JS NOTES

PAGE NO.:
DATE: / /

- Everything in JS happens inside an execution context.
- Execution Context

Thread of execution

Memory Management	Code
Key : value a : 10 fn : { }	

Variable Environment

- JS is a synchronous single thread language.

single thread :- JS can execute one command at a time.

Synchronous :- ~~It can go to the~~ control goes to the next line only when the present line execution finished.

PAGE NO.:
DATE: / /

var

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

var square2 = square(n);
var square4 = square(4);
```

Now, when I run this program, execution context will be created.

In first phase which means, memory execution, in that ⇒

n : undefined
square : {...} → (whole function will be copied here)
square2 : undefined
square4 : undefined

(In Memory Execution phase)

PAGE NO. :
DATE : / /

Now phase 2, code execution.

n : 2

Now, control will go to the square 2, square 2 will be invoke, so new execution context will be created.
Again, go through the memory creation phase 1

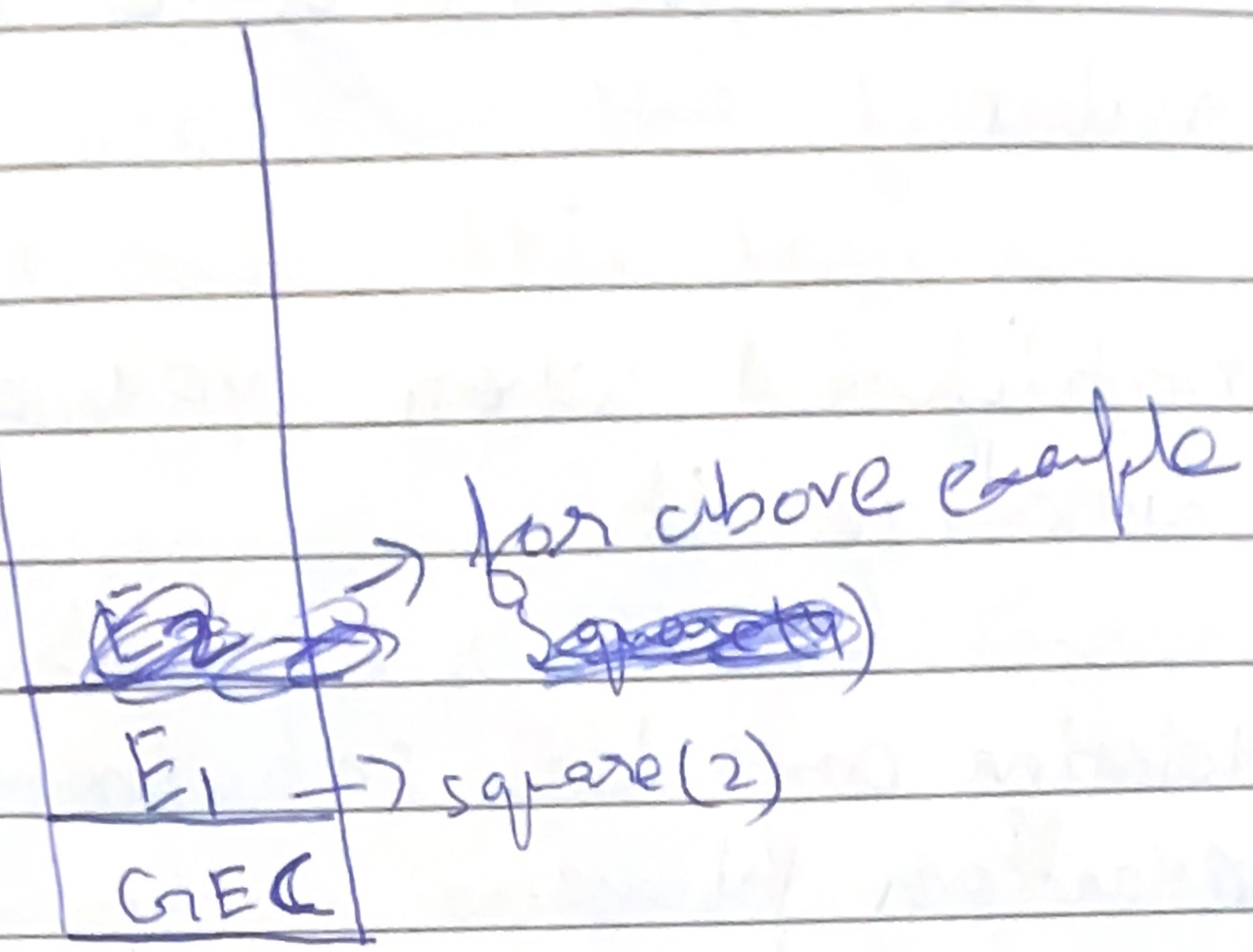
num : undefined
ans : undefined

second phase
num : 2
ans : 4
now we will move to return keyword now it will return the ans value and execution context will be deleted

Square 2 : 4
save for square 4, after that GEC also will be deleted.

PAGE NO.:
DATE: / /

JS handles all thing execution context creation and deletion using call stack.



Hoisting in JS

We can access the functions and variables before initialize that without giving any error.
(some value should come) like undefined (keyword "placeholder")

```
getName();
console.log(x);
var x = 7;
function getName(){
    console.log("Namaste JS");
}
```

Output => Namaste JS.
undefined.

6+1+25%

8

PAGE NO. :
DATE : / /

undefined and not defined.

undefined when you initialize it but didn't gave any value.

notdefined when you didn't initialize it.

→ Hoisting ans lies in memory creation phase.

var inside the function will be stored in the local scope.

GEC memory will be in the global and each function memory will be in the local scope. All are independent of each other

PAGE NO. :
DATE : / /

Shortest program in JS ⇒

empty file.

When we run the file window object and this keyword is created.

window is a global object which is created with the global execution context. and this variable will created.

this === window
(at the global level)

Global space :- Any code which we have written in the file outside the function.

local space :- Anything which create inside the function is called local space.

PAGE NO.:
DATE: / /

When we execute the code, whatever things written in global space gets attached to global window object.

```
var a = 10;

console.log(window.a) => 10
console.log(a) => 10
```

(so, when we are not writing anything before a, like window, it assumes that we are referring in the global obj.

JS is a loosely typed language it means it is not attaches it's variable to any data type.

* Scope means where you can access variables or functions.

PAGE NO.:
DATE: / /

When execution context is created, lexical environment also created.

lexical environment is the local memory along with its lexical environment of parent

```
function a(){
    var b = 10;
    c();
    function c(){

    }
}
a();
```

c() is lexical inside function a()

At the time of memory creation phase, we will get the reference of its parent in memory creation part.

PAGE NO. :
DATE : / /

Lexical parent ⇒

c() function is lexically inside a(), it means c() lexically parent is a().

a()'s lexical parent is global.

* Let and const declarations are hoisted but they are hoisted differently ⇒

```
console.log(a)
let a = 10;
```

~~a is hoisted~~ ~~because~~ ~~when~~ ~~we~~ without running even a single line a create memory in script undefined.

Temporal dead zone :- The phase ~~when~~ from a variable is hoisted till ~~it~~ is some value is given to it.

When the
we can't

Reference
When a

Syntax e
When a
with sa

le
le

(But it

let a;
let a =

c

cor

PAGE NO.:
DATE: / /

When the variable is in TDZ we can't access it.

Reference Error:-
When a variable is not defined.

Syntax error:-
When a variable is defined again with same name :-

let a = 10;	let a = 10;
let a = 20;	var b = 100;

(But it is possible in var)

let a;	But no in case
let a = 10	of const.

const b;	✗ Syntax error
b = 20;	

$\boxed{\text{const b = 20}}$ ✓

PAGE NO. :
DATE : / /

Type error :-

```
const b = 1000;

b = 20;
```

(Because assignment to constant variable) is not possible.

const > let > var

Block Scope and shadowing :-

Block :-

```
if (true) console.log("Hello").
```

This is the syntax
But why we are using block.
because JS doesn't expect multiple statements that's why we are wrapping multiple statements in a block.

PAGE NO. :
DATE : / /

Block Scope:-

```
{
  var a =10;
  let b = 20;
  const c = 30;
}
```

When we run this ⇒

Block
b : undefined
c : undefined

Global
a : undefined.

We can't access let and const outside block scoped.

```
{
  var a =10;
  let b =20;
  const c=30;
  console.log(a,b,c);
}
console.log(a,b,c);
```

Output :-
Inside the block (10, 20, 30)
outside (10, error, error).

PAGE NO.:
DATE: / /

Type error:-

```
const b = 1000;

b = 20;
```

(Because assignment to constant variable) is not possible.

const > let > var

Block Scope and shadowing:-

Block:-

```
if (true) console.log("Hello").
```

This is the syntax

But why we are using block because JS doesn't expect multiple statements that's why we are wrapping multiple statements in a block.

PAGE NO. :
DATE : / /

Block Scope:-

```
{
  var a =10;
  let b = 20;
  const c = 30;
}
```

when we run this ⇒

Block
b: undefined
c: undefined

Global
a: undefined.

we can't access let and const outside block scoped.

```
{
  var a =10;
  let b =20;
  const c=30;
  console.log(a,b,c);
}
console.log(a,b,c);
```

Output :-
Inside the block
(10, 20, 30)
outside
(10, error, error).

PAGE NO.:
DATE: / /

What is shadowing in JS?

```
var a=100;
{
  var a=10;
  let b=20;
  const c=30;
  console.log(a);
  console.log(b);
  console.log(c);
}
```

a will be shadowed inside block so the value will be 10 now.

var inside the block or in global space, it'll take the value in global object.
But let and const in global space will take the space in script but in block scope it take the value in block.

PAGE NO. :
DATE : / /

```
let b = 100;
{
    var a = 10;
    let b = 20;
    const c = 30;
    console.log(a);
    console.log(b); -> 20.
    console.log(c);
}
console.log(b) -> 100.
```

block {	script {
b : 20	b : 100
}	}

(In browser)

Shadowing not in the block, it behaves same with function.

a : 10
y : 83

Illegal shadowing :-

```
let a = 20;
{
    var a = 20;
}
```

[Error: Syntax error : 'a' has already been declared]

But we can use that ⇒

```
var a = 20;
{
    let a = 20;
}
```

Closures →

Function with it's lexical scope bundled together forms a closure.

PAGE NO.:
DATE: / /

Solid example:-

```
function x(){
    var a = 7;
    function y(){
        console.log(a);
    }
    return y;
}

var z = x();
console.log(z). ⇒ 7
z()
```

When function returned ~~it~~ ~~knows~~ ~~its lexical scope~~. they remember where they were actually present.

PAGE NO.:
DATE: / /

Closures questions :-

```
function x() {
    var a = 7;
    function y() {
        consoled.log (a);
    }

    a = 100;
    return y;
}

var z = x();
```

Output will be 100;
because ~~it~~ with the function, it sends the reference of it's parent lexical scope (means reference of variable not value)

— x — x — x — x — x — x —

PAGE NO. :
DATE : / /

```
function x() {
    for(var i = 1 ; i <= 5 ; i++) {
        setTimeout (function() {
            console.log(i);
        } , i * 1000);
    }
    console.log ("Hello");
}
x();
```

According to our expectation, it will not print →

6
6
6
6
6

Because it form a closure and JS will not wait for anything, when timer was expired the value was already i = 6 ; and ~~closure callback or~~ ~~callback~~ settimeout takes the

PAGE NO.:
DATE: / /

Callback attaches the timer and know the reference of i, meanwhile i will be 6. That's why it is printing 6.

Loop is running for 5 times all the 5 function's referring to same i.

We can use let to get rid of this situation. because let is a block scope, each time it will create a new i in block.

we can use with var also but need to bind that ~~for~~ settimeout inside the function.

PAGE NO. :
DATE : / /

```
function x(){
  for(var i=1 ; i<=5 ; i++){
    function demo(){
      setTimeout(function(){
        console.log(x);
      3 x * 1000;
    3
    demo(i);
  3
3
x();
```

It when we are calling function again and again with different i, so it will make new copies of i.

PAGE NO. :
DATE : / /

Closures Interview Questions ⇒

①
```
function outer(){
    var a = 20;
    function inner(){
        console.log(a);
    }
    return inner;
}
outer()();
```

Output ⇒ 20;

②
```
function outer(){
    function inner(){
        console.log(a);
    }
    var a = 20;
    }
    return inner;

Outer()();
```

Output ⇒ 20.

PAGE NO. :
DATE : / /

③
```
function outer(){
    function inner(){
        console.log(a);
    }
    let a = 10;
    return inner;
}
outer()();
```

Output : 10

④
```
function outer(b){
    function inner(){
        console.log(a,b);
    }
    let a = 10
    return inner;

var close = outer("helloworld");
close();
```

Output (10, hello world)