Javascript

Came into being in 1995. It is used to add programs to web pages by Netscape for its browser called Navigator. Later other browsers made themselves compatible with JS.

ECMA script is the same a JS. This is where JS was standardised by all browsers.

* To declare variables in JS:

1. Using var
   var a;

Variable initialization
a=10;

To print
console.log(a);
O/P : 10

* typeof operator : it tells us the type of data stored in a varia
  console.log(typeof a);
  O/P : number

* Javascript is a synchronous and single threaded language.

line-by-line in a specific order

* To run JavaScript file in browser:

1. Create a .js file (index.js)
2. Create a html file.

```
<|body>
<script src="index.js"><|script>
<|html>
```

So, script tag is used to link .js file

3. go live click on inspect and then console.

* ' ' single quotes
" " double quotes
` ` backtick ← Here \n is not required to write in a new line

these are all used to write strings

* \n Backslash n is used to write in new line.

* var num = 200;
console.log(`half of ${num} is ${num/2}`);
${} - makes computation
so, output is dynamic here as it changes according to the value user gives to num. o/p: half of 200 is 100.
if we write this in " " it will be printed like half of ${num} is ${num/2}

Page No.
Date
YOUVA

Problems with var :—

1. Redeclaration is allowed.
To over this problem 'let' keyword is used.

```
let l = 100;
console.log(l);

let l = "200";
console.log(l);
```

We get a syntax error as l has already been declared as number. We can reinitialize a ~~var~~ variable but not re-declare it inside a block.

```
let l=100;
console.log(l);
let l = 300;
console.log(l);
```

O/P 100
300

* Loops in js - same as java.

(2) var is function-scoped in case of fun^n otherwise var is global scope while 'let' is block-scoped. So, if we define a variable using var it allows access outside the block while let lets us access the variable declared only inside that block.

eg:

```
num=10;
for (var i=0; i<num; i++) {
    if (i%2==0) {
        console.log(i);
    }
}
console.log("value of i is"+i);
```

O/P
0
2
4
6
8

Value of i is 10

Had i been let i, line → would not have printed as let is block scoped.

So, i has been accessed outside for loop

* const keyword - can neither be re-declared nor re-initialised.

```
const a = 2;
console.log(a);
a=4;
```

← will create typeerror as 'a' has been assigned to constant variable.

doubt : fun scoped & global scoped difference?

* Three types of functions

1. Normal function

Fun^n Def^n → function function_name (parameter1, parameter2) {

}

Function invoking / calling

function_name (arg1, arg2);

eg:
```
function add (a, b) {
    return a+b;
}
```
← defining fun^n called add.

```
let ans = add(2,6);
console.log(ans);
```
→ calling add fun^n with arg 2,6 & storing in ans

() ← used to invoke

O/P : 8

* Functions are treated as first class citizens in javascript
i.e., functions are very important as
1) They can be returned
2) They can be passed as parameters / arguments
3)

② ~~Function Expression~~

Example –

```
function calculator ( str, a, b) {
    if ( str == 'add') {
        return function add () {
            console.log (a+b);
        }
```

make code for
sub
mult
div

```
LO    let returnedFunc = calculator ('add', 2, 3);
L1    console.log(" returned function is \n" + returnedFunc)
L2    returnedFunc();
```

← This line invokes returnedFun

O/P returned function is
return function add () {
console.log(a+b);
}
5

Here returnedFunc is passed as add function in L1
L2 invokes returned fun from L0 & computes 5.

③ Function Expression

```
let sayHi = function () {
    console.log (" Hello guys I'm function expression");
    }
console.log (" " + sayHi);
sayHi();
```

O/P :- function () { console.log (" Hello guys I'm function expression")
}
Hello guys I'm function expression

(A) IIFE (Immediately Invoked Function Expression)

To invoke a function at the same time when it is defined

Usually,

```
function add (a,b) {
    return a+b
}
add (2,3);
```

← invoking add funⁿ

In IIFE

```
let additionIIFE = (function add (a,b) {
console.log ~~return~~ (a+b);
})(20,30);
```

for invoking

× Array - It is a collection of elements. In javascript you can store values of different types in an array. An array has a contiguous allocation of memory

```
let cars = ['BMW', "Audi", "MG", 1, 2, 3, true],
```

Accessing elements of an array

```
console.log (cars[0]);
console.log (cars[3]);
```

O/P BMW
1

Adding elements in an array

```
cars [7] = "Honda",
console.log (cars);
```

O/P ['BMW', 'Audi', 'MG', ~~Maru~~ 1, 2, 3, true, 'Honda']

Page No.
Date

Replacing elements in an array

```
cars[3] = "Mahindra";
console.log(cars[3]);
```

O/P: ['BMW', 'Audi', 'MG', 'Mahindra', 2, 3, true, 'Honda']

Method of an array

① Pop-method - This method removes the element from the array

```
cars.pop();
console.log("after popping the elements\n"+ cars);
```

O/P ['BMW', 'Audi', 'MG', 'Mahindra', 2, 3, true, 'Honda']
after popping the elements
['BMW', 'Audi', 'MG', 'Mahindra' 2, 3, true]

② Push-method - This pushes a new element at the end of an array.

```
cars.push('Tata');
console.log("after pushing an element");
console.log(cars);
```

O/P: ['BMW', 'Audi', 'MG', 'Mahindra' 2 3 true 'Honda']
after pushing an element
['BMW', 'Audi', 'MG', 'Mahindra', 2, 3 true 'Honda', 'Tata']

③ unshift method - This adds element at the start of an array

```
cars.unshift("JLR");
cars.unshift("Mustang");
console.log(cars);
```

O/P ['JLR', 'Mustang', 'BMW', 'Audi', 'MG', 'Mahindra' 2 3 true 'Honda', 'Tata']

④ Shift Method - it removes element from 0th index of an array

```
cars.shift();
cars.shift();
console.log(cars);
```

O/P : ['BMW', 'Audi', 'MG', 'Mahindra' 2 3 true 'Honda', 'Tata']

* To get the length of an array

```
console.log(cars.length);
```

O/P : 8

* 2D Array - Array of arrays

```
let array2D = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

console.log(array2D);
console.table(array2D);
console.log(array2D[1][2]);
console.log(array2D[1]);
```

O/P [[1,2,3], [4,5,6], [7,8,9]]

(index)	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

6
[4, 5, 6]

* Array of strings ["a", "b", "c"]
* Array of numbers [1, 2, 3]
* let res = array2d[1];
* console.log(res);
console.log(res[2]);

} — chaining of arrays

O/P: [4, 5, 6]
6

* let array2D = [['BMW', 2, null, 45],
[4, true, 6,]
[7, '8', 9,]
[7, '8', 9]

console.log(array2d[1][3]);

O/P undefined

console.table(array2d);

O/P

index	0	1	2	3
0	'BMW'	2	null	45
1	4	true	6	
2	7	'8'	9	
3	7	'8'	9	

* (array2d.length) ← gives the no. of elements in our array. ← gives no. rows in a 2d array

There are 4 elements. Each element is an array is a sep thing.

* To get the no. of columns we select a row
console.log(array2d[0].length)
O/P - 4
console.log(array2d[1].length)
O/P - 3

* To put values in a 2d array

```
array2d[i][i] = false;
console.table(array2d);
```

O/P

index	0	1	2	3
0	'BMW'	2	null	45
1	4	false	6	
2	7	'8'	9	
3	7	'8'	9	

* To change directory
Type in terminal : cd foldernane (in MAC)
change directory

STRINGS

A string is a sequence of characters.

```
var str = "Hello Pepcoders";
//        0123456789101112131415
console.log(str);
console.log(str[4]);
console.log(str.length);
```

O/P Hello Pepcoders
0
15

String Methods

1) Extraction Method

i) Slice method :

str.slice (start index, end index) – Here start index element is included while element at end index is not included.

Here if end index < start index an empty string ("") is returned

```
var slicedstr = str.slice (6,15);
console.log (slicedstr);
console.log (str);
```

O/P : Pepcoders
Hello Pepcoders

```
var slicedstr = str.sliced (6);
console.log (slicedstr);
```

O/P Pepcoders

When no ending index is specified it means from 6 till the end

```
var slicedstr = str.slice (2, -4)
console.log (slicedstr);
```

O/P llo Pepco

start Printing from index 2 & leave the last 4 elements or print till (15-4=11th index)

(ii) Substring method

* str.substr (start, length)

substr is considered a legacy in ECMAScript & could be removed from future version.

eg: str.substr (2,6) – begin from index 2 and capture 6 elements then con. including index(2).

O/P llo pe

* substring() – it is different from substr.

Syntax: str.substring (start index, end index) – where ending index is not included. Eg: let anyString = 'Mozilla'

console.log (anyString.substring (0,1));
console.log (anyString.substring (1,0));

Both display 'm' → if end index < start index the effect of substring is as if they were sw

console.log (anystring.substring(0,7))
console.log(anystring.substring(0,10))

Both display Mozilla. so we can say any argument value < 0 or > string length is treated as if it was 0 / stringlength

if both the arguments are negative console.log(anystring.substring(-5, -2)) ⇒ "" ""

Page No. Date

* Replacing-strings

```
str = "YOLO";
console.log(str);
```

O/P YOLO

* console.log(str.toLowercase());

O/P yolo

* console.log(str.toupperCase());

O/P YOLO

* **Concatenation Method** : To combine strings

```
let firststr = "You guys";
let secondstr = " are smart";
let concatenatedstr = firststr + secondstr;
console.log(concatenatedstr);
```

O/P : You guys are smart

* **Concat method**

```
let concatstr = firststr.concat(secondstr, " and me");
console.log(concatstr);
```

O/P You guys are smart and me

* **Trim method** - Removes all the whitespaces of starting and ending in a string

```
let trimstr = "        Hello        how are you        "
console.log(trimstr);
console.log(trimstr.length);
```

```
console.log(trimStr.trim());
console.log(trimStr.trim().length);
```

O/P : Hello how are you

77

Hello how are you

23

* split() divides string on the basis of argument & puts them in an array

```
var str = "Hello my name is Dee"
console.log(str);
// Hello my name is Dee
var a = str.split("e");
console.log(a)
// [H', llo my nam', 'is D']

var a = str.split(" ");          // splits on the basis of spac
console.log(a)
// ['Hello', 'my', 'name', 'is', 'Dee']

var a = str.split("");           //Here ∄ any space between
console.log(a);                  जहाँ space नहीं है वो basis pe split
// ['H','e','l','l','o', 'm','y','n' 'a','m','e', 'i','s','D','e','e']
var str = "Hello hello my name is hello Dee"
console.log(str.split("hello"));
// Hello my name is Dee
```

The arguments on whose basis is split done that thing is removed from string.

Date:

Objects in javascript

var str = "Hello";

Properties - A variable inside an object

eg str.length // gives 5.

Two ways to access property

1. dot notation (like above) str.length
2. Square bracket notation str["length"]

Method - A function inside an object

eg: str.toUpperCase() // HELLO

↑ a method

Objects - Objects consist of properties and methods.
- They exist in Key value pair
- A collection of data

1. let obj = { }
console.log(obj);

On terminal we get { } // ← empty object

2. let person = {
name : "Dee",
age : 25,
phone : 9650534947,
gender : "female",
height : "154 cm"
},

values

each key-value pair is seperated by a comma.

Key - a key may be a property or a method here these are all properties

console.log(person); // prints the object called person ie, the whole information

console.log(person.name); // prints Dee

console.log(person.gender); // prints female

```
console.log(person["gender"]);   // female
                                 // This is square bracket
                                 // notation.
```

3. Let

```
captainAmerica = {
    firstName : "Steve",
    lastName : "Rogers",
    friends : ['Bucky', 'Tony Stark', 'Bruce Banner'],   // Object can contain array
    age : 102
    isAvenger : true,
    address : {                                          // object inside an object.
        state : "New York",                              // ie, Nesting
        city : "New York City",
        country : "USA"
    },
    sayHi : function() {                                 // Method - sayHi
        console.log('Hello my name is ${this.firstName}');
    }
};
console.log(captainAmerica);   // The whole information inside captainAmerica is printed
```

line 35 ?

A function inside an object is called method

a keyword 'this' means the object inside which we are. Here captainAmerica.firstName

```
console.log(captainAmerica.friends);   // ['Bucky', 'Tony Stark', 'Bruce Banner']
or console.log(captainAmerica["friends"]);
console.log(captainAmerica.friends[0]);   // Bucky
```

~~console.log~~

∵ sayHi is a funⁿ. calling a funⁿ needs ()

```
captainAmerica.sayHi();   // Method Accessing
                          Prints: Hello my name is Steve
```

* Accessing a method needs () while property access needs no ().
* 'in' keyword in JS is used to get keys in an object

```
for (let key in captainAmerica) {
    console.log(key);   // Prints all keys of captainAmerica
}
```

in keyword

~~will not~~

~~because it~~

just a variable Name may be anything Not necessarily key

console.log(captainAmerica[key]);

Syntax: console.log(objectName[variable]);

// Bracket notation: this statement gives the values binded in the variable called key. So values in key change. 1st binded value is firstName, 2nd lastName & so on.

* let arr = ["a", "b", 1, true];

```
let arr = {
    0 : "a",
    1 : "b",
    2 : 1,
    3 : true
}
```

In javascript everything is an object. Array in JS works inside like arrr.

* When we know a key inside an object we use • notation to access its value.

eg: captainAmerica.age will give 102

key - we can write kooth, ghoda, anything does not matter

```
for (let key in captainAmerica) {
    console.log(key);
    console.log(captainAmerica[key]);
}
```

// we get keys of object */

L1: console.log(key);
L2

Prints the value of the resp. key

This for loop works like:

L1 firstName
L2 Steve
L1 lastName
L2 Rogers
L1 friends
L2 ['Bucky', 'Tony Stark', 'Bruce Banner']
L1 age
L2 102
L1 isAvenger
L2 true
L1 address
L2 { state: "New York", city: "New York City", country: "USA" }

L1 sayHi

L2 [Function: sayHi] // looks like it because
// we did not put () with sayHi above

~~captain~~ To check the type of a key

console.log (typeof captainAmerica.isAvenger); // Gives booleans

console.log (typeof captainAmerica.friend); // gives object as in JS there is nothing called array Array is an object too

To see it practically: How to start debugger
1. Add the name of this js file in <script> of .html file
2. Right click on inspect, Select the lines
3. Click sources
4. Click scope & call stack by clicking on next line

Q- How does code executes in JS?

```
L1  var num = 5;
L2  function cube(num){
L3  var res return num * num * num;
L4  return res;
L5  }
L6  var cube1 = cube(num);
L7  var cube2 = cube(10);
L8  console.log(cube1);
L9  console.log(cube2);
```

If not commented these two will be displayed after L9 Global Ex. Context will be destroyed

Global Execution Context

Memory Creation Ph	Code Execution Phase
num = defined 5	num = 5
cube = { entire function }	when cube() is called: MCP: num = undef 5, res = undef 125; CEP: num = 5 — when return is encountered CEP is destroyed
cube1 = undefined 125	when cube(10) is called: MCP: num = undef 10, res = undef 1000; CEP: — return destroys CEP
cube2 = undefined 1000	

Phases in Execution Context

1. Memory Creation Phase
Skims through the code and allocates memory to all variables and functions.
Variables are not given value here hence we find undefined.
value = undefined also takes memory so memory has been allocated is proved

Suppose if code is
```
L1 var num = 5;
L2 function cube(num){
L3 var res = num*num*num;
L4 return res;
L5 }
L6 var cube1 = cube(num);
L7 var cube2 = cube(10);
```

2. Code Execution Phase
JS runs the code line-by-line. It is the point where all the functions and calculations are executed.
Like, variable are given values.
functions are performed. Whenever a function is called a ~~when~~ new execution context is created. This is local code execution context. L1 is executed in Global Execution Context.

- Local Execution contexts
After L7 code ends so Global Execution Context is destroyed

When L2 is reached we encounter a fun called cube
When L3 is reached we see cube(num) being called.
So a local ex. context is formed. In MCP num = undefined, res = undefined, in CEP when no value for num is found locally. System moves to 2 lines prior to it and if finds num like here num = 5 in L1 allocates this to local variable. So, local variables are preferred.

NOTE: Print commands do not take part in memory creation or Processing & code execution phase console works

M	W	T	F	S	S
Page No.:				YOUVA	
Date:					

so, num = 5 then L3 is executed res = 5×5×5 = 125
L4 returns res (=125) to L6 and local CEP is destroyed
var cube1 = 125

Then L7 is executed we find cube(10), a function. So, local execution context is formed

MEP	CEP

Here,

10 is passed as local variable. so, num = 10, L3 is executed res = 1000, L4 returns 1000. Local Ex. Code is destroyed. ∴ cube2 = 1000

After L7 has been executed code is completed and global execution context is destroyed

* if var ans3 = cube();
console.log(ans3);
When cube() is called no argument has been passed
In L2 num will be undefined
L3 res = undef * undef * undefined
L4 return res // will return NAN
i.e, Not A Number

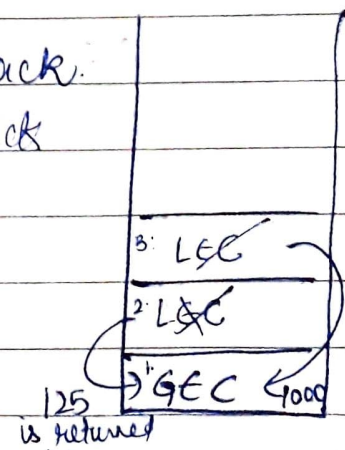
console | NaN |

NaN means non sensical calculation

---

* Call stack/ - It is used to manage execution context in J.S.
Program stack/ Control stack

1. Global execution context is pushed inside call stack.
2. Local execution context is pushed inside call stack
3. After return statement is encountered, LEC is destroyed (LEC is popped)
4. After all the lines of code are executed GEC is destroyed (popped)



125 is returned to GEC

Note: undefined and null are almost same.
Although They may be used inter-changeably due to system design

Page No.:
Date:

Automatic Type Conversion

Datatypes: number string boolean undefined null

Eg: ① var ans = "5" + 1; // 1 is converted automatically to string
console.log(ans);
[51] console

② var ans = 1 + "5";
console.log(ans); // 1 is converted to string
[15] console

∴ Precedence: string > int

③ console.log(null * 5); [0] ← console // Null is converted to 0 ∴ 0.
④ console.log(undefined * 5); [NaN]
⑤ console.log("5" - 1); [4] // String is converted into number
Remember + is the sign for concatenation ∴ the difference b/w ①, ② & ⑤ *

⑥ console.log(null * "5"); [0] // Null is converted 0 Here "5" is converted to number ∴ we get 0.

⑦ console.log("ten" * 3); [NaN] // String * num = Non sense

On comparing ⑥ & ⑦ we get a difference even though both kind of have string & No.: in ⑥ "5" can be mapped to a no. 5 ∴ 0*5=0 but in ⑦ "ten" cannot be mapped to a no. so we have String * Number = NON SENSE

⑧ console.log(3+4+"5"); [75]

Imp for interview

M	T	W	T	F	S	S

Page No.:
Date: 22.02.22
YOUVA

* Hoisting

```
L1  var x=10;
L2  function greet(){
L3    console.log("Hello")
L4  };
L5  console.log(x);
L6  greet();
```

Memory	Code
x: ~~undef~~ 10	L6 greet is called
L3 → greet: f{	M | C
}	no var in greet() → console

Console
10
Hello

Now,

```
L1  console.log(x);
L2  greet();
L3  var x=10;
L4  function greet(){
L5    console.log("Hello")
L6  };
```

Memory	Code
L1 x: ~~undef~~ 10	L2 greet() is called
L5 → greet(){	M | C
}	no var in greet()

on execution of L3

Console
undefined
Hello

console(x)

Now when we will reach L3 x=10 in Memory but as there is no print command for it henceforth, our console will look like [undefined, Hello] because when x was consoled it was undefined

JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables to the top of their scope, prior to the execution of the code.

Basically, it follows the memory diagram.

Advantage: If a variable is undefined it does not hamper the rest of the code.

Like here we have accessed variables and function even before we have ~~initialized~~ declared their values. (like x and greet)

* function expression where a fun^n is stored in a variable.

eg: ① function bye() { ← A fun^n

}

② var hello = function bye() { — A function expression

}

Examples

```
L1  console.log(a);
L2  greet();
L3  console.log(hello);
L4  hello();
L5  var a=10;

L6  function greet() {
L7      console.log("Hello Pepcoders");
L8  }
L9  var hello = function() {
L10     console.log("You guys rock");
L11 }
```

	Memory	Code
L6	a=und	M C
L7	greet: f {	
	}	
L8	hello=und	

L1 - No memory allocated

L2 -

L3 - No memory allocated until L4. L5: a= undefined

L6: greet: f{ } , L9: hello=undef.

Now code execution phase

L1: a=undefined consoled

L2: greet call so its own EC is formed

Console
undefined

```
L1 console.log(hello);
L2 hello();
L3 var hello = function () {
       console.log("You guys rock")
   }
```

M	Code
L3 hello: ud	

console
(Error in L2 hello is being called as a fun^n but it is a variable)

Now,
```
1. console.log(hello);
2. var hello = function() {
3.     console.log("You guys rock");
4. }
5 hello();
```

M creation	Code exe
hello = ud	hello call (M | Code)
f{	
}	

console:
hello
You guys rock

L5 calls hello
Inside hello we have a console so it is printed

Scopes in JS

```
1 function outer () {
2     inner ();
3     function inner () {
4         console.log(b);
5     }
6 }
7 var b=10;
8 outer();
9 console.log(b);
```

Memory	Code
outer {	outer(): (M | C: inner (M | C))
}	
b=~~ud~~ 10	

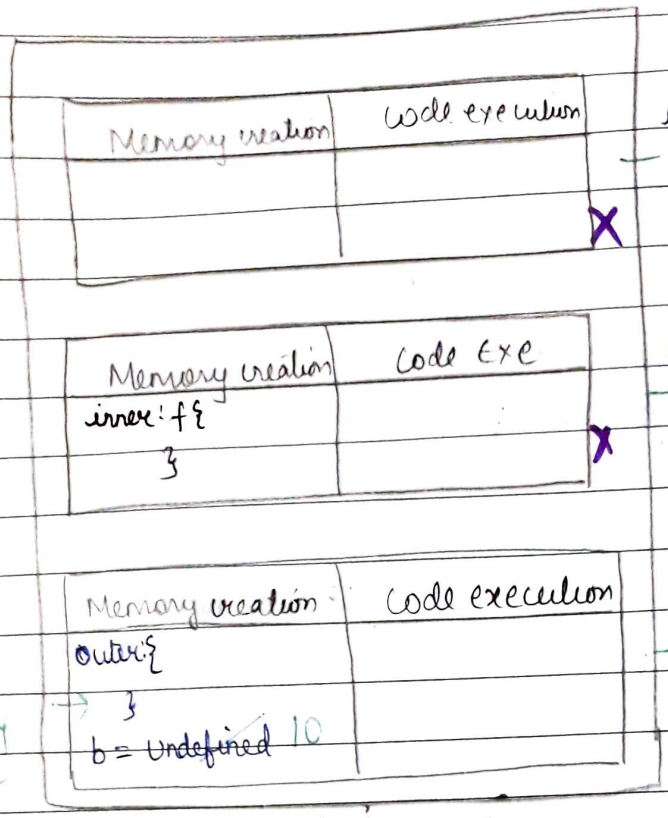
needs to console(b)
firstly we search in local scope of inner where we find a var called b we move its parent & look for
Not found again we move to outer parent and here we find b=10 so console() - gives 10

Explanation: L1 creates outer { } in memory. b = undefined by L7
L8, L9 will not do anything in memory creation phase
Code execution fun^n
L1 to L6 ~~happ~~ nothing happens. L7 makes b=10
when L8 is called - we move to L1, L2 is a call ⇒ no mem
A new code execution L3 ⇒ inner { } L4 & L5 do nothing

Console
10
10

Then L9 again prints 10

Lexical environment chaining



Call stack

inner EC = Has local memory of inner + lexical environment of outer (= Local memory of outer + local memory of GEC)

outer EC — Has Local Memory of outer + lexical environment of GEC

GEC - only has local memory
Global execution environment

x - popped

Memory creation
L1 - makes outer {
}

L7 b = undefined
L8 Rest lines no memory to be allocated

Code execution phase

L7 makes b = 10
L8 - New execution context is made

For outer
Memory creation
L2 : Nothing
L3 : inner : f{
}

Code execution
L2 : is call to inner so a new EC is formed

For inner
Memory creation
L4 has only console so no memory

Code execution
from L3 - Nothing
L5 - needs consolelog(b)

The search of b begins with the local environment of inner. ∄ no b in the memory of b so we move the search to its parent, outer where there is no b. We can do this because the environment context of inner = local memory + lexical environment of outer.

Now we move to the parent of outer and the environment context of outer = local memory of outer + lexical environment of GEC where we will find b = 10

so, L5 prints 10.

L6: pops EC of inner

L7: pops EC of outer

Now we come back to the code execution phase and go to L9 which again wants to print b. As b = 10 is found in the local memory of GEC it is printed

Console

10
10

* Scope - is a variable / function inside the scope / function OR where can this variable / fun^n be accessed in the code.
* Scope chain - chain of lexical environment and the parent references.
* Lexical environment - Local memory of oneself + its parent's lexical environment.

```
* const a = [1,2,3,4];
    a.push(5);          // [1,2,3,4,5]
    a.splice(0,2);      // a=[3,4]
    a=[...a,'*'];       // error ∵ a is const
                        // Otherwise ...a is spreader
                        // & wants a=[1,2,3,4,*]
    a.length=0          // makes length 0
    console.log(a.length);
    console.log(a);
```

Console
```
0
[]
```

```
                 0th level   level 01   level 02   02 ends   01 ends   0th level
* const arr = [ 1, 2, [ 3, [ 4,5 ] ] ] ]
console.log(arr.flat(1));
```

concatenates the first array inside arr.

```
[1, 2, 3, [4,5]]
```

```
console.log(arr.flat(2));
```

concatenates 2 arrays inside arr. Go 2 levels deep & remove arrays

```
[1,2,3,4,5]
```

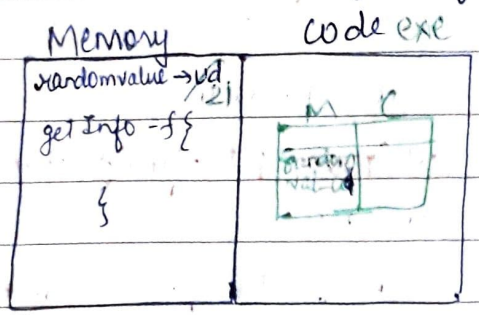
```
arr.flat();
```

Default value is 1
removes array the first array.

* when we do not know the depth of array nesting
  [. . , [. . [[ ]], [[]], [[[[]]]; . . .]
  To flat it out completely arr.flat(Infinity);
* Fact: type of Infinity is Number.
  So, Infinity basically is a value ~ not finite.

const is block-defined
1. ~~const~~ var randomvalue = 21;

```
2.  function getInfo() {
3.      console.log (typeof randomvalue);
4.      var randomvalue = "Arun";
5.  }
6.  getInfo();
```



Code Exe
1. ~~randomvalue = 21~~
fun^n def^n cannot be executed
then getInfo is called so a new execution context
In its memory phase rv = ud
then go on to its code exe.
console.log (typeof rv) = undefined
then rv = Arun but ud is already printed. L5 destroys local execution L7 destroys GEContext

Q:
```
const groceries = ['Banana', 'Apple', 'Peanuts'];
if ( groceries.Indexof(banana)) {
    console.log ('We've to buy bananas');
  }
else {
    console.log('NO need to buy bananas');
  }
```

Indexof is a method. So, groceries.Indexof(banana) = 0
0 is a falsy value.
so if ke andar false hai so if execute nahi hoga
else execute hoge.

console $\boxed{\text{NO need to buy bananas}}$

* Optional chaining (?)

agar age wali property hai toh uski value access ho jaegi warna undefined return kar dunga. Basically error nahi dega agar nahi mili property aur code fatne se bach jaega.

```
const person = {
  firstName: 'Lydia',
  lastName: 'Hallie',
  pet: {
    name: 'Mara',
    breed: 'Dutch Tulip Hound',
    },
  get fullName() {
    return '${this.firstName} ${this.lastName}';
    },
  };
```

```
console.log(person.pet?.name);
```
• pet mein kya name hai?
O/P Mara

```
C.L (person.pet?.family?.name);
```
• pet mein family hai kyā? NAHI
O/P : Undefined code work doesn't stop.

```
C.L (person.getfullName?.());
```
Koi parameter nahi hai to default pass hong
O/P : Lydia Hallie

```
C.L (member.getLastName?.());
```
Yahan member se pehle ? nahi hai toh hum check nahi kar rahe find kar rahe hai.
O/P : Reference error i.e., Jo refer karne ho wo nahi hai

```
function sum(x,y,z) {
    return x+y+z;
    }
```

```
console.log(sum([1,2,3]));                  || 1,2,3 undefined undefined
console.log(sum(...[1,2,3]));               || 6
console.log(sum([...[1,2,3]]));             || 1,2,3,undefined undefined
console.log(sum([1,2,3]));                  || [1,2,3]
```

```
const person = {
    name: "Lydia",
    age: 21,
    },
const changeAge = function(x = {...person}) {
        x.age += 1;
        };
const changeAgeandName = function(x = {...person}) {
        x.age += 1,
        x.name = "Sarah";
        };
```

Doubt 26 [illegible] mean convert 344 ??

Remind call 4 Bind

```
12    changeAge (person);
13    changeAgeandName ();
14    console.log (person);
```

L12 passes parameter person (so we have changeAge : funⁿ (person) {
person points to { name : Lydia
but x.age changes age to 22
person points to name : Lydia age : 22

L13 : passes no arguments so changeAgeandName funⁿ takes its default parameter x = {... person}
so, a new parameter x → { name : Lydia
age : 21 } — before funⁿ executes

the funⁿ changeAgeandName changes x & actually points to x = { name : Sarah
A : 22 }

L14 : prints [Lydia
22]

Q
```
1  function nums (a,b) {
2      if (a>b) console.log ("a is bigger");
3      else console.log ("b is bigger");
4      return
5      a+b;
6  }
```

* console.log (nums(4,2)); // a is bigger undefined (∵ ∄ with return

** console.log (nums(1,2)); // b is bigger undefined

CONSOLE

had line 4 been return a+b

console of * a is bigger
6

console of ** b is bigger
3

* const is block-defined. Example

```
function checkAge (age) {
    if (age <18) {
        const message = "Sorry, you're too young.";
        }
    else {
        const message = "Yay! Oldie.";
        }
    return message;
    }

console.log (checkAge (21));
```

// Gives Reference Error message is const & is defined in if block & else block. Return statement is outside these blocks. What we are trying to refer doesn't exist

Note: Had there been

```
if ( ) {
    var message = " ";
    }
else {
    var message = " ";
    }
return message;
}
```

← this would have worked.

Q

```
const numbers = [1,2,3];
numbers [10] = 11;
console.log (numbers);
```

// ind 3 to ind 9 will be empty

Console: [1,2,3, 7 x empty, 11]

or [1,2,3, <7 empty items>, 11]

Q 1

```
const add = function (x){
    function inner (y) {
        function innermost (z) {
            console.log (x,y,z);
            return x+y+z;
            }
        innermost(6);
        }
    inner(5);
    }
add(4);
```

Memory creation -	Code execution phase -

add = undefined
{
 	
}
	M - | CE -
	z = ~~undef~~ 6 ← when innermost is called

	M - | CE - y = ~~undef~~ 5 innermost { ; ← when inner (5) is called

	M - | CE -
	x = ~~undef~~ 4
	inner {
	}
	
	← when add(4) is called

Consoles | 4 5 6

Q- What happens when we run a js file

Page No.
Date: 03.03.22

Browser ke andar js ka code jab run hoga —

1. Ek GEC banega
2. it's then pushed on to stack
3. JS engine creates an object called window. Window object is creates on a global level & may have fun^n/methods, variables. It may be used by anyone.

This is all work to be done by browser's engine

JS engine creates "this" keyword on the global level. "this" points to the window object. (go to console on inspect page. To access window.property and this.property will give same results as this points to window. This can also be cross-checked using this===window & it'll give true.

Every object has its own 'this' keyword. Example:

```
captainAmerica = {
    name : "A",
    age :  ,
    friends : [ ]
    }
}
```

Here this.name will give A. Here "this" points to captainAmerica object

* window object is created for each tab. On reloading too it is created.

* Global Execution context contains the memory allocated to the fun^n & variables of js code.

Now, window object is created by js engine. window obj has inbuilt methods & the var & fun^n that we make in our code = (GEC)

* on inspect page anonymous is GEC

∴ window ≡ inbuilt + GEC. After running GEC is popped & window is left with what it started with. GEC is kind of a subset of window

Let var & const

* var

Hoisting - In js memory is allocated to variables and functions even without executing a single line of code.

* let

```
console.log(x);
var x=10;
let b=100;
```

console
undefined

∴ memory to x = ud

```
console.log(b);
var x=10;
let b=100;
```

console
Reference error says b has not been initialised

b is stored in script under scope
b is not stored in global scope ∴ for let variable whenever is allocated a memory, this memory is not with window object

They (let & const) can be hoisted but an error is thrown.

let & const are stricter than var

const > let > var — strictness

window.b ⇒ undefined ∵ we are trying find b in windows. b lies in script object.

console.log(b) ⇒ Reference error ∵ console goes through whole code. It knows about script object but has not been initialized i.e. are in TDZ & can't be accessed.

* Temporal Dead Zone (TDZ) is the time between when a variable is allocated memory & when it is initialised. When a variable is in TDZ it cannot be accessed. It is only for let & const.

In the above example b has not been initialized & lies in TDZ. Whenever a variable is in TDZ, accessing it will result in Reference error.

```
let b=100;
var x=10;
console.log(b);
```

* **Block** – A block is used to compound statements

```
{
  // block
}
```

Block scope – When variables are declared in a block, they can be accessed inside that block and are called block scoped.
Variables declared using let & const are block scoped.
var is function scoped.

example:

①
```
{
  var a = 10;        // memory allocated on global
  let b = 20;        // will be allocated memory on block.
  const c = 30;      //  "  "  "  "  "
  console.log(a);
  console.log(b);
  console.log(c);
}
console.log(a);
console.log(b);      // uncaught Reference error b is not defined
```

②
```
var a = 100;         // in global scope
{
  var a = 10;        // block scope
  let b = 20;        //  "
  const c = 30;      // block scope
  console.log(a);    →  10
  console.log(b);    →  20
  console.log(c);    →  30
}                    →  10
console.log(a);      →  Reference Error
console.log(b);
```

console

← a=10 has shadowed a=100

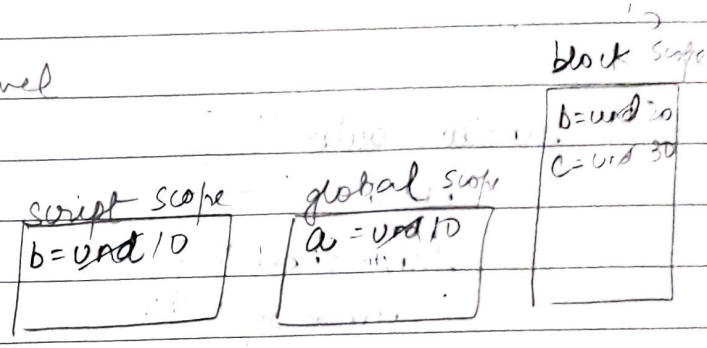
* After the block has been executed. The memory allocated to it i.e, the block scope's destroyed

var is global scoped. a ~~10~~ =100 has been over-shadowed by a=10

Shadowing can be done by var, let & const. There's also a concept of illegal shadowing (to be asked after a few classes by teacher)

eg: ③

```
1  let b=10;
2  {
3      var a=10;
4      let b=20;
5      const c=30;
6      c.l.(a);      ← 10   } block level
7      c.l.(b);      ← 20
8      c.l.(c);      ← 30
9  }
10 c.l.(b);          ← 10 ← script level
```



let & const if made globally they take memory on script (as in L1)

If they are in a block, they take memory on block (as in L4+L5)

Script memory is not destroyed upon completion of execution while block memory is destroyed after complete execution. L9 when executed pops block.

L4 has been able to redeclare let b due to different scopes of it & that of L1's let b. Redeclaration cannot be done in same scope.

* Closures

Function along with its lexical scope forms a closure

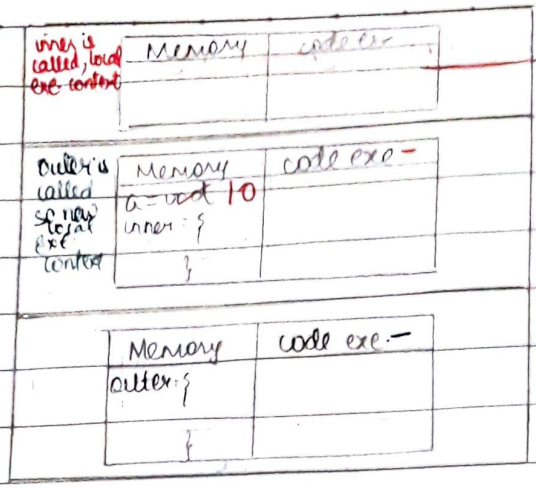
```
function outer() {
    var a=10;          ← lexical scope
    function inner() {
        console.log(a);  ← funⁿ
    }
    inner();
}
outer();
```

= closure

inner funⁿ's lexical has lexical of its parent, outer

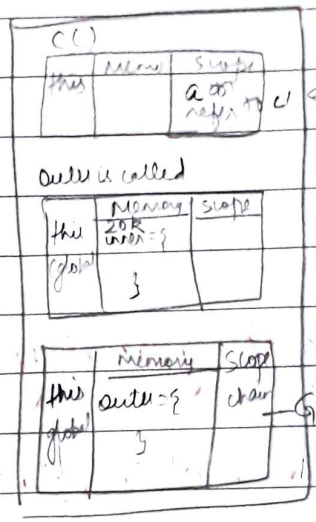
whenever a function is returned it comes with its lexical scope

ie



when console.log(a) is encountered inner looks into its local memory & lexical env of its parent (outer) It finds nothing in the local memory but a=10 is found in the lexical env of outer

```
1  function outer () {
2      var a=7;
3      function inner() {
4          console.log(a);
5          }
6      a=10;
7      return inner;
8      }
9  var c = outer();
10 console.log(c);
11 c();
```



← due to scope chaining

console

function inner() {
console.log(a);
}
10

← when c is called

a = ~~undf~~ 7/10
c = ~~undf~~ 20K.
Heap

call karne pe print hoga 10 because fun exp.

if L6 & L7 are exchanged

function inner() {
console.log(a);
}
7

← L10

L-11

Q
```
const colorConfig = {
    red : true,
    blue : false,
    green : true,
    black : true,
    yellow : false,
    };
const colors = ["pink", "red", "blue"];
```

console.log(colorConfig[colors[1]]);

start reading from left
so in colorconfig we try to colors which does not exist
so type error is thrown

while console.log(colorconfig[color[1]]),

Page No.
Date

* dot notation - used when we know that a property exists in obj
var obj = {
a : 12,
b : 10
}
← like here obj.a & obj.b

* Bracket notation - obj[ ] for indirectly finding
this is evaluated first
then we look for it in obj

* nums = [10, 11, 12, 13];
arrays are like objects internally in js
nums = {
"0" : 10,
"1" : 11,
"2" : 12,
"3" : 13
}

count = 0
for (var i in nums) {
traversion on keys of object
count += 1;
}
← everything except empty string is truthy value so for loop traverses on each key & each time count ↑ by

console.log(count) = 4 ←

Chrome's js engine is V8 which has been written in C++
javascript is synchronous

Page No.: Date: 5.03.21 YOUVA

Node.js

Node.js - It is a JS runtime environment
By ~~Rool~~ Ryan Dahl in 200. It is open source

Node.js is a non-blocking input-output model. It is event driven that is why it is efficient. ∴ works asynchronously
i.e, goes on from one line to another without waiting for responses.
जब input मांगा जाता है और output आता है इस बीच बाकी का code block रहता है मतलब idealy ऐसा रहता है।
लेकिन js में ये block नहीं होता जब तक I/O होता है node.js आगे का code भी run कर देता है।
event driven that is responds to events happening in app. events / System events / custom events (we write them)

* Node Modules - These are inbuilt modules (function libraries) which have pre-written codes. (kind of Pre-loaded)

* Node Package Manager (npm) - it contains the modules which have functionalities other than the ones which are in-built. we can create our own functionalities or install from outside. This is open source. (npmjs.com is where we get the functionalities created by others)
npm registry is a public collection of packages of open source code for node.js

* node.js is used to build servers. ∵ server is used by multiple people at once so all of them send different requests and as node.js is asynchronous so these requests are responded to as and when they are completed. These are not worked upon one by one. मतलब ये नहीं कि पहली request पहले solve करी जाएगी। not first come first serve

* Runtime environment is the environment in which a program or application is executed. It's the hardware and software infra that supports the running of a particular codebase in real time.

To make your command prompt work like node we write node & press enter. Now CMD works like runtime environment

* File System Module - an inbuilt ~~method~~ module
fs - file system module helps to make files, folders, read & modify data etc.
Suppose there's a file called fs.js (can be called anything
name it any variable

① const fs = require ("fs"); // require methods helps to access/call inbuilt modules. If we need to access something which is not inbuilt we will write its path inside ().

let res = fs.appendFileSync ("f1.txt, Hello I am f1 file");
appendFileSync appends data into a file, if that file does not exist, it is created and then data is appended.

Require - to call any module or function in the file we are working on. Require method goes to the file required, executes it (brings its output, if any) but ye Print kyu hua? Bina console.log kiye bhi?

require ("Path of the file") ← to call a file which we created we write its path
require ("fs") ← calling an inbuilt 'fs' module.
* console.log (fs) after ① we print all that exists in fs module.

console.log (fs) ← catches the exported modules from fs module. By default fs module is built with a command to export (module.export =) ie why ⊛ prints complete fs module

* To access a file in the same folder require ("./filename");
* To access a file which is outside our folder require ("../foldername/filename")

* appendFileSync -
1. let res = fs.appendFileSync ("f1.txt", Hello I am f1");
2. fs.appendFileSync ("f1.txt", "\n You guys are smart");
console.log (res);

L1 will create a file f1.txt if it does not exist already & write Hello I am f1
L2: will append You guys are smart.
L3: when executed : Hello I am f1
You guys are smart.

```
let data = fs.readFileSync("f1.txt");
console.log(data);
console.log(data + " ");
```

← gives buffered data

← By automatic type conversion buffer is converted into string

will print those two lines but encoded ∵ usually data is encoded by system. To get it in human readable form (decoded form) conver

```
let data = fs.readFileSyn("f1.txt", "utf-8");
console.log(data);
```

Hello I am S1
You guys are smart

II OS module - helps to give information about the operating system. It is inbuilt like fs. So we 'require' it.

Def^n: OS modules provides functions that you can use to retrieve information from the underlying operating system and interact with it.

```
const os = require("os");

let mySystemArch = os.arch();
console.log(mySystemArch);
```

// · arch returns the underlying architecture

```
let myCPUInfo = os.cpus();
console.log(myCPUInfo);
```

// · cpus returns info about CPU

```
let hostName = os.hostname();
console.log(hostName);
```

// · hostname returns the host name of the os as string.

```
let networkInfo = os.networkInterfaces();
console.log(networkInfo);
```

// returns an object containing network interfaces that have been assigned a network address

Date 08.03.22

* os.release(); // returns the latest release of operating system as a string

* os.totalmem(); // returns the total amount of system memory in bytes as an integer

* os.uptime(); // returns the system uptime in no. of seconds
let uptime in Hours = os.uptime() / 3600; // To get uptime in hours

* os.userInfo(); // returns the info about currently effective user

* os.platform(); // returns as string identifying operating system

* os.type(); // Returns the name of O.S