```
        console.log(a)
    }
}
```

⇒ 100 (If not found inside its lexical env, it tries searching in parent and so on...)

e.g.
```
const a=10 ⟶ In Script
{
    const a=20 ⟶ In Block
    {
        const a=30 ⟶ In Block
    }
}
console.log(a)
```

different blocks i.e.
→ Script a: 10
→ Block a: 20
→ Block a: 30

⇒ 10

Episode-10

Closures in JS

→ Closure : Function bundled with its lexical scope is known as closure.

→ e.g.

```
function x() {
    var a = 7
    function y() {
        console.log(a)
    }
    y()
}

x()
```

⇒ 7

→ So, here the function y() is bind together with its lexical environment. x() is the lexical parent of y(). So, y has access to x() variables and form a closure.

→ Check MDN docs for more info.

→ We can also return a function

→ Functions are so beautiful that when they are returned from another function, they still maintains their lexical scope. They remember where they were actually present.

→ Javascript is synchronous. After we call a function it is deleted from the execution context. All the variables, functions etc. are gone.

e.g.
```
function x() {
    var a = 7
    function y() {
        console.log(a)
    }

    return y  <-- we can return a function in JS
}

var z = x()  <-- when we call x(), its EC, variables everything is deleted. The returned y() func. gets stored in z. So, if we console log z, we get a function.
console.log(z)
z()
```

⟹
```
f y() {
    console.log(a)
}
7
```

But what if we call z, now, the var a=7 is deleted, the whole x() no longer exists, what will it return (console log)?. It will still log 7.
How? Here, comes closure into picture. So, when func. are returned from another func, they still remember / maintains their lexical scope. So, it remembers their was var a=7 and has strong binding.

When we return y, not just the function code was returned, but the closure was returned (Closure - func. ~~enclosed~~ along with its lexical scope)

→ We can also write

return function y() {
:
}

instead of return y().

→ Uses of Closure :

1) Module Design Pattern
2) Currying
3) Functions like "once"
4) Memoize
5) Maintaining state in async world
6) setTimeouts
7) Iterators
and many more.

✳ Episode - 11

✱ setTimeout + Closures Interview Question

→ setTimeout stores the function in a different place and attaches a timer to it. When the timer is finished, it rejoins the call stack and is executed.

→ Code:

```
function x() {
    var i=1
    setTimeOut(function () {
        console.log(i)
    }, 3000)
    console.log("Namaste JS")
}
x()
```

JS is synchronous. It won't wait 3 sec. for setTimeOut. It will move forward. When 3 s are completed, it runs.

⇒ Namaste JS
1

→ How to get this output?

1
2
3
4
5

1 after 1 s, 2 after 2 s, 3 after 3 s and so on...

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

When we use let, every time a new copy of variable is attached to the setTimeout So, for i=1, a copy is attached, for 2 different copy

When we use let, every time a new copy of variable is attached to the setTimeout function.

→ Because it is block scoped.

For i=1, setTimeOut
i=2, "
⋮

We cannot use var. If we use var then the value of i is global scoped. Even before the setTimeout starts, i will reach at the end of the loop everytime and will print 6.

6
6
6
6
6

(2) var with Closure

~~for~~ Without closure, the var reference gives the latest value as it does not retain the original value but rather has the reference, so any update in the value after timeout will be shown.

With closure, we can solve this problem. It was not working because i was refering to the same memory location [console.log(i) → to i in for loop]

So, we have to do something that i is supplied with a new value everytime.

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

We can use any name instead of x.

Here, there is a new copy of i everytime and so, we can achieve the desired output.