→ JS is a loosely-typed / dynamically-Typed / Weakly Typed language.
This means that it allows a variable to hold any type of value at any time. We need not specify any data type of the variable

```
var a = 8; → a holds a number
a = a * 2;
console.log(a); //OP: 2
a = "$" + String(a); ⟶ now a holds a string
console.log(a); //OP: "$2"
```

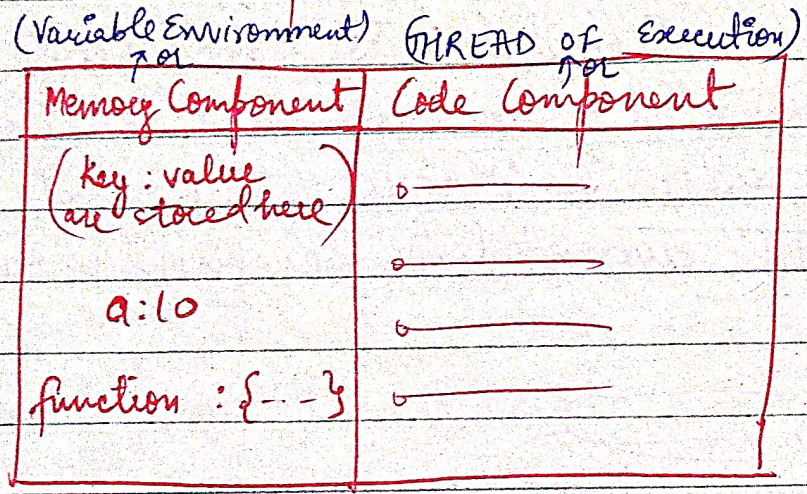
2.3.22 Namaste नमस्ते JavaScript जावास्क्रिप्ट

Episode - 1

How JS works & Execution Context

→ Everything in JS happens inside an Execution Context

* Imagine execution context to be a big box in which the whole JS is executed.
It has 2 components



* Memory Component has a fancy name "Variable Environment". It is the environment where all the variables & functions are stored as key-value pairs.

* Code Component has a fancy name "Thread of Execution". It is the place where JS code is executed. The entire code is executed line-by-line (one line at a time).

V. V. imp

"JavaScript is a Synchronous Single-Threaded language"

Synchronous : JS executes in a specific order.

Single-Threaded : JS can execute one command at a time.

Synchronous Single-Threaded : JS can execute one command at a time and that too in a specific order. So, it goes to the next line only after the current line has finished executing

~~Episode~~ Episode - 2

What happens when you run JS code?

* When you run a JS code, an execution context is created.
* Eg : (Code Example for Execution Context Creation)

The creation of Execution Context takes place in 2 pheses :

→ (1) Memory Creation Phese
→ (2) Code Execution Phese

Eg: ① var n = 2;
② function square(num) → this is the parameter
③ {
④ var ans = num * num;
⑤ return ans;
⑥ }
⑦ var square2 = square(n); → this is the argument
⑧ var square4 = square(4);

→ In this phase, JS skims through entire code line by line & allocates memory to variables & functions

Memory Creation Phase: In this phase, memory is given to the variables and functions.

→ Memory is allocated to variable n and a special value undefined is given to it.

Special case of functions (they are stored into execution context)

→ In next line of code, it encounters a function square() and allocates space to it. Actually the entire function body is copied into the memory component.

→ Similarly for square2 & square4, memory is allocated to them and they are given a special value 'undefined'.

Memory	Code
n: undefined	
square: {- - - -}	
square2: undefined	
square4: undefined	

Code Execution Phase: In this phase, after the memory has been allocated to all variables & functions, JS again goes through entire code and executes the code line by line.

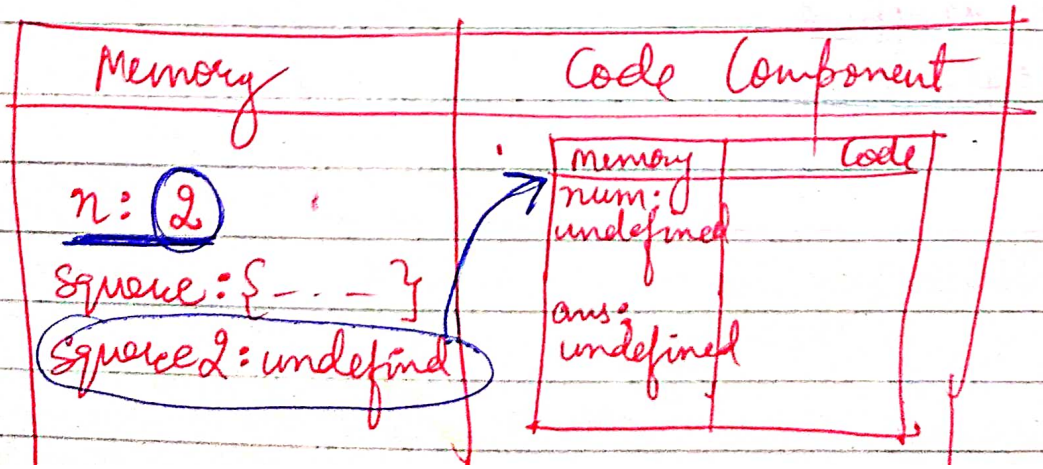
(i) * when `var n = 2` is read, JS actually places 2 as value of n in the memory (in place of undefined)

(ii) * In next line, JS (from line ② to ⑥), it finds nothing to execute because of function definition.

(iii) * It moves to line no. ⑦ and `var square2 = square(n)` is read

Here, function square() is being invoked
So here comes the interesting part:

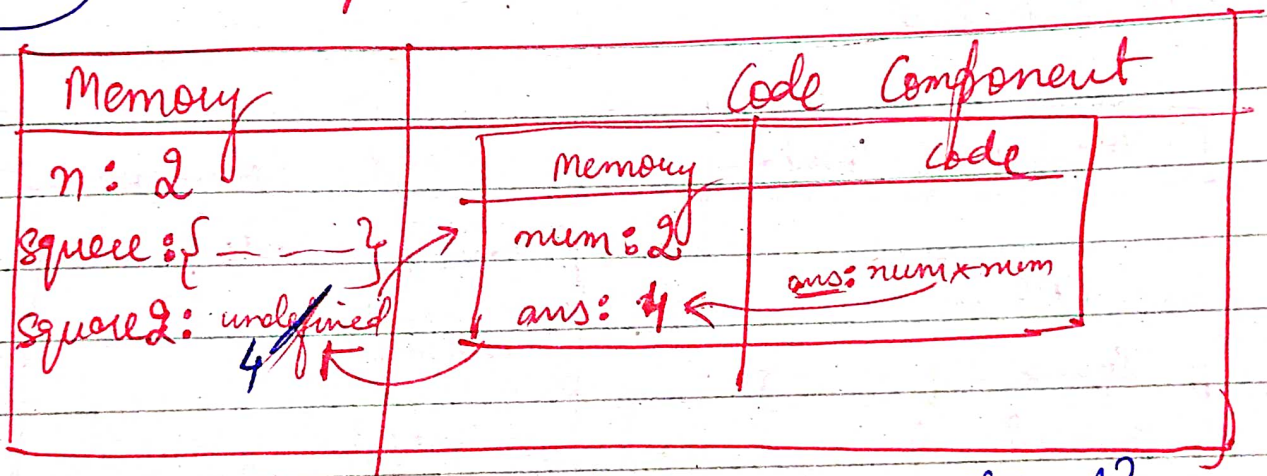
* Whenever a function is invoked in JS an entirely new execution context is created and again it takes two phases to create that Execution Context.



When function square(n) is invoked, an entire new execution context is created and it again goes through Memory execution & Code Execution phase.

→ In Memory execution, *num is allocated memory, and assigned undefined
*ans is also allocated memory and assigned undefined.

→ In code execution phase, num is assigned the value of n ie 2
ans is assigned the value num * num = 2 * 2 = 4



** What happens when return is encountered?

(1) It moves to next line inside function square() [return ans] line no: 5, so the control goes back to point where the function square() was invoked. So control goes out of the local execution context to the global execution context to square2.

* And square2 gets the value of ans ie 4
4 replaces the undefined

What happens when return is executed?

(2) Another thing that happens when return statement is executed is that after we come out of the local execution context, this execution context is deleted.

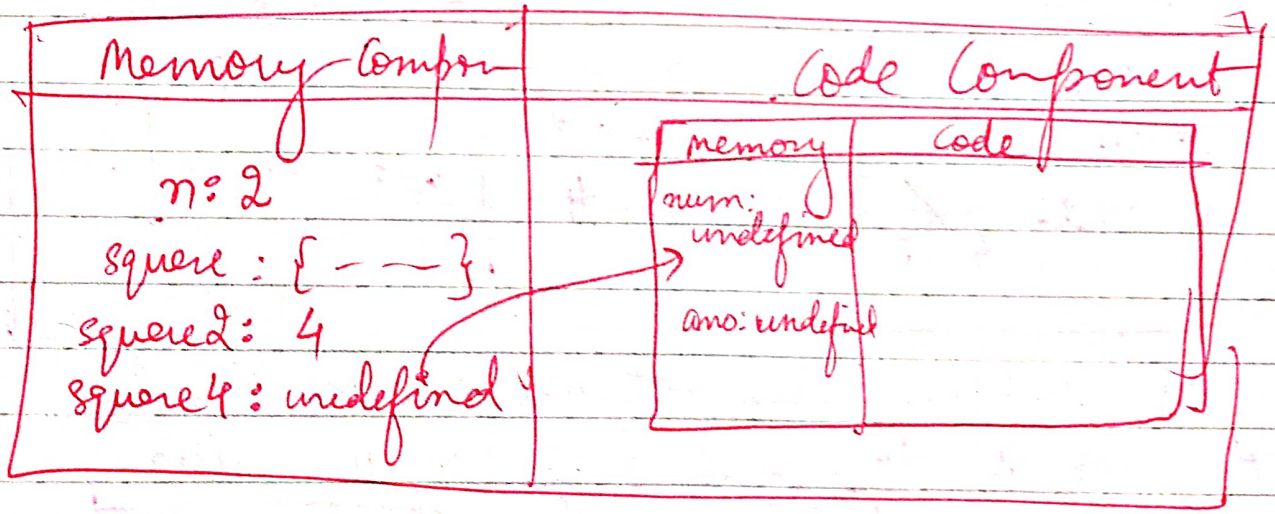
[So once we return from a function, its local execution context is deleted]

(IV) * Next in the Code Execution phase of Global Execution Context,

* we move to next line

ie `var square4 = square(4)`

Again function square(4) is being invoked so a brand new execution context will be created.



So we get into local Execution Context and ~~4 is passed to num~~ it goes through 2 pheses (Memory & Code)

→ In Memory Phase, num is allocated memory and assigned undefined. Same

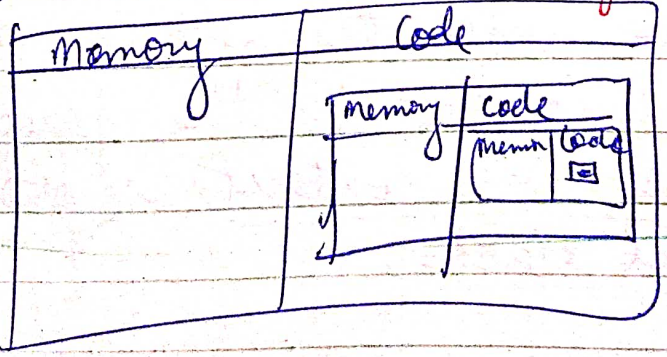
with ans, memory is allocated to ans & undefined is assigned to it.

→ In Code Execution Phase, num is assigned 4 (i.e. ~~value of~~ argument is passed to parameter) and ans gets value 4*4 = 16 (in place of undefined).

→ Then return ans is encountered, so control comes from local execution context to ~~global~~ the place where the function was actual invoked
And this execution context (local wala) is deleted.

(v) And we are done with the Code Execution Phase of global Execution Context, ~~so it becoz~~ the JS is done with entire program execution! So the entire Execution Context gets deleted.

Well, it is really difficult to do all this stuff. Suppose there is a function invocation inside function invocation and so on... & we would have execution context inside another execution context inside another execution context. So it might be very diff to create, delete, manage contents



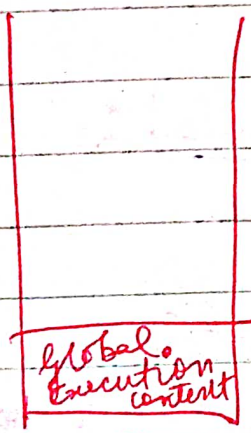
But JS handles this very beautifully :)

Call Stack in JavaScript

JS is beautiful

JS uses call stack to manage all the creation and deletion of execution contexts.

→ At the start of program execution, call stack gets the Global Execution Context (GEC)



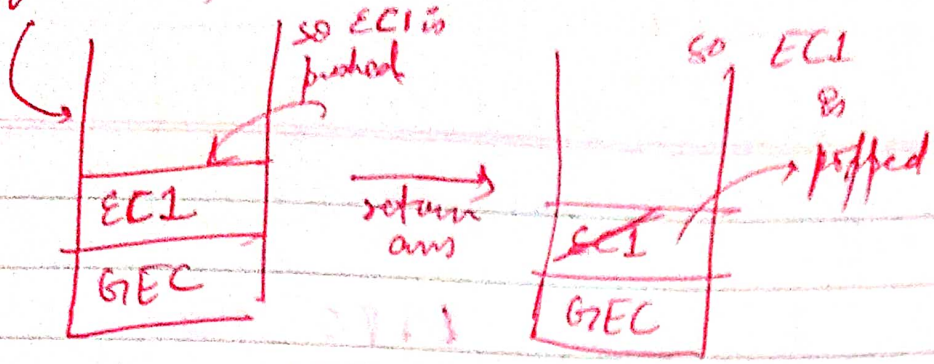
Call Stack

Whenever the program is run, the call stack gets the Global Execution Context into it, so it is present in the bottom of the stack.

→ Then later, whenever there is a function invocation, and new execution context is created, inside global execution context that new execution context is pushed on top of stack.

Once we encounter return statement ie return is executed in the function, that new execution context is popped out of the call stack.

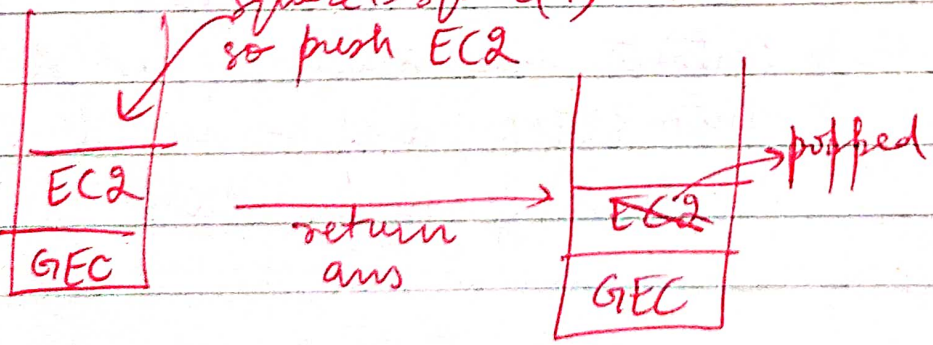
square(num) ie invokation



And after popping the new execution context, the control goes back to Global Execution Context

Eg:

```
var n = 2;
function square(num) // ~~Execute~~
{
  var ans = num * num;
  return ans; //control comes back to
}
var squared2 = square(num); //// new execution context is made & control goes to square() and the new context.
var square4 = square(4);
```



→ At the end, when entire program is executed, the GEC (Global Execution Context) is also popped and the call stack becomes empty.

"Call Stack Maintains the Order of Execution of Execution Contexts"

As we have seen previously (LIFO) (the new execution contexts are executed before the previous (older) execution contexts.

"Call Stack is known by many fancy names"

1. Execution Context Stack —
2. Program Stack —
3. Control Stack —
4. Runtime Stack —
5. Machine Stack —

} these are all the same thing

~~Hoisting~~

Refer to the Source in Chrome developer tools to see the CallStack.
Notice how the execution contexts are added (pushed into stack) and removed (popped) from the stack.

Episode - 3

3.3.22

Hoisting in JS

Eg:

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

O/P: Namaste JS
7

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

// we are trying to access getName() & x before initialising them

O/P: Namaste JS → from getName()
undefined → from x

Notice, function getName() is called before it is defined but it still prints "Namaste JS" ~~and same~~ and for x, we are trying to console x before initialising it and this prints undefined in output

This phenomenon is something peculiar to JS. Other languages might flag an error while JS still gives O/P. This is becoz of a concept known as "Hoisting"

Hoisting is a phenomenon by which you can access variables and functions even before initialising them. This won't give any error.

7 Functions Hoisting Behind the Scenes

Eg:

```
console.log(x);
console.log(getName);   // note, we are not invoking getName(), we are just consoling it on o/p.
var x = 7;
function getName()
{
    console.log("Namaste JS");
}
```

o/p:

```
undefined   // from x
getName()   // from getName
{
    console.log("Namaste JS");
}
```

Notice the output. When (x) is consoled, it gives undefined whereas when getName() is consoled, it prints the entire function definition.

This is because weird in JS. But actually, this happens because of the creation of Execution Context.

Let's understand 2 phases of Execution Context Creation

1. Memory Creation Phase: Here, in this phase, memory is allocated to

variables, and in case of functions, the entire function definition is copied to Execution Context.

Brief of whole

So for this code, even before the code execution starts, memory is allocated to x and undefined is assigned to it. Similarly, function getName() gets the entire function definition copied into it.

Memory	Code Component
x: undefined getName: {...}	

(2) Code Execution Phase: In this phase, JS runs each line of code (line-by-line)

1st line: * console.log(x) → prints undefined

2nd line: * console.log(getName) → prints entire function definition

3rd line: var x = 7; → x is assigned 7 (in place of undefined)

4th line: function getName() → The function definition is encountered, so nothing has to be done!

{

}

** So because of creation of Execution Context, Hoisting actually comes into picture.

Difference b/w Undefined & Not Defined

Undefined is the special keyword in JS that is assigned to the variables before they are initialised any value. See the case of x in our example

Not Defined: It is an error that shows up when we are trying to access a variable that has not been allocated memory space ie any variable that has not been declared

Eg:

```
console.log(x)
console.log(getName);

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

O/P: ReferenceError : x is not defined

After Reference Error, the program crashes and no other output is available

getName() ; // for getName()

"Not Defined" error comes because in the execution context, we don't have any memory allocated to x or say x does not exist there in the memory. This is the time of Memory Allocation Phase/ Memory Creation Phase.

Memory	Code
getName: {---}	

In Code Execution phase, the JS reads each line of code and executes it.

1st line: console.log(x) → gives "not Defined" error

2nd line: console.log(getName) → gives the entire function getName

After 1st line, the program execution stops because of Reference Error.

because we are trying to access x which does not exist in the memory.

"What happens in case of Arrow Functions? Do they behave the same as Normal Function in case of Hoisting?"

Eg:

```
console.log(x)
console.log(getName);
var x=7;
var getName = ()=> // arrow function
    { console.log("Namaste JS");
    }
```

The Output: undefined (for x)
undefined (for getName)

We notice that console.log(x) gives undefined as output, and
console.log(getName) also gives undefined.
This is because the arrow function is considered as a variable, and just like any other variable, it is assigned undefined in the memory creation phase.

Memory	Code
x : undefined getName : undefined	

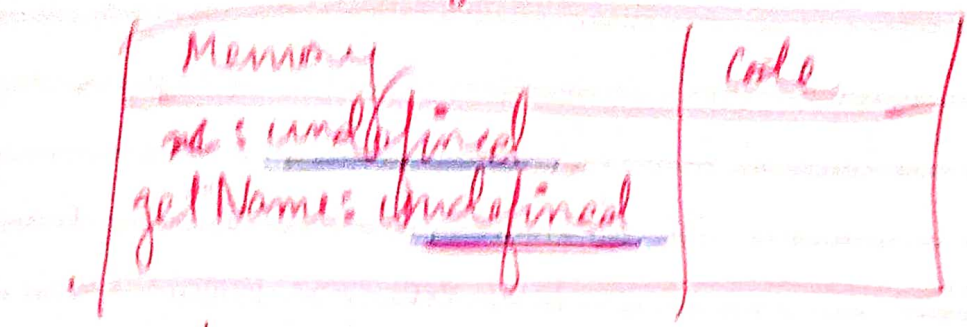
So in code execution phase, x is printed as undefined, and same for getName arrow function.
So they behave diff in case of hoisting as compared to (function getName() {}).

Another case
Eg

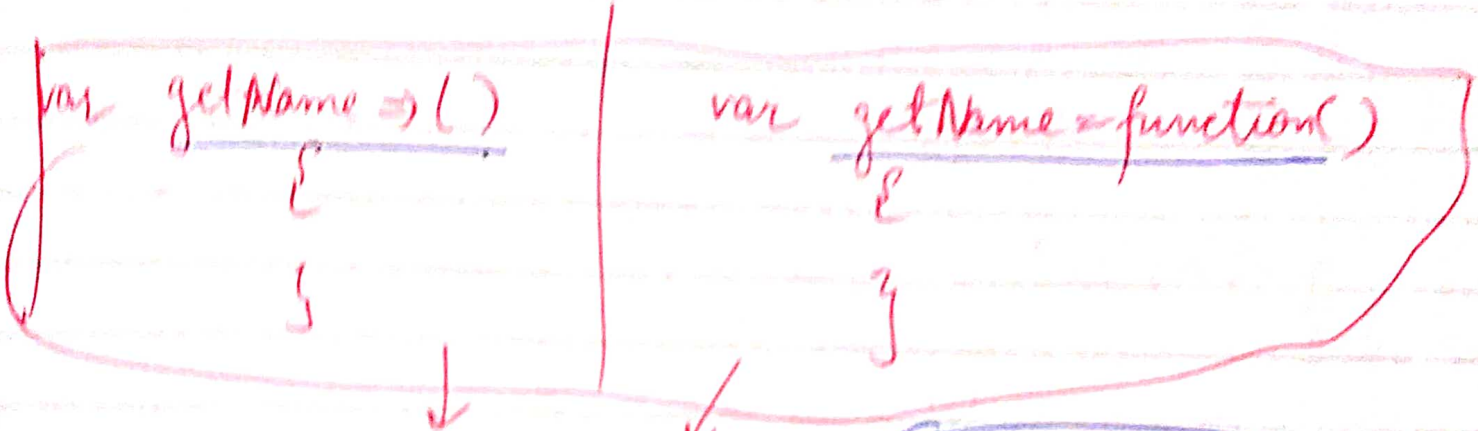
```
console.log(x);
console.log(getName);

var x = 7;
var getName = function () { // again, this function is also treated as a variable
    console.log("Namaste JS");
};
```

O/P: undefined (for var x)
undefined (for getName)



again getName() behaves just like a variable



In both these cases i.e. arrow function and another syntax of function, getName() behaves like a variable. So undefined is assigned to it.

Ref. Hoisting (functions vs variables) functions first

```
foo()
var foo;
function foo()
{
  console.log()
}
```

```
foo = function()
{
  console.log(1);
}
foo = function()
{
  console.log(2)
}
```

Episode - 4

"How Functions Work in JS, & ~~Variable~~ Environment"

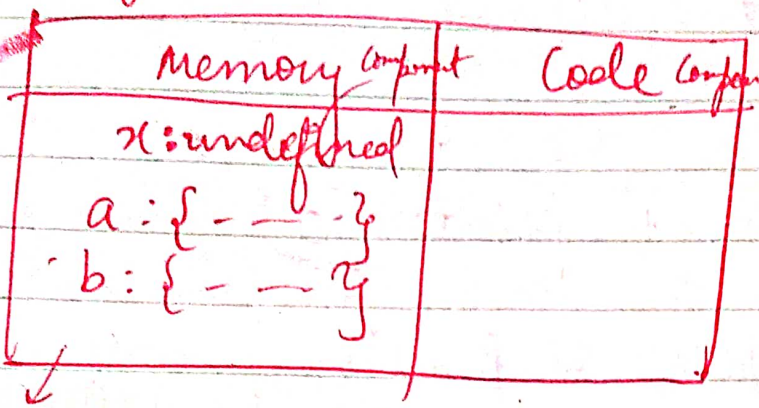
Eg:
```
① var x = 1;
② a();
③ b();
④ console.log(x);
⑤
⑥ function a()
⑦ {
⑧     var x = 10;
⑨     console.log(x);
⑩ }
⑪ function b()
⑫ {
⑬     var x = 100;
⑭     console.log(x);
⑮ }
```

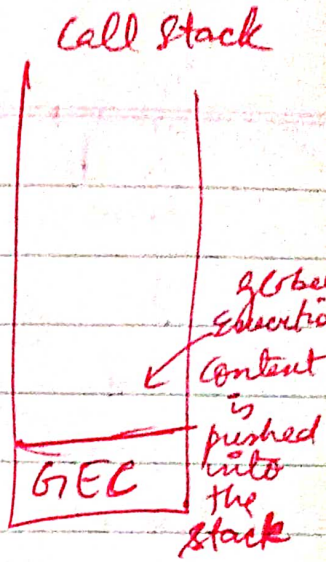
// we are accessing a(), b() before // defining them. But hoisting occurs // so we ~~can~~ can access them // before defining them.

O/P:
```
10
100
1
```

How are we getting this output?
Let's dive deep into what is happening behind the scenes.
(Actually all this is because of the Execution Context (creation))

① Memory Execution Phase:



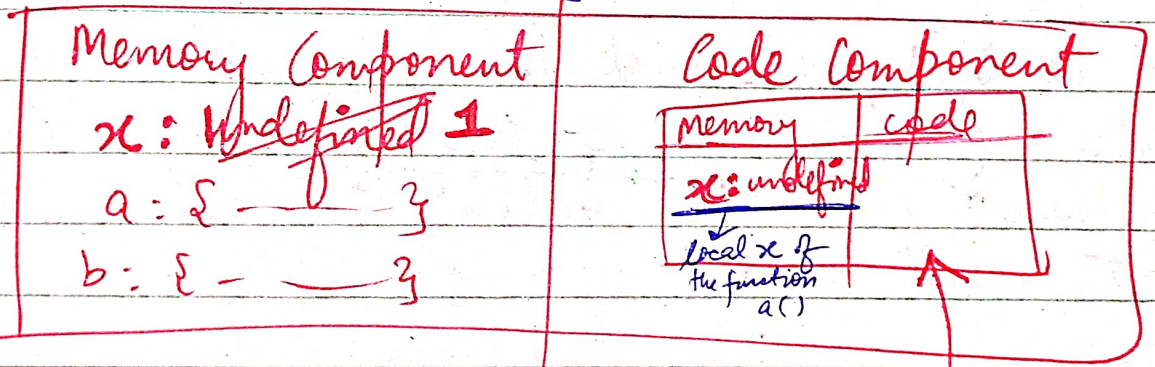


memory is allocated to x, a() & b() in the memory ~~co~~ execution phase.

→ x is assigned undefined.

→ a() & b() will point to the exact function code of a() & b() respectively.

② Code Execution Phase:



(i) 1st line is var x = 1 → so x is assigned 1

(ii) 2nd line → a() → i.e. a() is being invoked, so an entirely new execution context will be created for a(), and is pushed on call stack.

~~Now~~ Now that local execution context will also undergo memory & code execution phases.

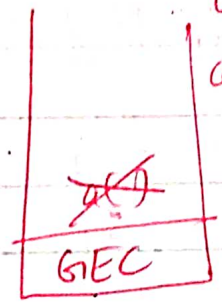
~~Now control goes to line number 8~~

In memory execution phase, that local x will be allocated memory & be assigned undefined. This x is totally different from the x of the global execution context.

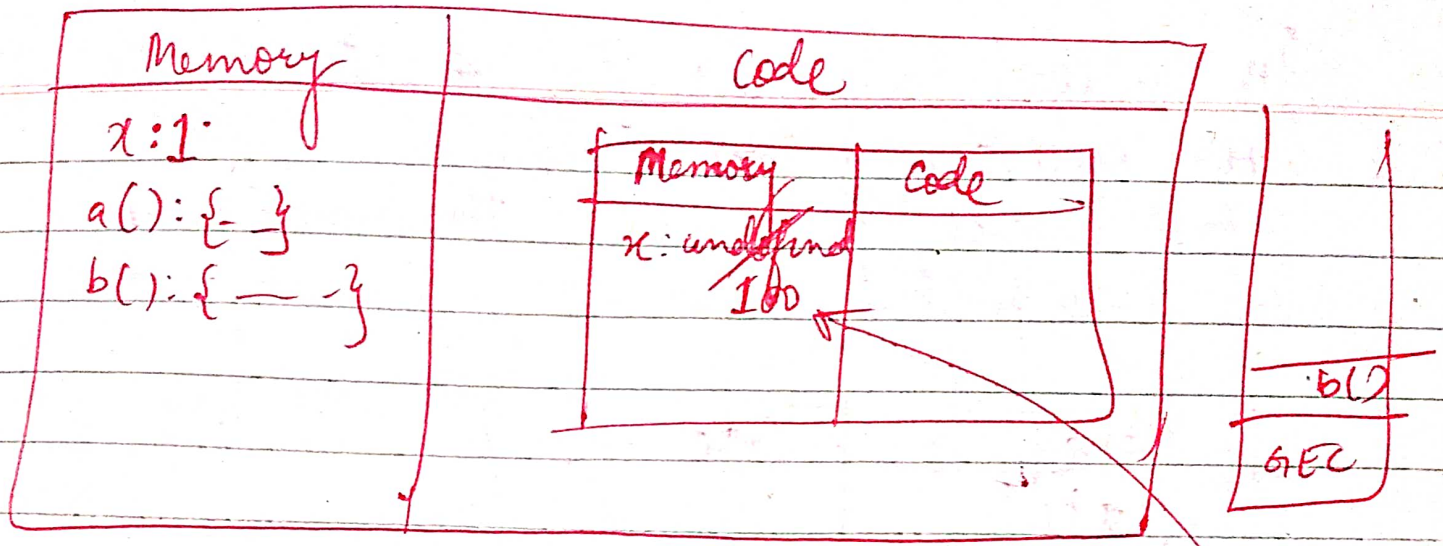
In code execution phase → x : 10 [control is at line number 8]

and console.log(x) prints 10 because JS engine finds the value of x in the local execution context i.e. in the local scope of the function, and if it finds x = 10 in the local memory & prints 10

Then we are done with a()'s execution, so the execution context of a() is deleted from the call stack:



(iii) The control ~~goes back to~~ line ~~no~~ number 3, where b() is being invoked. So again a new execution context is created & pushed into call stack.



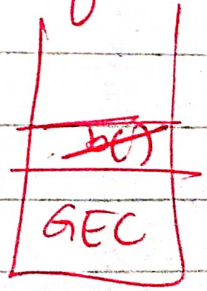
Now in memory execution phase, x is allocated memory & assigned undefined. This x is just limited to the local execution context of b() and is independent of the x of global execution context.

In code execution phase this local x is assigned 100 [our control is on line number 13.

Memory Next line → console.log(x) → prints 13

This is because JS engine finds value of x in the local scope of the function, and finds x=100 in the local execution context, and prints 100

Then we are done with b()'s execution, so the execution content of b() is deleted from the call stack.

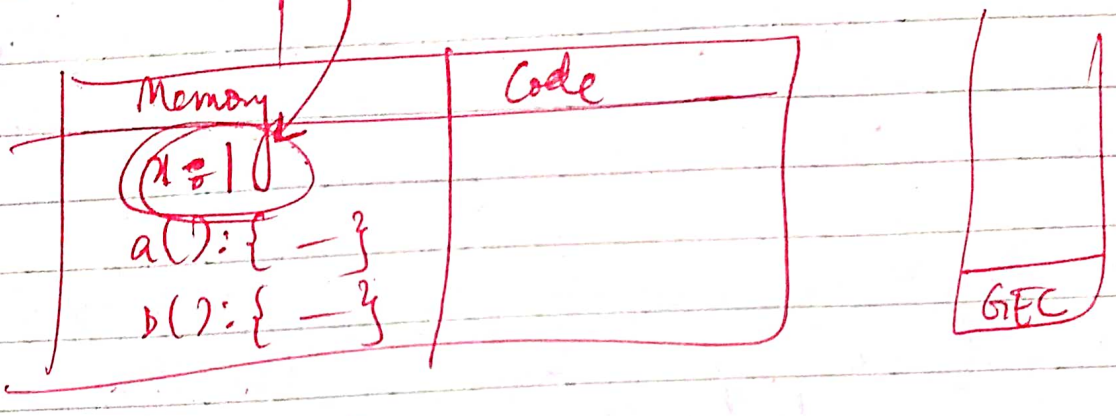


(iv) And control goes back to line number 4

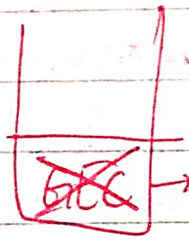
↳ console.log(x) will print 1

This is because JS engine finds the value of x

in the local memory, which is actually the global execution context itself.
And inside the GEC, the value of x = 1
so 1 is printed.



(v) After this, the control goes to line 6 (but nothing is to be done because there is a function definition) same with line (11), and the entire program execution is finished. So the GEC also gets deleted from the call stack



So output is: 10
100
1

:) (I hope you understood!)

```
var a;
console.log(a)
let a;
```

gives error
Uncaught SyntaxError: Identifier 'a' has already been declared.

4-3-22

Episode - 5

"Shortest JS Program"

(Concepts of Window & this keyword)

Do you know what is the shortest JS program?

→ It is the empty JS program!!!. Exactly! Actually, there is a lot happening behind the scenes even if the program has 0 lines of code, and there is nothing to execute.

"Behind the scenes of the Shortest JS Program"

(steps to observe the "behind the scene stuff"):

→ Open the Inspect and go to Source.

→ Place debugger on line 1 of script.js code file and Run the code (Ctrl+R).

① → We see that a Global execution context is created [Observe the call stack in inspect]

② In the Global execution context, some memory is allocated (although we haven't written anything in the code to which memory should be allocated) But still JS does its job.

→ (window is a global object created along with Global execution context in case of browsers

② JS also creates a window object (even if we had an empty code). This window object is a big object with lots of functions & variables. These functions & variables are created by JS itself in the Global space, so we can access them from anywhere in the code.

Created by js engine

Just like "window", js engine also creates a "this"

What is a global level/space: Anything that is not inside a function is in a global space.
Whenever a global execution context is created, a "window" object is also created.
In case of global level,"this" points to window object.

Eg:
```
var a = 10; // variable a is in the global space
function b()
{
    var x = 10;
}
console.log(window); // see that a will be present
console.log(window.a); // we can access the a variable by doing window.a
console.log(a); // to make things less cumbersome, we can even avoid writing window.a and just write a.
```

'a' gets attached to window , since it is in global space

Also, since at global level, 'this' poins to window, so, this.a also gives 10

The global object in Node.js is called 'global', while in browser, it is 'window'

* JS automatically gets to know that we are refering to the variable a in the global space.

* So, whenever we try to access any variable without (console.log(window.variablename)), and instead, we use the console.log(variablename), the JS engine assumes that we are refering to the variable in the global space.

If u try accessing a key in an object that does not exist, it will give undefined

(2) console.log(x); → gives not defined error

while console.log(window.x); → gives undefi

"What if I skip writing anything before a variable defined in the local space?"

Eg:
```
var a = 10;
function b()
{
  var x = 10;
}
console.log(x); // this gives us Reference Error & says that x is not defined
```

inside window object

as we have not written anything before x [like this.x]

This is because, JS assumes that we are refering to a variable x that exists in the global space. So it searches the global space but finds no x there (because x does not exist in global space, it is just in b()'s local space). So it returns an error saying "x is not defined"

window.a is undefined, since let variables are not attached to window.

Doubt → (1)

```
let a = 10;
var b = 100;
console.log(window.a);
console.log(window.b);
console.log(window.x);
console.log(x);
```

O/P: undefined
100
undefined
x is not defined

why O/P is undefined? x has never been allocated memory, still window.x gives undefined

Here we are trying to access x from window object, but x has never been declared. Still it gives undefined as O/P and not "x is not defined"