How JavaScript Code is Executed and Call Stack

Sunday, 16 July 2023 7:43 PM

How JS code is executed and Call Stack

Everything in JS happens inside an execution context.

→ What happens when you run a JS code?

ans. Whenever JS code is executed an execution context is created.

eg:

```
1 var n =2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);
```

→ How is the above code executed?

ans. When the above code is executed, firstly an execution context is created.

Now, this EC is created in 2 phases:

1) Memory Creation phase 2) Code Execution Phase

- In the memory creation phase, JS will allocate memory to all the variables and fn's.

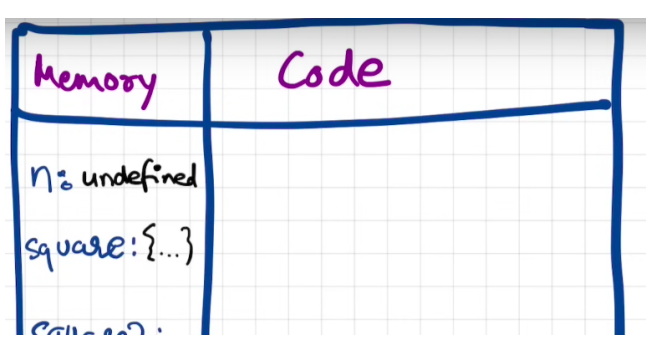
So, JS will first allocate memory to 'n', then it encounters a fn, so it will allocate memory to 'square()'.

While allocating memory to 'n' it stores a special value undefined.

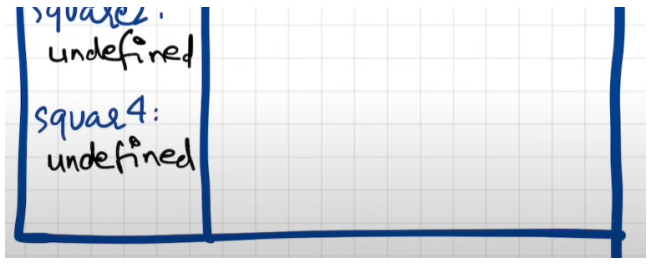
In case of fn's, it stores the whole code of the fn inside the memory.

Finally, square2 and square4 are also allocated memory.

Since they are variables so JS stores them as undefined.



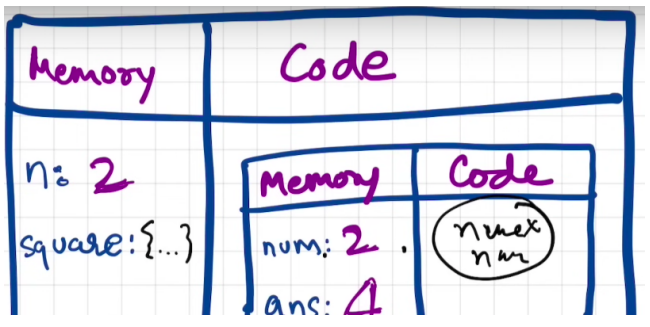
(this is how the EC looks like right now)



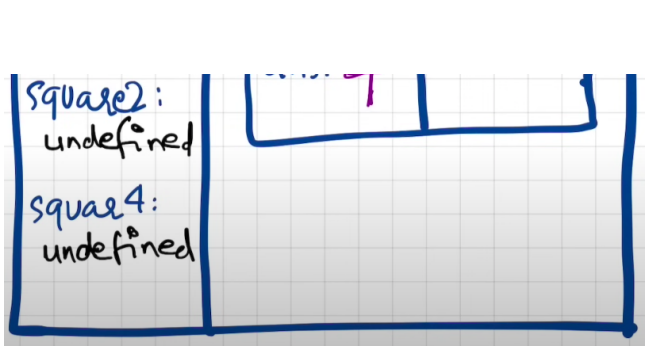
- In the 2nd phase, i.e the Code Execution phase, JS once again runs through the whole code line by line and it executes the code now.
  As soon as it encounters the 1st line, it replaces value of 'n' from undefined to '2'.
  It sees that there is nothing to execute in lines 2-5 so it skips that and moves to line 6.
  In line 6, fn is being invoked. When a fn is invoked a new execution context is created inside the global EC.
  Now, when this execution context is created it again has 2 components : 1) Memory and 2) Code

→ Now in the memory creation phase, memory is allocated to the parameter and variables. Memory is allocated to 'num' & 'ans' and they will be stored as undefined.
→ Now, in the code execution phase, first of all value of 'n' which is 2 is passed to 'num' (which is a parameter), So, value of 'num' is changed from undefined to 2.
(Where, n is argument and num is a parameter).
In the next line, value of 'ans' is replaced from undefined to 4.
Whenever 'return' is encountered, it returns the control of the program to where the fn was invoked.
'return ans' finds the value of 'ans' in the local memory and returns the value of ans back to the 6.

After the whole fn is executed, the local EC will be deleted and value of variable square2 will be replaced from 'undefined' to '4'.



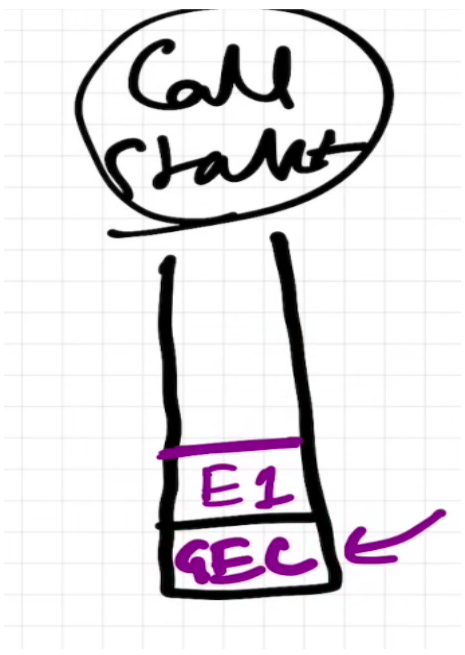
(EC just before encountering return statement)



Note: Similarly, all the process is repeated for line 7.
And finally, when all lines of code have been executed the GEC is also deleted.

→ What is a call stack?

ans. It is a stack. Whenever any JS program is run, the call stack is populated with a GEC.
The whole GEC is pushed into the call stack.
When a fn is invoked a local EC is pushed into the call stack. When the fn code is executed the local EC is popped from the stack.
And control goes back to the GEC.
And finally the GEC is also popped from the call stack once the whole JS code is executed.



← the call stack

→ The call stack is for managing the ECs.
→ The call stack maintains the order of execution of the execution contexts.

- Call stacks are also known by:
  1) Execution Context stack
  2) Program stack
  3) Control Stack
  4) Runtime stack
  5) Machine Stack