



Topic : Closures

1. LET KEYWORD

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

1.1. 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 letTest() {  
    let x = 1;  
    if (true) {  
        let x = 2; // different variable  
        console.log(x); // 2  
    }  
    console.log(x); // 1  
}
```

Let in for loop – Consider the following for loop.

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

The loop will print 5 5 5 5 instead of 1 2 3 4. The short answer for this phenomena 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.

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

This loop will print 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 function is created in loop captures a different a instance.

EXTRA:

You can read about let from the links below -

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>
<https://www.geeksforgeeks.org/difference-between-var-and-let-in-javascript/>

2. EXECUTION CONTEXT AND LEXICAL ENVIRONMENT

2.1. Execution Context

When your code runs in the JavaScript Engine. Each statement of your code is executed in a certain Execution Context. In JavaScript environment there are 2 main types of Execution Context. First is Global Execution Context, when your code is initially run even if it's spread across to a page using a <script /> tag, JavaScript creates one **Global Execution Context** in which your code was placed in when they execute and runs inside the browser. Second is the **Function Execution Context** from the word itself it was created when you invoked the function that you define.

Each time you invoked a function it will create a new Function Execution Context.

```
var message = 'Hello there';
function foo(message) {
    bar(message);
}

function bar(message) {
    console.log(message);
}
foo(message);
```

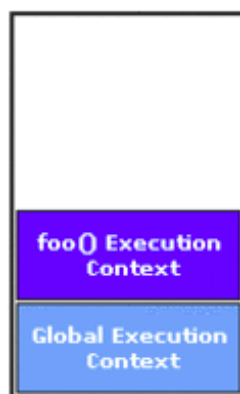
Initially Execution Context Stack is empty.



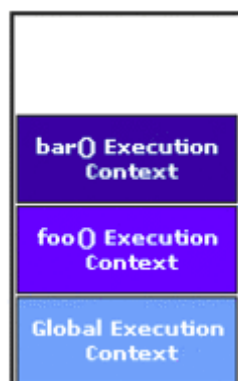
When this code runs, JavaScript engine **create one Global Execution Context** and push it to Execution Context Stack.



When we call the function foo, Global Execution Context was paused because JavaScript is a single threaded environment they can only execute one code at a time. After that JavaScript engine will **create a new Function Execution Context for foo** and push it into Execution Context Stack.



When foo function is executed we invoked the bar inside foo definition. JavaScript engine paused the execution context in foo function and **creates a new Function Execution Context for bar** and pushed it into stack.



After the **bar was finished executing it will popped out** in the Execution Context Stack and go **back to foo** and resume its execution. **Same process will applied to the foo until we finished and go back to the Global Execution Context** and resume the execution.

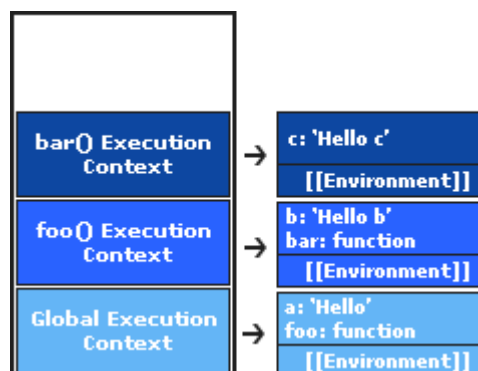
2.2. Lexical Environment

Consider the following code

```
var a = 'a';
function foo() {
  var b = 'b';
  function bar() {
    var c = '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 foo environment. When invoking the foo() we also called the bar() function which is also creates a new environment for their definition.

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

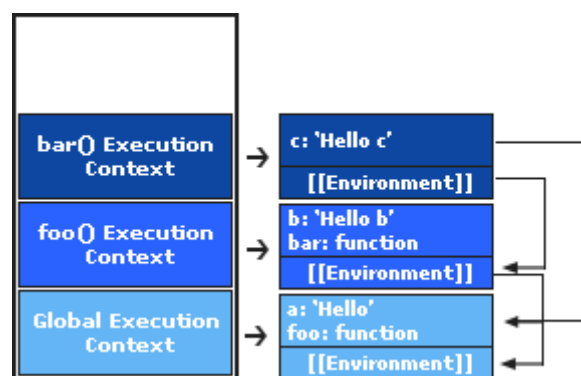


Inside the bar function we do logging to checks if the variables that we create is visible and if we can access it in their environment.

When variable c was logged in the bar environment, it was displayed successfully obviously because it was inside his environment.

However when we do logging the variable b and variable a it was also successfully displayed. How did this happen?

This is because, In the second logging we call the variable b which is not in 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 has reference to foo function the foo function environment is not pop out in the Execution Context! When variable a was logged it was also successfully displayed because variable a is stored in the Global Execution Context. Everyone can access the Scope in the Global Execution Context.



EXTRA:

You can read about them from the links below -

<https://blog.bitsrc.io/understanding-execution-context-and-execution-stack-in-javascript-1c9ea8642dd0>

3. 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:

- it has access to its ***own scope*** — variables defined between its curly brackets
- it has access to the ***outer function's variables***
- it has 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);  
// foo equals 3
```

From the above example, we can make following observations -

The **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 the 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.

EXTRA:

You can read about closure from the links below -

<https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36>