

→ JS is a loosely-typed / dynamically-Typed / weakly
Typed language.

This means 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 नमस्कार JavaScript जावास्क्रिप्ट

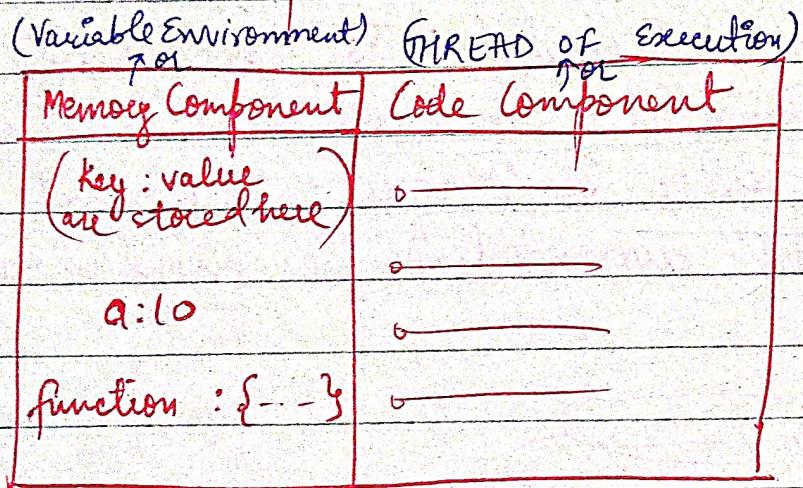
Episode - 1

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).

"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 phases :

- ① Memory Creation Phase
- ② Code Execution Phase

Eg: ① var n = 2;

② function square (num)

③ {

④ var ans = num * num;

⑤ return ans;

⑥ }

⑦ var square2 = square (n);

⑧ var square4 = square (4);

this is the parameter

this is the argument

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

→ In next line of code, we encounter 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	

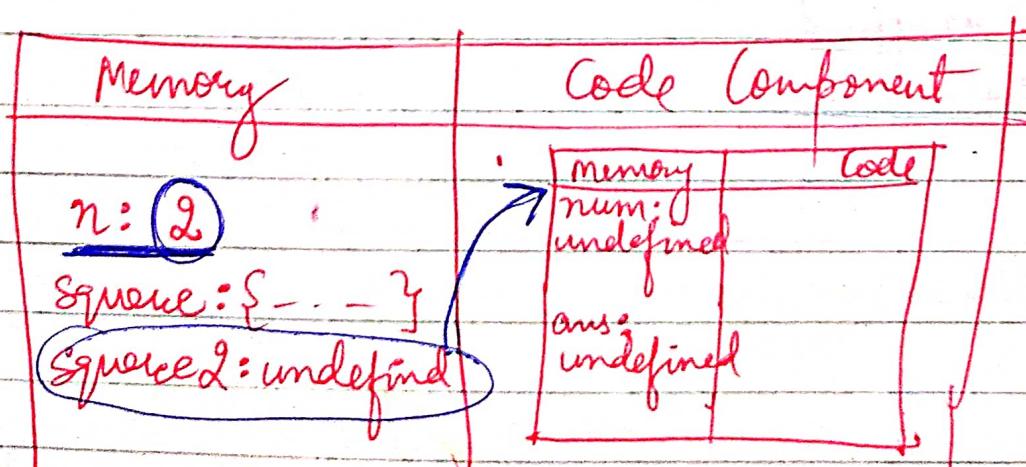
acc n: undefined
if n == 0
{
 return 1;
}
else
{
 return n * acc(n - 1);
}

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 at ~~the place of n~~ in the memory (in place of undefined)
- (ii) * In next line, if (from line ② to ⑥), it finds nothing to execute ~~and so~~ because of function definition.
- * It moves to line no. (7) and `var squared = 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 $num \times num = 2 \times 2 = 4$

Memory	Code Component	
<code>n: 2</code> <code>square: {</code> <code> <code>num: 2</code> <code> <code>ans: 4</code> <code> <code>ans: num * num</code> </code></code></code>	memory	code

* What happens when `return` is encountered?

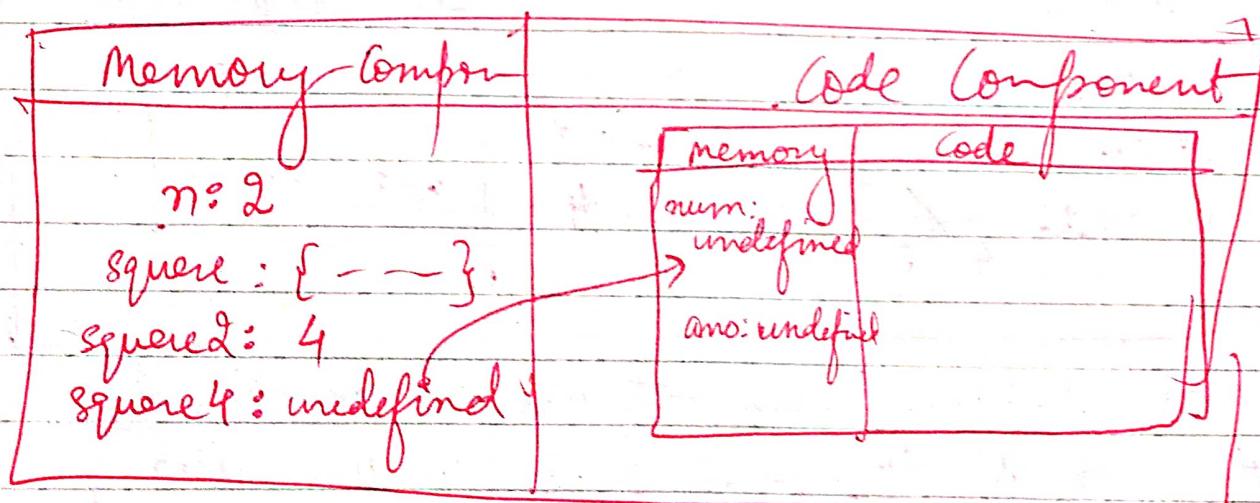
- (1) It moves to next line `return ans`
 line no: 5 inside function square so the control goes back to point from where the function `square()` was invoked. So control goes ~~out~~ of the local execution context() to the global execution context to `squared`.
 And `squared` gets the value of `ans` ie `4`
`4` replaces the undefined

What happens when return is executed?
↓
(second point)

- Q 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
→ When we move to next line, Content,
ie `var squared = square(4)`

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



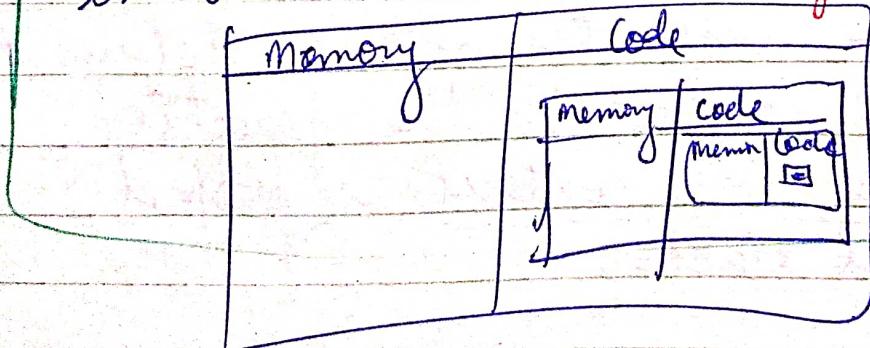
So we get into local execution context
and 4 is passed to num it goes
through 2 phases (Memory & Code)

→ 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 $4 \times 4 = \underline{\underline{16}}$ (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 var) is deleted.

(v) And we are done with the Code Execution Phase of Global Execution Context, so it becomes 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 contexts



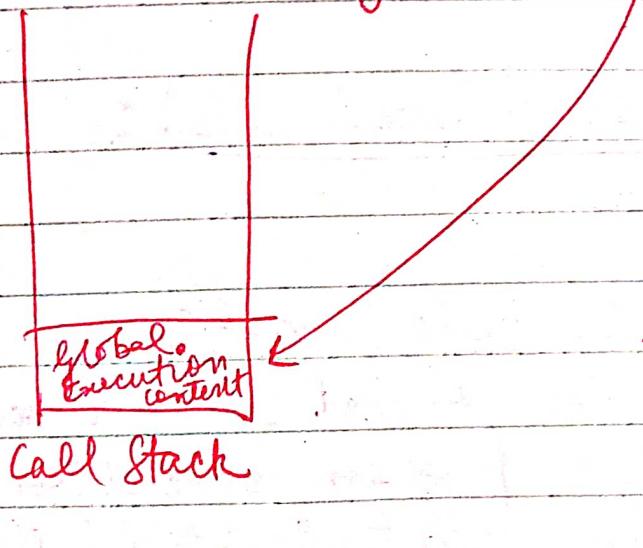
But JS handles this very beautifully :-)

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)

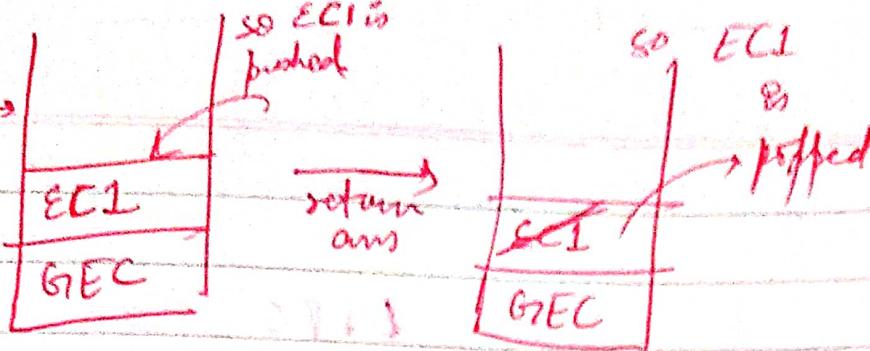


Whenever the program is run, the call stack gets the Global Execution Context into it, so it is present in the bottom of the stack.

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

square(num) re-invocation



And after popping the new execution context, the control goes back to global execution context

Eg:

`var n = 2;`

`function square(num) {`

`}`

`var ans = num * num;`

`return ans; // control comes back`

`}`

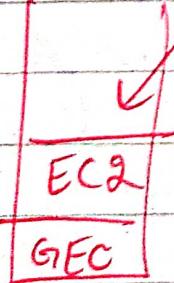
`var squared = square(n); // new execution context`

`var square4 = square(4);`

`square4 = square(4)`

so push EC2

✓



is made & control goes to square() and the new content.

→ 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 execution contexts.
(older)

"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 call stack.

Notice how the execution contexts are added (pushed into stack) and removed (popped) from the stack.

3.3.22

Episode - 3

Hoisting in JS

Eg:

```
var x = 7;  
function getName()  
{
```

```
    console.log ("Namaste JS");  
}
```

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

O/P: Namaste JS
7

```
getName(); // we are trying to  
console.log(x); // access getName() & x before initiali-  
var x = 7; -sing them.  
function getName()  
{  
    console.log ("Namaste JS");  
}  
  
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

This phenomenon is something peculiar to JS. Other languages in output might flag an error while JS still gives

O/P. This is because 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.

"Functions Hoisting Behind the Scenes"

```
Ex:  
1) console.log(x);  
   console.log(getName());  
   var x = 7;  
   function getName()  
   {  
     console.log("Namaste JS");  
   }  
3
```

O/P:
→ ~~get Name~~ undefined → from x
get Name() → from getName
{ console.log("Namaste JS"); }
3

Notice the output. When x is console, it gives undefined whereas when getName() is console, 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.

(not of memory)

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: { ... }	

L(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 in function 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:

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

var x = 2;

var getName;

function getName()

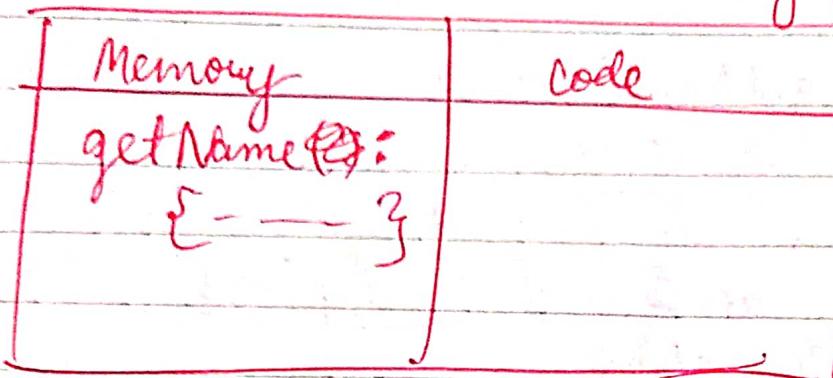
console.log("Namaste JS");

g

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

O/P: ReferenceError : x is not defined
getname(); // for getName()

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



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

1st line: `console.log(x)` → gives "notDefined" error
2nd line: `console.log(getName())` → gives the entire function getName

After 1st line,
the program stops
execution because of reference
error.

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

Q: 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());`
if `var x = 7;`
`var getName = () => console.log("Nameste JS");`

The output : undefined (for `x`)
undefined (for `getName`)

We notice that `console.log(x)` gives undefined
~~as output~~ 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
<code>x: undefined</code>	
<code>getName: undefined</code>	

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.
Another case (to function `getName()`)

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

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

O/P undefined (for `n`)
undefined (for `getName()`)

Memory	Code
<code>n = undefined</code> <code>getName() undefined</code>	

again `getName()` behaves just like a variable

var getName => ()
 {
 }
 {

var getName = function()
 {
 }

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

Refer Hossny. (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 = 0;

⑨ console.log(x);

⑩ }

⑪ function b()

⑫ {

⑬ var x = 100;

⑭ console.log(x);

⑮ }

O/P:

10

100

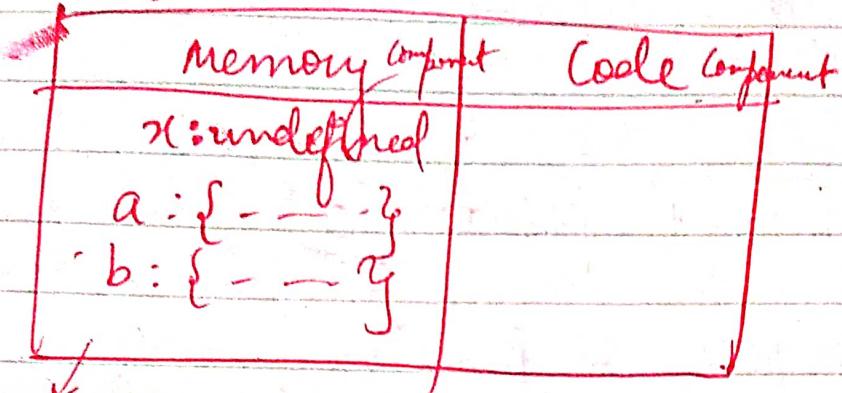
1

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:



Call Stack

GEC

Global
Execution
Content
is pushed
into the
stack

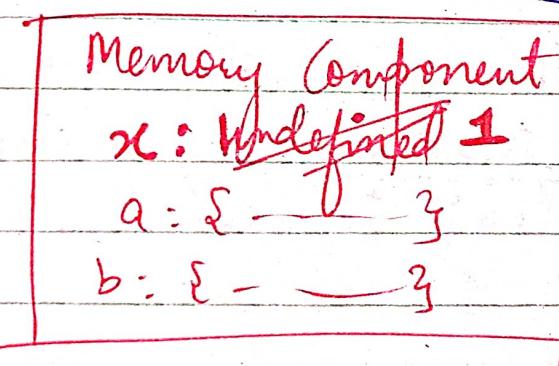
memory is allocated

to `x`, `a()` & `b()` in the memory execution phase.

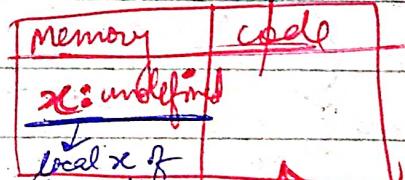
→ `x` is assigned `undefined`.

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

② Code Execution Phase:



Code Component



call stack

`a()`
GEC

- The 1st line is `var x=1` → so `x` is assigned `1`
- 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.

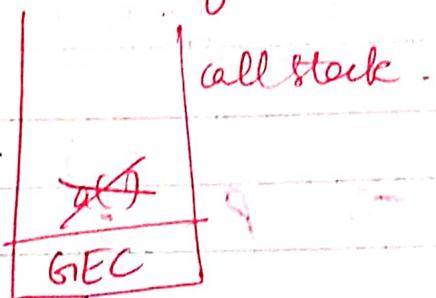
Now that local execution context will also undergo memory & code execution phases.

~~Now control goes to line number 8~~

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

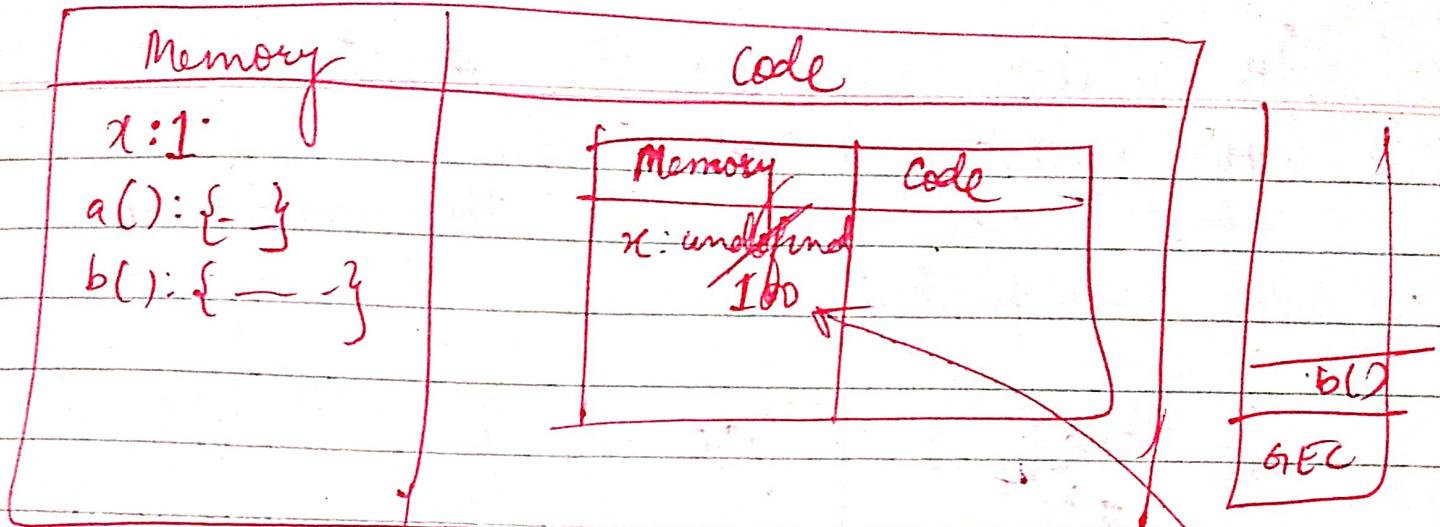
In code execution phase $\rightarrow x = 10$ [Control is at line number 8] and `console.log(x)` prints 10 because JS engine finds x in the local execution context i.e. in the local scope of the function, and if it finds $x = 10$ in the local memory & prints 10

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 ~~number 3~~ where b() is being invoked. So again a new execution context is created & pushed into call stack.





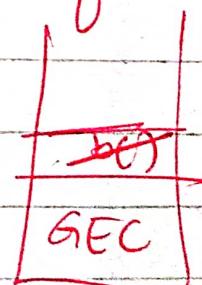
Now in memory execution phase, x is allocated memory & assigned undefined. This x is just limited to the local execution context of $b()$ and is independent of the x of global execution context.

In code execution phase, x is assigned 100
 (our control is on line number 13). ~~13~~

Memory Next line \rightarrow `console.log(x)` \rightarrow prints 13
 This is because JS engine finds 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 context of $b()$ is deleted from the call stack.

(iv) And control goes back to line number 4

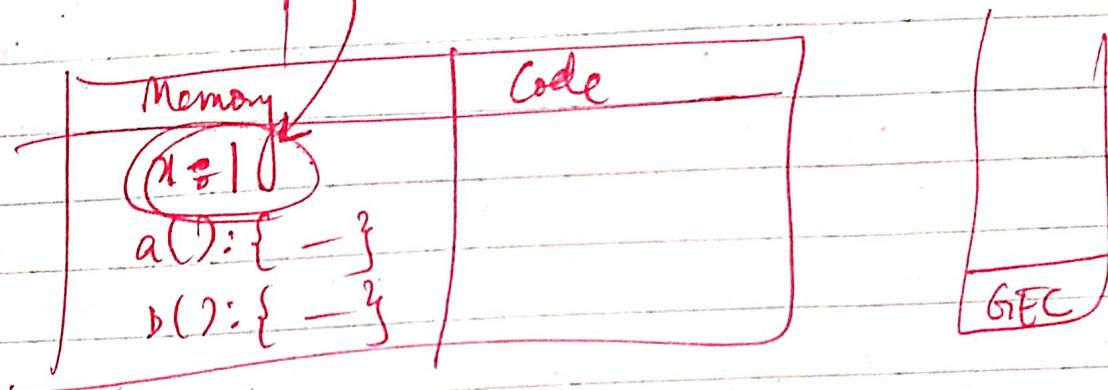


\hookrightarrow `console.log(x)` will print 1

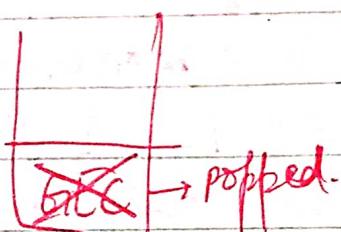
This is because JS engine finds the value of x

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 cell stack.



So output is: 10
100
1

;) (I hope you understood!)

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

B] gives error
Uncought Syntax
Error: Identifier 'a'
has already been
declared.

"Shortest JS Program"

(~~var~~ ~~const~~ 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 debugger 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, ~~kesome~~ 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)

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.

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`~~
~~`window`~~
`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.`

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

* So, whenever we try to access any variable
without ~~console.log~~ ~~window~~ ~~(variable name)~~, and instead,
we use the ~~console.log~~ `variable name`,
the JS engine assumes that we are referring
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 undefined

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

inside
window object

Eg: `var a = 10;
function b()
{`

`var x = 10;
}`

`console.log(x); // this gives us`

Reference error
says that x is not

This is because, JS assumes that we are referring 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".

Doubt → ① `let a = 10;
var b = 100;`

`console.log(window.a);` → O/P: undefined
`console.log(window.b);` → 100
`console.log(window.x);` → undefined
`console.log(x);` → 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".