# JAVA SCRIPT

* Everything in Javascript happens inside an "execution context"

* execution Context
  ↳ like a big box and have two component

Variable environment

| Memory | Code |
|---|---|
| key : value<br>a : 10<br>fn : {...} | 1 ——<br>2 ——<br>3 ——<br>4 —— |

↙ Thread of execution
↓
like a thread in which whole code is executed one line at a time.

* JS is a synchronous single threaded language.
  ↓
  can run one command at time in specific order.

## How Javascript Code Works

↳ When we run a program a execution context is created.

| Memory | Code execution | Code |
|---|---|---|
| n : ~~undefined~~ 2<br>square : {...} ← whole code.<br>square2 : ~~undefined~~ 4<br>square4 : ~~undefined~~ 16 | Nothing to do. square(n) ← function invocation | memory \| Code<br>num : ~~undefined~~ 2<br>ans : ~~undefined~~ 4 |

(phase-1 - Memory allocation)

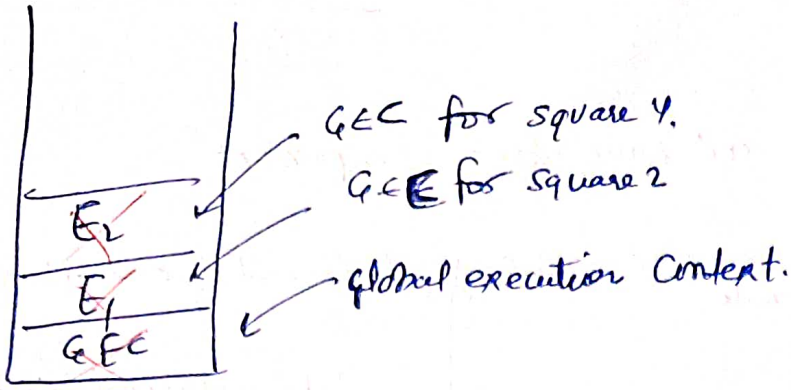
after return it will delete.

Same thing is in

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

It will delete after all code compiled

All these thing Javascript Engine handles by Call Stack.




" Call Stack maintains the order of execution of execution contexts".

Call Stack is also known as

- ↳ Execution Context stack
- ↳ Program Stack
- ↳ Control stack
- ↳ Runtime stack
- ↳ Machine Stack

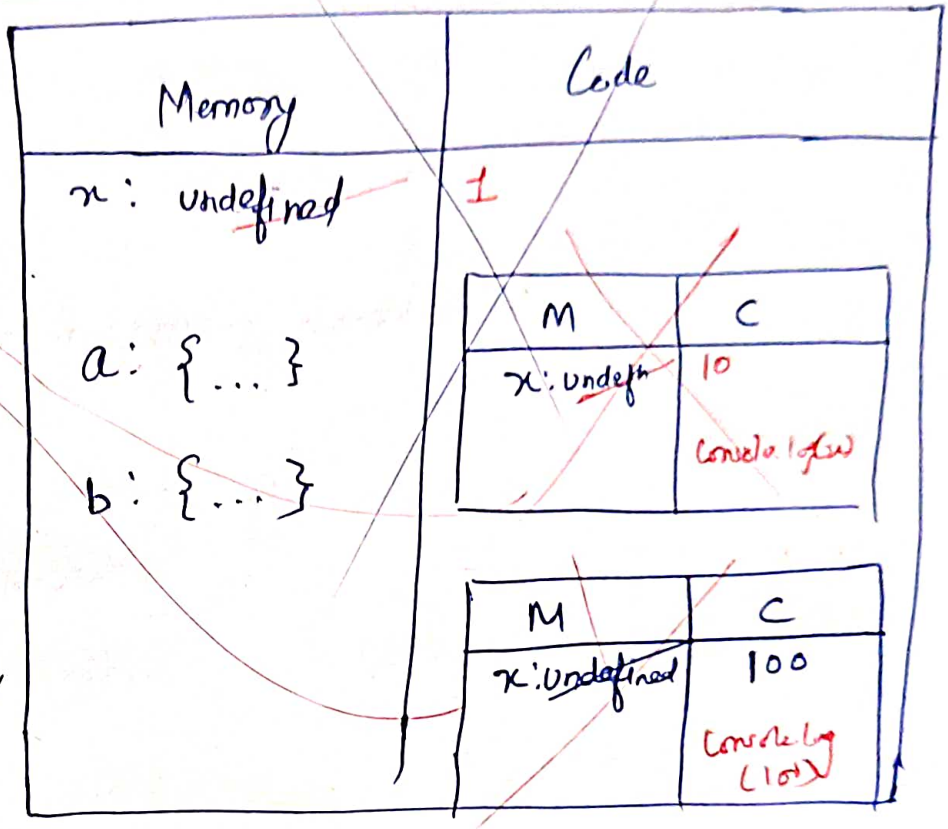
# Hoisting in Javascript

Hoisting is a phenomenon in JS by which you can access variables and functions even before you have initialized. (w/o any errors)

Read from Git for more example

## How Javascript fn Works

```
var x = 1;
a();
b();
console.log(x);
function a(){
    var x = 10;
    console.log(x);
}
function b(){
    var x = 100;
    console.log(x);
}
```




Call Stack


O/P

10
100
1