

Class 10 - scopes, lexical scope and closures

Revision of Execution Context

```
console.log(a)
var a = 10
function test(){

console.log('This is a test function')
}
test();
```

If I try to console log A before initializing the value A? What do you think the output should be?

This will run, The output will be undefined

Now if I move the function call before initialiing the function

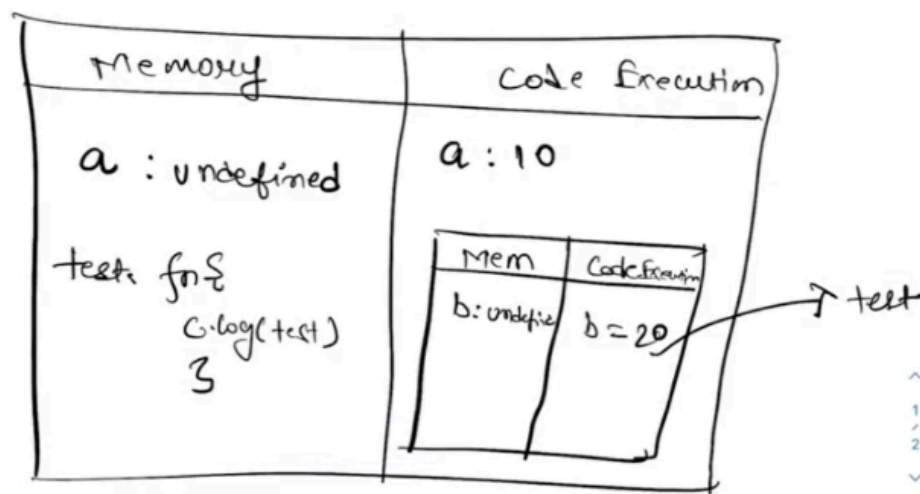
```
console.log(a)
var a = 10
test();
function test(){
  console.log('This is a test function')
}
```

If I try to invoke this function before declaring this, before declaring this, then what should be the output?

So if I now run this code, the function still runs. There is no undefined or there is no error or something like that, it still runs. Although I have defined this function later and I'm calling this earlier. But still this function runs.

But this behavior is not acceptable when you do this with a variable. this behavior can be explained by knowledge of execution context.

1.7 GEC.



Dissecting Scope

1. What would be the output

```
function greet(){
  console.log(msg)
}

var msg = "Hello World"

greet()
```

2. When the function invocation happens, JS engine will try to find the msg variable in the local memory space (Scope) or in local memory of greet's execution context
3. Since it prints the value "Hello Worl" so somehow it was able to access the value which is outside the function
4. What if we have another nested function

```
function greet(){
  sayHNW()
  function sayHNW(){
    console.log(msg)
  }
  // console.log(msg)
```

```
}  
  
var msg = "Happy New Year"  
  
greet()
```

5. What if we move the variable inside the function and access it outside

```
function greet() {  
  var msg = "Happy New Year";  
  sayHNW();  
  function sayHNW() {  
    console.log("inner",msg);  
  }  
}  
  
greet();  
console.log("outer",msg)
```

6. Scope

- a. Just like the cafeteria, common sitting area, gaming area, etc., in a college, where everyone can come and go, the global scope in programming is where variables and functions are accessible to the entire program.

b. Liek Example below

```
var a = 20  
  
function parent() {  
  console.log(a);  
}
```

```
parent()
```

- c. The classrooms in a college, where only students of that particular class or subject go, are akin to local (or function) scopes in programming.

- d. Code below

```
function parent() {  
  var a = 20  
  console.log(a)  
}  
console.log(a)
```

```
parent()
```

- e. So how can we define scope - **Scope refers to where in a program a variable or function is accessible. It determines the visibility and lifetime of variables and functions** within different parts of your code
- f. Lets see one more example

```
function parent() {  
  var a = 20  
  function child() {  
    console.log(a);  
  
    function child2() {  
      console.log(a)  
    }  
    child2()  
  }  
  child()  
}
```

```
parent()
```

- g. So you see, whatever functions you are writing inside this parent function, they have the access to this 'a' variable.

h. Lets revisit this pattern

```
function parent() {  
  function child() {  
    console.log(a);  
  
    function child2() {  
      var a = 20;  
      console.log(a)  
    }  
    child2()  
  }  
  child()  
}
```

```
parent()
```

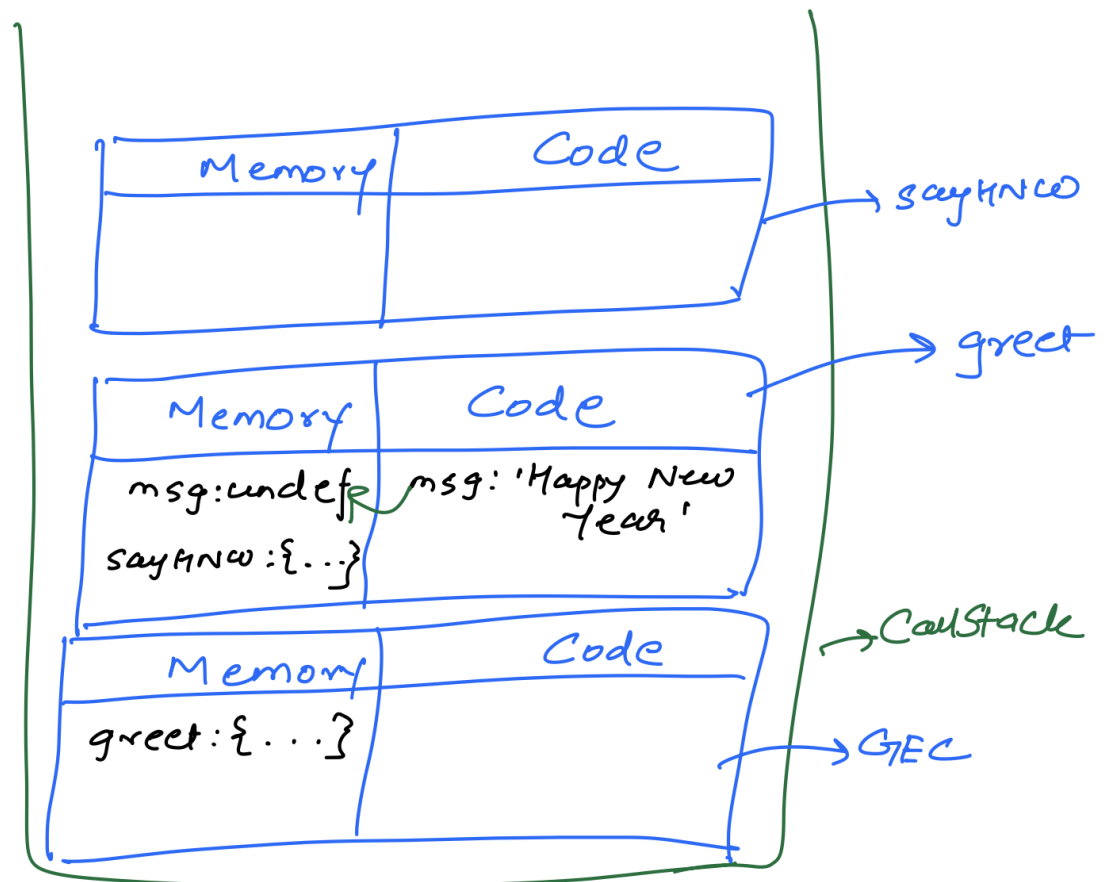
- i. we have an error. 'a' is not defined . 'a' is inside this function scope, and this cannot be accessed outside.
- j. This is the simple example of lexical scope that **Every Nested function will have access to its parent's properties or variables and they will be able to work with them and the scope that is formed is lexical scope**

7. Lexical Environment

- a. Lexical environment is the local memory plus the lexical environment of its parent
- b. Let's take the previous example of greet

```
function greet() {  
  var msg = "Happy New Year";  
  sayHNW();  
  function sayHNW() {  
    console.log("inner",msg);  
  }  
}
```

```
greet();
console.log("outer",msg)
```



c.

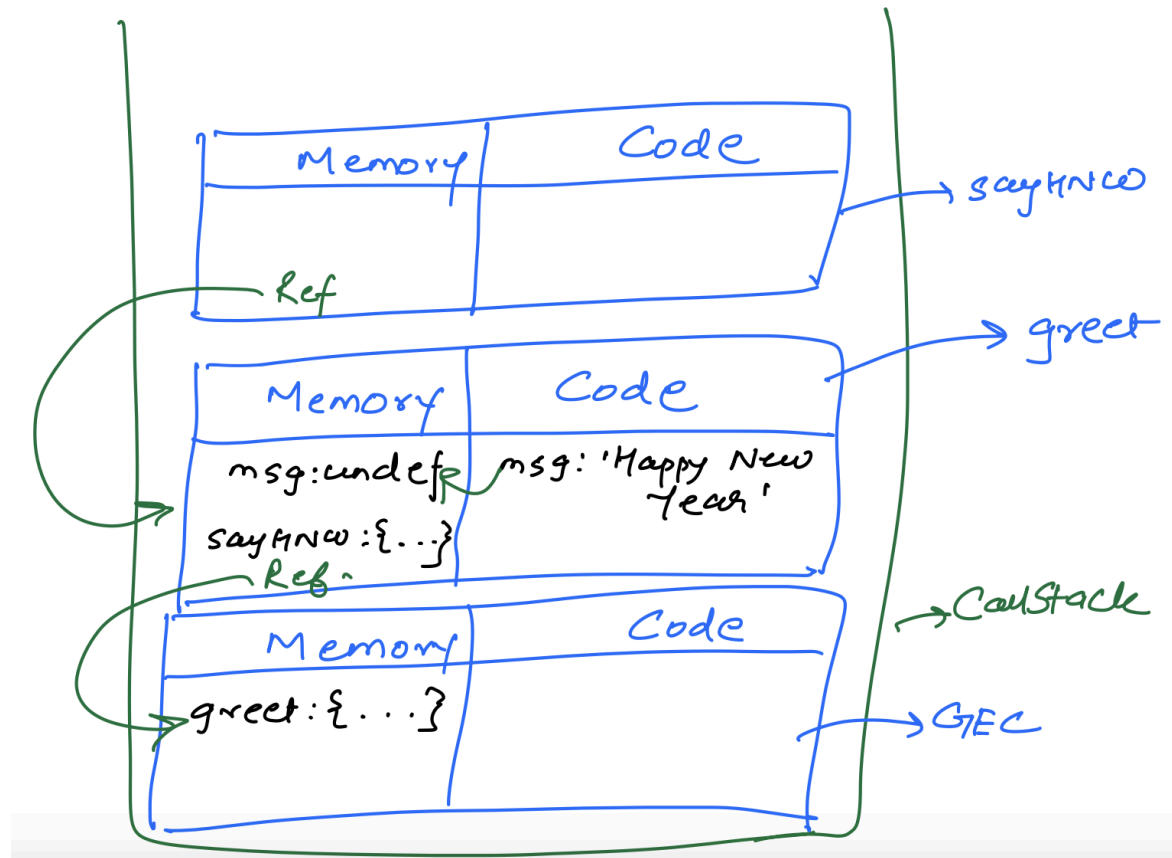
d. So when we talk about lexical environment, we say where it is placed in hierarchy

e.

f. We say the `sayHNW` is lexically inside `greet` function

g. In simple words we are saying `sayHNW` is written or placed inside the `greet` function and `greet` is lexically inside the global scope

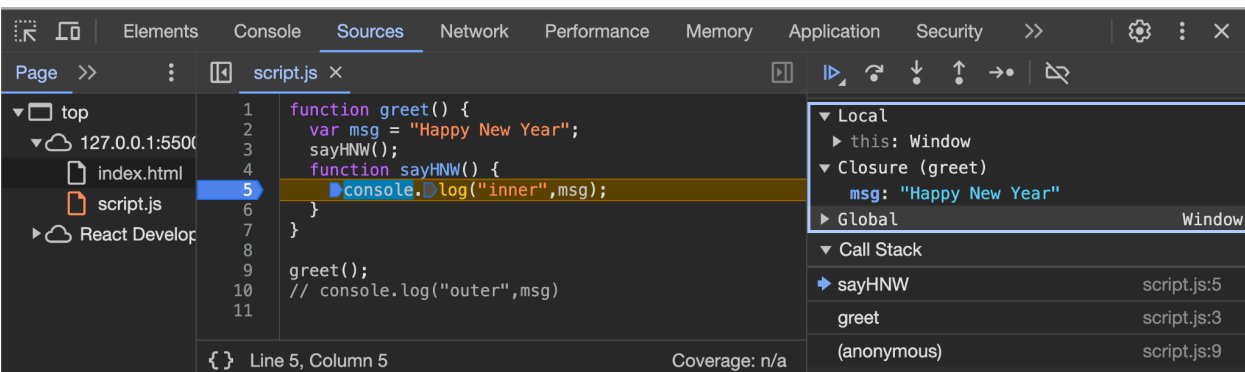
- h. Whenever the execution context is created, we also get access to the lexical environment of our parent



- i. Using this arrangement of lexical environment is how we are able to access variables present in parent's scope
- j. This arrangement or chain of lexical environments is known as **SCOPE CHAIN**
- k. The scope chain is a series of linked lexical environments.
- l. If a variable isn't found in the current lexical environment, JavaScript follows the reference to the outer environment and looks for the variable there. This process continues,

moving outwards in the chain, until the variable is found or the global scope is reached.

- m. This chain of lexical environments (from local to global) determines the accessibility of variables and is known as the scope chain.
- n. So when we place a debugger inside the inner function, we can see the function's lexical environment which consists of its scope, its parent scope and the global scope

o. 

- p. Lexical scope refers to the concept where a child (nested) function has access to the variables and scope of its parent function. This means the function is 'lexically bound' to the environment in which it was declared, not where it is called.
- q. Every nested function in JavaScript has access to the variables and properties of its parent function's scope. This access forms the basis of lexical scoping, enabling the nested function to 'remember' and interact with the environment it was created in, regardless of where it is executed.

Scope difference of var, let and const

1. We saw few differences already between a var, let and const declared variables
 - a. Redeclaration possible for var
 - b. Hoisting difference
 - c. Reinitialization differences
 - d. There is one more imp difference related to scope of variables declared using let, const and var keyword
2. Let and const are block scoped
 - a. What is a block in JS

```
{  
    // this is a standalone block  
}  
  
if(true){  
    // this is a block inside an if statement  
}
```

- b. Block scope means that a variable declared with either let or const is only accessible within the block of code in which it is defined.** A "block" is typically defined by curly braces {} and can be a function, an if statement, a loop, or any other kind of statement that uses curly braces.
 - c. Var declared variables are function scoped and let and const are block scoped**

```
if (true) {  
    let blockScopedVar = "I am inside an if block";  
    console.log(blockScopedVar); // Works fine  
}
```

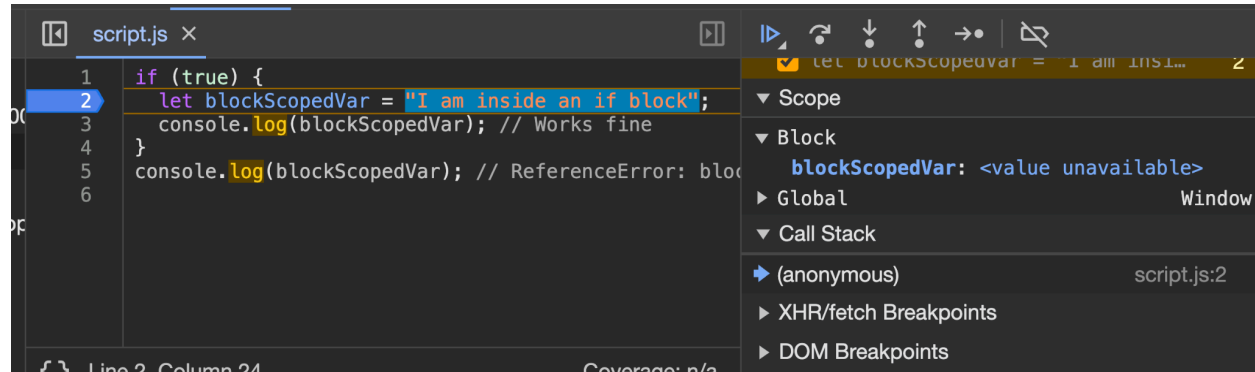
```

}

console.log(blockScopedVar); // ReferenceError:
blockScopedVar is not defined

```

d. see the block scope on developer tools



e.

f. What will be the output here

```

let val = 'some msg'

if(true){

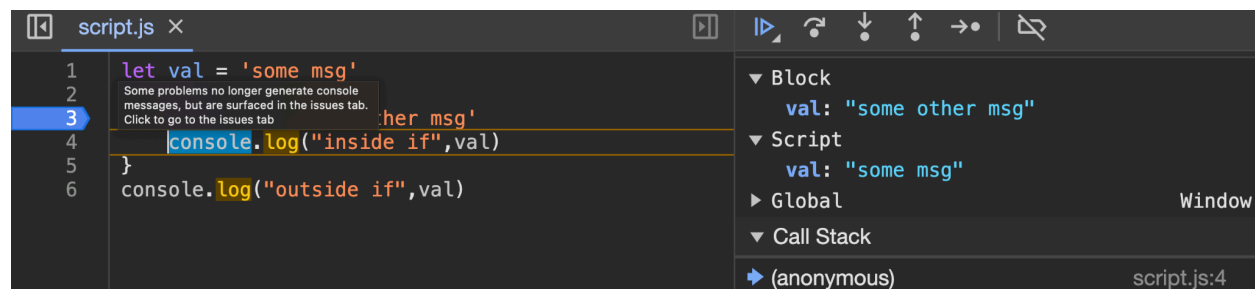
    let val = 'some other msg' // shadowing

    console.log("inside if",val)

}

console.log("outside if",val)

```



g.

h. Var declared variables are visible beyond the blocks

```

if (true) {

    let blockScopedVar = "I am inside an if block";

    console.log(blockScopedVar); // Works fine

```

```

}

// console.log(blockScopedVar); // ReferenceError:
// blockScopedVar is not defined

if(true) {
  var functionScopedVar = "I am var variable";
}

console.log(functionScopedVar); // Works fine

```

- i. What is meant by var declared variables are function scoped

```

function greet(){
  var msg = 'Hello World';
}

greet()

console.log(msg) // error

```

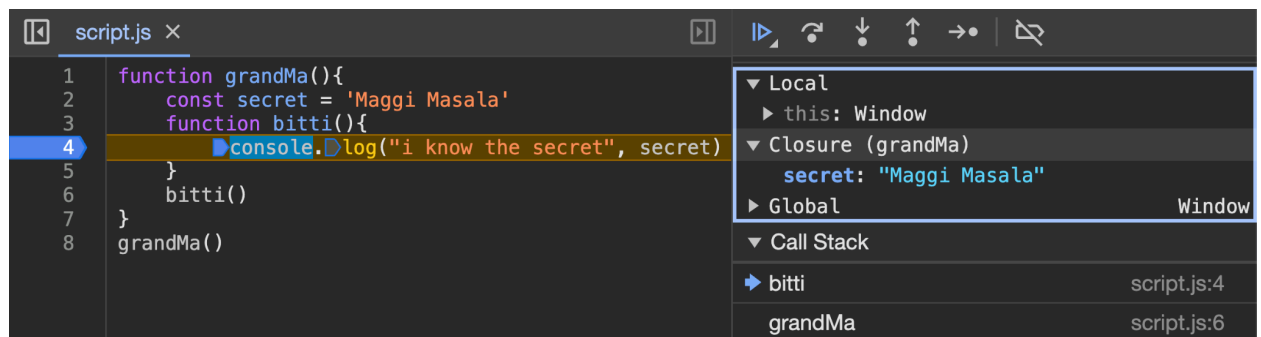
Closures

- a. In JavaScript, the inner function remembers the environment in which it was created. So, it can access the outer function's variables anywhere it's called
- b. in JavaScript, when the outer function has finished executing, the inner function still retains access to the outer function's variables via the closure.

Code

```
function grandMa() {  
  const secret = 'Maggi Masala'  
  function bitti() {  
    console.log("i know the secret", secret)  
  }  
  bitti()  
}  
grandMa()
```

1. When we invoke bitti, it first looks for secret in its memory and when it can't find it, it goes to its lexical parent and gets the secret from its parent



- 1.
2. So closures are nothing very complicated. **It is a function bundled with its lexical scope**
3. Returning a function
 - a. Now let's make things interesting

```
function grandMa() {  
  const secret = 'Maggi Masala'  
  function bitti() {  
    console.log("i know the secret", secret)  
  }  
  return bitti  
}
```

```
}  
const recipe = grandMa()  
  
// some other code  
  
recipe()
```

- b. Instead of invoking , we now return the function
- i. Functions are first class citizens in JS. they can be assigned to a variable, passed as params in another function or returned
 - ii. When grandMa is invoked, a new execution context is created
 - iii. After it is done, the execution context should have been destroyed and along with all the variables and functions inside it
 - iv. Then what happens when we call the recipe function
 1. It still prints the secret because it remembers its lexical scope
 2. The inner function remembers the **reference** of the variable

Question:

Create a function called `createCounter`. This function should contain a local variable `count` initialized to 0 and return another function. The returned function, when called, should increment `count` by 1 and return the new value. Each time `createCounter` is called, it should create a new scope and `count` independently.

```
function createCounter() {  
  // Your code here  
}  
  
const counter1 = createCounter();  
const counter2 = createCounter();  
  
console.log(counter1()); // Should return 1  
console.log(counter1()); // Should return 2  
console.log(counter2()); // Should return 1  
console.log(counter2()); // Should return 2
```

Solution:

```
function createCounter() {  
  let count = 0;  
  return function() {  
    count += 1;  
    return count;  
  };  
}
```

What happens when you move the `count` update inside the inner function