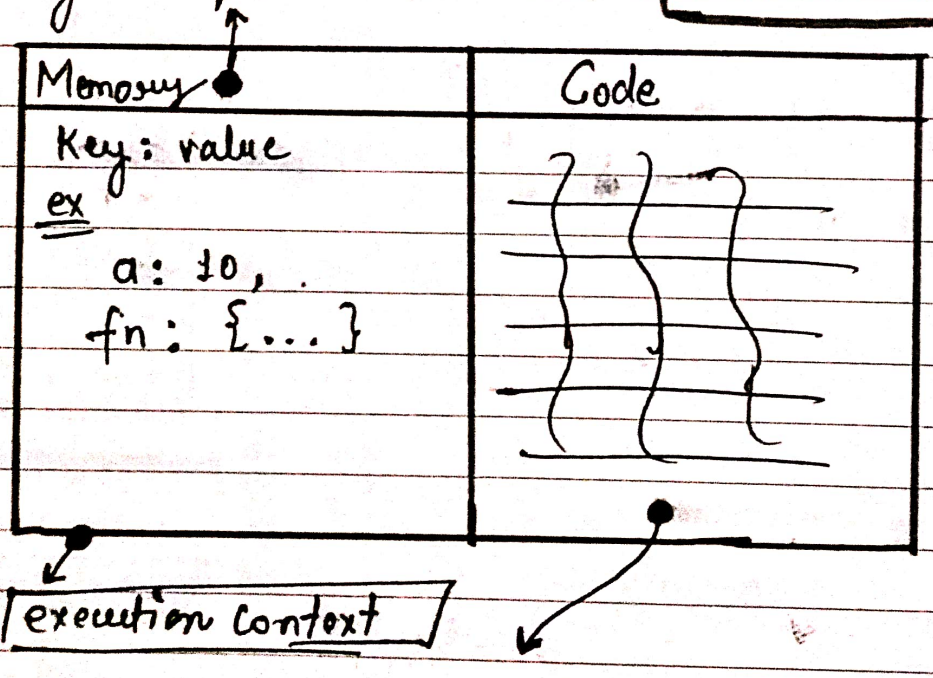
Day 1st

Everything happened inside javascript "Execution Context"

It is kinda big box in javascript and everything happens in this execution context.

Memory Component is used to carry the variables and function in the form of key-pair pattern.
Memory Component also called **Environment Variable.**

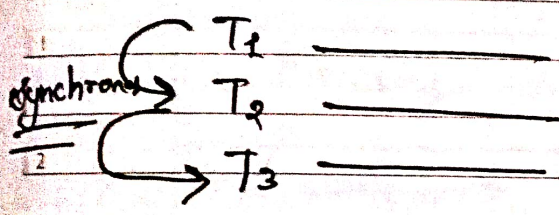


Code Component is responsible for execution of code line by line. And it is also called as **Thread of Execution**

"Javascript is Single-threaded language"

Javascript is synchronous single-threaded language.

In code component the threads are executed one at a time and untill the first execution finished it cannot execute the next one, this called as synchronous.



How Code is Executed?

When code executed is created an execution Context.
It created in two phase —
→ Memory Creation Phase -
→ Code ~~Creation~~ Execution Phase.

1. Memory Creation Phase —
→ interpreter or compiler go through all whole program.
→ Store variables and functions.
→ where variable contains undefined. And functions contains whole function code.

2. Code Execution Phase–

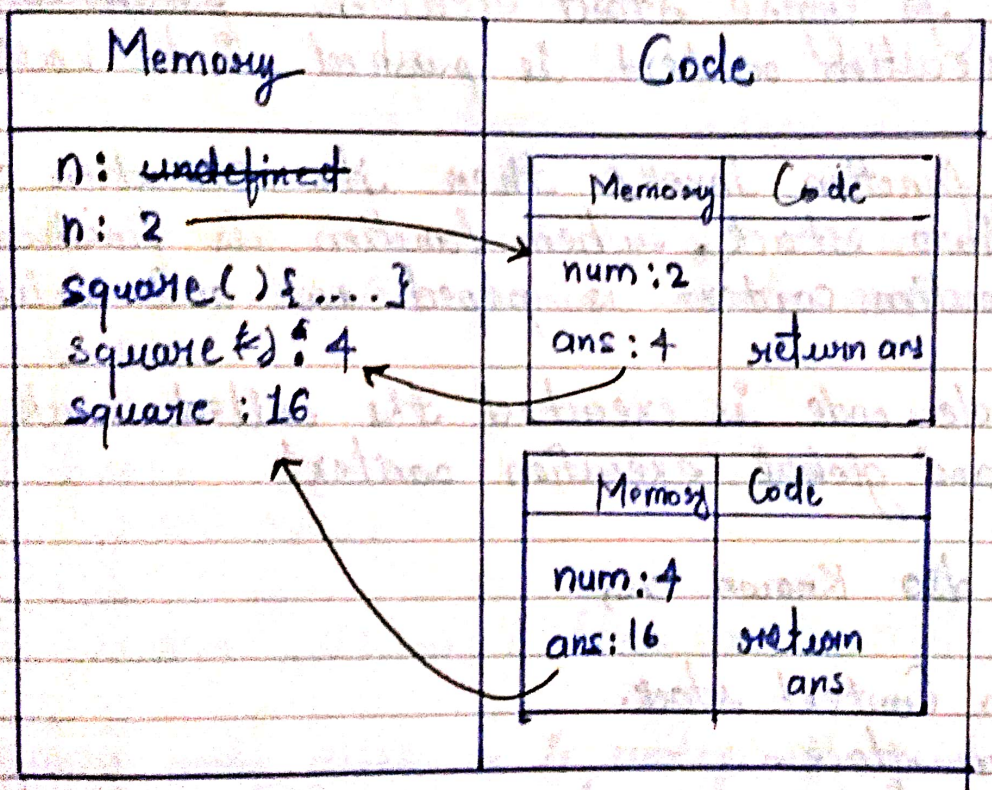
→ Code executed one by one.

Example

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

step • 1. Global Execution Context created.
step • 2. Store variable as undefined and function with code.
step • 3. n got 2 in context
• 4. function is ignored.
• 5. Here square2 is function invocation (calling)
then,
→ n which is 2 passed to function square in num.
→ before passing a new local execution context is created
→ num and ans stored with undefined value.
→ then num got 2
→ ans gets 4
→ return ans · returns to function invocation
→ and square2 gets 4.

6. Square 4 is function invocation.
7. Again a new execution context created.
8. Memory get num and ans with undefined value.
9. 4 is passed to num
10. ans gets 16
11. Returns to function invocation.
12. Square 4 gets 16.
13. And whole global execution context get deleted.



Global Execution Context

Javascript Engine manages all that using a callstack.

[Callstack]

A callstack is manages the execution pattern of execution of code. Or manges order of execution.

Whenever js engine start execution ~~push into~~ global execution context is pushed into callstack.

When a function invoke then its execution context pushed into stack, when function is finished then its execution context is poped out and deleted

After whole code is executed the called stack is empty and global execution context.

Callstack also known as,

a) Execution Context stack.
b) Program stack.
c) Control stack.
d) Runtime stack
e) Machine stack.

2019 WK 02 • 007-358

[Hoisting]

JANUARY MONDAY 07

"Hoisting is a functionality which let you use variable or functions before defining it."

Example

In hoisting, when a code or program runs, it allocate memory to every function and variable in global execution context.

```
Console.log(x);
```

```
var x=7;
```

→ throws undefined.

when above code is executing first x is stored in GEC then execution phase starts and we got x as undefind.

```
Const Console.log(x)
      getName();
      var x=2;
      function getName(){ console.log("Hi"); }
```

The above code gives x is undefined and function is executed. Because x has undefind and function has whole code of it body at phase-1.

```
var x = 10
function getName(){
    console.log("Hi");
}
console.log(x);
getName();
```

Now it give x = 10 and function executed and gives "Hi".

How it works —?

```
a)  console.log(x);
b)  getName();
c)  var x = 10;
d)  function getName(){
e)      console.log("Hi")
f)  }
```

Phase-1 Memory Allocation

Phase-2 Code execution

Phase-1 Allocate memory to variable and function.

b) → getName : f(getName){....}

c) → x : undefined.

Phase-2 Code execution.

a) Console.log x = undefined.

b) getName() = Hi

c) x = 10

d) getName already executed.

Note-

getName = () => { } are treated as variable
const a = getName = () => { } are treated as variable.

Difference - Undefined vs Not defined.

using left page program -
if we don't assign value to x and not ~~int~~ declare it then, memory component x variable does not find it when code is executed then it gave error (reference error : x is not defined)

Refference error: x is not defined.

But it is declared and assigned too, but used before assignment then called Undefined.

How function works in javascript -

This helps to understand scope of variable, lexicality behaviour, etc

2019 FEBRUARY

Mon		4	11	18	25
Tue		5	12	19	26
Wed		6	13	20	27
Thu		7	14	21	28
Fri	1	8	15	22	
Sat	2	9	16	23	
Sun	3	10	17	24	

indication of line number.
indicates flow of data.

```
1  var x = 1;
2  a();
3  b();
4  console.log(x);

5  function a(){
6      var x = 10;
7      console.log(x);
8  }

9  function b(){
10     var x = 100;
11     console.log(x);
12 }
```

Memory	Code
x: undefined	x = 1 (1)
x: 1	
a: {...} (2)	
b: {...}	
	console.log(x) (4)

Memory	Code (2)
x: undefined	x = 10 (6)
x: 10	console.log(x)

Memory	Code (3)
x: undefined	x = 100 (10)
x: 100	console.log(x)

Console
10
100
1

When nothing to do for JS engine, then whole global execution context is vanished and call stack too.

[Scope & Lexical Environment]

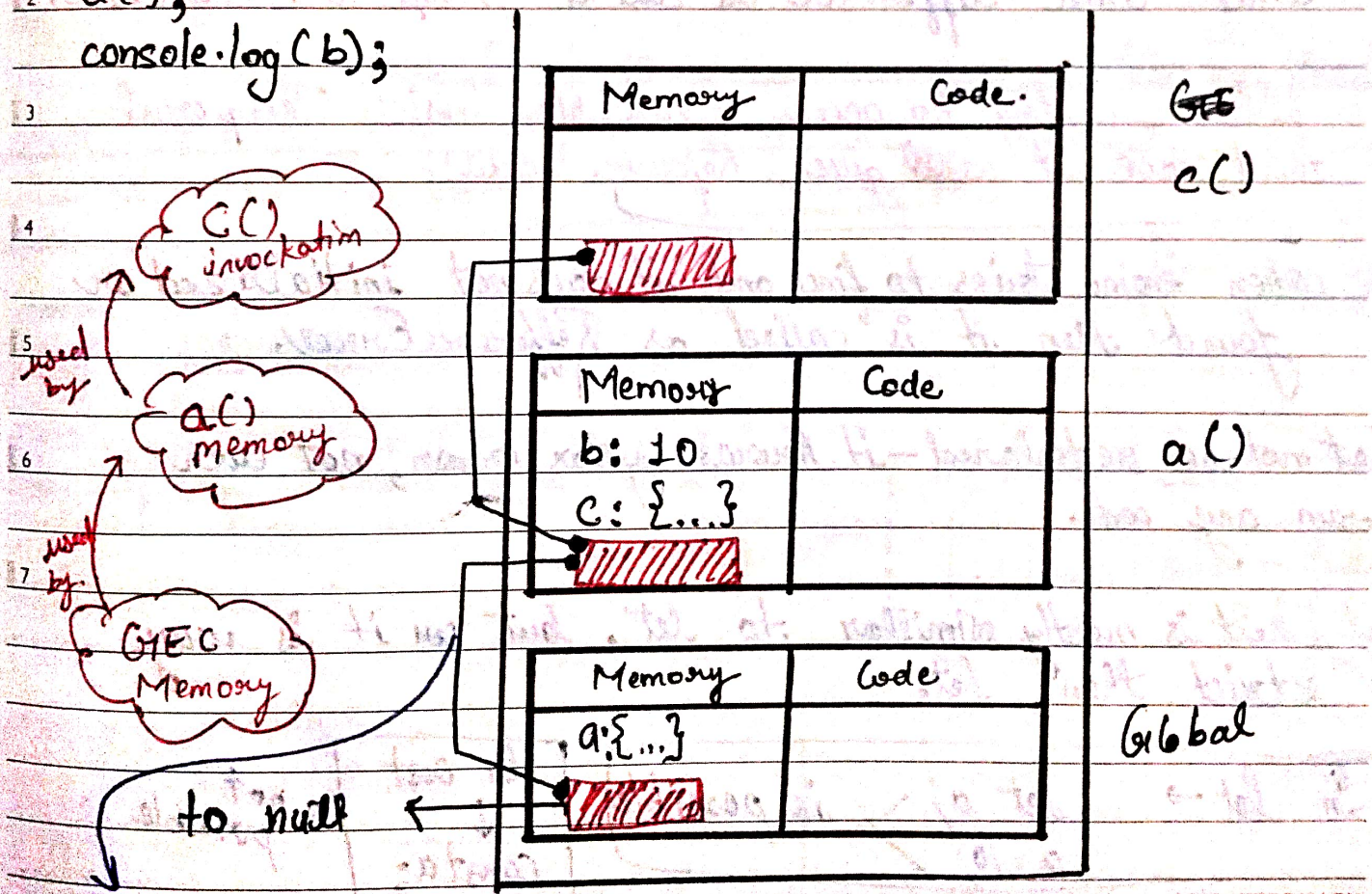
- Word lexical means "in hierarchy".

Lexical environment is local memory along with its parents.

- Lexical environment means where the function is sitting physically in code.

```
function a(){
    var b = 10;
    c();
    function c(){
        // ....
    }
}
a();
console.log(b);
```

- Here in example, function c is lexical in function a. function a is lexical in global execution context.



These way to finding is called as Scope chaining.

Call stack

2019 FEBRUARY

Mon		4	11	18	25
Tue		5	12	19	26
Wed		6	13	20	27
Thu		7	14	21	28
Fri	1	8	15	22	
Sat	2	9	16	23	
Sun	3	10	17	24	

15 JANUARY TUESDAY 2019 WK 03 • 015-350

[let vs Const] as per Hoisting

let

Variables declared with let and const, are not stored different memory space in that is script.

For let and Const =>

when code memory allocation is finished then, they are assigned with undifined in different memory space.

Until they initiallized with their actual value that time difference is called Temporal Dead Zone.

When you try to access a variable inside Temporal dead zone it will gives ReferenceError:-

when engine tries to find and it does not initiallized or found then it is called as RefferenceError.

→ Let not be redeclared – it throws syntax error, not even run any code.

Const → It is mostly similar to let, but it is way strict than let.

In let → let a; a=10; → is possible

In cost. → Const a; a=10; → not possible

If throws SyntaxError: Missing Initializer in const declaration.

→ Const expecting declaration + initiallization in oneline.

Const can't be redeclared.

[Blocking scoping & Shadowing]

Block

Block is also known as compound statement. Used to combined multiple code in one group. These g block used where is expect single statement

means, where we want to write multiple statements instead of single statement.

ex `if (true) true;` → no error → single statement

```
if (true) {
    multiple statements
}
```
Block.

Block scope

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

17 JANUARY THURSDAY

2016

WK 03 • 017

When you run the snippet,

▼ Block
  b: undefined — [let]
  a: undefined — [const]
output
▼ Global
  a: undefined — [var]

Means, let and const are block scoped. And not accessible after execution known as block scoping

But a is assesible because it is in global scoped.

[closure]

A closure is a function is bind together with its lexical environment OR

Function along with its lexical scope forms a closure.

Ex

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

→ Here function x is closure because it contains function y and it is lexical scope to function x.

→ when a function is returned function x() is vanished but function y() is still maintain its lexical scope.

Closure functions are functions once. Means they are executed once

2019 FEBRUARY

Mon		4	11	18	25
Tue		5	12	19	26
Wed		6	13	20	27
Thu		7	14	21	28
Fri	1	8	15	22	
Sat	2	9	16	23	
Sun	3	10	17	24	

Lecture - 15

16

AUGUST
FRIDAY

Lexical Scoping

See 14 jan Monday page

2019

WK 33 • 228-137

But first see this

Execution happen in two phase.

- Memory creation
- Execution.

Auto Global variable → if a variable does not have any data type or scopes then, compiler assign it globally.

```
var naam = "abc";
function func1(){
    clg(naam);
    hello = "def";
    clg(hello);
}
func1();
clg(naam);
```

→ hello is globally scoped.

But if you use strict mode then it won't helped.

```
"use strict"
hello = "def"
```

Now this line will give error

Lexical Scoping lexical scope refers to how variable scope is determined by the physical location of the code when its written.

```
function greet(){
    let name = "Alice";
    function sayHello(){
        clg("Hello" + name);
    }
    sayHello();
}
greet();
```

More explanation

2019 AUGUST SATURDAY 17

VK 33 • 229-136

Here, sayHello() can access name variable becoz it is lexically (physically) inside the greet()

So lexical scope is all about where functions and variables are declared in code-structure.

Conclusion → so name is accessed due to its lexical scope in sayHello(). Although name is sitting physically in greet() but it ~~acces~~ is accessed because it is lexical scoped in sayHello().

Closures.

A function that return function another with its lexical scope.

```
function makeFunc() {
    const name = "Mozilla";
    function displayName() {
        log(name);
    }
    return displayName;
}
```

this surrounding is lexical environment to function displayName because it uses name variable.

When this returns then it also carry the lexical env with function.

And that is closures.

when displayName execute then it access name and name is in its lexical env, this way both function displayname and lexical env used so that is called closure.