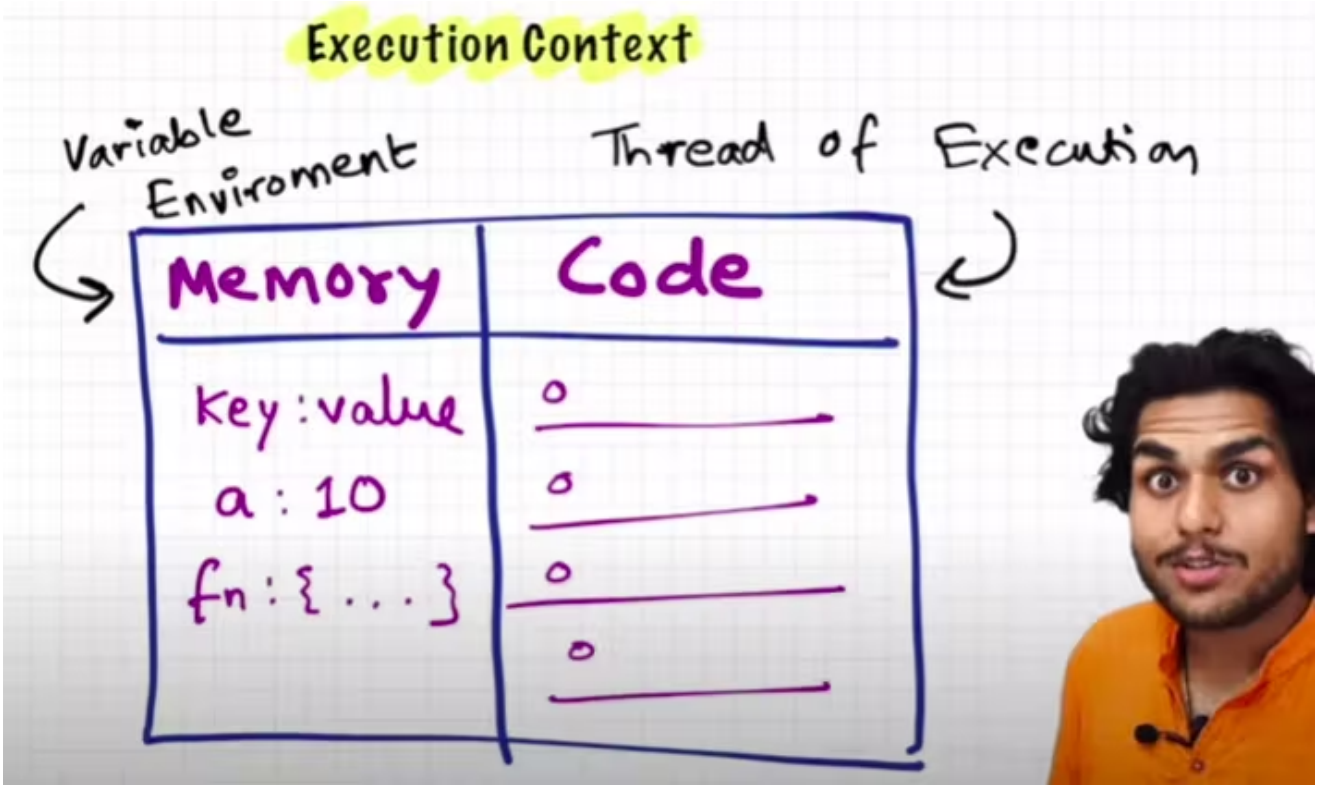
Execution Context:

1. Everything in JS happens inside the execution context. Imagine a sealed-off container inside which JS runs.
2. 

3. In the container the first component is ==memory component== and the 2nd one is code component
4. Memory component has all the variables and functions in key value pairs. It is also called ==variable environment==.
5. Code component is the place where code is executed one line at a time. It is also called the ==Thread of Execution==.
6. **JS is a ==synchronous, single-threaded language==**
7. Synchronous: one command at a time (?). follows a particular order of execution. That is, it goes from one line of code to the next and then to the next. It can't skip any, or jump to previous one.
8. Single-threaded: Execution happens on a single thread ^[f1] only.

Code Execution:

1. When a JS program is ran, a ==global== execution context is created.
2. The execution context is created in two phases. The **creation phase/ memory creation phase** and **code execution phase**
3. In the first phase (memory creation), JS will allocate memory to variables and functions.
4. Take this,

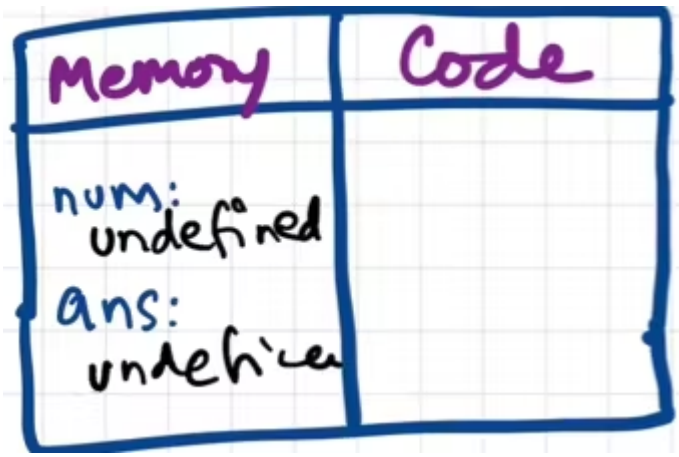
COPY

```
var n = 2;
function square(num) {
 var ans = num * num;
 return ans;
}
var square2 = square(n);
var square4 = square(4);
```

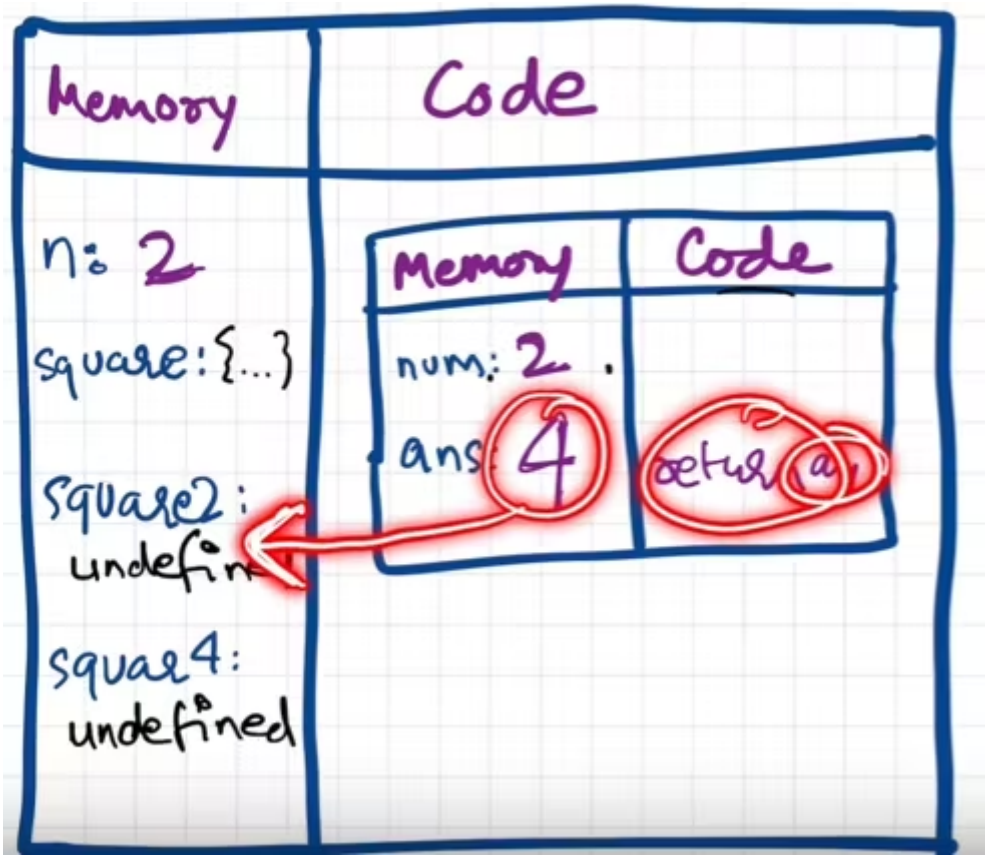
into consideration. Here, it goes to line one, and allocates a memory space for variable 'n' and then goes to line two, and allocated a memory space for function 'square'. When allocating memory for n it stores 'undefined', a special value for 'n'. For 'square', it stores the whole code of the function inside its memory space. Then, as `square2` and `square4` are variables as well, it allocates memory and stores 'undefined' for them, and this is the end of first phase i.e. memory creation phase.

5. Now, in 2nd phase i.e. code execution phase, it starts going through the whole code line by line. As it encounters `var n = 2`, it assigns 2 to 'n'. Until now, the value of 'n' was undefined.
6. For line 2-5, there is nothing to execute. As these lines were already dealt with in **memory creation phase**.
7. Coming to line 6 i.e. `var square2 = square(n);`, here functions are a bit different than any other language. A new execution context is created

altogether. In this new execution context, in memory creation phase, we allocate memory to num and ans the two variables. And undefined is placed in them.



8. Now, in code execution phase of this execution context, first 2 is assigned to num. Then `var ans = num * num` will store 4 in ans. After that, `return ans` returns the control of program back to where this function was invoked from.



. Afterwards, square2 will get 4 assigned to it. And now, this execution context which was created for function, will be deleted.

9. Same thing will be repeated for square4 and then after that is finished, the global execution context will be destroyed.
10. Now, it gets messy in a JS program as multiple execution Contexts are created. Sometimes, nested multiple times too. For managing this, JS maintains a ==callStack==.

Call Stack:

1. A stack which holds different executions contexts. At the bottom, there is the GEC [^f2]. And as the control goes from line to line, it creates/ pushes new execution contexts on top of the GEC. After that new execution context is finished, it get popped/ deleted from the stack and the control goes back to GEC [^f2]. Finally everything is popped and callStack is empty. So in essence, ==callStack maintains the order of execution of execution contexts==.
2. Unfortunately, callStack is known by following names.
    1. Call Stack
    2. Execution Context Stack
    3. Program Stack
    4. Control Stack
    5. Runtime Stack
    6. Machine Stack

Hoisting in JS:

1. Code:

COPY

```
getName();
console.log(x);
var x = 7;
function getName() {
 console.log("Namaste Javascript");
}
```

Output: `Namaste Javascript`
`undefined`

2. It should have been an outright error in many other languages, as it is not possible to even access something which is not even created (defined) yet.
3. However, JS executed the function and printed undefined ([[#Code Execution]]: point 4) for the variable. We know that in memory creation phase it assigns undefined and puts the content of function to function's memory. And in execution, it then executes whatever is asked. Here, as execution goes line by line and not after compiling, it could only print `undefined` and nothing else.
4. Above phenomenon, is not an error. However, if you remove `var x = 7;` then it gives error. `Uncaught ReferenceError: x is not defined`
5. `console.log(getName);` prints the content of the function.
6. Hoist in English means a tool for lifting or lowering the load.
7. If `getName` is declared as an arrow function (`var getName = () => {...}`) or like this,

COPY

```
var getName = function () {
 ...
}
```

then it is just like any other variable. Hence, when we print getName to console, it prints undefined instead of its content, and when we try `getName();` it gives error as it is not a function.

COPY

```
console.log(getName) //output: undefined
getName(); //output: Uncaught TypeError: getName is not a function
var getName = () => {
 console.log("Hello!");
}
```

8. **==Interview tip:==** If searched online, it is typically written that hoisting is just moving all the declarations to the top of the code. But that would be misleading as all of this is happening due to the 1st phase (memory creation phase) of the [[#Execution Context]].

Working of functions:

1. Code

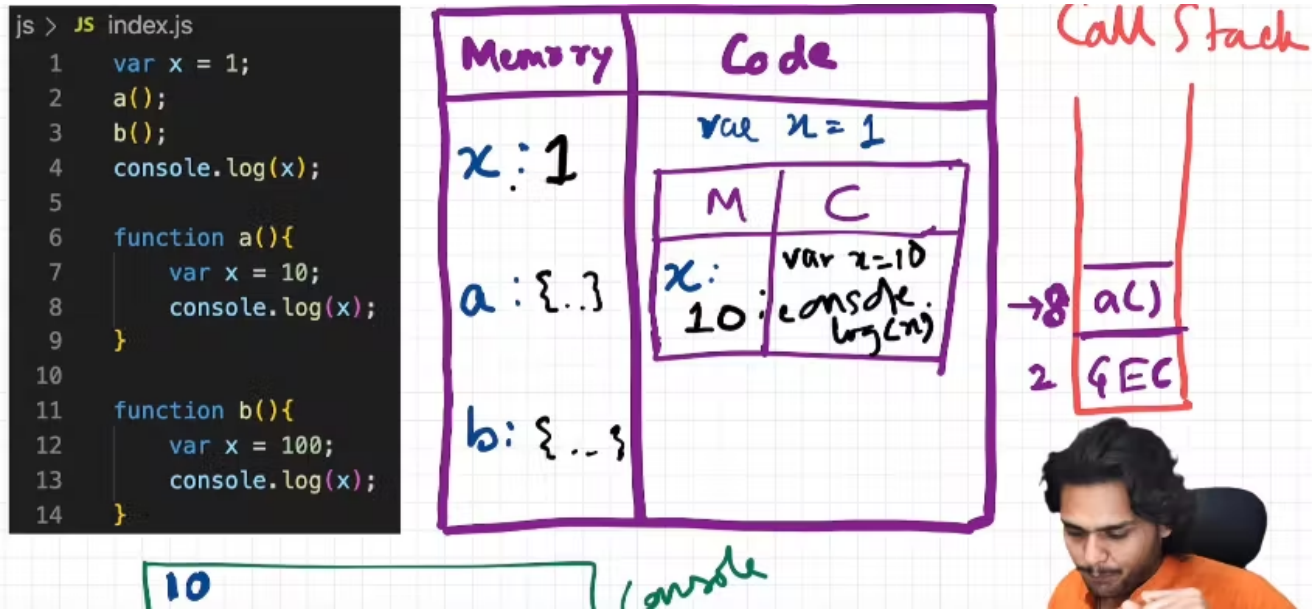
COPY

```
var x = 1;
a();
b();
console.log(x);
function a() {
 var x = 10;
 console.log(x);
}
function b() {
 var x = 100;
 console.log(x);
}
```

```
/*
output:
10
100
1
*/
```

2. Explanation:

    1. GEC [^f2] is created. memory assigned to x: undefined and a: points to function code, b: points to function code.
    2. Execution of GEC begins, at line 1 x: 1. At line `a()`, local execution context is created, x: undefined. Execution starts for local execution context, x: 10, 10 is printed on console. Local execution context is popped from call stack. Control goes to GEC.
    3. Same process is repeated for `b()`. 100 is printed on console.
    4. Control goes back to GEC. 1 is printed on console. GEC is popped.

3. 

Shortest JS program:

1. The shortest JS program is empty file. Because even then, JS engine does a lot of things.
2. As always, even in this case, it creates the GEC [^f2] which has memory space and the execution context.
3. JS engine creates something known as 'window'. It is an object, which is created in the global space. It contains lots of functions and variables.
4. These functions and variables can be accessed from anywhere in the program.
5. JS engine also creates a this keyword, which points to the window object at the global level.
6. So, in summary, along with GEC, a global object (window) and a this variable are created.
7. In different engines, the name of global object changes. Window in browsers, but in nodeJS it is called something else. **At global level, this === window**
8. If we create any variable in the global scope, then the variables get attached to the global object.

COPY

```
var x = 10;
console.log(x);
console.log(this.x);
console.log(window.x);
/*output:
10
10
10*/
```

Undefined and Not-Defined:

1. Undefined is something which has been allocated memory but any value is not assigned to it yet. However, not defined is something which doesn't even have any memory allocated to it yet, let alone any value being assigned to it.

COPY

```
var a;
console.log(a);
a = 11;
console.log(a);
console.log(x);
/*output:
undefined
11
Uncaught ReferenceError: x is not defined at app.js:5
*/
```

2. JS is a ==loosely types language==. That is, it doesn't attach its variables to any specific datatype. One variable can hold all different types of values.

COPY

```
var a;
console.log(a);
a = 11;
console.log(a);
a = "Hello";
console.log(a);
a = true;
console.log(a);
a = 11.87;
console.log(a);
/*output:
undefined
11
Hello
```

```
true
11.87
*/
```

Scope, scope-chain and Lexical environment:

COPY

```
a();
function a() {
    console.log(b);
}
var b = 10;
/*output:
undefined
10
*/
```

1. The above code indicates that somehow, function a is able to access the variable b which is not in its local scope but in the global scope.

COPY

```
/*CODE A*/
function a() {
    c();
    function c() {
        console.log(b);
    }
}
var b = 10;
a();
/*output:
10*/
```

```
/*CODE B*/
function a() {
    c();
    function c() {
        var b = 100;
        console.log(b);
    }
}
var b = 10;
a();
/*output: 100*/
```

1. Even in the above code A, 10 is printed. It means that within nested function too, that global variable can be accessed. However, in code B, 100 is printed. It means that local variable of the same name took precedence over a global variable.

COPY

```
function a() {
    c();
    function c() {
        var b = 100;
        console.log(b);
    }
}
a();
console.log(b);
/*OUTPUT:
100
Uncaught ReferenceError: b is not defined at app.js:9*/
```

1. The above code lets us know, that a function can access a global variable, but the global execution context can't access any local variable. **Note: At least in the above code, when the GEC [^f2] tries to print b, there is no 'b' in any of the execution contexts in the entirety of call stacks. The line `a()` is when the function 'a' is executed and after that, its execution context is popped of the stack, destroying variable 'b' in it as well. Now the question is, what happens if the GEC [^f2] tries to print 'b' before the execution of `a()`. Again, there is no b in the call stack as the execution context for function a has not even been created yet, and it gives error, but this time, it doesn't execute `a()` because perhaps JS stops executing after 1 error.**
2. **==TLDR; == An inner function can access variables which are in outer functions even if inner function is nested deep. In any other case, a function can't access variables not in its scope.**

let & const in JS:

Preview:

1. Are let and const hoisted in JS?
2. What is the temporal dead zone?
3. What is the difference between SyntaxError, ReferenceError and TypeError?

Are let and const hoisted in JS & the temporal dead zone?

Yes, they are. But in a different manner than var. And they are in the temporal dead zone until they are initialized.

COPY

```
// console.log(a, b);
console.log(c)
let a = 10;
```

```
const b = 1000;
var c = 100;
```

For the above code, all a, b, and c are assigned memory. But the difference is that c being a var, is attached to the global object, while a and c are assigned memory in a different memory space. This is the reason they remain inaccessible, and console.log throws `ReferenceError can't access a/ b before initialization`. This phenomenon is called the 'temporal dead zone', where temporal implies something relating to time. Basically, the time since a let or const is hoisted (assigned memory) till it is initialized with some value. As soon as let, const values are initialized they become accessible.

Errors:

ReferenceError:

When JS tries to access something which does not currently exist in the current scope, it throws a ReferenceError.

SyntaxError:

When a syntactically invalid code is tried to be interpreted, SyntaxError is thrown Let and const are so strict, that if any variable is declared twice, then no code is run at all, and it throws a SyntaxError. Which is not the case for var as it allowed for re-declaration.

COPY

```
let a = 10;
let a = 100;

// OUTPUT: Uncaught SyntaxError: Identifier 'a' has already been decl
```

TypeError:

A `TypeError` may be thrown when:

> an operand or argument passed to a function is incompatible with the type expected by that operator or function; or
>
> when attempting to modify a value that cannot be changed; or
>
> when attempting to use a value in an inappropriate way. const is stricter than let by not allowing re-assign a value to a variable which already has it.
> TypeError - MDN

COPY

```
const a = 10;
a = 100;

// OUTPUT: Uncaught TypeError: Assignment to constant variable.
```

Block scope and shadowing in JS:

What is a block?

COPY

```
{
}
```

These curly braces, is a block or a code-block. Why is it used? To call a bunch of statements as a single piece of code. It is also called 'compound-statement'. These are used in if-else, for, forEach, functions, etc. And, block-scoped tells us what are all the variables and functions which can be accessed in a particular block.

let & const are block scoped:

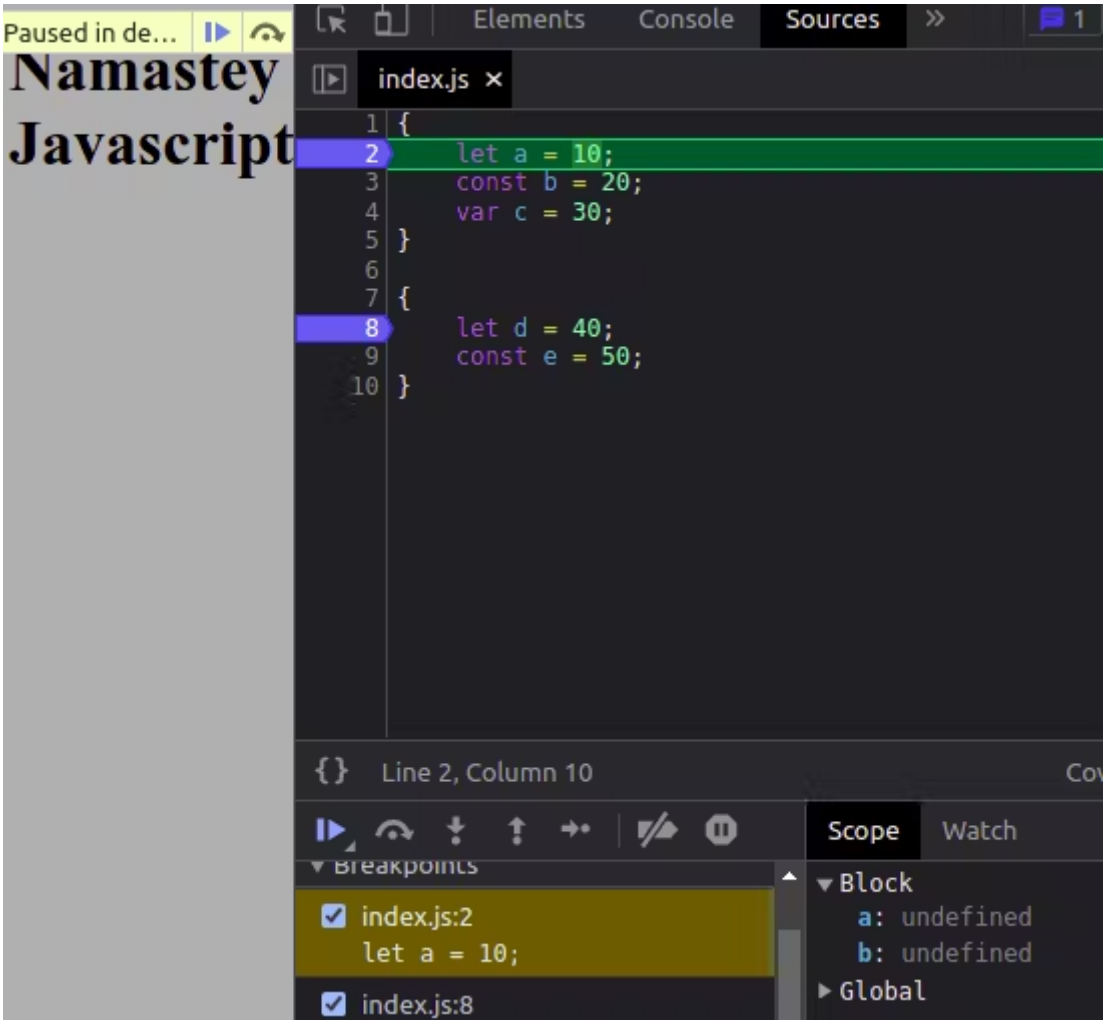
What are all the variables and functions inside of a block.

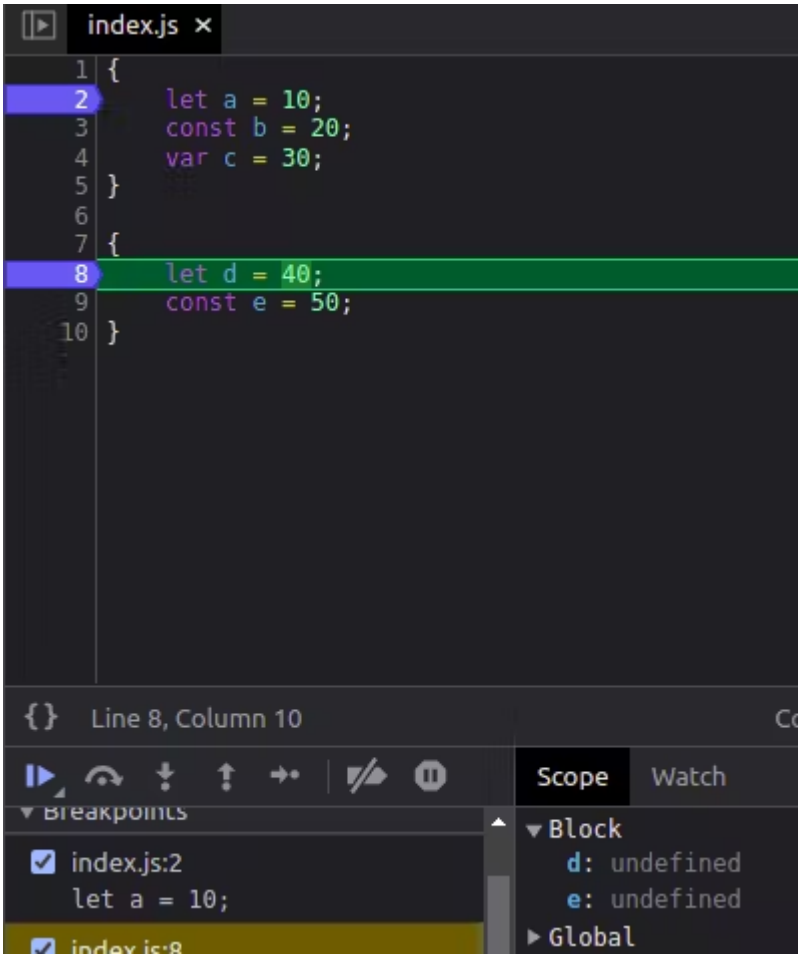
COPY

```
{
    let a = 10;
    const b = 20;
    var c = 30;
}

{
    let d = 40;
    const e = 50;
}
```

So, if you put debugger at `let a = 10` and `let d = 40`, then you'll see a and b undefined in the block scope and after resuming execution, again d and e in the block scope with value undefined.


Paused in de...
Namastey
Javascript
Elements
Console
Sources
index.js
1 {
2 let a = 10;
3 const b = 20;
4 var c = 30;
5 }
6
7 {
8 let d = 40;
9 const e = 50;
10 }
Line 2, Column 10
Scope
Watch
Breakpoints
index.js:2
let a = 10;
index.js:8
Block
a: undefined
b: undefined
Global


index.js
1 {
2 let a = 10;
3 const b = 20;
4 var c = 30;
5 }
6
7 {
8 let d = 40;
9 const e = 50;
10 }
Line 8, Column 10
Scope
Watch
Breakpoints
index.js:2
let a = 10;
Block
d: undefined
e: undefined
Global

shadowing:

COPY

```
var a = 10;
{
    var a = 100;
}
console.log(a);

// OUTPUT: 100
```

The var a from inside the block shadows the var a from outside the block. Since, var is globally scoped in this situation, it is attached to the global object (window for browser) and when it is initialized again, the value is changed in the global object.

COPY

```
let a = 10;
{
    let a = 100;
    var x = 10;
}
```

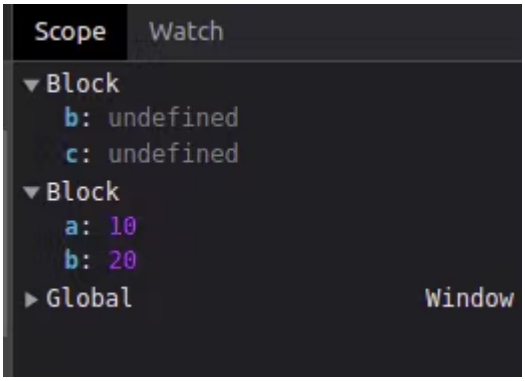
For the above code,

1. At line 1 x is assigned memory in global scope, and a is assigned memory in script scope.
2. At line 3, a in script scope is assigned value 10, and let a is assigned memory in block scope.
3. At line 4, x in global scope is assigned value 10.

We know that when there are two separate blocks, then memory is assigned in order inside the block scope. Now what would happen, if there were nested blocks?

COPY

```
{
    let a = 10;
    let b = 20;
    {
        let b = 30;
        let c = 40;
    }
}
```



Here, new a separate block scope is created

Illegal shadowing:

COPY

```
let a = 10;
{
    var a = 11;
}
```

Here, let a is assigned memory in script scope. And var a tried to shadow it. However, a SyntaxError: 'a' has already been declared, is thrown. However, it is possible to shadow a let variable using another let variable, and also var can be shadowed by a let. Like below:

COPY

```
let a = 10;
{
    let a = 20;
}
console.log(a);

// OUTPUT: 10 since let is block scoped
```

COPY

```
var a = 10;
{
    let a = 100;
    console.log(a);
}
console.log(a);
```

In summary, var can be shadowed by var, let and also const. But a let can only be shadowed by a const or a let, but not by a var.

Why is this the case? If a variable is shadowing something, then it can not cross the scope boundary of its *shadowee* (variable being shadowed).

COPY

```
let a = 10;
function sayHello() {
    var a = 20;
```

```
    console.log("hello");
}
```

Above code won't be a problem because a var's scope boundary ends at its current execution context Scope of a var - MDN .

Scope rules are same for arrow functions and function declarations.

Closures:

1. JS has a lexcial scope environment. If a function needs to access a variable, it first goes to its local memory. When it does not find it there, it goes to the memory of its lexical parent. Below code tells us a demo of that. So, function y along with its lexical scope (function x) would be called a closure.

COPY

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

/*OUTPUT: 7*/
```

1. As Akshay puts it, **a function bound together with its lexical environment is called closure**.
2. Now, in the below code, definition of function y will be printed on the console. Also, at line *, the function x vanishes from the call stack and there is no record of it. But, since x itself returns y, something interesting happens.

When function y is returned, the whole closure of y is returned and not just the function. Hence, at lin #, 7 is printed.

COPY

```
function x() {
    var a = 7;
    function y() {
        console.log(a);
    }
    return y;
}
//line *
var z = x();
console.log(z);
//several lines of code
//line #
z();
/*OUTPUT: 7*/
```

1. **Functions when returned, still maintain their lexical scope, i.e. the whole closure is returned.**
2. Uses of closures:
    1. Module Design Pattern
    2. Currying
    3. Memoize
    4. etc.

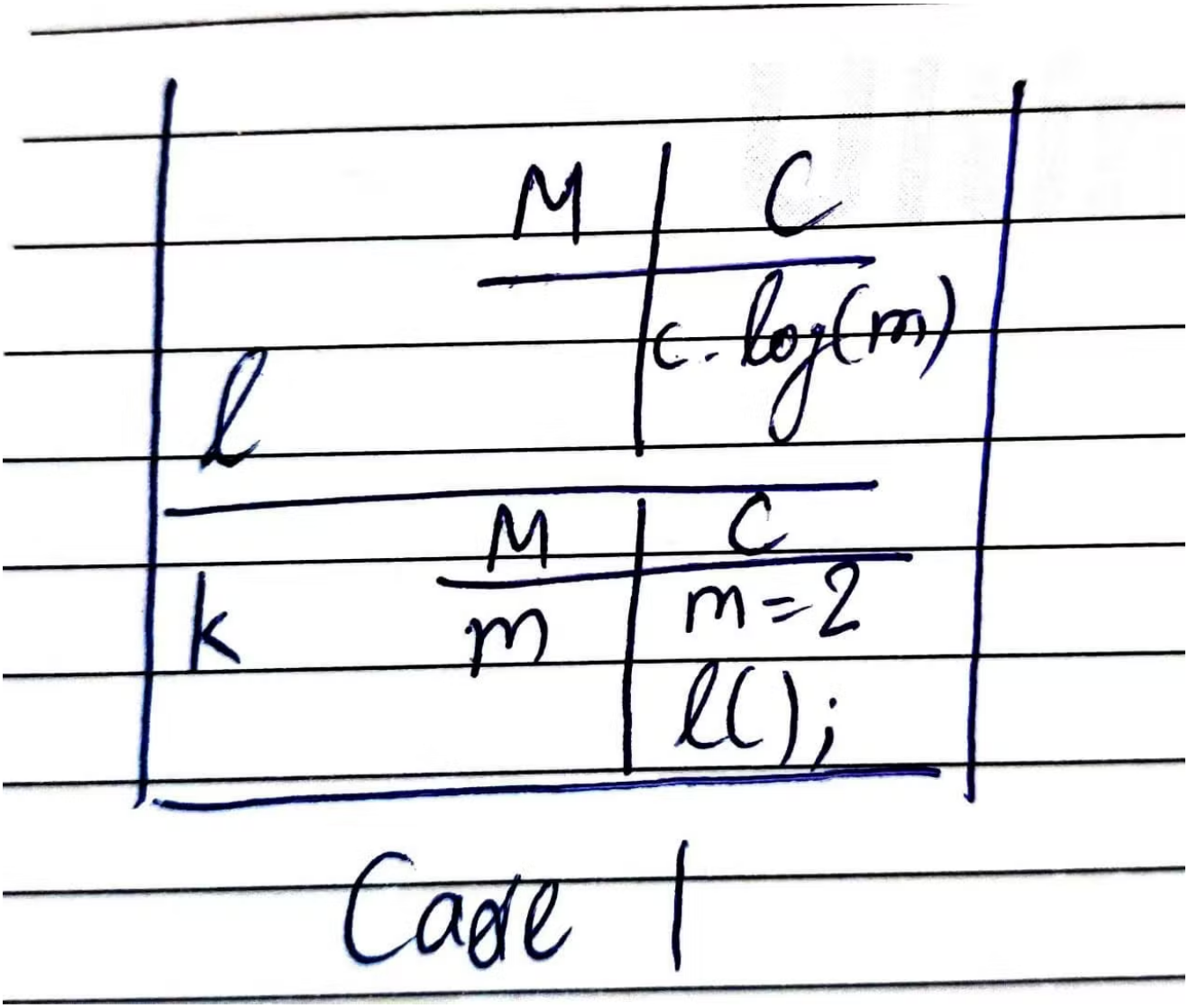
Questions:

1. What is the difference between arguments and parameters?
   Ans: StackOverFlow Answer

2. Is it possible for one function to access a variable which is only declared in another function?

COPY

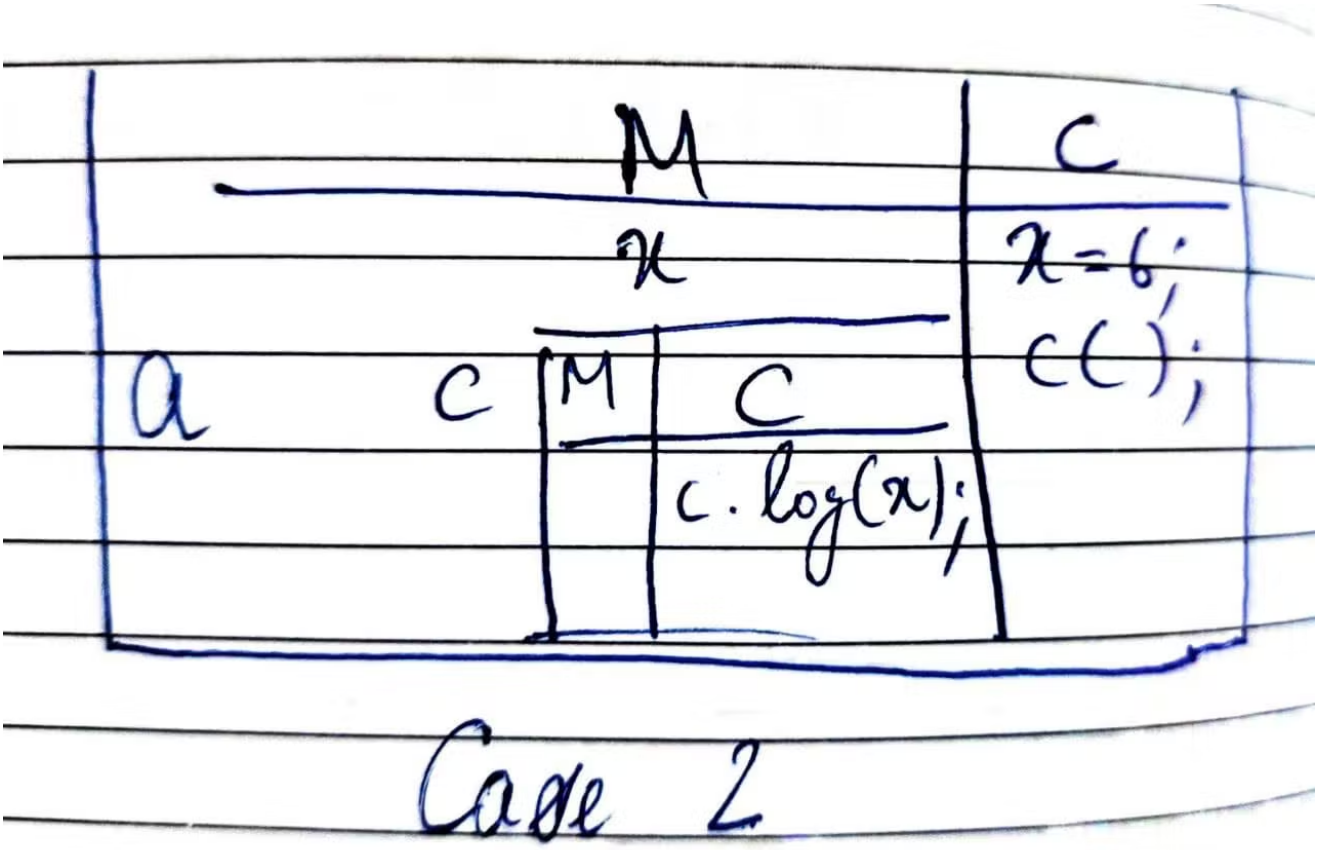
```
/*CASE 1: Seperate non-nested functions*/
function k() {
 var m = 2;
 l();
}
function l() {
 console.log(m);
}
k();
/*OUTPUT:
Uncaught ReferenceError: m is not defined
 at l (app.js:30)
 at k (app.js:27)
 at app.js:32*/
```



In the above case, functions 'l' and 'k' have non-nested execution contexts. When 'l' tries to print m, it can't find it. Neither in local scope nor in the scope in which local scope exists, i.e. the global scope.

COPY

```
function a() {
 var x = 6;
 c();
 function c() {
     console.log(x);
 }
}
a();
/*OUTPUT:
6*/
```



In 2nd case, x is declared and initialized before the execution of function c. Now, when function c tries to print x, it finds x in its (dare I say) parent scope, and 6 is printed on the console.

3. Is it possible for a function nested deep to access outer function's variables? Yes, indeed! It is possible for any function to access a variable which lies in outer execution context(s)

COPY

```
function a() {
 var d = 10;
 b();
 function b() {
     c();
     function c() {
         console.log(d);
     }
 }
```

```
}
a();
/*OUTPUT:
10*/
```

Footnotes:

[f1]: A thread is a path of execution within a process. A process can contain multiple threads. more reading

[f2]: Global Execution Context

JavaScript

MORE ARTICLES

Saksham Bhatt