Web Dev Course

5 main sub-topics

- → JavaScript (JS interview, Node, Scrapping & Automation)
- → Front End (HTML, CSS, DOM, Front End Interview)
- → React (React Interview)
- → SQL (SQL Interview)
- → Backend (MongoDB, Mongoose, Express, Node)

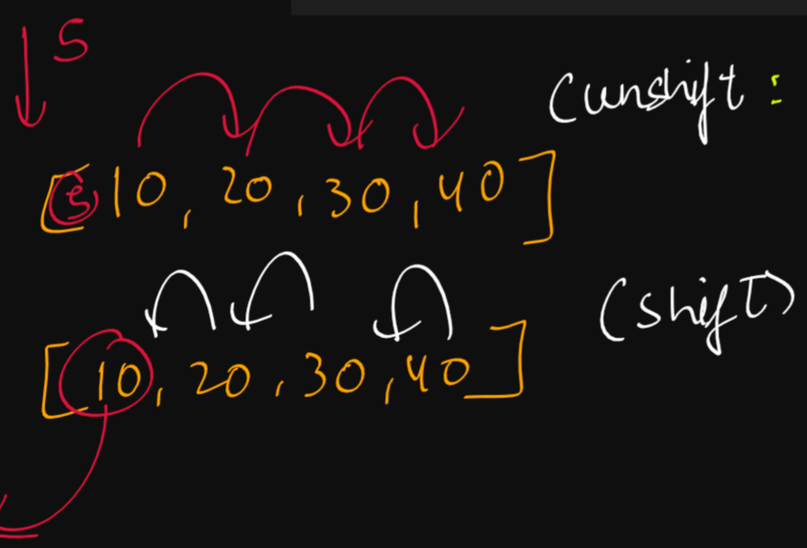
Arrays

Array in JS is not just a collection of similar datatype.

[1, "Hello", true, {Name: "Guneet", Age: 21}]

1 → Number
"Hello" → String
true → Boolean
{Name: "Guneet", Age: 21} → Object

```
let array = [1,"Hello",true,{Name: "Guneet", Age: 21}]
console.log(array)

//traverse
for(let i=0;i<array.length;i++) {
    console.log(array[i])
}
```



(unshift : since shifting is in left ← direction)

```
//remove last
array.pop();
console.log(array);   [1, "Hello", true]

//add last
array.push(20);
console.log(array)    [1, "Hello", true, 20]

//For arrays-> Shifting means moving at the previous place (left direction)

// add first
array.unshift(5)
console.log(array)    [5, 1, "Hello", true, 20]

//remove first
array.shift();
console.log(array)    [1, "Hello", true, 20]
```

slice → to get a subarray
→ original array is not changed
↳ gives us a COPY OF THE PART OF THE ARRAY.

array.slice(startIdx, endIdx)
↳ endIdx is not included

```
// slice -> get a part of the array
// original array remains unchanged
let slicedArray = array.slice(0,2) //starting index is included but ending is not
console.log(slicedArray)  [1, "Hello"]
console.log(array)    [1, "Hello", true, 20]
```

splice → generic delete function (specific del func → pop & shift)
↳ original array is affected
↳ it returns an array of deleted elems

```
// splice -> generic delete function
// first param is starting index
// second param is count of the elements to be deleted

let splicedArray = array.splice(1,1); // 1 is the starting index and we want to del 1 ele

console.log(splicedArray); ["Hello"]
console.log(array) //original array is also changed   [1, true, 20]
```

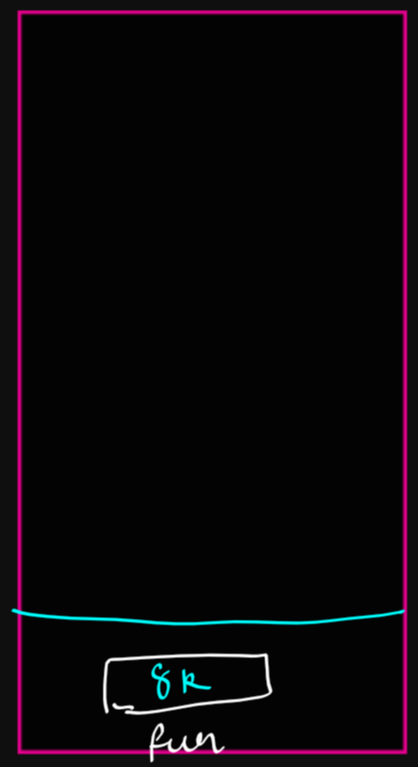
functions

```
function fun() {
    console.log("Hi, I am a function")
}

fun()
```

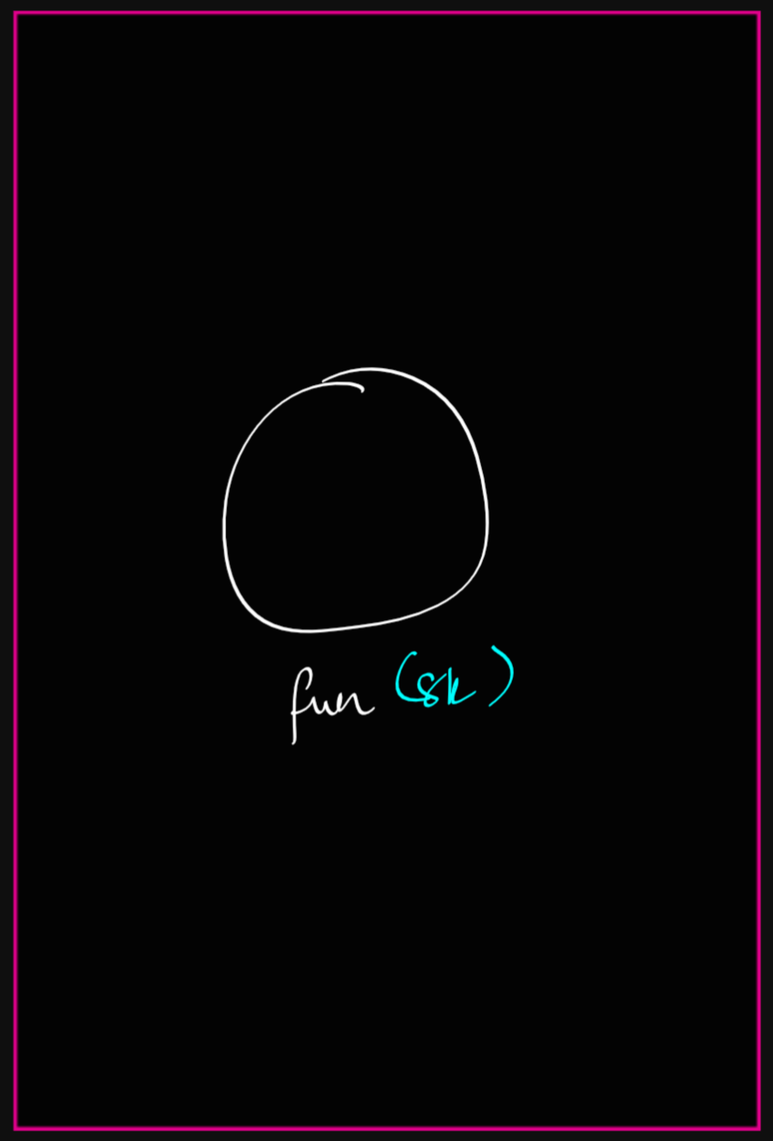
① In JS, code execution takes place in 2 steps

(i) Parsing : Reading entire code & allocating memory to variables & functions

(This brings variables & functions to the top of the code called Hoisting)



Stack



Heap

(ii) Execution

** functions are first class citizens in JS

(They behave like variables (move like objects))

→ JS is synchronous i.e code executes line by line.

parameter

```
function sayHi(name) {
    console.log("My name is " + name);
    return name;
//     return "Hello";

}

console.log(sayHi("Guneet"))
```

We can return anything-

In JS we do not have to define the return type of the function & it can return anything

But remember only one thing can be returned.

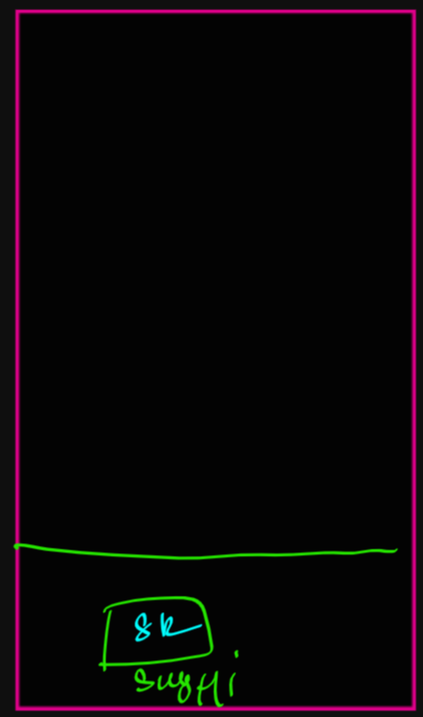
return true

or

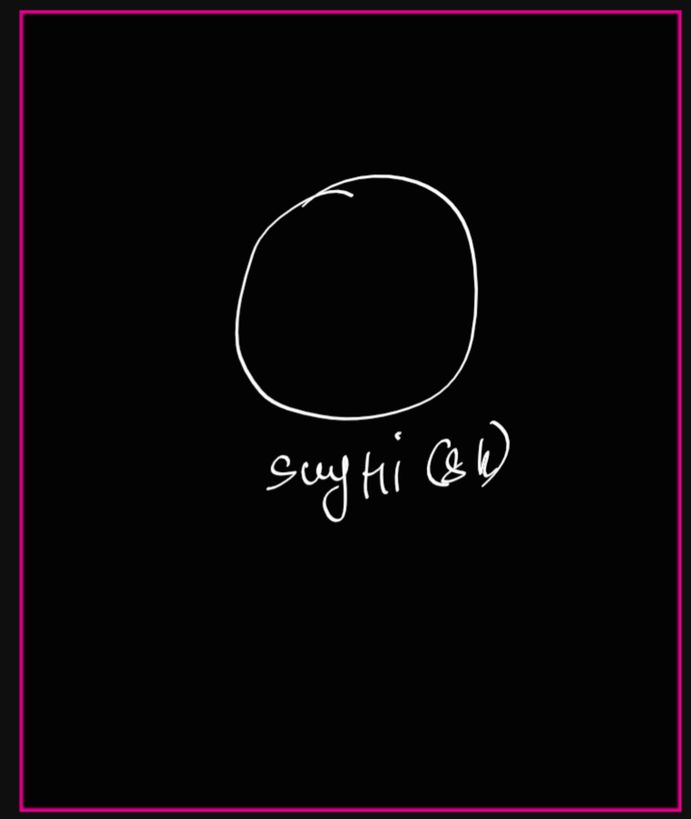
return [1,2,3,4]

or

return { Name: "Guneet", Age: 21 }



Stack



Heap

```
console.log(sayHi("Guneet"))  //This will print the return value

console.log(sayHi) //Here, we have not made any call. So, no return value will be printed
// [Function: sayHi] -> this will be printed instead of the address of the sayHi function
```

Address will not be printed as JS is a HLL. The toString() func is designed in such a way.

** function as a parameter to another function is called as a callback function and the function that takes a function as a parameter is called a higher order function.

HOF

```
function fun(param) {
    console.log("Hey there!!");
    param();
    return param;
}

function smallFun() {
    console.log("I am a small Function")
}

// console.log(fun);
// console.log(smallFun)
// console.log(smallFun());
console.log(fun(smallFun))
```

callback

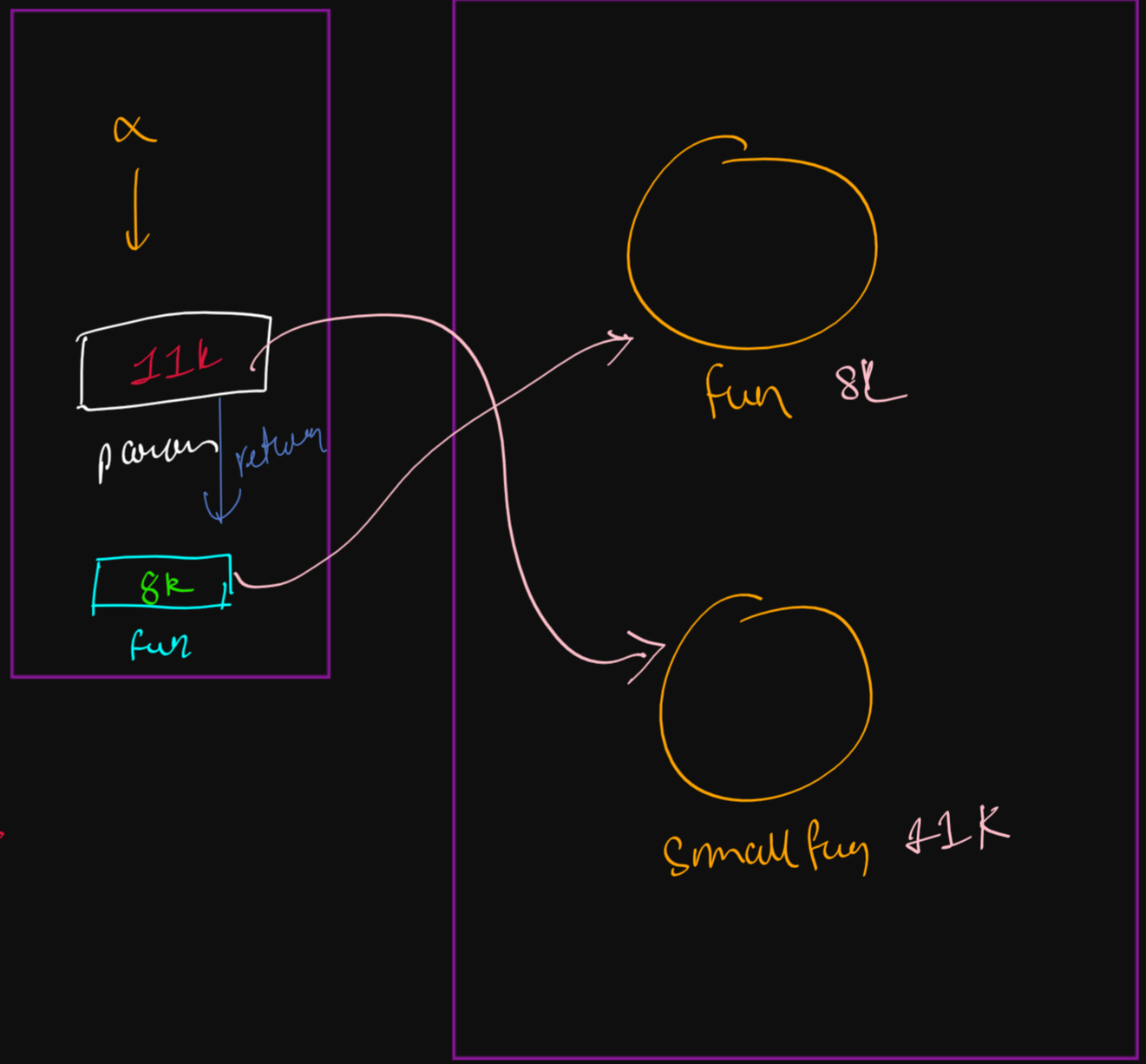
fun is called

execution context for smallFun

execution context for fun

global execution context

(we will study this further)



① Hey there!!

② I am a small function

③ [function: smallfun]

Some Higher Order functions

① Map : Applies the same func for every element of the array.

```
function squarer(number) {
    return number*number;
}

let arr = [1,2,3,4,5];

let squaredArray = arr.map(squarer);
console.log("Squared array" , squaredArray);
console.log("Original Array", arr);
```

[1, 2, 3, 4, 5]

map → iterates through every element of the array & applies the same func.

** Original array is not changed.

callback fun
→ Not returning anything

```
let arr = [1,2,3,4,5];

function cuber(number) {
    console.log(number * number * number);
}

let cubedArray = arr.map(cuber);
console.log(cubedArray)
```

output →

```
$ node rev.js
1
8
27
64
125
[ undefined, undefined, undefined, undefined, undefined ]
```

from this output, try to think about the polyfill of map.

Map Polyfill

```
Array.prototype.myMap = function(callBack) {

    let newArr = [];
    for(let i = 0;i<this.length;i++) {
        let newVal = callBack(this[i]);
        newArr.push(newVal);
    }

    return newArr;
}
```

② filter: Applies callback to every ele of the array & if the callback returns true then puts that ele to the new Array.

↳ Original array is not changed.

```
let arr = [1,2,3,4,5,6,7,8,9,10];

function oddTest(number) {
    return number % 2 == 1;
}

function evenTest(number) {
    return number % 2 == 0;
}

let evenArray = arr.filter(evenTest);
let oddArray = arr.filter(oddTest);

console.log("Even array", evenArray);
console.log("Odd array", oddArray);
console.log("Original Array",arr); //This remains unchanged
```

→ only the ele that pass evenTest will be here.

→ only the ele that pass oddTest

In a way, filter is used when callback returns a boolean value

Polyfill of filter

```
Array.prototype.myFilter = function(callBack) {

    let newArr = [];

    for(let i = 0;i < this.length;i++) {
        if(callBack(this[i]) == true) {
            newArr.push(this[i]);
        }
    }

    return newArr;
}
```

⑤ for Each: Applies a function to every element of the array. Here, the callback does not have a return type unlike that in map.

```
let arr = [1,2,3,4,5];

function printEle(elem) {
    console.log(elem);
}

arr.forEach(printEle)
```

Example

```
// For each polyfill

Array.prototype.myForEach = function(callback) {

    for(let i=0;i<this.length;i++) {
        callback(this[i]);
    }
}

arr.myForEach(printEle)
```

for Each Polyfill

④ Every : Checks whether all the elements of an array follow a particular condition. Even if 1 ele fails the condition, ans will be false.

→ Returns boolean value (T/f)

```
let arr = [1,2,3,4,5];

function checkLessThan10 (elem) {
    if(elem < 10) return true;
    return false;
}

console.log(arr.every(checkLessThan10))
```

Example

```
// Every Polyfill

Array.prototype.myEvery = function(callback) {

    for(let i=0; i<this.length;i++) {
        if(callback(this[i]) == false) return false;
    }

    return true;
}

console.log(arr.myEvery(checkLessThan10))
```

Polyfill

** → Callback returns a boolean value

⑤ Some: Checks whether atleast one element of the array follows a particular condition or not - If even ele holds a condition true, ans will be true.

→ Returns a boolean value (T/F)

```
let arr = [1,20,30,40,50];

function checkLessThan10 (elem) {
    if(elem < 10) return true;
    return false;
}

// console.log(arr.some(checkLessThan10))
```

Example

```
// Some Polyfill

Array.prototype.mySome = function(callback) {

    for(let i=0; i<this.length;i++) {
        if(callback(this[i]) == true) return true;
    }

    return false;
}

console.log(arr.mySome(checkLessThan10))
```

Polyfill

** Callback returns a boolean value

⑥ findIndex: Returns the firstIndex of the any element which fulfills a particular condition.

↳ Returns the integer (index)

```
let arr = [10,20,3,40,50];

function checkLessThan10 (elem) {
    if(elem < 10) return true;
    return false;
}

console.log(arr.findIndex(checkLessThan10))
```

Example

```
//Find Index Polyfill

Array.prototype.myFindIndex = function(callback) {

    for(let i=0; i<this.length;i++) {
        if(callback(this[i]) == true) return i;
    }

    return -1;
}

console.log(arr.myFindIndex(checkLessThan10))
```

Polyfill

Arguments

Let us say there is a function that has 2 arguments in function definition / declaration.

```
function fun(a,b) {
}
```

While calling the function, JS allows to pass less or more arguments. In fact, we might not even pass any argument. This is because when we call a function, the parameters that we pass go into the arguments (kind of array, not exactly an array) of that function. So, the parameters that we write at the time of declaration are just copies of a few arguments.

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64 /d/FJP WebDev/fun
$ node arg.js
[Arguments] { '0': 'hello', '1': 'hi' }
[Arguments] { '0': 'hello' }
[Arguments] { '0': 'hi' }
[Arguments] {}
[Arguments] { '0': 'hello', '1': 'hi', '2': 'bye' }
```

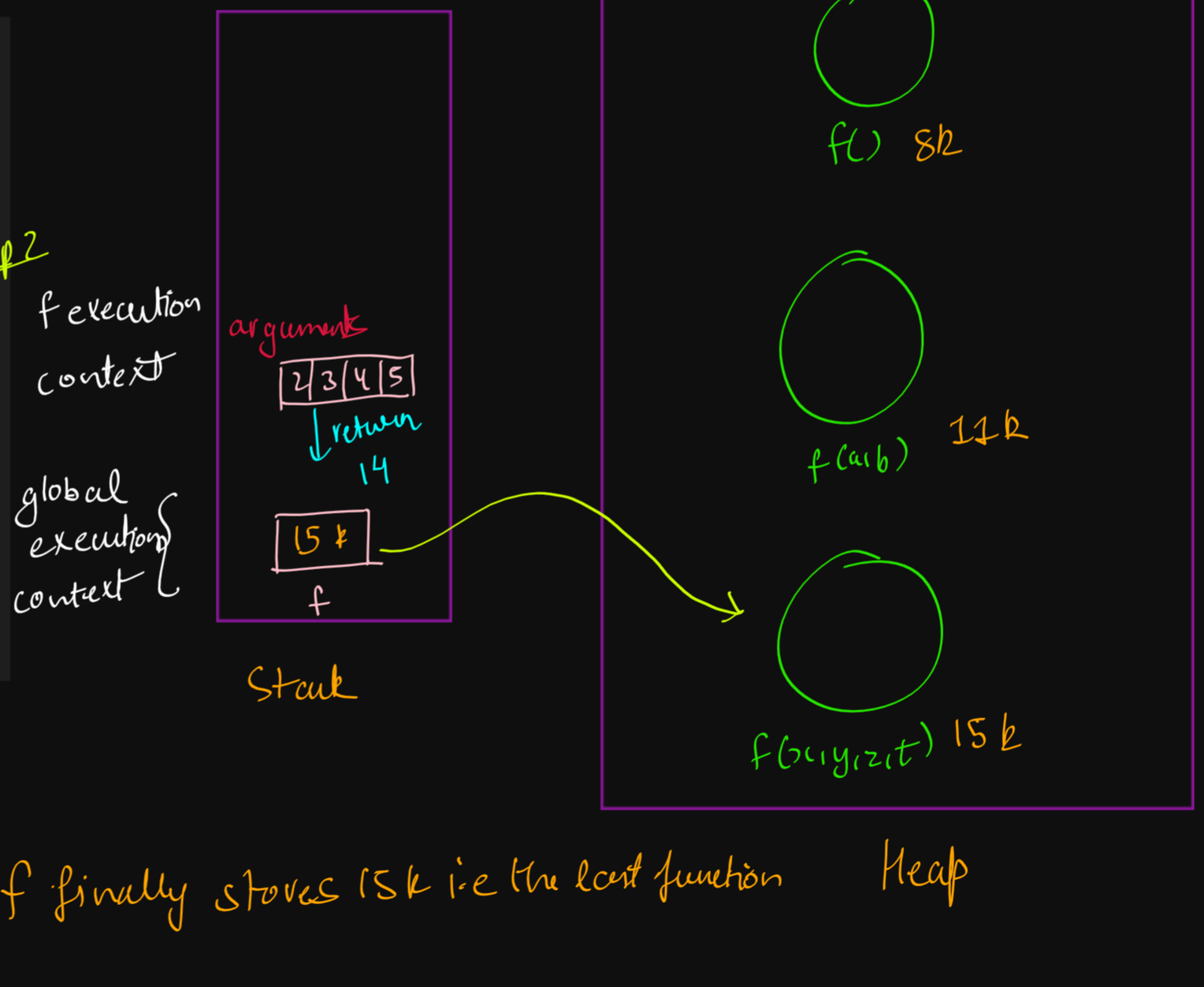
```
function fn(param1,param2) {
    console.log(arguments);
    // console.log(arguments["0"],arguments["1"])
}

fn("hello","hi")
fn("hello") //hello undefined
fn("hi") //hi undefined
fn() //undefined undefined

fn("hello","hi","bye") //basically the arguments object/array will contain the parameters
//When a function func(p1,p2) is defined these p1 and p2 do not matter at all
//The parameters while calling will be entering inside the arguments/objects array
```

Execution Context & Hoisting (function Hoisting)

```
//Any code in JS will execute in an execution context
//The default execution context is global execution context
//An execution context is created whenever a function is called
// The first step in an execution context is memory allocation and the second step is code execution
```

```
//Find the output of the following

function f() {
    console.log(arguments);
}

function f(a,b) {
    return a + b;
}

console.log(f(2,3,4,5));

function f(x,y,z,t) {
    return x + y + z + t;
}

console.log(f(2,3,4,5));
```

GEC step 2



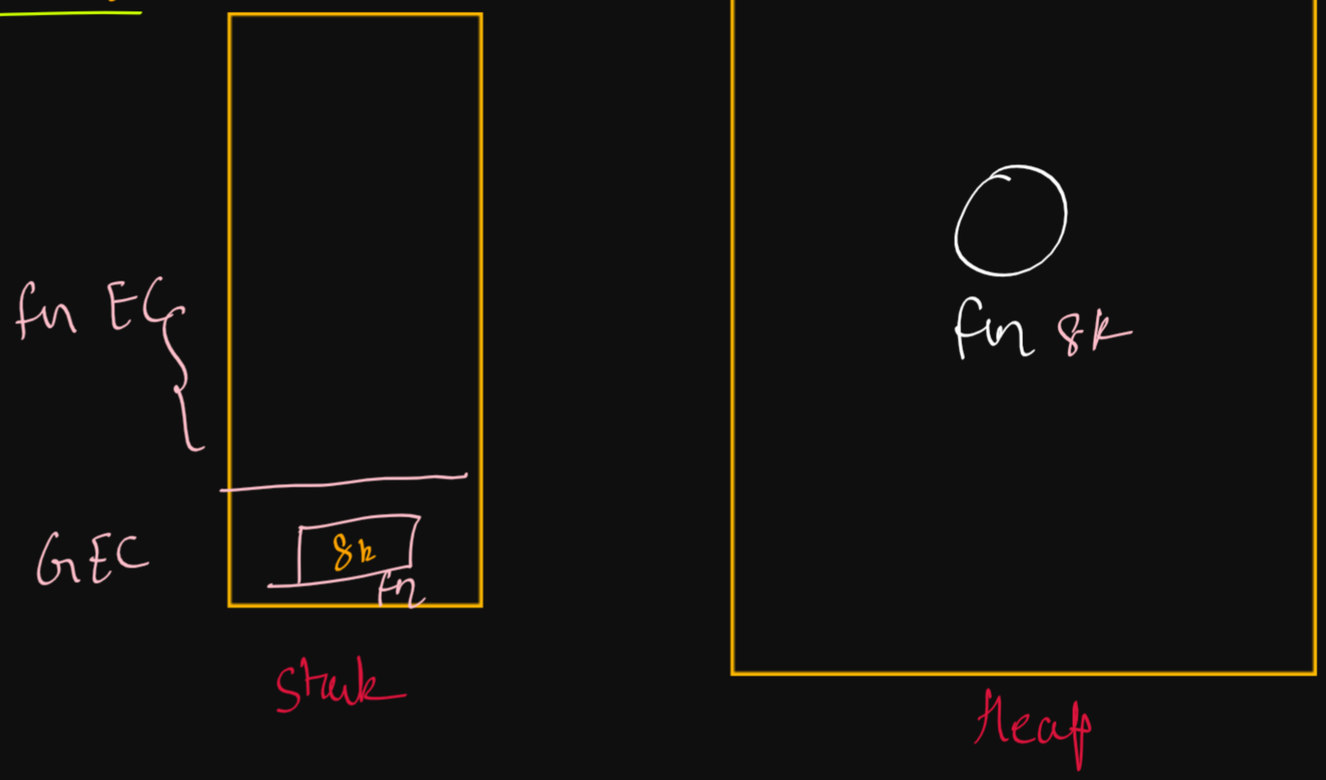
Step: 1 Memory Allocation

→ It is due to hoisting that f finally stores 15k i.e the last function

Some types of functions

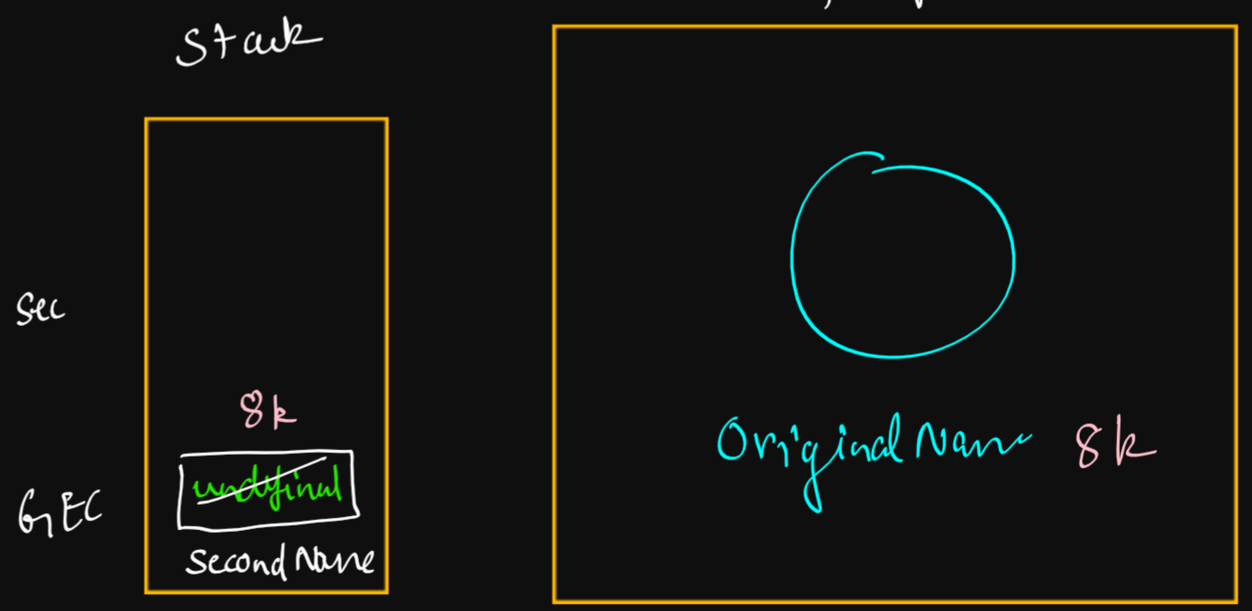
① Normal function definition:

```
//function definition

function fn() {
    console.log("I am a function")
}

fn();
```



② function Expression:

```
//function expression

//assigning function to a variable is function expression
let secondName = function originalName() {
    console.log("I am a function expression")
}

secondName()
```



IIfE

(Immediately Invoked function Expression)

```
// Immediately Invoked Function Expressions (IIFE)
console.log("Before");

//anonymous functions
(function () {
  console.log("Board is immediately drawn")
})()

let secondName = function() {
    console.log("I am a function expression")
}
```

We need not give the name

func call

IIfEs are used when we have to do something immediately as soon as the page loads. Say, we have to display some data from an API call as soon as the page loads.

To conver

```
(function() {
    let timeUnits = parseInt(prompt("How much to count?"));
    let interval = parseInt(prompt("Log after how much interval?"));

    let iid = setInterval(handleCalls,interval*1000);

    handleCalls.orgTU = timeUnits;//functions can be used as a store of properties (much like objects)

    function handleCalls() {
        console.log(timeUnits + "left");
        timeUnits -= interval;

        if(timeUnits == 0) {
            clearInterval(iid);
            alert(handleCalls.orgTU + " has been counted");
        }
    }
})()
```

Static function object (function is used as an object to store a property)

Anonymous functions

A function with no name is called as an anonymous function.

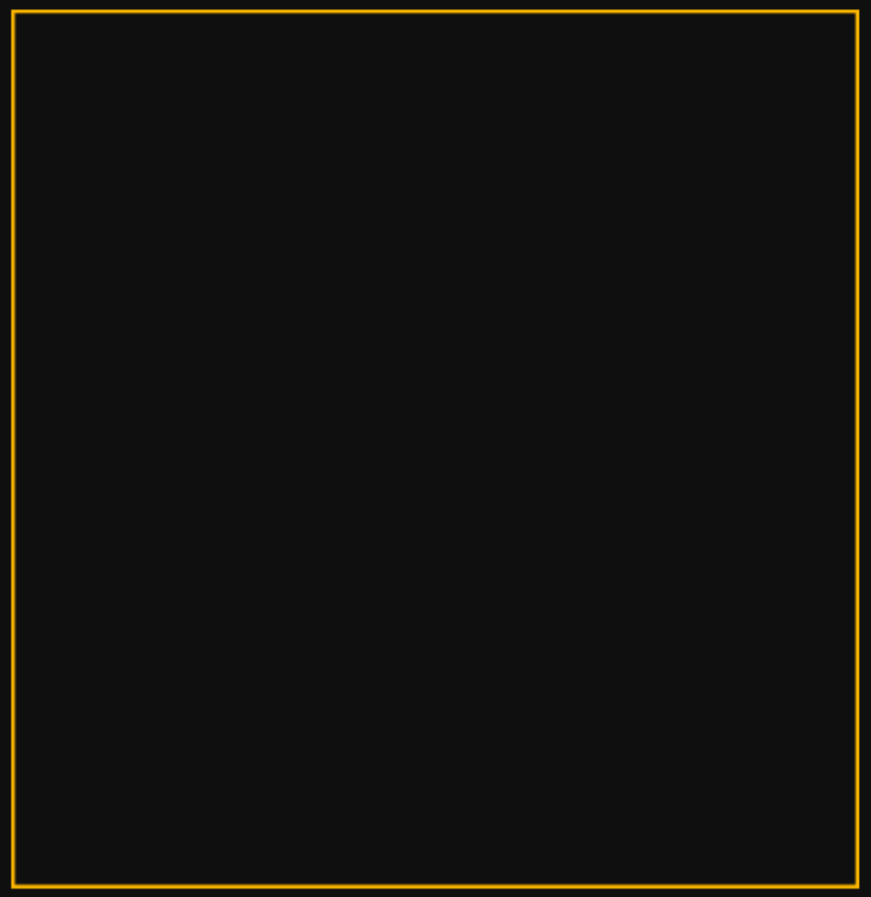
```
//anonymous functions
(function () {
  console.log("Board is immediately drawn")
})()

let secondName = function() {
    console.log("I am a function expression")
}
```

few Questions based on types of funcs & func Hoisting

```
//What is the output of this code?

fun();

var fun = function() {
    gun();
}

var gun = function() {
    console.log("I am inside gun")
}
```

GEC {

Stack



Heap

① Memory Allocation (GTEC)

② Execution

fun() → fun is not a function

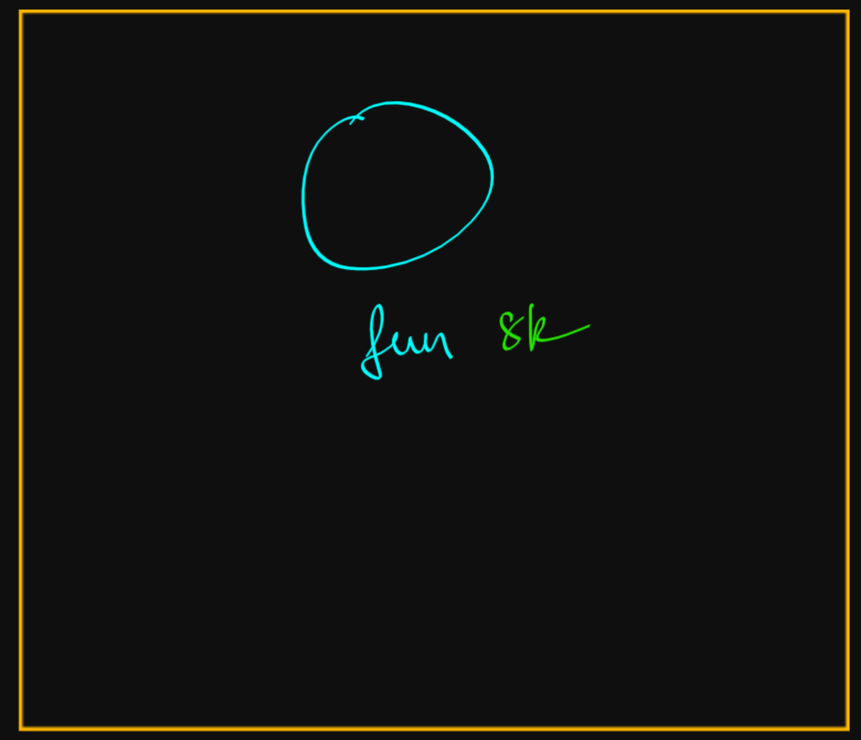
Here, fun Execution Context will not be formed as fun is not a func

```
//What is the output of this code?

var fun = function() {
    gun();
}

fun()
var gun = function() {
    console.log("I am inside gun")
}
```

fun EC

Global execution context

Stack



Heap

② Execution

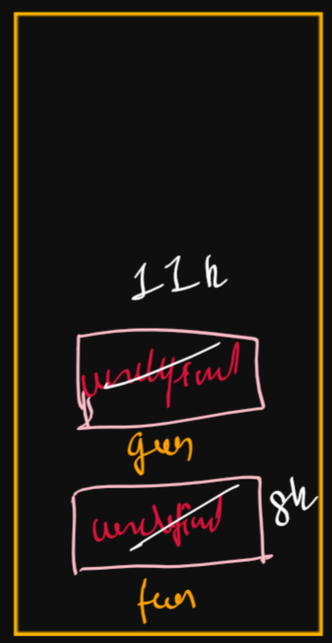
var fun = ...

fun() → gun is not a function

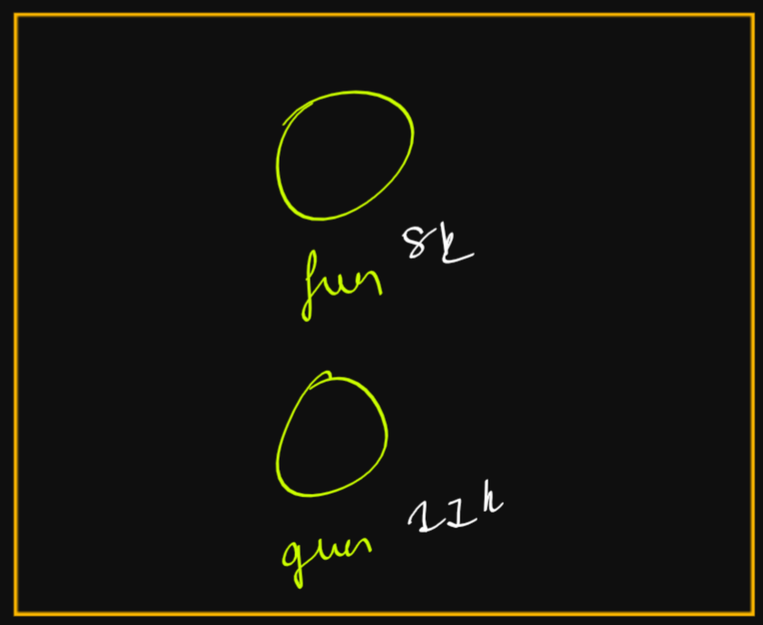
```
//What is the output of this code?

var fun = function() {
    gun();
}

var gun = function() {
    console.log("I am inside gun")
}


fun()
```

gun EC

fun EC



Stack



Heap

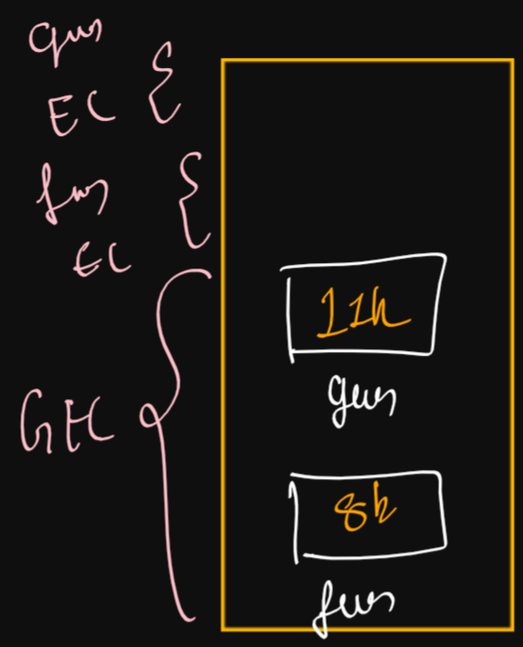
Execution

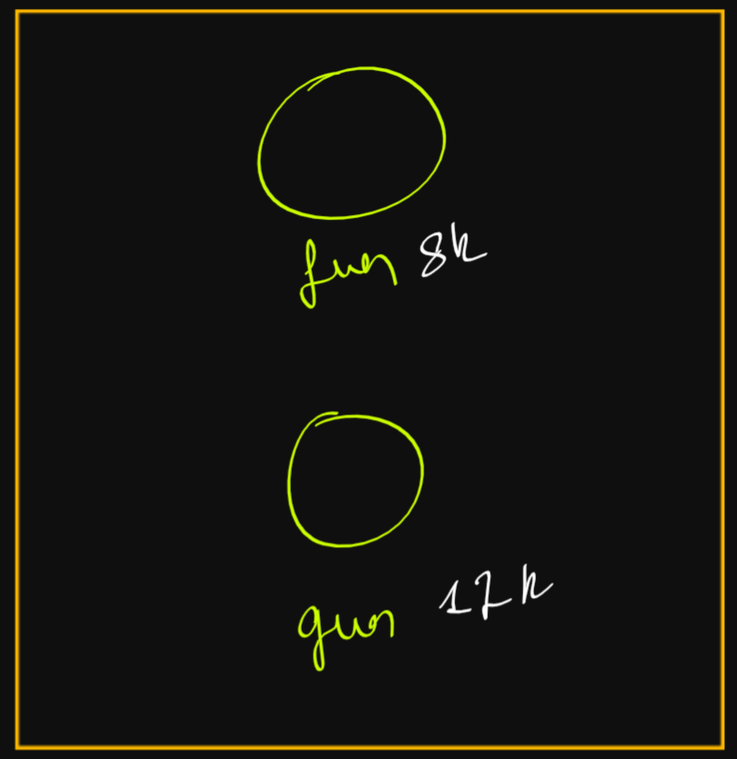
fun() → I am inside gun

```
//What is the output of this code?

fun();

function fun() {
    gun();
}

function gun() {
    console.log("I am inside the gun function");
}
```

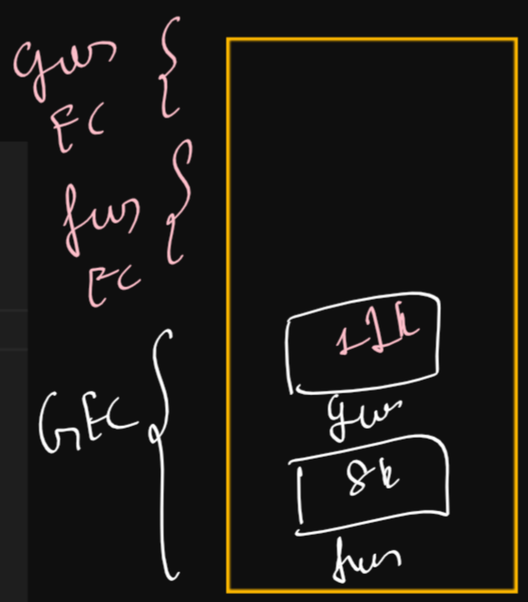
① Memory Allocation

② Execution : fun()



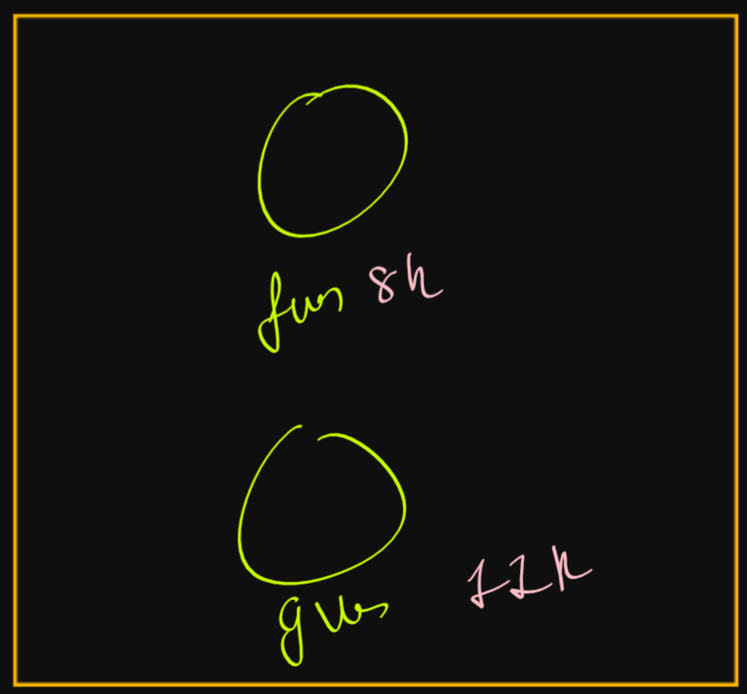



```
//What is the output of this code?

function fun() {
    gun();
}

fun();

function gun() {
    console.log("I am inside the gun function");
}
```



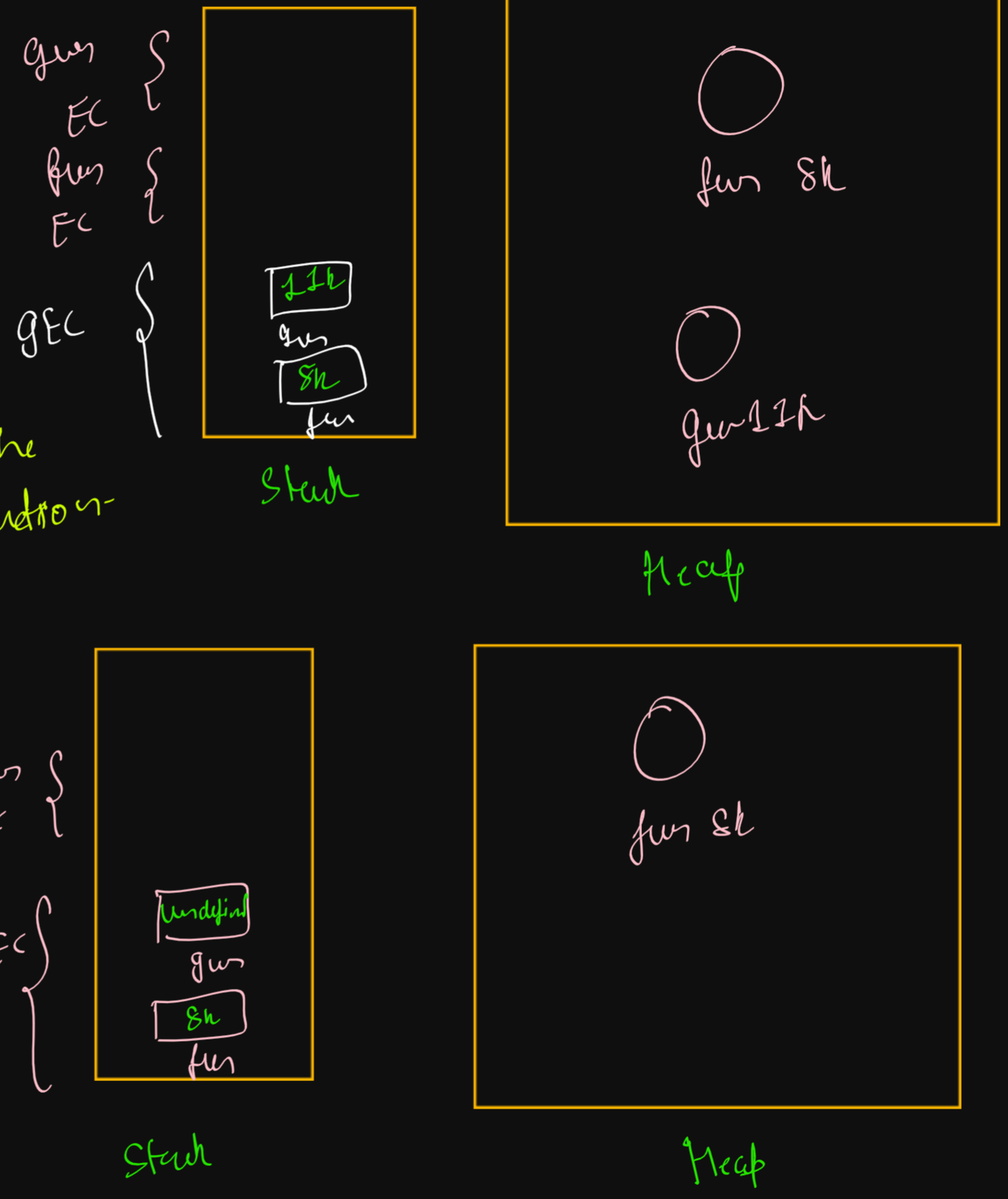
Execution: fun(): I am inside the gun function

Stack



Heap

```
//What is the output of this code?

function fun() {
    gun();
}

function gun() {
    console.log("I am inside the gun function");
}

fun();
```



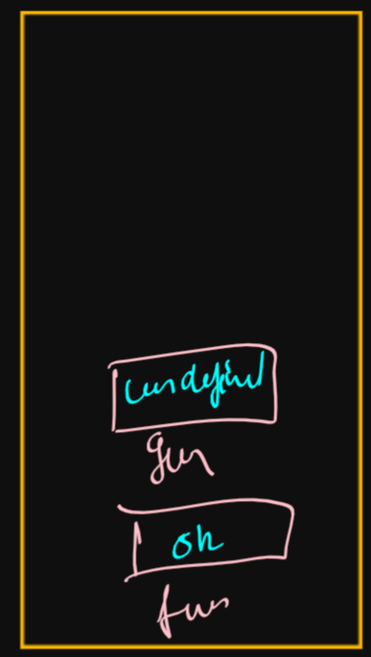
Execution: fun() -> I am inside the gun function

```
//What is the output of this code?
fun();

function fun() {
    gun();
}


var gun = function() {
    console.log("I am inside the gun function");
}
```


Execution: fun() -> gun is not a function

```
//What is the output of this code?

function fun() {
    gun();
}

fun();

var gun = function() {
    console.log("I am inside the gun function");
}
```

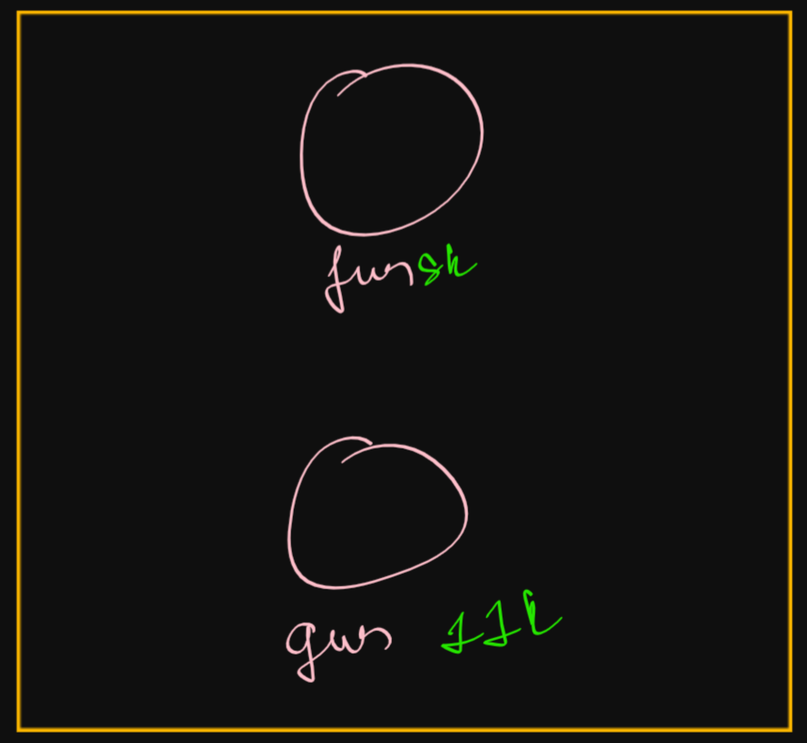
funEC { GEC {



Execution: fun() → gun is not a function

```
//What is the output of this code?

function fun() {
    gun();
}


var gun = function() {
    console.log("I am inside the gun function");
}

fun();
```

gun { EC }
fun { EC }
GEC {



Execution: fun() → I am inside the gun function

Call, Bind & Apply

There are various ways of calling a function. Like this normal way:

```
let obj = {

    fun1 : function(frnd1,frnd2) {
        console.log("I am a function on Object. This person is " + this.Name + ". His age is " + this.Age);
        console.log(this.Name + " says hello to " + frnd1 + " and " + frnd2);
        console.log(arguments)
    },

    Name: "Guneet",
    Age: 21

}

obj.fun1()
```

Call: It is a function available for every function. It is used to add a property of another object to our object. Basically, we change the default this.

obj.func.call(newThis, parameters to the function func)

↑

newObj (because obj is the default this)

Apply: Apply is similar to call. Just the parameters are passed as an array.

```
obj.func.call(newThis, array of parameters)
              ↑
            newObj
```

Bind: Bind is a little different. It does not call a function rather it returns a new function that can be called. If the new function (boundfunction) is called without any params, it is same as calling the call() func. If some param is given to bind, it just adds up to the arguments array.

```
let boundfun = obj.func.bind(newThis, params)

boundfun(new params)
```

How to use Call

```
let obj = {

    fun1 : function(frnd1,frnd2) {
        console.log("I am a function on Object. This person is " + this.Name + ". His age is " + this.Age);
        console.log(this.Name + " says hello to " + frnd1 + " and " + frnd2);
        console.log(arguments)
    },

    Name: "Guneet",
    Age: 21

}

let o2 = {
    Name: "Rohan",
    Age: 30
}

obj.fun1.call(o2,"George","Ben","Nik","Stefan");
```

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64 /d/FJP WebDev (main)
$ node rev.js
I am a function on Object. This person is Rohan. His age is 30
Rohan says hello to George and Ben
[Arguments] { '0': 'George', '1': 'Ben', '2': 'Nik', '3': 'Stefan' }
```

How to use Apply

```
let obj = {

    fun1 : function(frnd1,frnd2) {
       console.log("I am a function on Object. This person is " + this.Name + ". His age is " + this.Age);
       console.log(this.Name + " says hello to " + frnd1 + " and " + frnd2);
       console.log(arguments)
    },

    Name: "Guneet",
    Age: 21

}

let o2 = {
    Name: "Rohan",
    Age: 30
}

obj.fun1.apply(o2,["George","Ben","Nik","Stefan"]);
```

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64 /d/FJP WebDev (main)
$ node rev.js
I am a function on Object. This person is Rohan. His age is 30
Rohan says hello to George and Ben
[Arguments] { '0': 'George', '1': 'Ben', '2': 'Nik', '3': 'Stefan' }
```

How to use Bind

```js
let obj = {

    fun1 : function(frnd1,frnd2) {
       console.log("I am a function on Object. This person is " + this.Name + ". His age is " + this.Age);
       console.log(this.Name + " says hello to " + frnd1 + " and " + frnd2);
       console.log(arguments)
    },

    Name: "Guneet",
    Age: 21

}

let o2 = {
    Name: "Rohan",
    Age: 30
}

let boundFunc = obj.fun1.bind(o2,"George","Ben","Nik","Stefan");
boundFunc("Henry", "David");
```

```
$ node rev.js
I am a function on Object. This person is Rohan. His age is 30
Rohan says hello to George and Ben
[Arguments] {
  '0': 'George',
  '1': 'Ben',
  '2': 'Nik',
  '3': 'Stefan',
  '4': 'Henry',
  '5': 'David'
}
```

Polyfill of Bind

```
Function.prototype.myBind = function() {

    let callerFunc = this; //obj.fun1 (basically fun1)
    let args = Array.from(arguments);
    let newThis = args[0];
    let paramsArray = args.slice(1);

    function boundFunc () {
        let paramsToBoundFun = Array.from(arguments);
        let totalParams = paramsArray.concat(paramsToBoundFun);

        newThis.fun1 = callerFunc;
        newThis.fun1(...totalParams);
        delete newThis.fun1;
    }

    return boundFunc;
}
```

Closure

Polyfill of Call

```
Function.prototype.myCall = function() {

    let callerFun = this;
    let args = Array.from(arguments);
    let newThis = args[0];
    let paramsArray = args.slice(1);

    newThis.fun1 = callerFun;
    newThis.fun1(...paramsArray);
    delete newThis.fun1;
}
```

Polyfill of Apply

```
Function.prototype.myApply = function() {

    let callerFun = this;
    let args = Array.from(arguments);
    let newThis = args[0];
    let paramsArray = args[1];

    newThis.fun1 = callerFun;
    newThis.fun1(...paramsArray);
    delete newThis.fun1;
}
```

functions Behaviour

→ functions are variables (Hoisting)

→ functions can be passed as parameters to another funs (HOF & callBacks)

→ functions' reference can be stored in another variable (function expression)

→ function can return another function (Closure)

```
function outerfun()
{  var a;
   let arr = [1,2,3,4];
     function innerfunc() {
     }
}
```

→ the variables of outerfun that are used by the inner function will be stored in a closure even if outerfunc gets deleted from the stack.

```
function outer(firstNumber) {
    console.log("Inside outer");
    return function(secondNumber) {
        console.log("Inside Inner");
        return firstNumber * secondNumber;
    }
}

let rVal = outer(30);
let finalAns = rVal(20);
console.log(finalAns);
```

rVal has firstNo stored in its closure

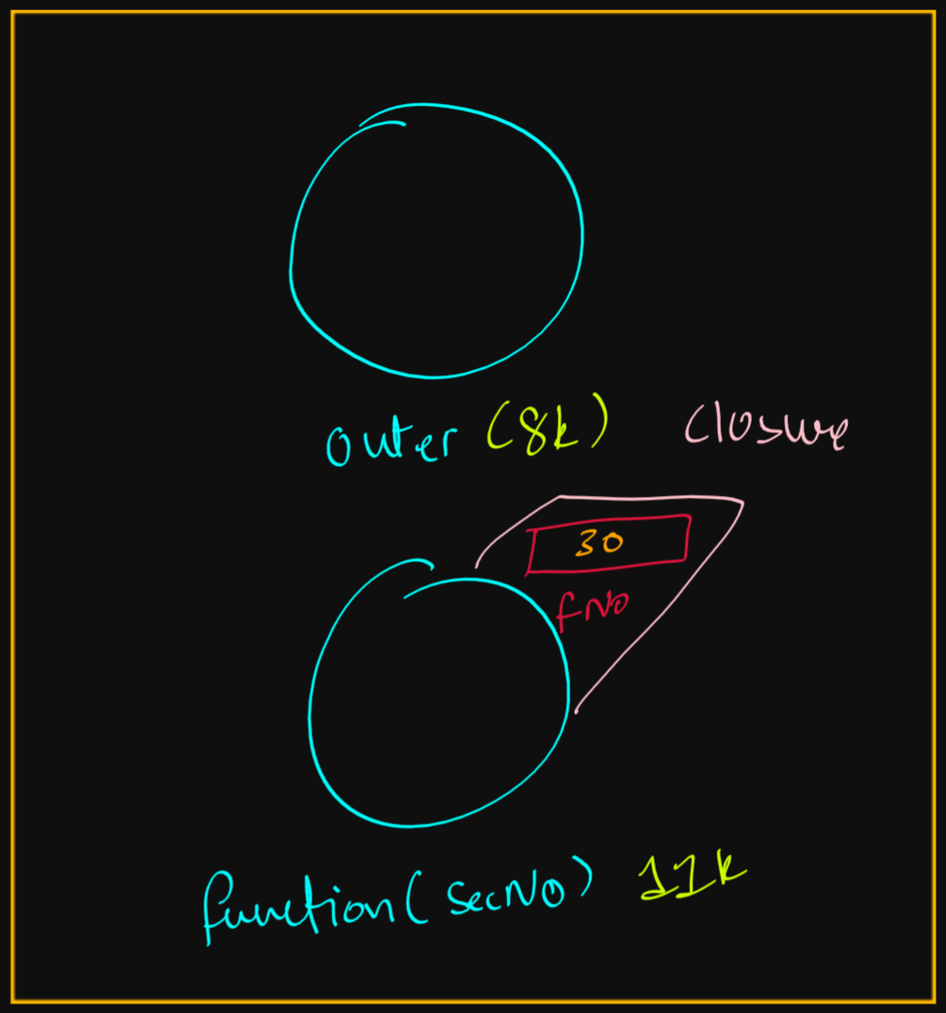
① Inside outer

② Inside inner

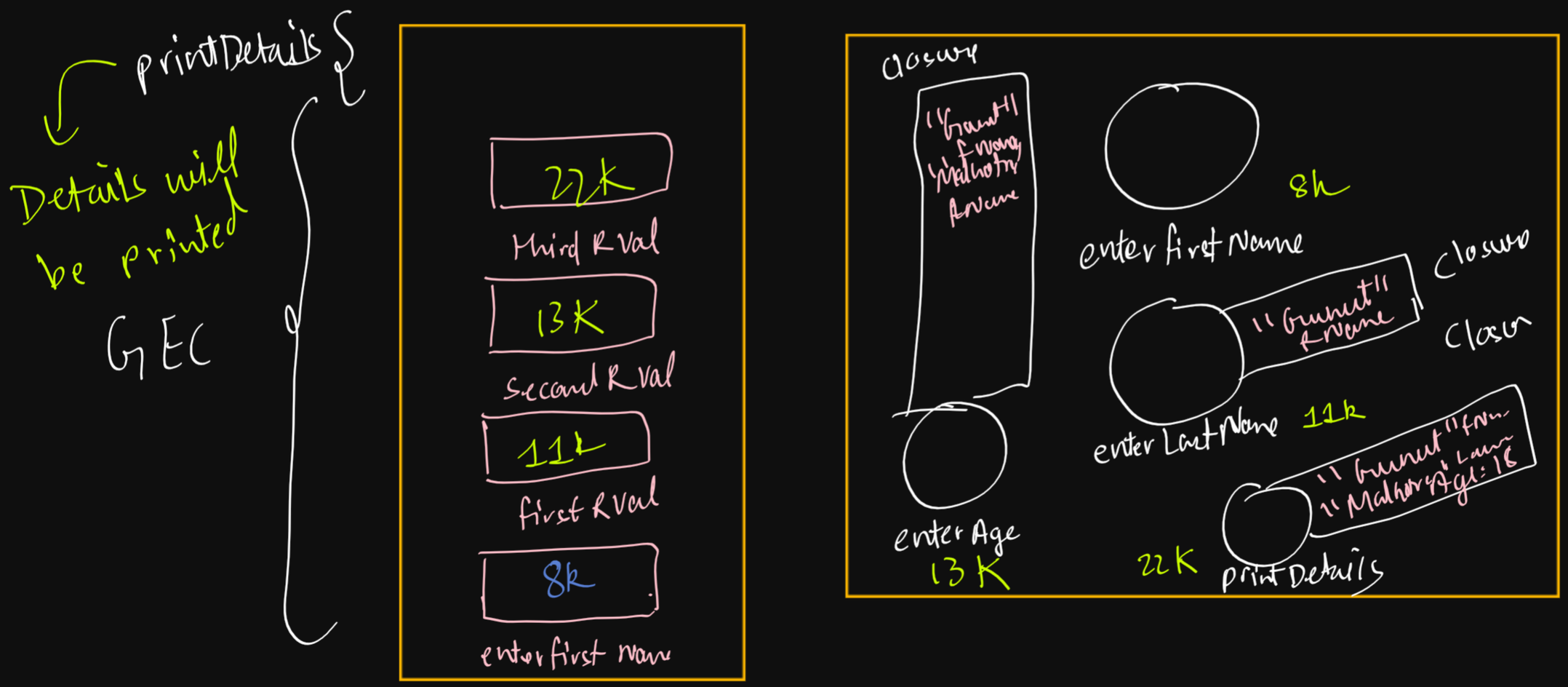
③ 600

Global Execution Context

20 × 30 = 600



```
function enterFirstName(first) {
    return function enterLastName(lastName) {
        return function enterAge(age) {
            return function printDetails() {
                console.log("Your name is" ,first,lastName,"and your age is",age);
            }
        }
    }
}

let firstRVal = enterFirstName("Guneet");
let secondRVal = firstRVal("Malhotra");
let thirdRVal = secondRVal(21);
thirdRVal();
```



JS Interview Series

What is Javascript?
fun facts

SpaceX Rocket → Dragon 2 → used JS for building UI

↳ Mobile Apps
↳ Desktop Apps
↳ Backend Server
↳ Websites
↳ Smart Devices

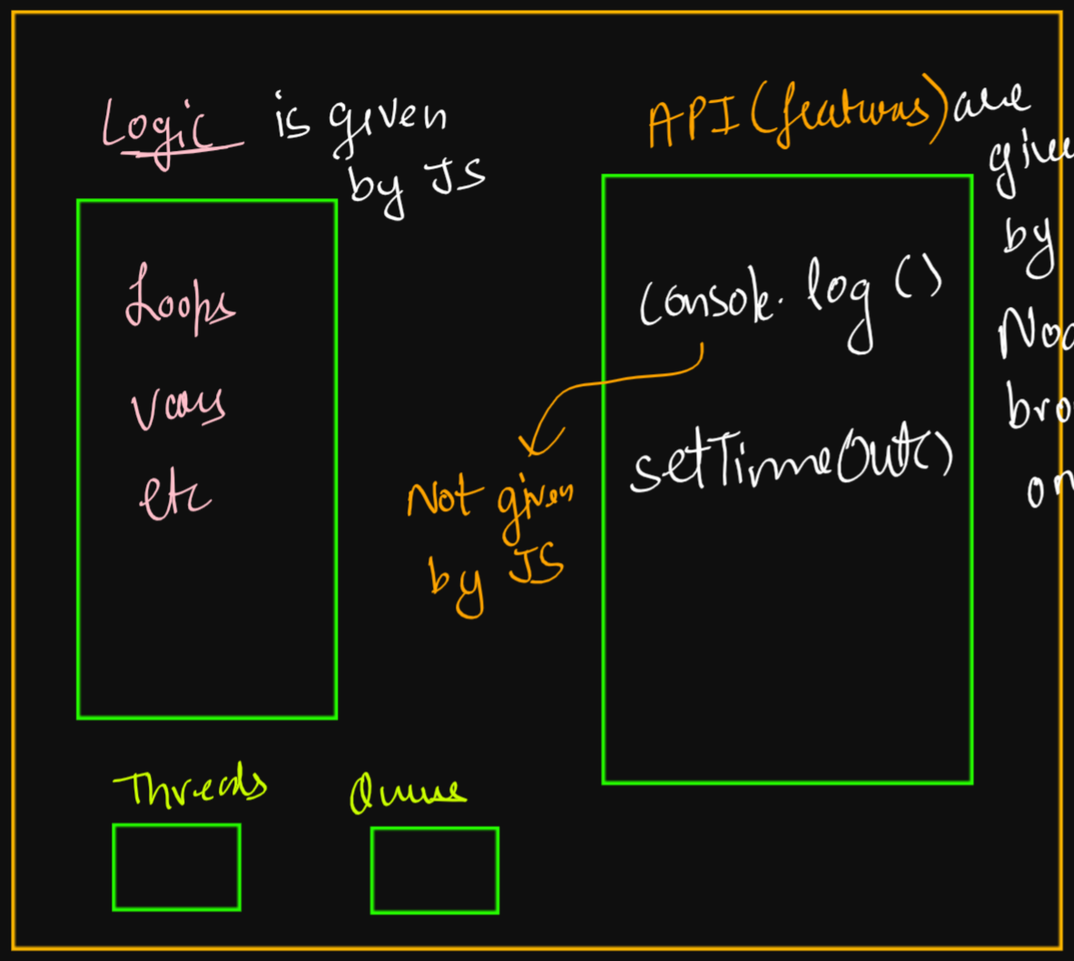
} JS uses/opportunities

*↳ Oracle owns the trademark to JS

Chrome → V8

Mozilla → SpiderMonkey

Apple (safari) → Webkit



(Javascript Runtime Environment)

↳ Basically, we do not need to install JS separately.

↳ We just install JSRE & JS comes with it.

↳ Browser

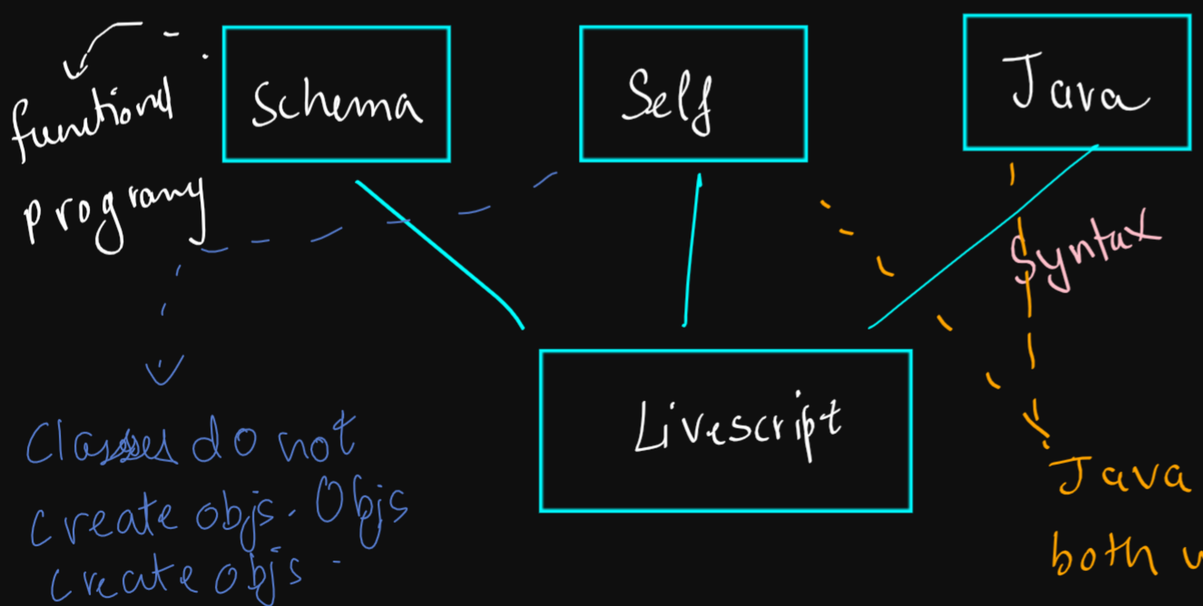
↳ Node js

↳ Node JS is the environment for running JS

↳ Vanilla Js → the Normal JS

↳ ES: ECMA Script (Now has various versions)

Created by Brendan Eich of Netscape in just 10days



Java & Self both were made by Sun Microsystems (Now purchased by Oracle)

Angular JS / Vue JS / React JS

↓

Optimised UI change

These are just frameworks.

Types of functions

→ function Statement

→ function Expression

→ IIfE

→ Anonymous function

→ Arrow function → very helpful in react (avoid use of this)

```
// arrow function
let fn = (a,b) => {
    return a * b;
}

// this can also be written as
let fn2 = (a,b) => a*b;

console.log(fn2(2,3));

// to find whether a number is prime or not
let isPrime = number => {

    for(let i = 2; i < number;i++) {
        if(number % i == 0) return false;
    }

    return true;
}

console.log(isPrime(14))
```

first class citizens

↳ can be assigned to a variable

↳ function can be passed as a parameter (callback)

↳ functions can be returned from a function (closure)

Lexical Scope and Scope Change

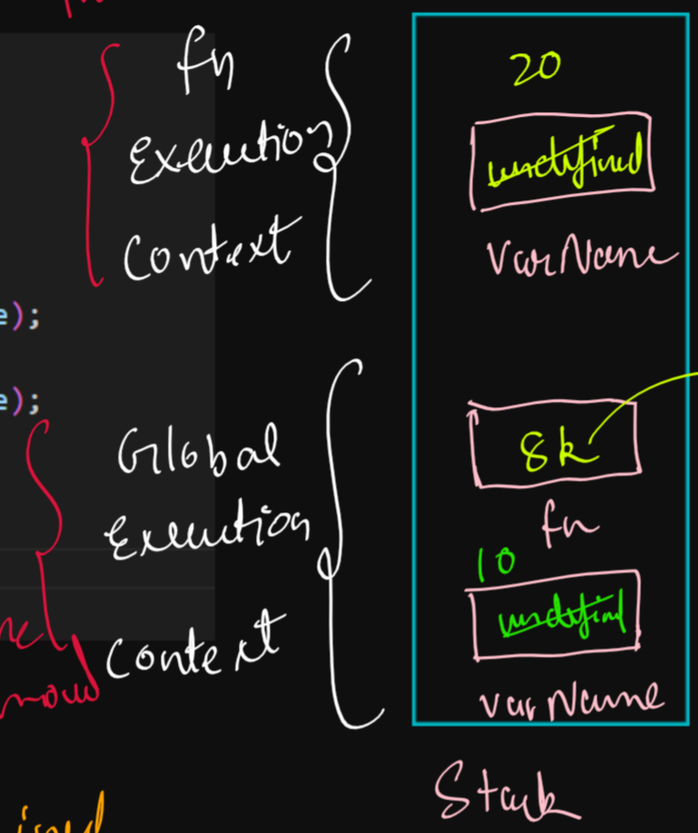
This will be removed

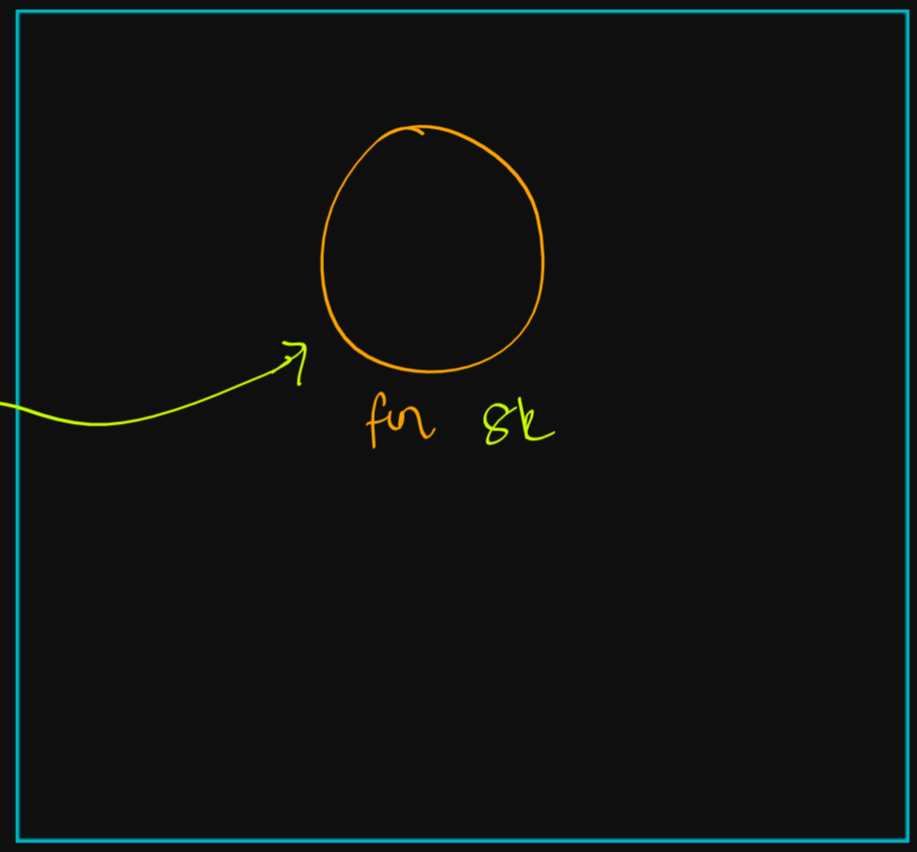
```
console.log("Line number 1", varName);

var varName = 10;
console.log("Line Number 4",varName);

function fn() {
  console.log("Line Number 7",varName);
  var varName = 20;
  console.log("Line Number 9",varName);
}

fn();
```

fn Execution Context

Global Execution Context

They this will be removed





① Line Number 1 undefined
② Line Number 4 10
③ Line Number 7 undefined
④ Line Number 9 20

Scope: Area where a function or a variable can be found.

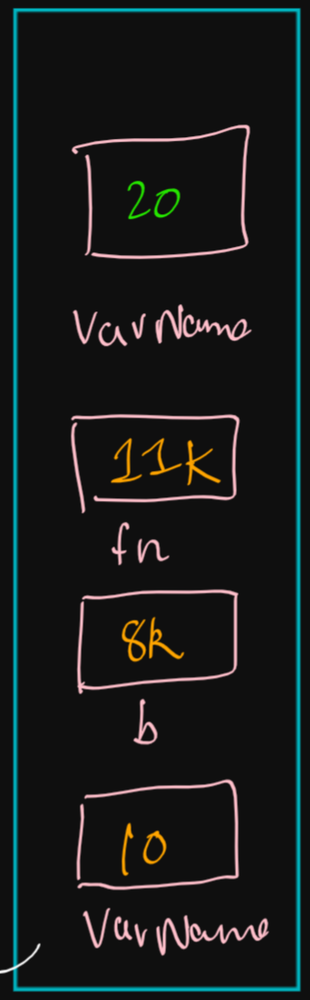
Scope Chain: When a variable is not found, func searches it outside then more outside called scope chain.

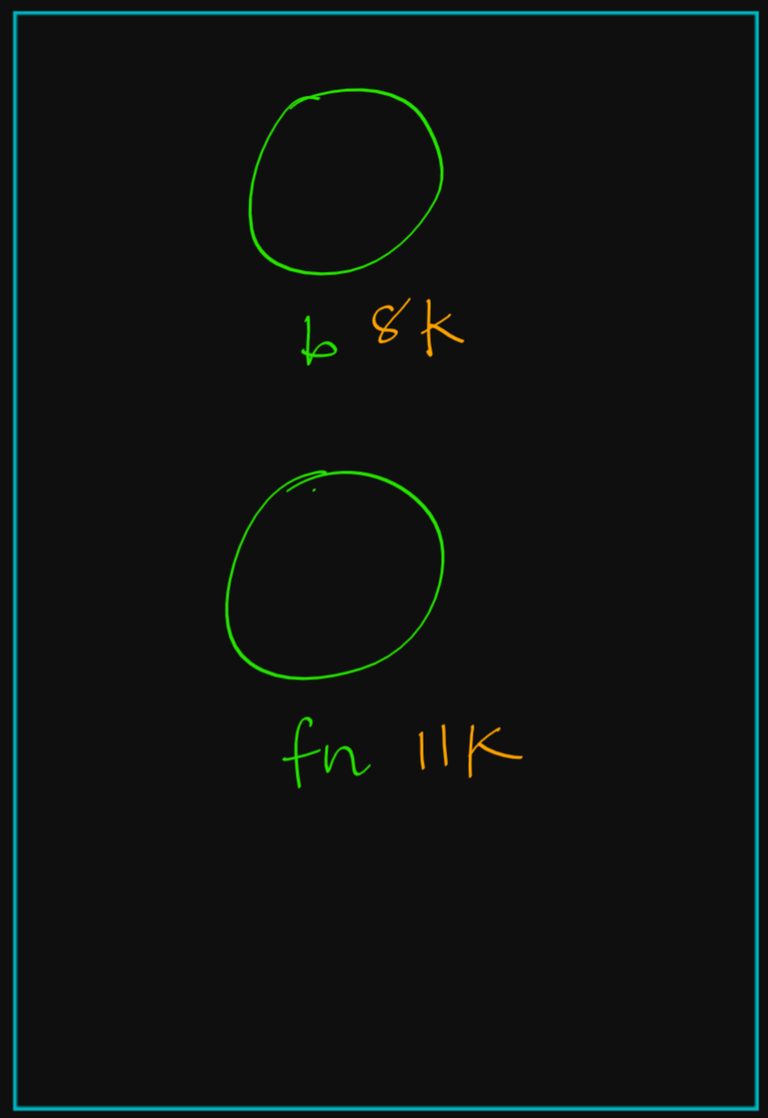
Lexical Scope: A variable i.e not inside a fun will be searched outside but outside of its declaration will be considered not call

```js
console.log("Line number 1", varName);
var varName = 10;

function b() {
    console.log("Line Number 5",varName);
}

console.log("Line Number 8",varName);

function fn() {
    console.log("Line Number 11",varName);
    var varName = 20;
    b()
    console.log("Line Number 14",varName);
}

fn();
```

fn Execution Context

Global Execution Context





① Line Number 1 undefined

② Line Number 8 10

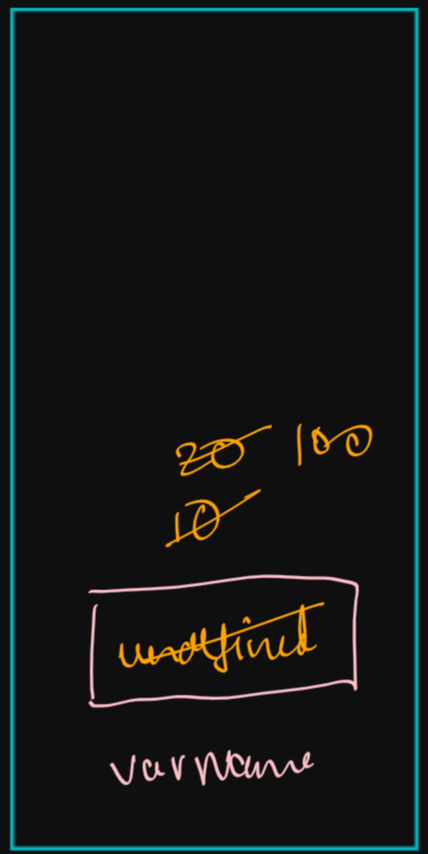
③ Line " 11 undefined

④ Line Number 5 10 → Due to lexical scoping, b() does not have varName & it searches for it outside the declaration of b & not the call.

⑤ Line Number 14 20

```
// hoisting
console.log("line number 2", varName); → undefined

// declare
var varName;

// assign
varName = 10;
console.log("line number 9",varName); → 10

// reassign
varName = 20;
console.log("line number 13",varName); → 20

// redeclare
var varName;
console.log("line number 17",varName); → 20

var varName = 100;
console.log("line number 20", varName); → 100
```

Global Execution Context



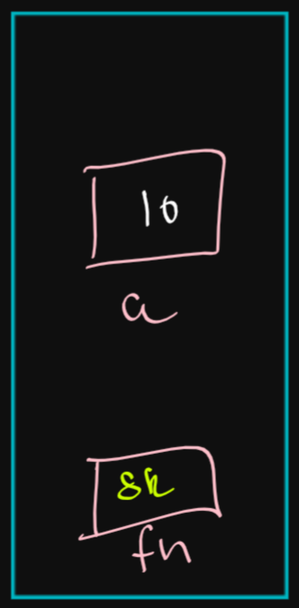
Stack

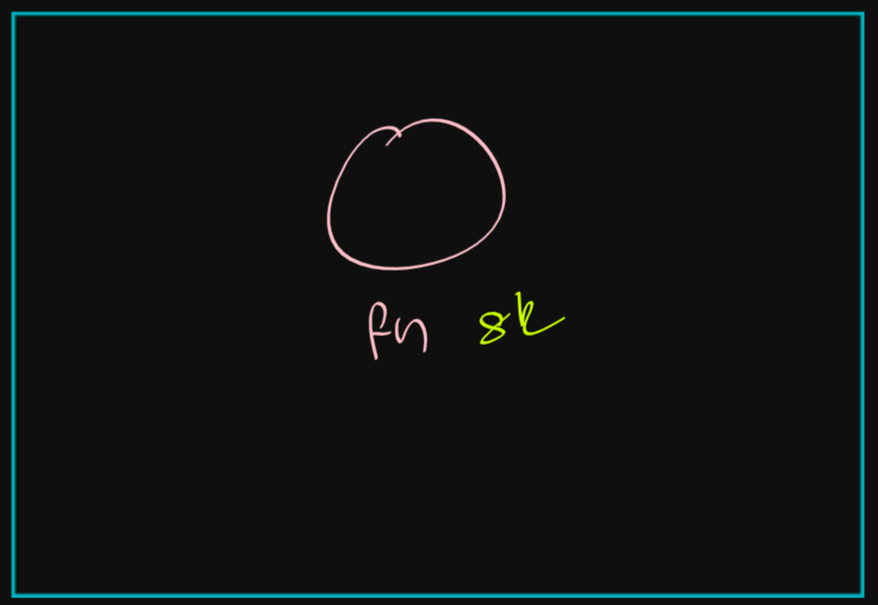
Var is function Scoped

```
function fn() {
    console.log(a); → undefined
    var a = 10;
    console.log(a); → 10

    if(a == 10) {
        var a;
        console.log(a); → 10
    }

    console.log(a); → 10
}

fn();
```

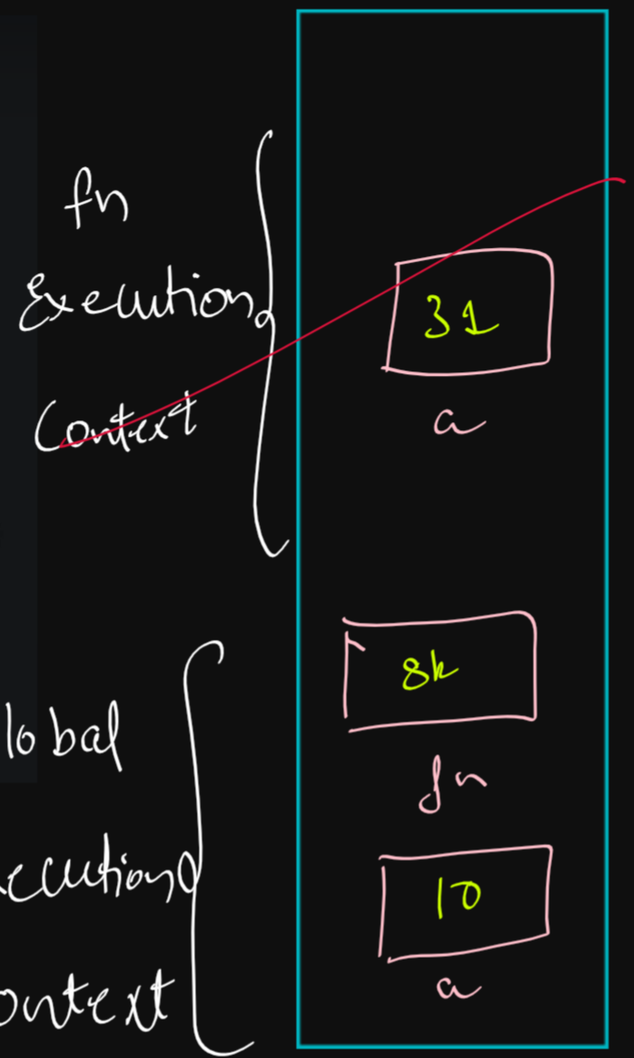
fn EC

GEC





```js
var a = 10;
console.log("line number 2", a);
function fn() {
    console.log("line number 4", a);
    var a = 20;
    a++;
    console.log("line number 7", a);
    if (a) {
        var a = 30;
        a++;
        console.log("line number 11", a);
    }
    console.log("line number 13", a);
}
fn();
console.log("line number 2", a);
```



line number 2 10
line number 4 undefind
" " 9 21
line number 11 31
line number 12 31
" " 2 10

This is a famous interview ques based on var.

Interview Qus based on Var & let

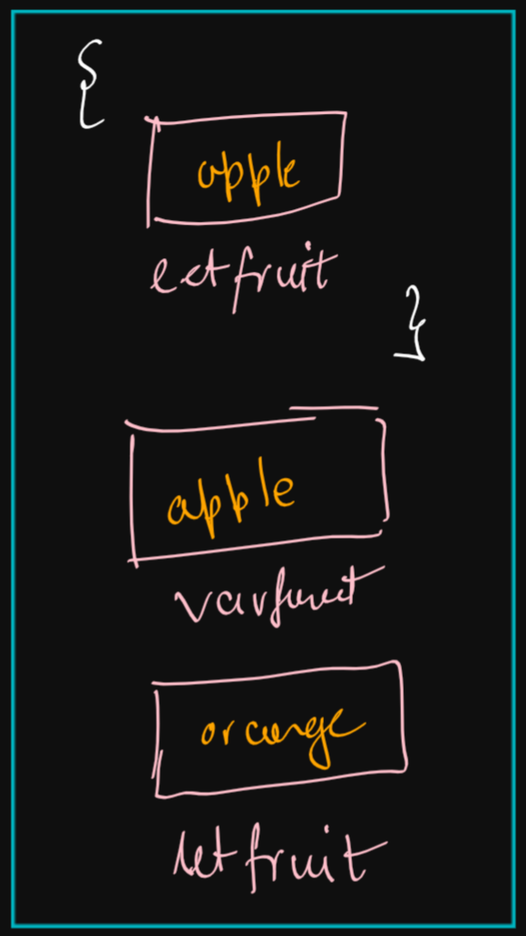
```
let letFruit = "orange";
var varFruit = "orange";
console.log("letFruit:",letFruit,"varFruit:",varFruit);

{
    let letFruit = "apple";
    var varFruit = "apple";
    console.log("letFruit:",letFruit,"varFruit:",varFruit);

}
console.log("letFruit:",letFruit,"varFruit:",varFruit);
```

↳ orange ↳ orange

↳ apple ↳ apple

orange apple

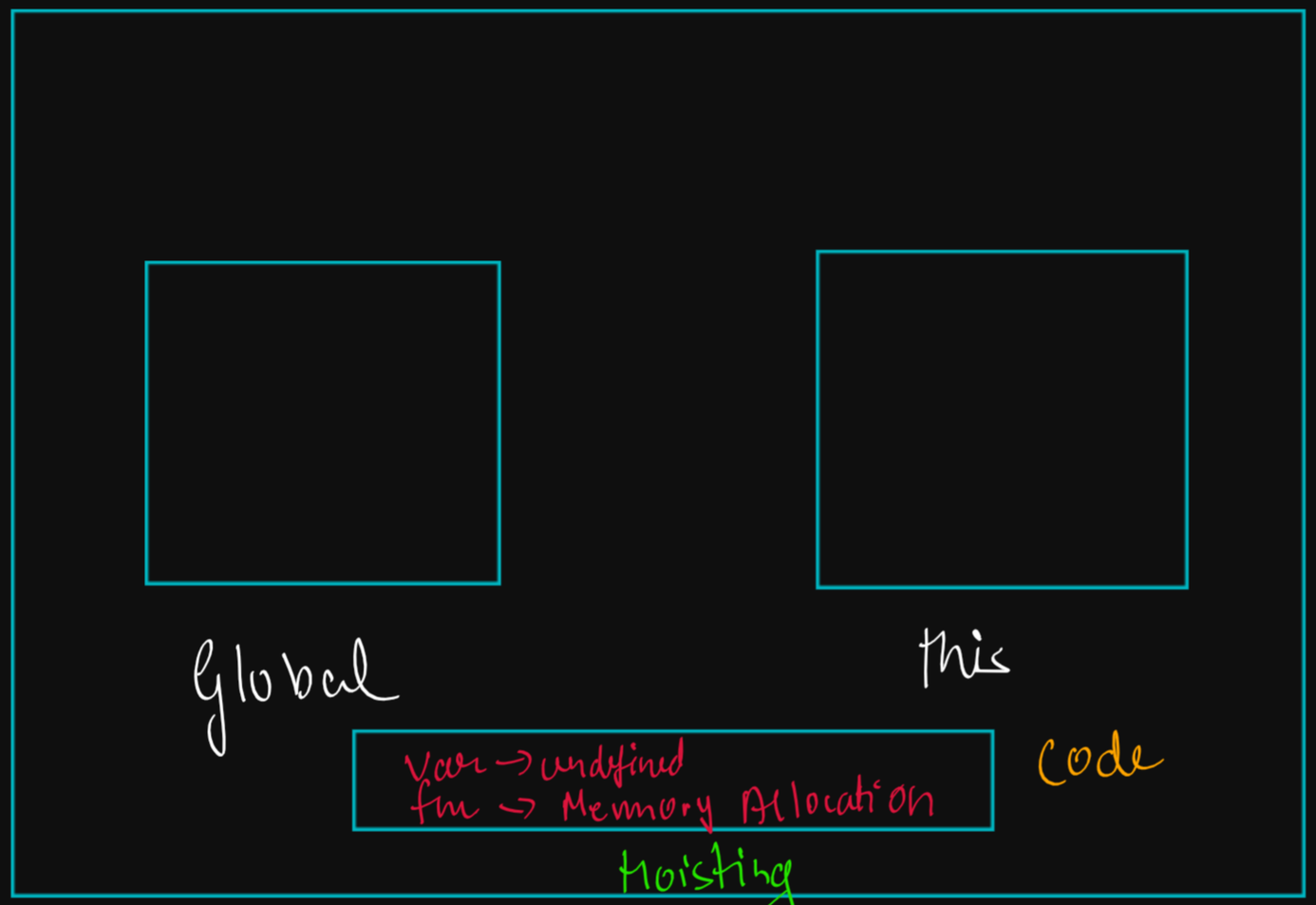
Global Execution context



Execution Context (Global & this)

Execution Context (Wrapper)

Creation Phase



in browser we get a window object which is the same as global in node js.

Also, this is an empty object by default in node whereas this = window in browser

Example Qus for Hoisting

```
console.log("varName", varName);   → undef
var varName;
console.log("varName", varName); → undef
varName = "Captain America";
console.log("varName", varName); → Captain Am
fn();  → Hello from fn
function fn() {
    console.log("Hello from fn");
}
fn(); → Hello from fn
fnContainer(); → Error: fnContainer is not a fun
var fnContainer = function () {
    console.log(" I am an Expression");
}
fnContainer();
```

Let and Const

Like var, this is also undefined

```
let varName;
console.log("varname value at line number 3",varName)

// assignment
varName = 10;
console.log("varName value at line number 7", varName);

// re-assignment
varName = 20;
console.log("varName value at line number 11", varName);
```

→ undefined

→ 10

→ 10

However JS is an interpreted lang, the re-declaration of let & const is checked at the time of hoisting only. Hence, not even a single line will be executed in that case & we will get an error.

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64 ~/Desktop/Web Development Complete/JS Interview Series (main)
$ node let.js
C:\Users\Guneet Malhotra\Desktop\Web Development Complete\JS Interview Series\let.js:14
let varName;
    ^

SyntaxError: Identifier 'varName' has already been declared
←[90m    at Object.compileFunction (node:vm:352:18)←[39m
```

Temporal Dead Zone

** Let & const are Block scoped

Temporal Dead Zone for let and const variables is that area from where the file starts parsing to where the variable is declared. A let/const variable in TDZ cannot be accessed. This means that we cannot access let/const variables before declaring.

```js
// Temporal Dead Zone

console.log("Hello")
console.log(varName);

let varName;
console.log("varname value at line number 3",varName)

// assignment
varName = 10;
console.log("varName value at line number 7", varName);

// re-assignment
varName = 20;
console.log("varName value at line number 11", varName);
```

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64 ~/Desktop/Web Development Complete/JS Interview Series (main)
$ node let.js
Hello
C:\Users\Guneet Malhotra\Desktop\Web Development Complete\JS Interview Series\let.js:4
console.log(varName);
            ^

ReferenceError: Cannot access 'varName' before initialization
    at Object.<anonymous> (C:\Users\Guneet Malhotra\Desktop\Web Development Complete\JS Interview Ser
    at Module._compile (node:internal/modules/cjs/loader:1101:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1153:10)
```

Hello got printed. This means that this does not create a problem during hoisting. It is not an error, just a safety measure.

** const is same as let.

Just it cannot be re assigned.

So, assign value while creation only.

Shadowing takes place in case of both let and const.

```
// variable shadowing
let fruits = "apple";
console.log(fruits); → apple
{
    // console.log(fruits) TDZ
    let fruits;
    console.log(fruits); → undefined
    fruits = "orange";
    {
        console.log(fruits) → orange ("orange" has shadowed the "apple" variable)
    }
    console.log(fruits);
}
console.log(fruits);
```

this is variable shadowing.

** Also, if var is outside & let is inside, this is also a case of legal shadowing.

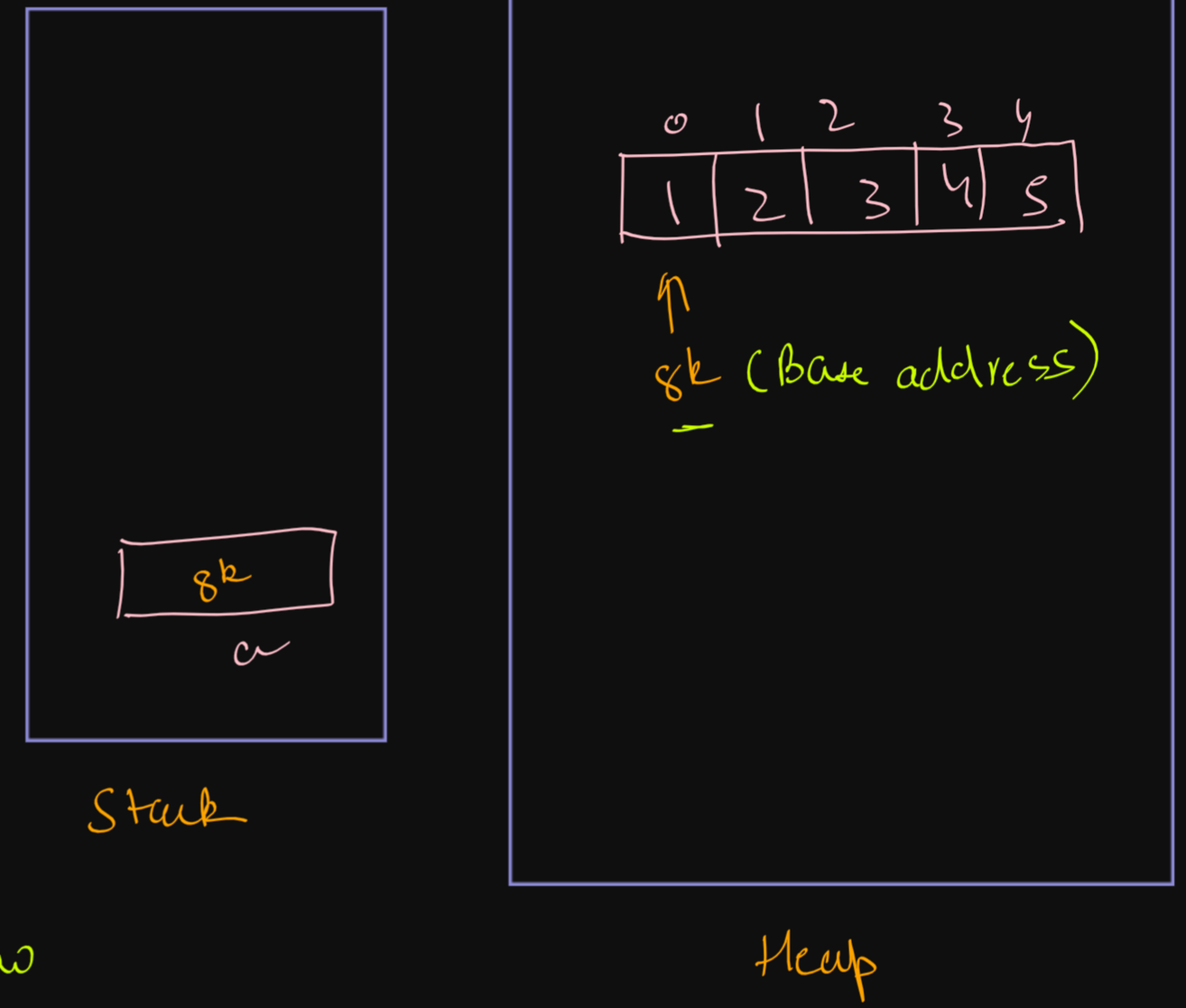
outside the block let/const & inside var is illegal shadowing. (DO NOT do this)

Table for Help

Keyword	Scope	Reassign	Redeclare	TDZ
var	function	✓	✓	X
let	block	✓	X	✓
const	block	X	X	✓

```
// const a = 10;
// We should get an error as we are re-assigning the const value
const a = [1,2,3,4,5];
console.log(a);

a.pop();
console.log(a)

// How and why is this allowed?
a.shift();
console.log(a);
```

We should not be allowed to remove first ele from a const array, but it is. So, what exactly are arrays in JS & how are they working?



JS does not have any Arrays

```js
let array = [1,2,3,4,5]
console.log(array)

array.myProp = "myProp";
console.log(array)

array.fun1 = function() {
    console.log("Hello from the array");
}

console.log(array)

array[95] = "95th element";
console.log(array)

// length only returns the highest key value in number + 1
console.log(array.length)

console.log(array[10])
```

```
$ node array.js
[ 1, 2, 3, 4, 5 ]
[ 1, 2, 3, 4, 5, myProp: 'myProp' ]
[ 1, 2, 3, 4, 5, myProp: 'myProp', fun1: [Function (anonymous)] ]
[
  1,
  2,
  3,
  4,
  5,
  <90 empty items>,
  '95th element',
  myProp: 'myProp',
  fun1: [Function (anonymous)]
]
96
undefined
```

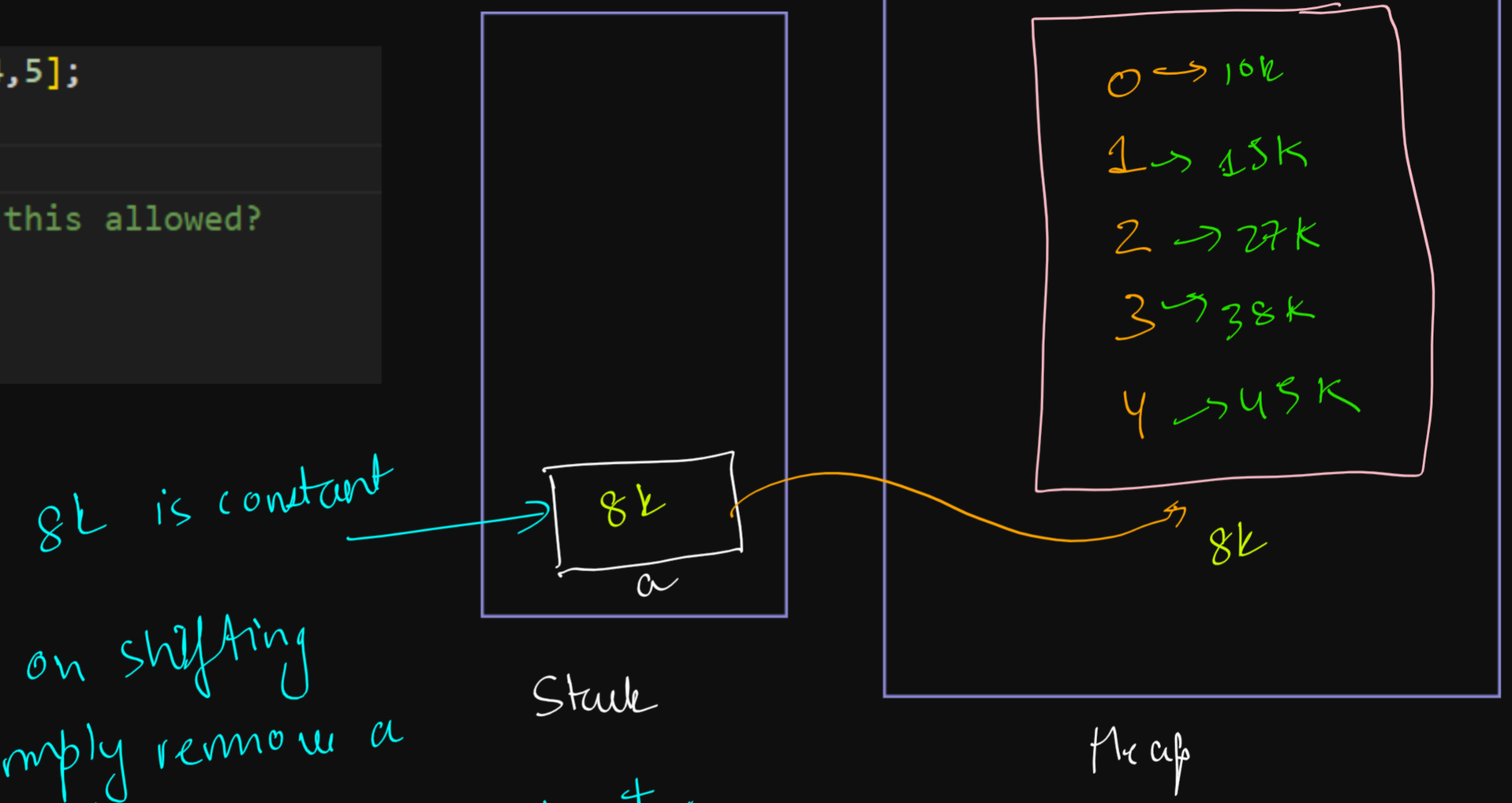
→ just for visual

for in loop works same in the array as for an object.

```js
for(let key in array) {
    console.log(key, ":", array[key])
}
```

```
0 : 1
1 : 2
2 : 3
3 : 4
4 : 5
95 : 95th element
myProp : myProp
fun1 : [Function (anonymous)]
```

```
const a = [1,2,3,4,5];
console.log(a);

// How and why is this allowed?
a.shift();
console.log(a);
```



So, on shifting
we simply remove a
key-value pair from the object.
& the address &k is still constant

function As an Object

```js
function fn() {
    console.log("I am a function");
}

fn.myProp = "property of a function";
fn.myMeth = function() {
    console.log("I am a method of a function");
}

console.log(fn);
fn()
fn.myMeth()
console.log(fn.myProp)
```

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64 ~/Des
Web Development Complete/JS Interview Serie
in)
$ node functionAsAnObject.js
[Function: fn] {
  myProp: 'property of a function',
  myMeth: [Function (anonymous)]
}
I am a function
I am a method of a function
property of a function
```

functions are also objects (key/value pairs)

extra feature → code property

that can be executed when we invoke that function

Everything is object. Only 6 primitives: number, string, boolean, null, undefined & symbol

Introduction to functional Programming

Imperative vs declarative code writing

```
// check whether the square of a number is even or not

// Imperative way

const number = 4;

const noSquared = number * number;

let isEven;

if(noSquared % 2 == 0) {
    isEven = true;
} else {
    isEven = false;
}

console.log(isEven)

// Declarative way
let isSquareEven = x => ((x*x) % 2 == 0)
console.log(isSquareEven(5))
```

Pure & Impure functions -

```
// add 2 numbers

let a = 2;

function add(num) {
    console.log("The sum is", num + a);
}

// If we change the value of a then the output will be diff.
// This means that the add function gives diff result with same parameters
// This is an impure function
add(5);
```

```
// This is a pure function as it will give same output for the same params always

function addition(a,b) {
    console.log("The sum is ", a + b); //Since we are printing on the console, it is having an external effect.
    // This is called as side effect.
}

addition(5,10)

// In functional programming, we try to write pure functions without side effects.
// However, pure funcs with side effects if needed are also good. No problem with that.

// pure function without side effect
function addition2(a,b) {
    return a + b; //apart from the result of the func, nothing is affected hence there are no side effects
}

console.log(addition2(10,20))
```

Mutability vs Immutability

Mutability is a problem

```
const obj1 = {
    Name: "Guneet",
    Age: 21
}

const obj2 = obj1;

obj2.Name = "Lavish"; //This change will also be reflected in obj1

console.log(obj1);
console.log(obj2);
```

```
Guneet Malhotra@LAPTOP-GUNEET MI
$ node Immutable.js
{ Name: 'Lavish', Age: 21 }
{ Name: 'Lavish', Age: 21 }
```

We can overcome this by using Object.assign() or spread operator.

```
const obj1 = {
    Name: "Guneet",
    Age: 21
}

const obj2 = Object.assign({},obj1);

obj2.Name = "Lavish"; //This change will not be reflected in obj1

console.log(obj1);
console.log(obj2);
```

```
Guneet Malhotra@LAPTOP-GUNEET MINGW64
$ node Immutable.js
{ Name: 'Guneet', Age: 21 }
{ Name: 'Lavish', Age: 21 }
```

const obj2 = {...obj1}

This will also do deep copy