

## Episode 2 : How JS is executed & Call Stack

---

- When a JS program is ran, a **global execution context** is created.
- The execution context is created in two phases.
  - Memory creation phase - JS will allocate memory to variables and functions.
  - Code execution phase
- Let's consider the below example and its code execution steps:

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

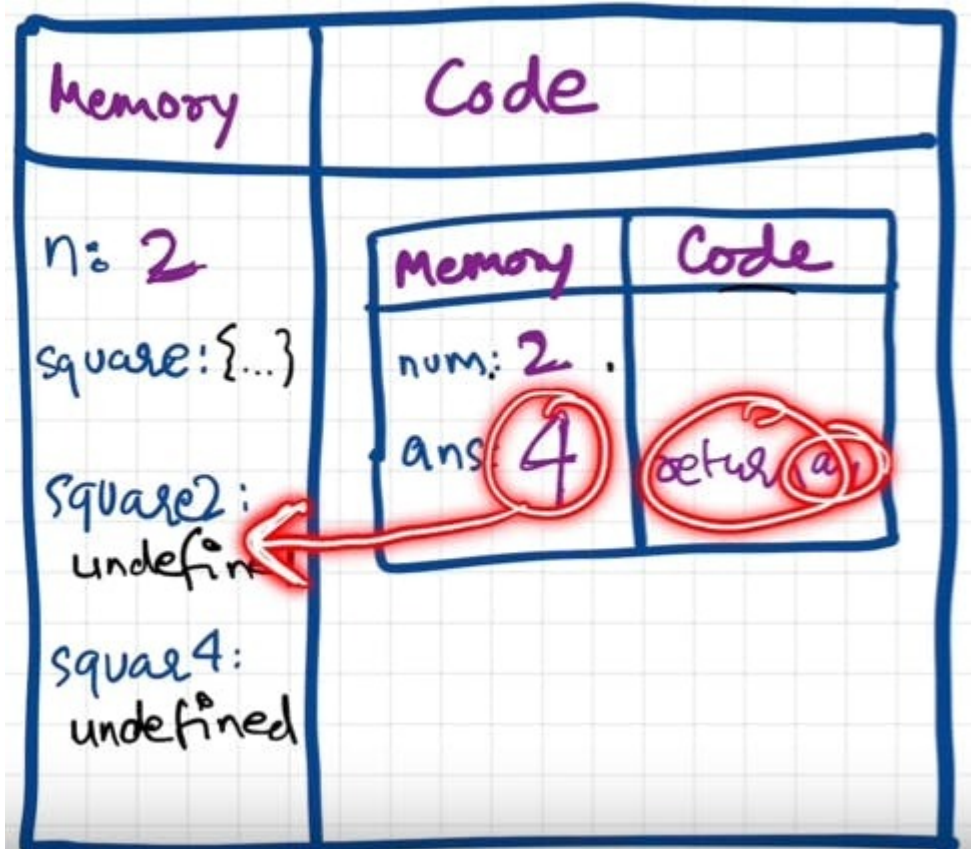
The very **first** thing which JS does is **memory creation phase**, so it goes to line one of above code snippet, and **allocates a memory space** for variable '**n**' and then goes to line two, and **allocates a memory space** for **function 'square'**'. When allocating memory **for n it stores 'undefined'**, 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.

So O/P will look something like

Memory	Code
$n = \text{undefined}$ $\text{square} = \{ \dots \}$ $\text{square2} = \text{undefined}$ $\text{square4} = \text{undefined}$	

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. For function, there is nothing to execute. As these lines were already dealt with in memory creation phase.

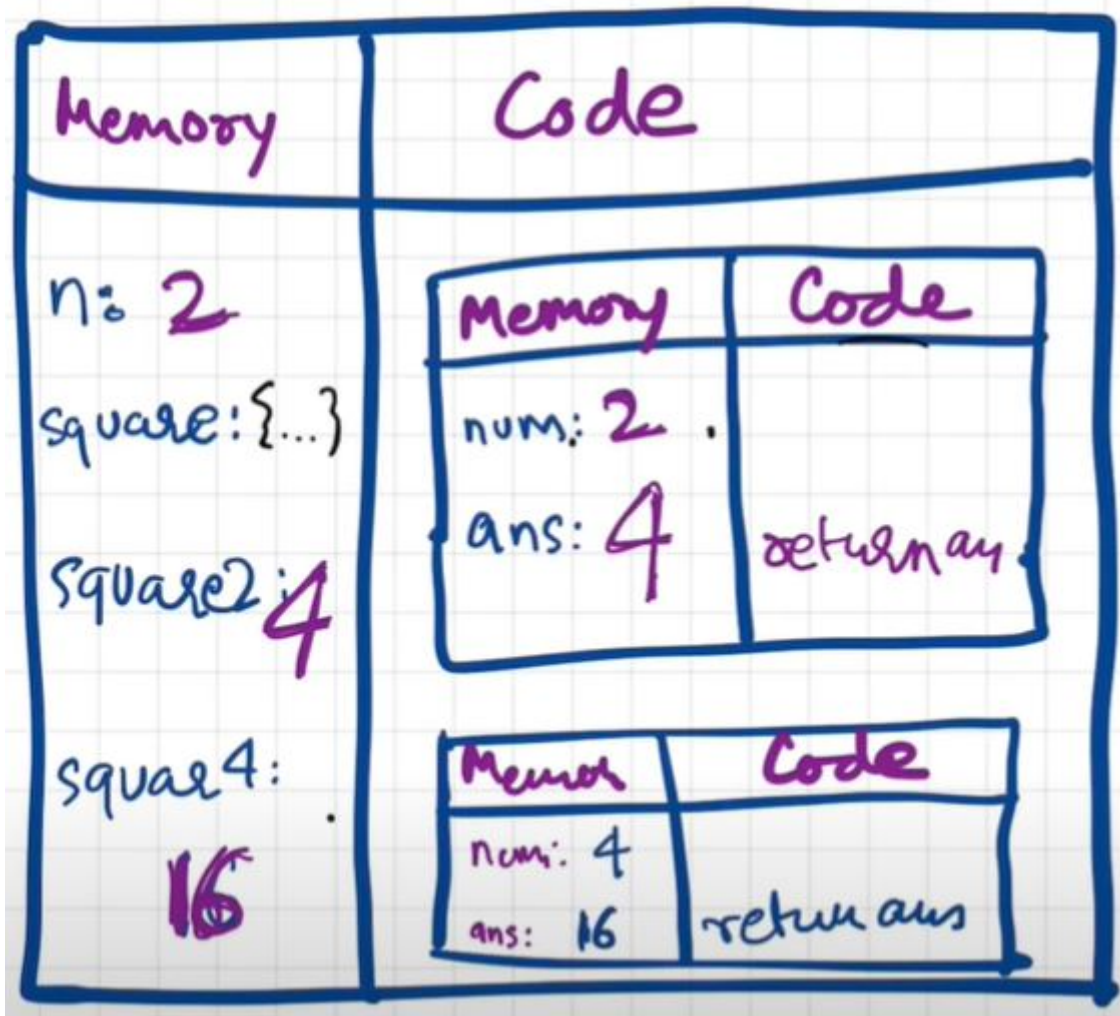
Coming to line 6 i.e. `var square2 = square(n)`, here **functions are a bit different than any other language. A new execution context is created altogether**. Again in this new execution context, in memory creation phase, we allocate memory to num and ans the two variables. And undefined is placed in them. 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.



When **return** keyword is encountered, It returns the control to the called line and also **the function execution context is deleted**.

Same thing will be repeated for square4 and then after that is finished, the global execution context will be destroyed.

So the **final diagram** before deletion would look something like:



- Javascript manages code execution context creation and deletion with the the help of **Call Stack**.
- Call Stack is a mechanism to keep track of its place in script that calls multiple function.
- Call Stack maintains the order of execution of execution contexts. It is also known as Program Stack, Control Stack, Runtime stack, Machine Stack, Execution context stack.

---

Watch Live On Youtube below:

