

A closure is a feature in JavaScript where an **inner function has access to the outer (enclosing) function's variables** — a scope chain.

The closure has three scope chains:

- Access to its **own scope** — variables defined between its curly brackets
- Access to the **outer function's variables**
- Access to the **global variables**

A closure is created when an inner function is made accessible from outside of the function that created it. This typically occurs when an **outer function returns an inner function**. When this happens, the inner function maintains a reference to the environment in which it was created. This means that it remembers all of the variables (and their values) that were in scope at the time. The following example shows how a closure is created and used.

```
function add(value1) {
 return function doAdd(value2) {
          return value1 + value2;
  };
 }
 var increment = add(1);
 var foo = increment(2);  //Returns 3
```

Thee add() function **returns its inner function doAdd( ).** By returning a reference to an inner function, a closure is created.

“value1” is a local variable of add( ), and a non-local variable of doAdd( ).
Non-local variables refer to variables that are neither in the local nor the global scope. “value2” is a local variable of doAdd( ).

When add(1) is called, a **closure is created and stored in “increment”.** In the closure’s referencing environment, “value1” is bound to value one. Variables that are bound are also said to be closed over. This is where the name closure comes from.

When increment(2) is called, the closure is entered. This means that doAdd( ) is called, with the “value1” variable holding the value one.