Ep-01 How Javascript Works

1.1) Execution context

→ Everything in Javascript happens inside an execution context.

→ Execution context can be assumed as a big box or container in which whole Javascript code is executed.

→ Execution context has two components:

1. Memory component / variable environment
   → It stores all variables and functions as key:value pair. eg voraible : var a = 10 stored as a:10.
2. code component / Thread of Execution
   → It is the place where the code is executed one line at a time.
   → It's also know as thread of execution, which is just like a thread in which the whole code is executed one line at a time.

Memory component / variable environment	Code component / Thread of execution
key : value	o ———
a : 10	o ———
fun : {...}	o ———
	o ———

1.2) Javascript is Synchronous Single threaded language.

→ Single-threaded means Javascript can only execute one command at a time.

→ Synchronous single threaded means that Javascript can only execute one command at a time and in a specific order. That means it can only go to the next line of code once the current line of code has been executed.

Note: Though Javascript is a "synchronous single threaded language", but also supports "asynchronous operations" like AJAX requests. This capability allows Javascript to perform non-blocking operations, enabling it to handle tasks like fetching data from servers without halting the entire program.

Ep-02 How Javascript code is executed

2.1) Understanding JS Execution context

→ Everything in JS happens inside an execution context.

Q- What happens when you execute an Javascript program?

• An execution context is created having two components: Memory component & code component.

→ Global execution context is created in two phases.

1) First Phase: Memory Creation phase

2) Second Phase: Code Execution phase

eg. code

```
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);
```

* Let's see how this piece of code runs behind the scenes

→ when you run this code, an global execution context is created.

→ This execution context is created in two phases:

1. First Phase - Memory creation phase.
2. Second Phase - Code Execution phase.

1) → In first phase: Memory Creation Phase

→ In memory creation phase, Javascript will allocate memory to all the variables and functions.

→ As soon as the JS encounter line 1, it will allocate memory to n.

→ Similarly at line 2, JS will allocate memory to function named square.

⇒ Q- what does it store, while allocating memory:

→ When JS allocates memory to n, it stores an special value "undefined." in the first phase.

→ In the case of function, it will store the whole code of the function inside the memory phase.

→ At line 6, it will allocate memory to square2 and at line 7 to square4. it will store undefined in square 2 and square 4.

Summary - In the first phase, JS skims through the whole program line by line and allocates memory to all the variables and functions. As well as allocates value undefined to variables and whole code to the functions.

↳ undefined is like a placeholder in JS special keyword

Memory component	Code component
n: undefined	o ———
square: {...}	o ———
square2: undefined	o ———
square4: undefined	o ———

Fig Execution context

2) In Second Phase: Code Execution Phase

→ After memory allocation, JS once again run through the whole program line by line and it executes the code now. (This is the point where functions and every calculation take place.)

→ As soon as it encounters first line, where n=2. it actually place 2 as value in n. (undefined is replaced with actual value of n i.e. 2)

→ Moving to next, at line 6 there is function invocation.

→ when you invoke a function, a "new execution context" is created. This execution context again has two components: Memory and code.

Now we will again go through the two phase: Memory creation and code execution phase.

⇒ Execution context for function invocation:

2.1.1 Memory creation Phase:

→ Memory is allocated for variables & functions which are inside the function. including parameters. variables

→ For given function square, memory will be allocated for parameter num and variable ans. variables inside the function and undefined will be stored as value.

```
function square (num) {
    var ans = num * num;
    return ans;
}
```

Memory component	Code component
n: 2	
square: { var ans = num * num; return ans; }	
square2: undefined	
square4: undefined	

Global Execution context

Memory	code
num: undefined	o ———
ans: undefined	o ———

→ Execution context for function square()

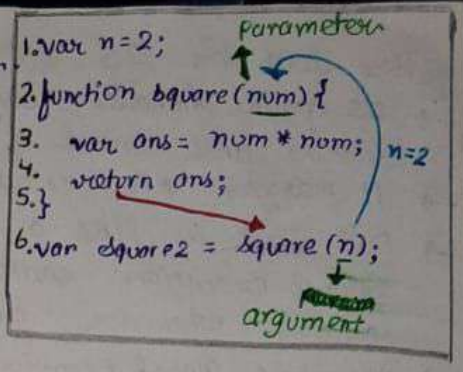
fig.

2.2.2 Code Execution Phase

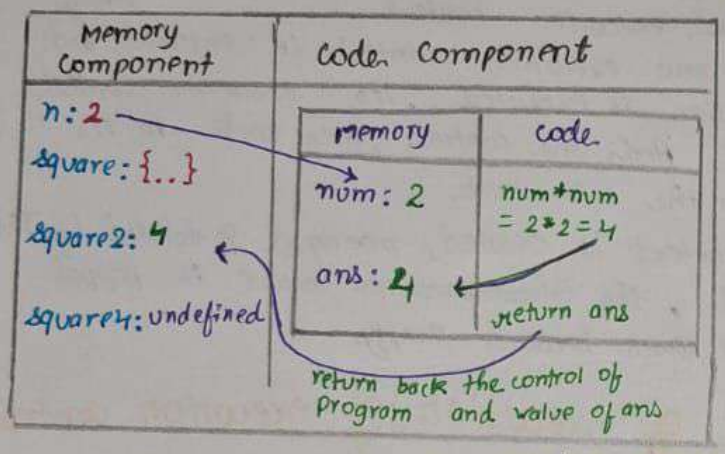
→ In this phase, JS executes whole function line by line.

→ When the function is involked, the value of n i.e. 2 is passed to num.

→ Now, moving to the next line no. 3, it will do the calculation num*num and put the result in variable ans. i.e. in execution context → in code component it will perform calculation num*num and replaces the undefined in ans in memory component with resultant value.



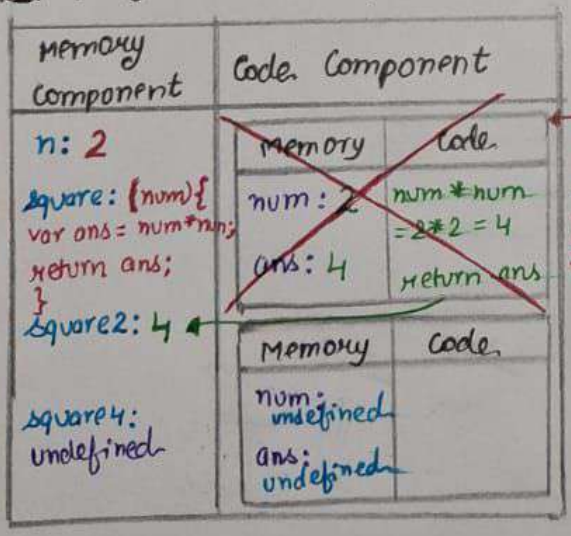
→ After finishing line no. 3, control goes to line no. 4 where JS encounter special keywords return. This return keyword states that now return the control of the program, to the place where this function was involked.



→ After the whole function is executed, the whole execution context for that instance of that function will be deleted.

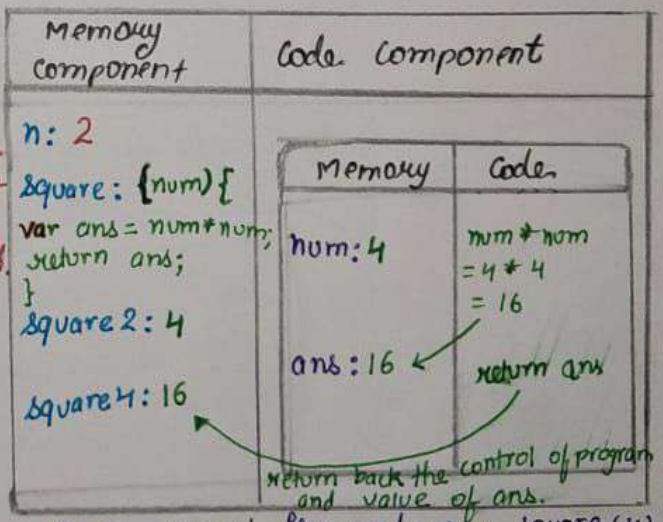
→ After the replacement of undefined in square with return value 4, JS will go to line no. 7. In this line there is function invocation square(4) Here, we are passing the argument 4 directly. The function will be executed in the same as it executed for square(n). A brand new execution context will be created: memory & code component and it will again through the two phases.

2.2.1 Memory creation phase for square(4)



After the execution of return statement, the execution context for the function square(n) is deleted.

2.2.2 Code Execution Phase for square(4)



→ After the execution of function square(4) the execution context for it is deleted.

→ After the execution of whole program, the whole global execution context will be deleted.

Q. How does the JS engine manage all the stuff explained above?

→ JS engine handles everything to manage this execution context creation, deletion and control of program.

→ It manages it using a stack known as "Call Stack."

→ Call stack is like a stack. At the bottom of this there will always be Global execution context stored as it the first execution context to be created whenever any JS program is executed.

→ The whole Global Execution context (GEC) is pushed inside the stack.

→ whenever a function is invoked or a new execution context (EC) is created it is pushed inside the stack.

→ After the function is executed and its ~~the~~ execution context is popped off the call stack. and the control goes back to the global execution context.

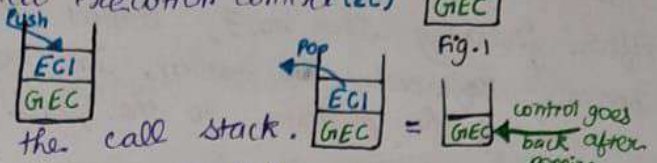
→ Similarly on any function invocation a new execution context is created and pushed into the stack. After the function is executed, its execution context is popped off the stack and deleted. And the control goes back to its previous execution context which is in the call stack.

→ This is how the whole execution context is created, managed & deleted by JS engine.

→ After the whole program is executed, the Global Execution context is popped off the stack & deleted and the call stack becomes empty.

"Call Stack"

Fig. 1



"Call Stack maintains the order of execution of execution context"

Call Stack is also knowns as:
1. Execution Context Stack
2. Program Stack
3. Control Stack
4. Runtime Stack
5. Machine Stack