Ep 1. How JS works & Execution Context.

* "Everything in JS happens inside a Execution Context."

Execution Context is a like a big box which has two components.
1. Memory Component (variable Environment)
2. Code Component (Thread of Execution)

Memory	Code
key: value	0
b = 20	
function: ? {}	

→ Code component is the place where code is executed one line at a time.
→ In memory component, variables & functions are stored as key: value
* "Javascript is a synchronous single-threaded language".
→ JS can execute one command at a time and in specific order.

Ep2: How JS code is executed?

Q What happens when we run JS code?

⇒ An execution context is created.

Execution context is created in two phases.
1) Memory Creation Phase
2) Code execution Phase.

- In Memory creation Phase, memory is allocated to variables & functions.

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

Memory	Code
n: undefined → 2	memory: num: ~~undefined~~ → 2, ans: ~~undefined~~ → 4 / code: ~~num = 2~~, ~~ans = 2*2 = 4~~, ~~return 4~~ return.
square: {...}	
square2: ~~undefined~~ → 4	memory: num: 4, ans: 16 / code: return ans
square4: ~~undefined~~ → 16	

2nd phase :- Code execution

Whenever a function is invoked, a brand new execution content is created.

* "call stack maintains the order of execution of execution contexts."

Ep : 3

✔Hoisting in JS : In most of the programming language if we try to access the variable and function even before declaring it , it gives an error but in JavaScript, we can access variables and functions even before their initialization.

✔The reason behind hoisting is the execution context.

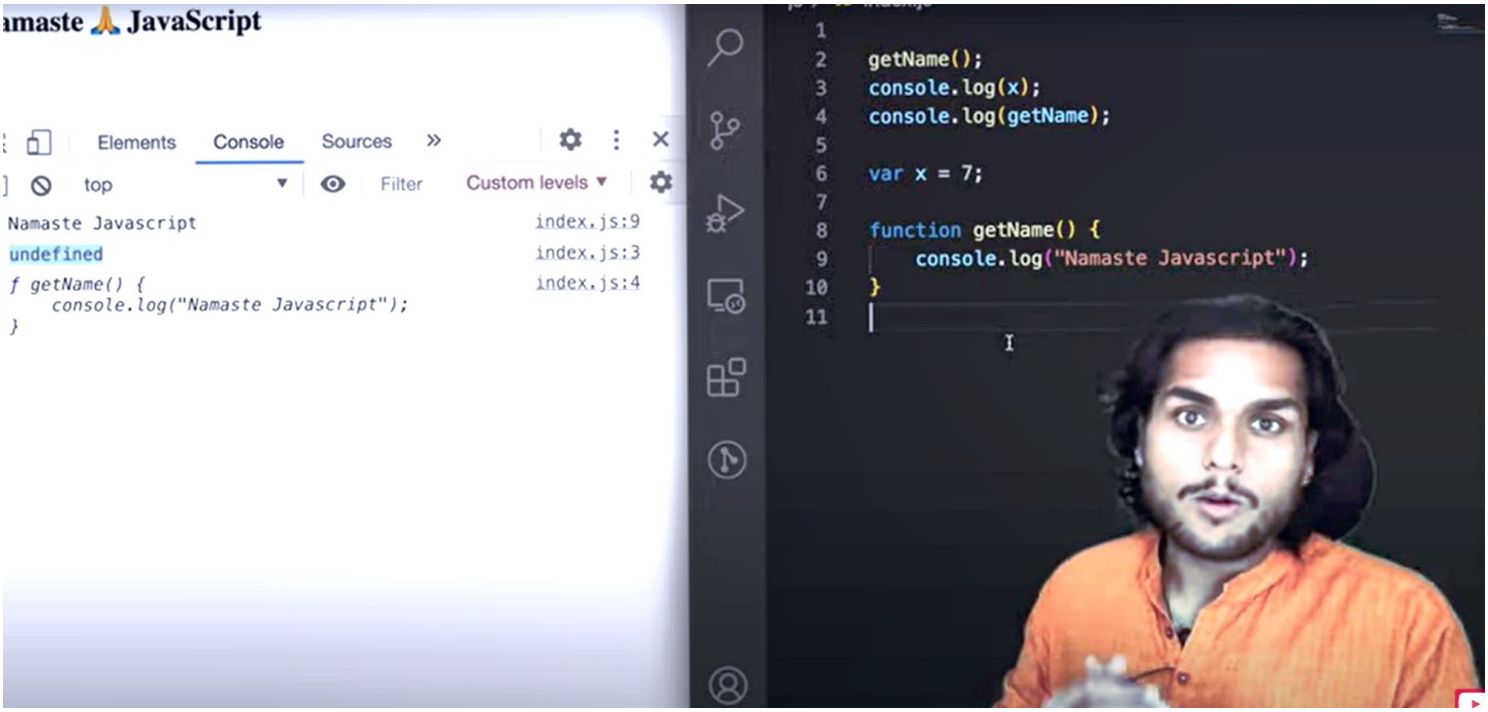
✔ undefined: memory has been allocated but not initialised a value

✔not defined : memory has not been allocated.

✔Working of functions:

When a function is invoked , a new execution context is created and that execution context is pushed into the call stack . When the whole code inside the function is executed, the execution context is popped off the call stack.

Example of Hoisting in JS :

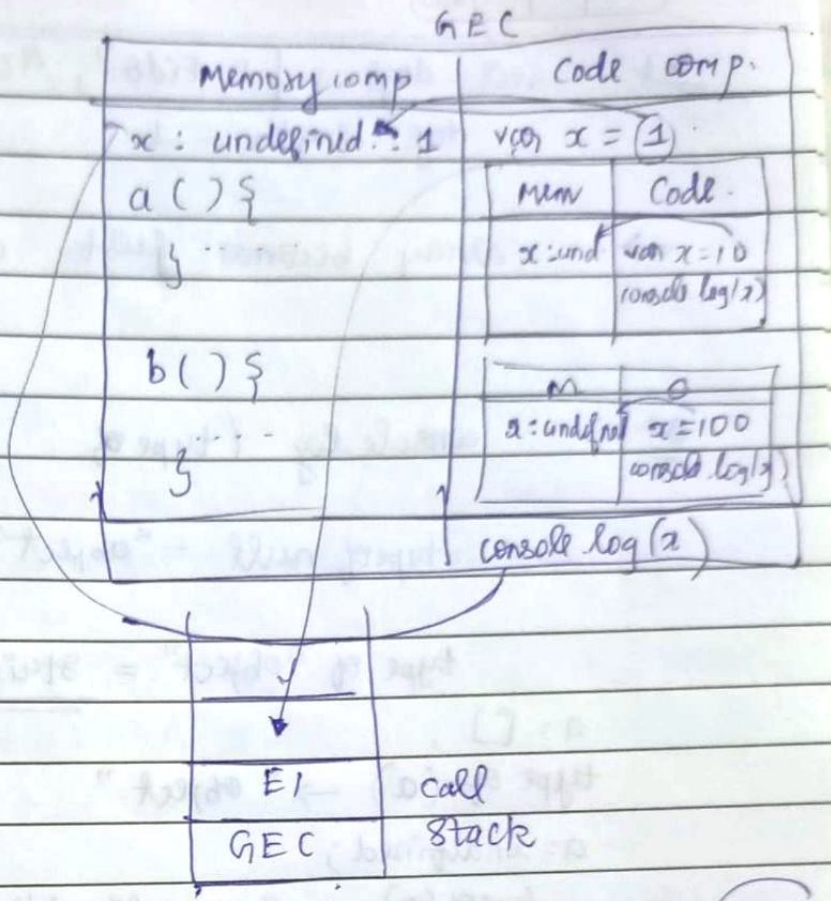


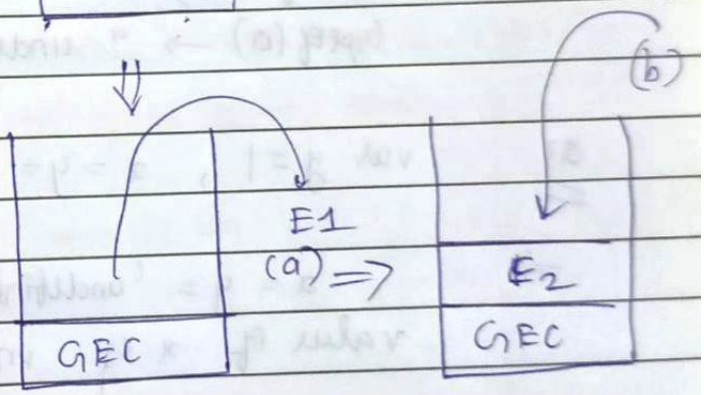
Ep: 4

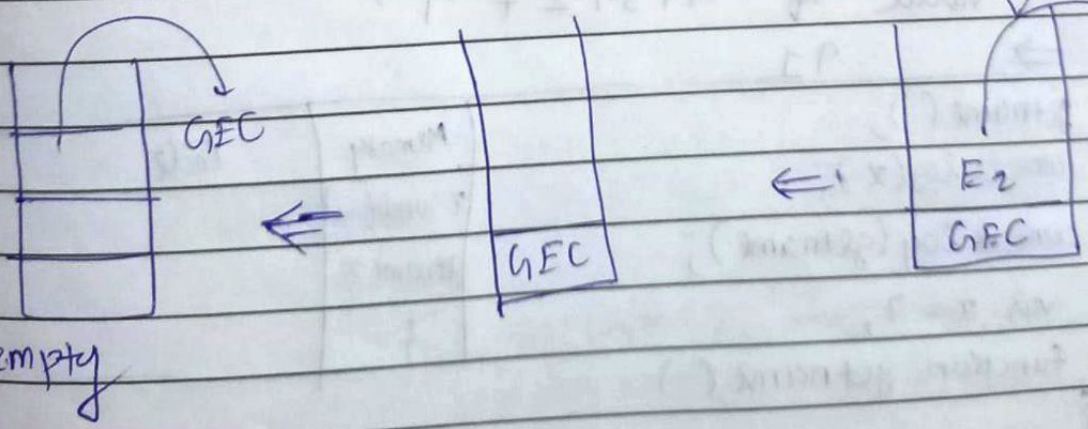
e.g of functions in JS

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

o/p
10
100
1







Ep : 5,6

Day 5 : # NamasteJS

🐡 Whenever a JS program is executed, a global object is created along with global execution context and "this" variable by JS engine.

🐡 At global level, "this" points to global object i.e window in case of browser.

🐡 window consists of lot of variables and functions which can be accessed anywhere in the program.

🐡 Shortest JS code : No code

🐡 Even when the js file is empty, JS engine will create a global object.

🐡 Anything which is not inside the function is the global space.

🐡 Whenever we create any variable or function in global space, it gets attached to the global object.

🐡 undefined is a placeholder which is kept inside a variable and it states that in the whole code that variable was not assigned anything.

🐡 not defined is an error that occurs when a variable hasn't assigned any memory in the memory allocation phase during creation of execution context.

🐡 JS is loosely (weakly) typed language. It does not attach variables to any specific data type.

🐡 Never assign undefined to a variable.

Ep:- 7 The Scope Chain, Scope & lexical Environment

→ Scope means where we can access a specific variable or function in our code.

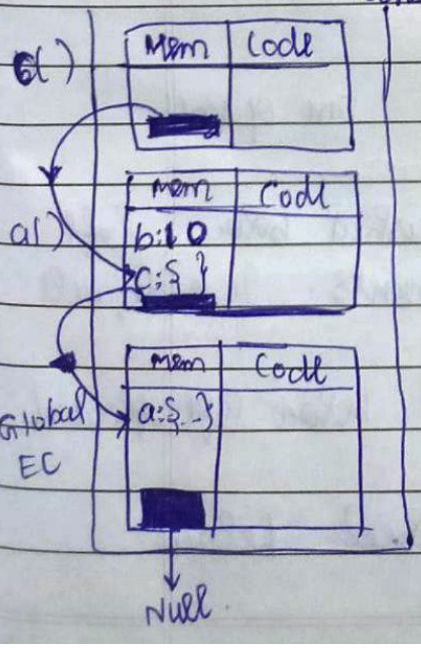
1) What is the scope of a variable?
2) Is 'b' inside the scope of a function?

→ Scope is directly dependent on lexical environment.

→ Whenever an execution context is created, a lexical environment is created. lexical environment is the local memory along with lexical environment of its parent.

(in hierarchy, sequence / in order.)

e.g
```
function a(){
    var b = 10;
    c();
    function c(){                // c is lexically inside a
        console.log(b);
    }
}
a();
console.log(b);                  // Error.
```



→ Scope chain: chain of lexical environment is known as scope chain. It defines whether a variable or function is present inside the scope or not.

Ep: 8 let & const in JS, Temporal Dead Zone.

let & const declarations are hoisted.

➡ console.log(a)
let a = 10;

* Temporal Dead Zone.

→ It is time since when let variable was hoisted till it is initialized some value.

→ Whenever we try to access a variable inside a temporal dead zone it gives an reference error.

→ const should be declared & initialize in the same line.

```
const b = 1000;
b = 10000;
```

Type Error :- Assignment to const variable.

```
const b;
b=10;
```

Syntax Error :- Missing initializer in const declaration

```
let a = 100;
let a = 10;
```

Syntax Error

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

Reference Error : cannot access a before initialization.

* use const wherever possible, if not try to use let keep var aside, use it consciously.

Ep :8

BLOCK SCOPE & SHADOWING IN JS

🐟Block is also known as compound statement. It is used to combine multiple JS statement into one group.

🐟We group multiple statements together in a block so that we can use it where JS expects one statement.

🐟 Block scope means what all variables & functions we can access inside a block.

🐟When we declare a variable using let and const keyword, a memory is allocated for those variables in the script scope and for var , memory is allocated in the global scope.

🐟e.g let d = 10; // present in the script scope.

```
{
let a = 2; //block scope
const b = 5;. // block scope
var c = 9; // global scope
}
```

here a and b are in the block scope and c in the global scope.

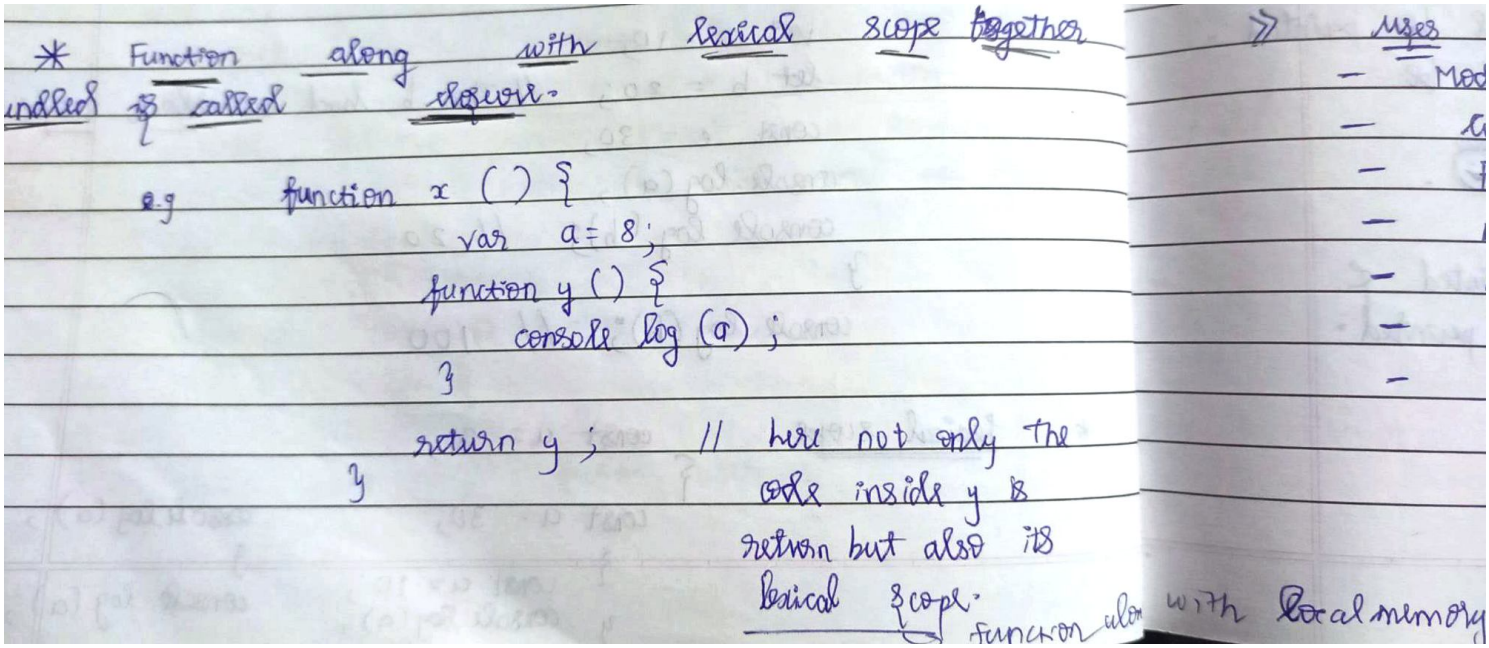
we cannot access a and b variable outside the block because it is present in the different scope.

Shadowing : when a variable is declared in a certain scope having the same name defined in the outer scope & when we call the variable from the inner scope , the value will be changed as it shadows outer variable.

e.g var a = 10;
{
var a = 20; // this variable shadows upper variable.
console.log (a) ; // prints 20
}
console.log(a) ; // also prints 20

Illegal shadowing: while shadowing a variable, it should not cross the boundary of scope. i.e we can shadow
" var " variable by " let " variable but cannot do opposite.

Ep:10 Closures in JS



* Function along with lexical scope together called is closure.

e.g
```
function x () {
    var a = 8;
    function y () {
        console.log (a);
    }
    return y; // here not only the code inside y & return but also its lexical scope.
}
```

» uses
- Mod
-
-
-
-
-
-
-

function also with local memory

* Functions can be assigned to a variable.
e.g var a = function y(){
console.log(a);
};

var z = x();
console.log(z);
z();

o/p:- y() {
console.log(a)
}

* Function can be passed as an argument into a function.

we can return a function inside a function.
e.g return y;

a function along with the reference to those variables not just the value is returned.

» Function bundled with lexical environment is known as a closure. Whenever function is returned, even if its vanished in execution context but it still remembers the reference it was pointing to.

» Closure is an ability of a function to remember the variables & functions that are declared in its outer scope.

» Uses of closures :-
- Module Design Pattern
- Currying
- Functions like once
- memoize
- maintaining state in asyn world
- setTimeouts
- Iterators and many more ...

Ep 11. SetTimeOut + closures Interview Question

Suppose we have a function.

```
function x() {
    var i = 1;
    setTimeOut (function () {
        console.log (i);
    }, 1000);
}
x();
```

The above code will print the value of i after 1 second.

Lets do some changes

```
function x() {
    var i = 1;
    setTimeOut (function() {
        console.log (i);
    }, 1000);
    console.log ("Namaste JS");
}
x();
```

We expect that the value of i will be printed after 1s & then Namaste JS will be printed But ....

JS, time & tide waits for none  .

First Namaste Javascript will be printed & then after 1 sec value of i will be printed.

Reason:- The function setTimeOut forms a closure. It remembers the reference to i. So wherever the function goes it takes the reference of i along with it.

The setTimeout takes the callback function and stores it somewhere & executes after the time expires.

8. Write a program to print numbers 1 to 5.

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

O/p:
```
Namaste Javascript
6    // after 1 sec
6    // after 2 sec
6    // after 3 sec
6    // after 4 sec
6    // after 5 sec.
```

This happens because of closure. Even when a function is taken out in other scope still it remembers its lexical environment.

When the setTimeout takes this function & stores somewhere & attaches the times so that function remembers a reference to i. (not the value of i)

So when the loop runs for the 1st time it takes the function attaches a timer & also remember the reference of i & similarly all 5 copies of this function are pointing to same reference of i. because the environment for all this f^ are same.

JS does not wait for anyone it just runs the loop again & again does not wait for times to expire, the value of i value became 6 & when the callback runs it prints 6.

⇒ This problem can be fixed with using 'let' instead of var. as a new copy of i is created for every iteration.

⇒ It works with let because let is block scoped & var isn't.

Q How to implement the above functionality without using let?

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

```
    console.log("Namaste Javascript");
}
x();
```

→ For every iteration function close has a new parameter.

o/p.
1 // after 1 second
2 // after 2 second
3 // after 3 second
4 // after 4 second
5 // after 5 second

Ep 12 Closures Interview Question

Q1 Define closure :- A combination of function and its lexical scope bundled together forms a closure.

Each & every function in JS has access to its outer lexical environment which means access to variables & functions which is present in the environment of its parent.
Even when this function is executed in other scope it still remembers its outer lexical scope in which where it was originally present. That is what closure is.

Q2 Example to demonstrate closure :-

```
function outer(){
    var i=10;
    function inner(){          //This inner f^ has access to i variable.
        console.log(i);
    }
    return inner;
}
```

```
outer()();
```

Q3. Use of double parenthesis ()() in JS?

```
outer()()
```

outer() returns the inner function; outer()() calling the inner function

Can also be written as -

```
var z = outer(); // returns the y function.
z(); // calls the y function.
```

Q4 What if we move the line `var i = 10;` below the inner function? Still it forms a closure or not?

```
function outer(){
    function inner(){
        console.log(i);
    }
    var i=10;
    return inner;
}
var z = outer();
z();
```

NO, it will still forms a closure. This inner function forms a closure with outside environment not in passicular sequence where it is present in code.

Q5. Are let declaration closed over?

→ No, it still forms a closure.

Q6 Are function parameter closed over?

```
function outer(b) {
    function inner() {
        console.log(a,b); // 10 hello world
    }
    let a = 10;
    return inner;
}
var z = outer("hello world");
z();
```

Here b is also a part of outer environment of inner function. inner function() forms a closure with b also.

Q7. What if the outer function() is nested inside other function?

⇒

```
function outest() {
    var c = 20;
    function outer(b) {
        function inner() {
            console.log(a,b,c);
        }
        let a = 10;
        return inner;
    }
    return outer;
}
var z = outest()("hello world");
z();
```

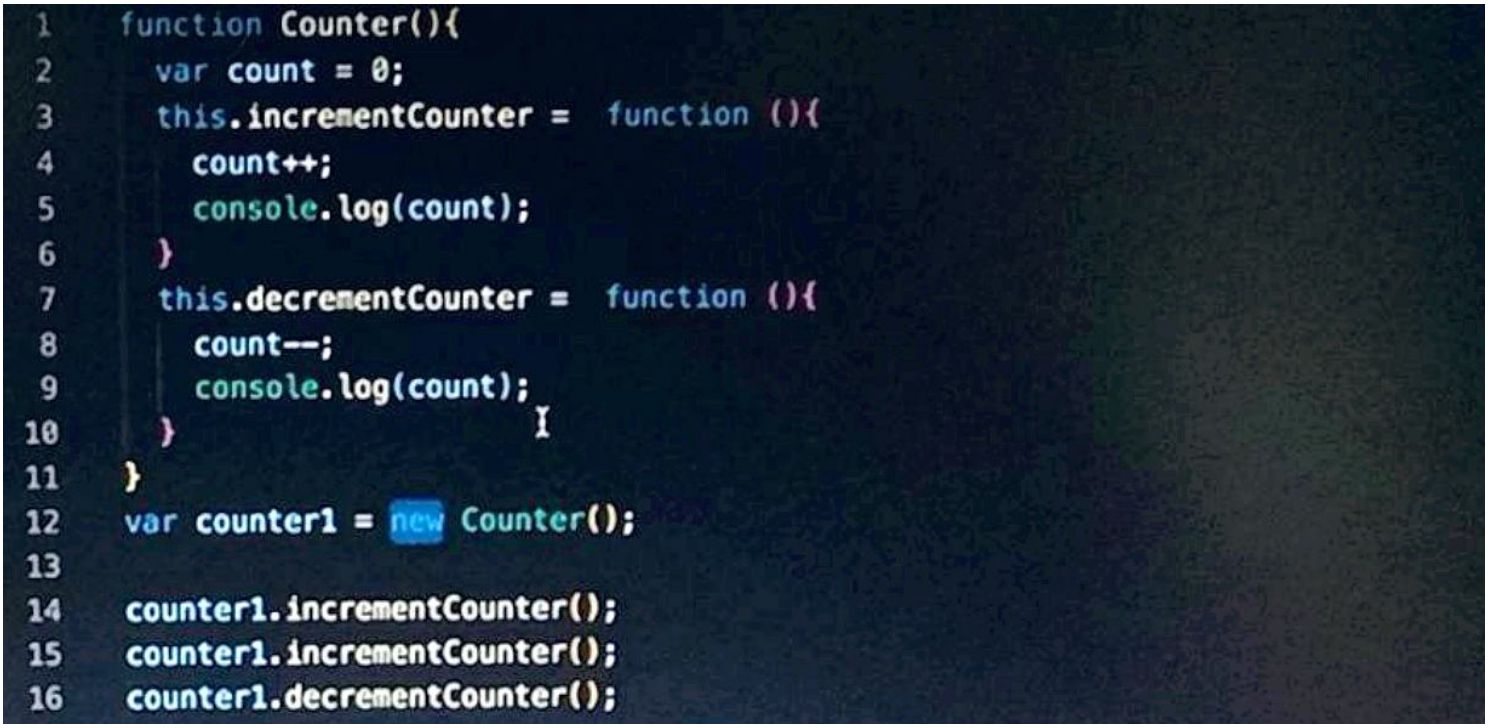
inner function stills forms a closure with c also

🐟 conflicting name global variable -
inner function first tries to find the variable in the parent function first if it finds prints the value but if not it tires to find the variable in the parent parent's function or the global scope.If that variable is not present inside the global scope it prints the reference error .

🐟 Advantages of closures
👉 Module pattern
👉 function currying
👉higher order functions
👉 data hiding and encapsulation

🐟 data hiding : We want to have data privacy over variable so that other functions or part of program cannot acces it .

🐟constructor function in JS
It is used to create objects .
Example in below image .



```
function Counter(){
  var count = 0;
  this.incrementCounter =  function (){
    count++;
    console.log(count);
  }
  this.decrementCounter =  function (){
    count--;
    console.log(count);
  }
}
var counter1 = new Counter();

counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();
```

Disadvantages of closure :

👉over consumption of memory

👉 the variables declared inside the closure are not garbage collected .

Garbage collector : It is like a program in browser or JS engine which freeze up the unutilised memory.
when the variables are no longer needed, Garbage collector takes those variables out of memory.

Relation between garbage collector and closures : the variable which is not used in the function gets smartly collected by the JS engine and it will be no longer be in the memory.
e.g function a( )
{
var x = 0, z = 10;
return function b ( ) {
console.log (x) ;
}
}
var y = a( );
y( );

Here z is not been used . So when b ( ) function is returned z variable is garbage collected smartly and x is not .

Ep:13

First class functions

Function Statement:

e.g function a( ) {
console.log ("Namaste JS");
}
a( );

The above way of creating a function is called Function Statement. It is also known as Function Declaration.

Function Expression :

e.g : var b = function ( ) {
console.log(" Namaste Duniya ");
}
We can assign a function to a variable also. Here function acts like a value.
This is known as Function Expression.

Difference between Function Statement and Function Expression:

If we call the function statement even before it's declaration it prints the whole code inside it . But if we call the function expression even before it's declaration it gives reference error .

Difference between Function Statement and Function Expression:

If we call the function statement even before it's declaration it prints the whole code inside it . But if we call the function expression even before it's declaration it gives reference error .

Anonymous Function: A function without any name is called anonymous function. It does not have its own identity. It gives syntax error if we run .
They are used in place where function are used as values.

Named function expression:

```
e.g var b = function xyz ( ) {
console.log (" Namaste Duniya");
}
b( );
xyz ( ) ; // we cannot call this way because xyz function is present in the local scope .
```

Difference between parameters and arguments:

```
function a( parameter1)
{
console.log(parameter1);
}
a(10); // here 10 is argument.
```

First Class Functions: The ability to use function as values is called First Class Functions.
We can even pass function inside another function as argument as well as return function from another function.
e.g var b = function (param1) {
console.log(param1);
}
function xyz(){

}
b(xyz); // passing another function inside function

Functions are First Class Citizens is same as functions are First Class Functions .

Ep:14 Callback functions

Callback function :
A callback is a function which is passed as an argument to another function.
Advantages:
It gives access to whole asynchronous world in a synchronous single threaded language.
e.g

```
setTimeout ( function ( ) {
console.log ( "hello" );
},3000);
```

here the setTimeout function takes the callback function and attaches a timer of 3 seconds.
Blocking the main thread : If a function takes long time to execute till then it does not allow any other function to execute . So the call stack will be blocked.
Event Listeners:

```
document.getElementById( "clickme").addEventListener("click", function ( ) {

});
```

here clickme is name of I'd and click is the event listener which calls the callback function so it will come in the call stack when button is clicked.

Closure along with Event Listeners:
example in the image below .
callback function forma a closure with outer environment i.e count varaible

Garbage collection and remove Event Listeners:
Event listener are heavy i.e it takes memory. If there are more Event Listeners in our program, our page can go slow.

Event Loop , Callback Queue , Microtask Queue

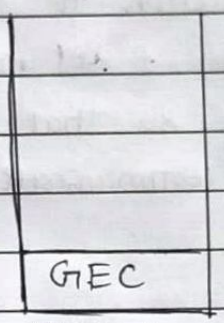
Ep. 15 Asynchronous JS & Event Loop from Scratch

* How JS engine executes the code using Call Stack.

e.g

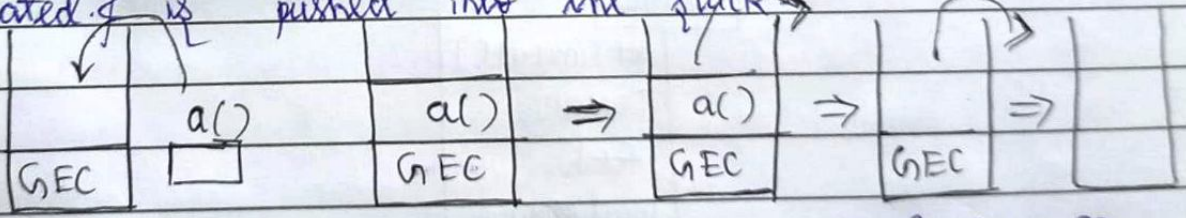
```
function a() {
    console.log("a");
}
a();
console.log("End");
```

We know than whenever JS code is run, a global execution context is created & is pushed into the call stack.



Call stack.

In GEC, the function a() will be allocated a memory in the memory component. When the function a() is called, a new execution context is created & is pushed into the stack



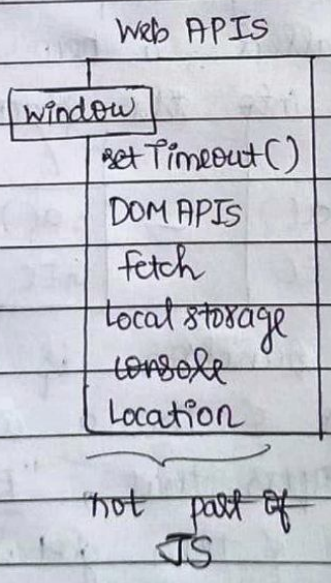
The code inside the function is executed & prints "a" on the console & the a will be popped out of the call stack. After that, "End" will be printed on the console & then global execution context will also be popped out of the call stack.

* If anything comes inside the call stack it quickly executes it. It does not wait for anyone.

* What if we want certain program to run after 't' time? We need something extra superpower because call stack doesn't have a timer.

* Call stack is present in the JS engine & JS engine is present inside the browser. Browser has —
1) Local storage 2) Timer 3) URL section (https://—)
4) can make connection to the server
5) displays the UI & lot more.
We need a way so that our JS code can make connection / communicate with the browser functionalities.

* WEP APIs in JS: To access all the superpower present in browser we have WebAPIs.



Browsers give JS engine facility to use all the above superpower through a keyword "window".
eg if we write window.setTimeOut() in our JS code, it gives access to timer.

* How setTimeout Works ?

e.g
```
console.log ("start");
setTimeout (function cb() { ...
    console.log ("callback");
}, 5000);
console.log ("End");
```

Here first GEC will be pushed into the call stack & then it executes the code line by line. First "start" will be printed on the console using console WEB API, The nextline setTimeout will called the setTimeout API & gives access to the timer features of browser. The setTimeout functions registers the callback function cb() & attaches a timer of 5 sec. Now the JS code moves on to nextline, it does not wait for anyone & prints "End" on the console. The GEC will be popped out of the call stack.
After 5 seconds, the cb() function needs to be executed When timer expires, cb() function goes into the CallBack queue & EventLoop checks if there is something in CallBack queue & if yes, eventLoop pushes it into Call stack. So the "callback" will be printed on the console.

* How EventListeners work in JS ?

e.g
```
console.log ("start")
document.getElementById ("btn")
.addEventListener ("click", function cb() {
    console.log ("callback");
});
console.log ("End");
```

→ Here first the GEC will be created & push into the call stack & code is executed line by line. So first it calls the console WEB API method & log "start" to the console.
Now the next line document.getElementById("btn").addEventListener will execute.

→ addEventListener — superpower given by browser to JS engine through window object in form of WebAPI which is DOM API.

→ document.getElementById — means it calling the DOM API which fetches something from DOM (Document Object Model)
Here the id is "btn".

→ .addEventListener - registers a callback cb() & an event "click" is attached to it in the WebAPI environment.
Next the "end" will be printed on the console.
Then GEC pops out of the call stack.

→ The registered callback function still resides in the WebAPI environment.

→ When user click on the button, callback method is pushed inside the callback queue

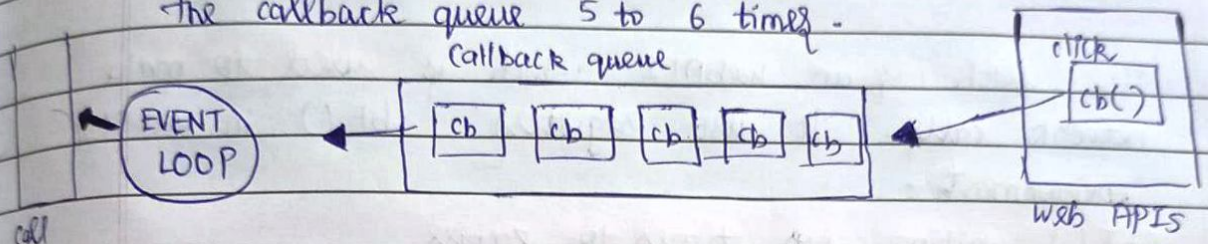
→ The job of Event loop is to continuously monitor the call stack & callback queue.

→ Now Event Loop pushes the callback method into the call stack & call stack quickly executes it & "callback" will be printed on the console.

* Why do we need call Event Loop?

* Why do we need CallBack Queue?
↳ also called Task Queue

Suppose if user clicks on the button 5 to 6 times, so the callback function will be pushed into the callback queue 5 to 6 times.



Now the eventloop takes the first cb function pushed into call stack, the call stack executes it & pops it out. Again, the event loops sees whether call stack is empty then it takes the next cb() function & pushes it onto call stack and so on ——

* How fetch() function works?

e.g.

```
console.log ("start");

setTimeout (function cbT()) {
        console.log ("CB SetTimeout");
    }, 3000);

fetch ("https://api.netflix.com").then (function cbF()){
        console.log ("CB Netflix");
});

console.log ("End");
```

→ The fetch function requests the API call. It returns a promise. The callback function cbF() will be executed once promise is resolved.

→ In the code, first Start will be printed then setTimeout registers cbT() & attaches a timer of 3sec in WebAPI environment.

→ The fetch is an WebAPI which is used to make network calls. It also registers cbF() in WebAPI environment.

→ cbT() waiting for timer to expire
cbF() waiting for data to be returned from Netflix server

* MicroTask Queue in JS

→ It has higher priority than Callback Queue. Once the data is received from the Netflix server, the cbF() function will be pushed into the MicroTask Queue

Suppose now the timer also expires then the callback function cbT() will be pushed in the CallBack Queue. First "End" will be printed on the console & once the global execution context is poped out of stack then event loop sees the MicroTask Queue & CallBack Queue. Due to higher priority of MicroTask Queue, cbF() will be pushed into call stack & then "CB Netflix" will be printed on console & cbF will be popped out of call stack. Then Eventloop takes the cbT() function & now the "CB TimeOut" will be printed on consale.

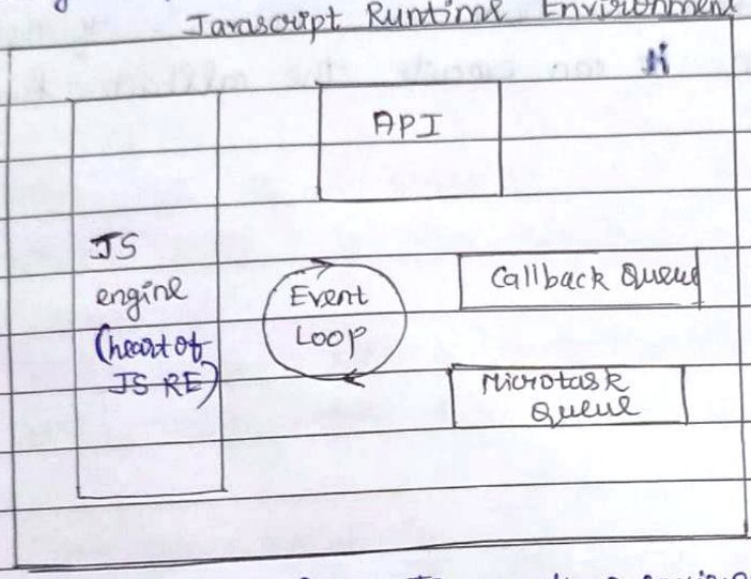
* What are MicroTasks in JS?

⇒ All the callback function which comes through promises will go inside MicroTask Queue.

Mutation Observer:- It keeps on checking whether there is some mutation in DOM tree or not. If there is some mutation it can execute the callback function

Ep 16 - JS Engine EXPOSED

JS can run inside a browser, server, smartwatch, lightbulb, robots. This is possible because of Javascript Runtime Environment.
it is like a big container which has all the things required to run JS code.



Every browser has JS runtime environment to execute JS code.

→ setTimeout & console API are present in both Browser & Node.js but internal implementation is different.

Browsers	JS Engine
Edge	Chakra
Firefox	Spider Monkey
Chrome	V8

→ ECMAScript - it's like governing body of JS language. JS engines needs to follow the ECMA Script language standard.

→ Brendan Eich – created the JS programming language & first JS engine. Now it is known as SpiderMonkey.

* JS engine is not a machine. It is just a piece of code.

* JS Engine Architecture :- It takes the human readable code as input. This code goes through 3 major steps:-
  1) PARSING
  2) COMPILATION
  3) EXECUTION

1] PARSING: During this phase, the code is broken down into tokens.
Syntax Parsers :- The job of syntax passers is to take the code & convert it into Abstract Syntax Tree (AST).

Abstract Syntax Tree :- A tree representation of program source code. It is data structure widely used in compilers, due to their property of representing the structure of program code.

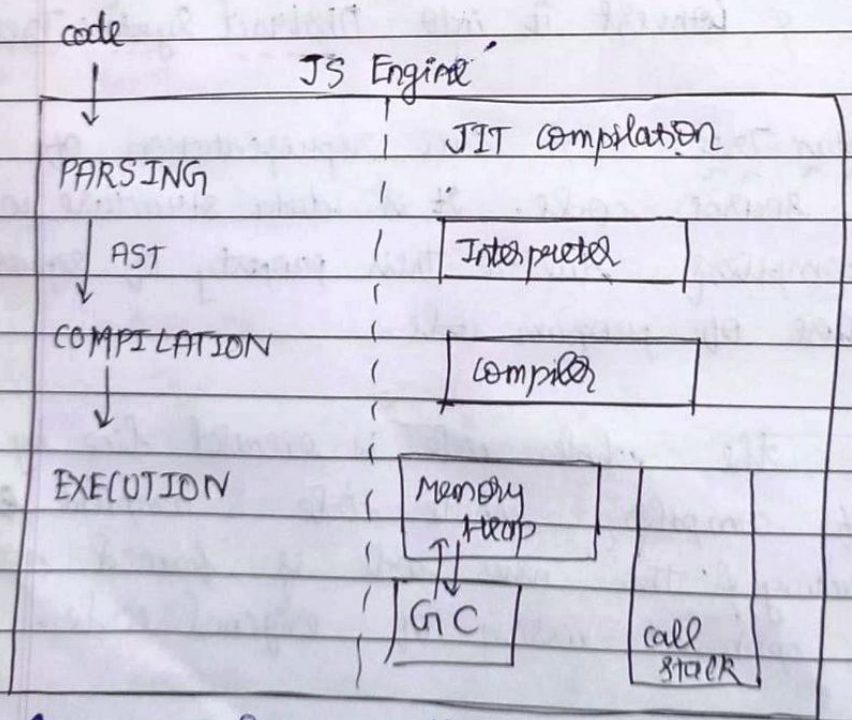
→ In Interpreter, the whole code is executed line by line.
→ In case of compiler, whole code is compiled even before executing & the new code is formed which is the optimised version of original code.

→ JS can behave like an interpreted language as well as compiled language, everything is dependent on JS engine.

* JIT compilation: JS engine can use interpreter along with compiler that makes it just in time compiled language.

* AST goes to the interpreter, interpreter converts a HL code to byte code & that code moves to the execution step. While doing, it takes help of compiler to optimize the code.
The job of compiler is to optimize the code as much as it can on the runtime. ∴ It is also known as JIT compilation.

3] EXECUTION: Execution of code is not possible without two major components:- 1) Memory Heap 2) Call Stack.



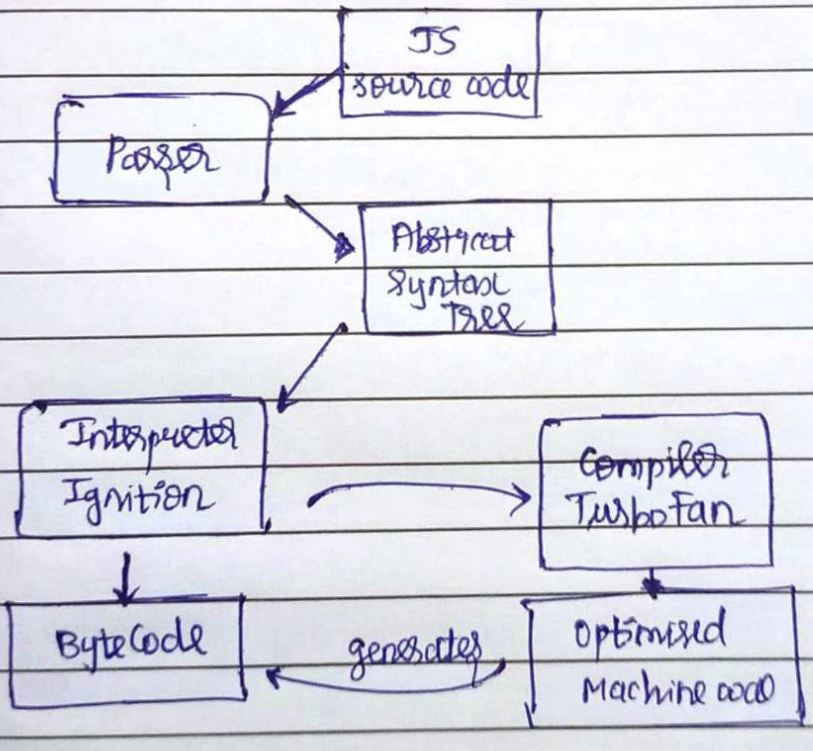
In memory heap, all the variables & fn are assigned memory.

Inlining, copy elision, inline caching

* Garbage Collector using Mark & Sweep Algorithm.
Mark & Sweep Algo looks out for objects 'which are unreachable' rather than objects 'which are no longer needed.

* Fastest JS engine - V8. It has a interpreter named "Ignition". & "Turbofan" which is optimised compiler.

V8 JS Engine



* MARK AND SWEEP ALGORITHM

This algo goes through 3 steps:

1) Root: It is global variable used in code. A window object in JS can act as a root. This algo uses global object root to find whether the objects are reachable or unreachable.

2) This algo then monitors every root & their children. While monitoring -
   objects which are reachable — marked
   " " " not reachable - unmarked

3) Objects that are unmarked means which are unreachable will be garbage collected.

* MARK PHASE: In this, we are able to find elements which are marked & unmarked.

e.g
```
var obj = {
            prop: "hello"
          }
```

Here the global object 'root' used by algo can reach obj & its property "hello". So its marked now.

Let us assign a null value to this obj.
Now new assigned 'null' will be marked & property 'hello' will be unmarked. (not reachable)

e.g
```
obj = null
```

* SWEEP PHASE:

→ It sweeps the unreachable objects.

→ object with property 'hello' is unmarked, it will be garbage collected in this phase.

Ep. 17 TRUST ISSUES with setTimeout()

e.g
```
function cb() {
    console.log("callback");
}
setTimeout(cb, 5000);
```

A setTimeout with delay of 5 sec here does not always wait for 5 seconds. It does not guarantee that the callback cb() function here will be called after 5 seconds.

* The setTimeout callback function will only execute when the global execution context is popped out of the stack. It does not always run after the 't' seconds.

e.g Code Demonstration of the setTimeout delay

```
console.log("start");
setTimeout(function cb() {
    console.log("Namaste JS");
}, 5000);
console.log("end");

let startDate = new Date().getTime();
let endDate = startDate;
while (endDate < startDate + 10000) {
    endDate = new Date().getTime();
}
console.log("While expires");
```

Here newDate() gives us 'Current Day, Month, Date, Year . Time

newDate().getTime() → gives the current time in milliseconds

startDate takes the current time & wait for 10 seconds. Basically the while loop is used to provide the delay.

→ First global execution context will be pushed onto Stack

→ First "start" will be printed on console by using the console WebAPI method & then setTimeOut WebAPI registers the cb() function in the webAPI environment & attaches a timer of 5 seconds. Then "End" will be printed on the console. Now the while loop will run till 10 seconds. But the times for callback of cb() also expires but it will have to wait in the Callback Queue until the whole code is executed ie until the GEC is popped out of the stack.

→ Then after 10 sec "While Expires" will be printed. Now the EventLoop keeps on monitoring if there is something inside the callstack. As soon as the GEC is popped out, it takes the cb() & pushes it into the call stack & it prints "Callback" on the console.

→ So it is not necessary the delay that we provided ie of 5 seconds ensures that the cb() will execute after 5 seconds. Here it executes after 10 seconds.

* setTimeOut (0) :-

e.g
```
console.log ("start");
setTimeout ( function cb(){
        console.log ("callback");
},0);
console.log ("end");
```

Here the delay provided is of 0 ms. Hence we expect that start, callback & end will be printed on console. But it is not.
Here the callback f^n cb() will be register in the webAPI environment then it will have to wait in callback Queue & then it will be executed.

So o/p is :-
```
start
End
callback
```

Ep. 18 Higher-Order Functions.

* Higher order Functions :- A function which takes another function as argument or returns a function from it is known as higher order functions.

eg
```
function x () {
    console.log("Namaste");
}

function y (x) {
    x();
}
```

Here function y is called higher order function & x is called callback function.

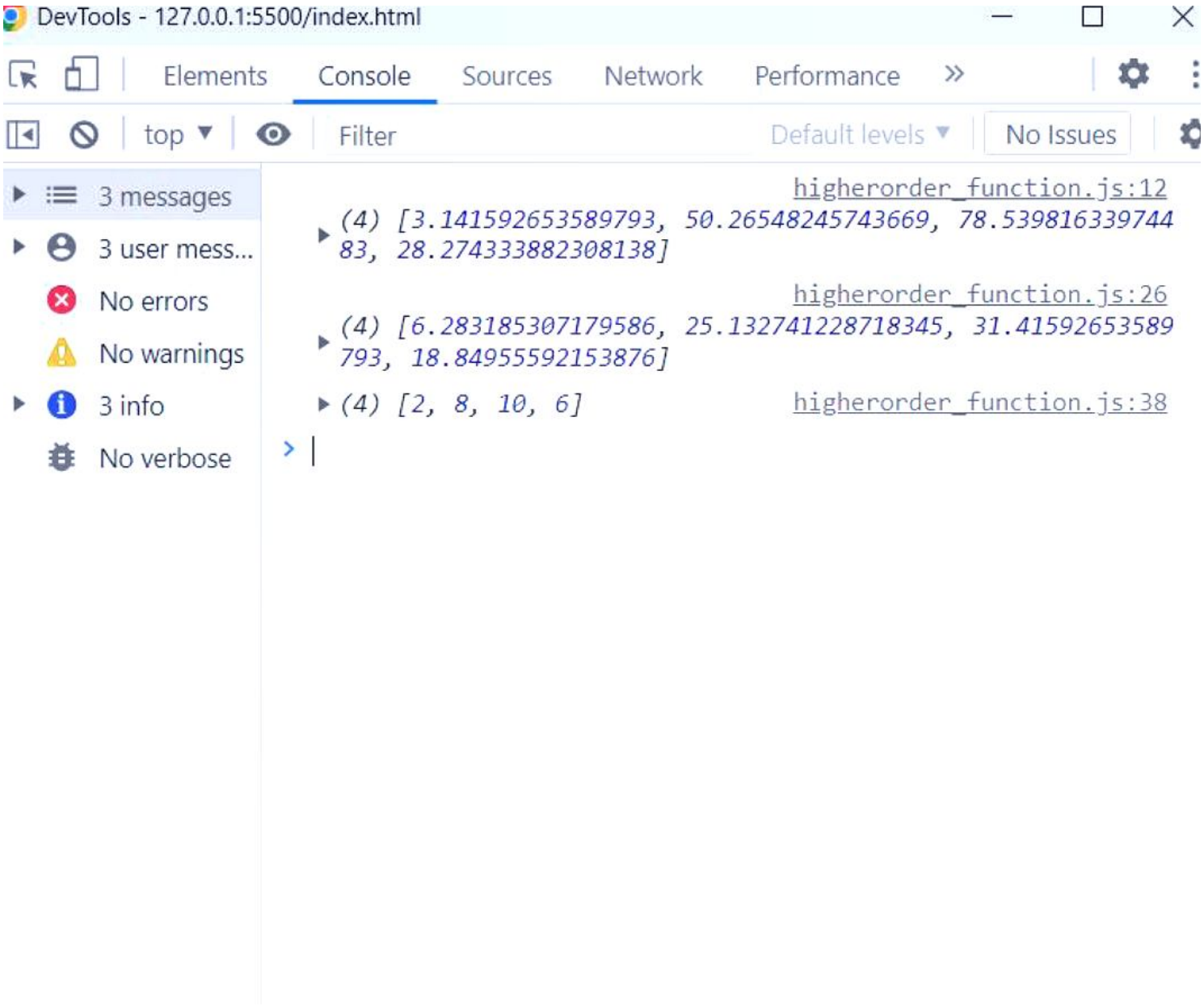
* Beauty of functional programming :-
In the example, each of every function has its own responsibility. ie area function calculates area, circumference function calculates circumferena. They are performing just one task.
The calculate function task is to create a new array 'output', push the elements into that array & return the array.

→ Reusability,
→ Modularity
→ Passing function into another function

* Polyfill for map function in JS:

JS higher_order_function_optimised_code.js > area

```
1   const radius=[1,4,5,3];// radius of 4 circles
2
3   const area=function(radius)
4   {
5       return Math.PI*radius*radius;
6   }
7
8   const circumference=function(radius)
9   {
10      return 2*Math.PI*radius;
11  }
12  const diameter=function(radius)
13  {
14      return 2*radius;
15  }
16
17  const calculate=function(radius,logic)
18  {
19      const output=[];
20      for(let i=0;i<radius.length;i++)
21      {
22          output.push(logic(radius[i]));
23      }
24      return output;
25  };
26  console.log(calculate(radius,area));
27  console.log(calculate(radius,circumference));
28  console.log(calculate(radius,diameter));
29
```


DevTools - 127.0.0.1:5500/index.html
Elements
Console
Sources
Network
Performance
top
Filter
Default levels
No Issues
3 messages
3 user mess...
No errors
No warnings
3 info
No verbose
higherorder_function.js:12
(4) [3.141592653589793, 50.26548245743669, 78.53981633974483, 28.274333882308138]
higherorder_function.js:26
(4) [6.283185307179586, 25.132741228718345, 31.41592653589793, 18.84955592153876]
(4) [2, 8, 10, 6]
higherorder_function.js:38

JS higherorder_function.js > ...

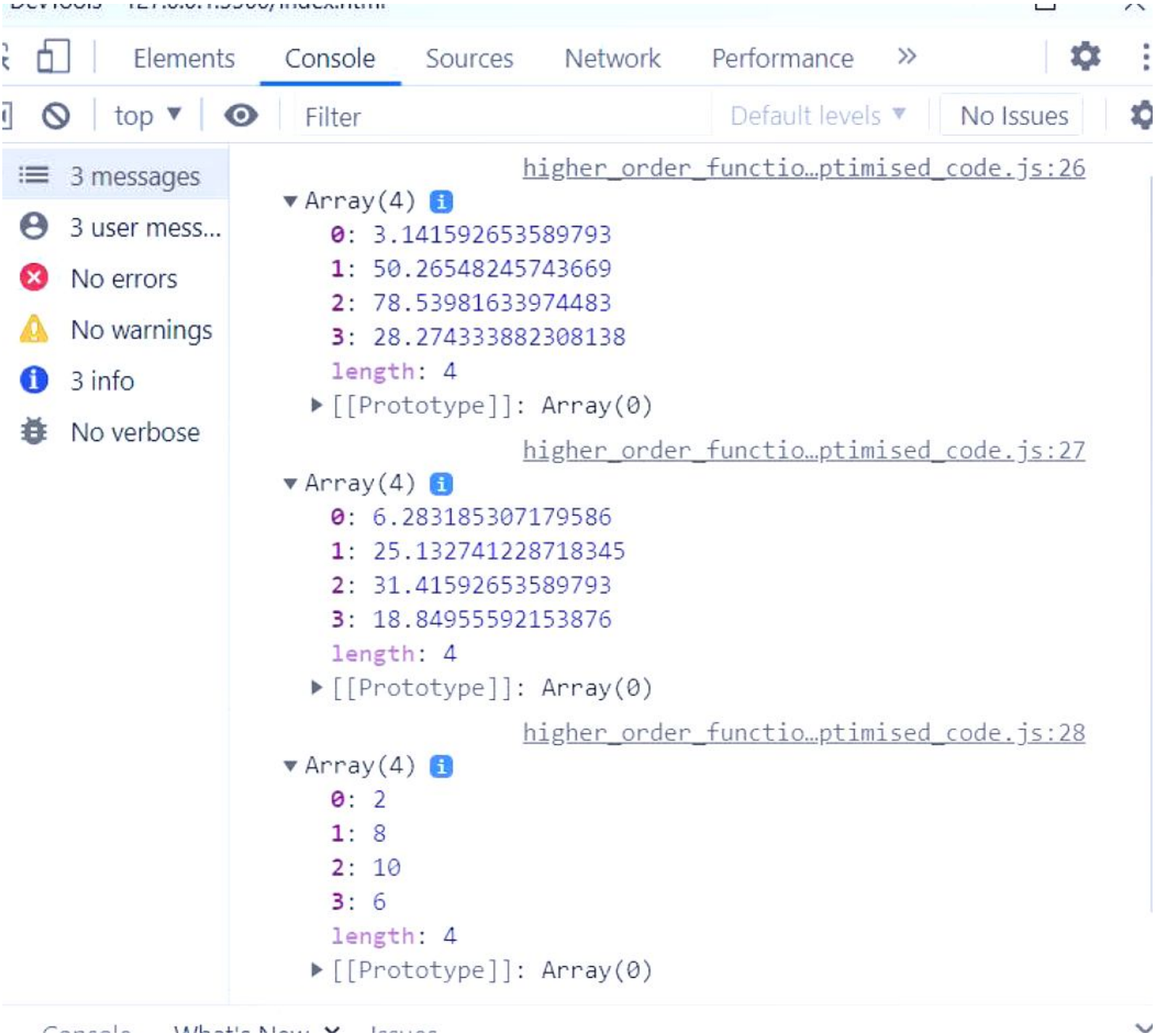
```
const radius=[1,4,5,3];// radius of 4 circles

const areaCalculation=function(radius)
{
    const output=[];//to store areas of 4 circles
    for(let i=0;i<radius.length;i++)
    {
        output.push(Math.PI*radius[i]*radius[i]);
    }
    return output;
};
console.log(areaCalculation(radius));//prints the area of 4 circles


//to calculate circumference

const calculateCircumference =function(radius)
{
    const output=[];//to store circumference of 4 circles
    for(let i=0;i<radius.length;i++)
    {
        output.push(2*Math.PI*radius[i]);
    }
    return output;
};
console.log(calculateCircumference(radius));//prints the circumference of 4 circles

//to calculate diameter
const calculateDiameter =function(radius)
{
    const output=[];//to store circumference of 4 circles
    for(let i=0;i<radius.length;i++)
    {
        output.push(2*radius[i]);
    }
    return output;
};
console.log(calculateDiameter(radius));//prints the diameter of 4 circles
```


Elements
Console
Sources
Network
Performance
top
Filter
Default levels
No Issues
3 messages
3 user mess...
No errors
No warnings
3 info
No verbose
higher_order_functio…ptimised_code.js:26
Array(4)
0: 3.141592653589793
1: 50.26548245743669
2: 78.53981633974483
3: 28.274333882308138
length: 4
[[Prototype]]: Array(0)
higher_order_functio…ptimised_code.js:27
Array(4)
0: 6.283185307179586
1: 25.132741228718345
2: 31.41592653589793
3: 18.84955592153876
length: 4
[[Prototype]]: Array(0)
higher_order_functio…ptimised_code.js:28
Array(4)
0: 2
1: 8
2: 10
3: 6
length: 4
[[Prototype]]: Array(0)

Ep. 19 MAP, FILTER & REDUCE

* Map, Filter, Reduce are higher order functions in JS.

* ARRAY.map() function in JS:

→ It creates a new array from calling a function for every array element.
→ It does not change the original array.
→ map() calls a function once for each element in an array.
→ map() does not execute the ~~array~~ function for empty elements.

e.g 1

```
const arr = [1, 2, 3, 4, 5]

// To transform the array, suppose we need
// to convert into array named Double = [2, 4, 6, 8, 10]

function double (x) {
    return x * 2;
}

const output = arr.map(double);
console.log(output);
// o/p:- (5) [2, 4, 6, 8, 10]
```

Ex. 2

```
function triple (x) {
    return x * 3;
}

const output2 = arr.map(triple);
console.log(output2);
// prints (5) [3, 6, 9, 12, 15]
```

eg 3 To convert the binary format

```
const arr = [1, 2, 3, 4, 5]

function binary (x) {
    return x.toString (2);
}

const output3 = arr.map (binary);

console.log (output3);
```

// o/p:- ["01", "10", "11", "100", "101"]

2nd way of writing the code :-

```
const output3 = arr.map ( function binary (x) {
    return x.toString (2);
});
```

3rd way : using arrow function.

```
const output = arr.map ((x) => x.toString(2));
```

* filter ( ) function :- → creates a new array filled with elements that pass a test provided by a function.
→ The filter ( ) method does not execute the function for empty elements.
→ The filter ( ) method does not change the original array.

e.g
```
const arr = [1, 2, 3, 4, 5]
// filter odd values

function isOdd(x) {
    return x%2 ; // if it is even return false else True
}
const output = arr.filter(isEven);
console.log(output);
```

o/p :- (3) [1, 3, 5]

Similarly, we can check for isEven etc.

→ We can also pass the function directly in filter function as well as we can use array function.

e.g
```
const output = arr.filter((x) => x>4);
```

* Reduce() function :- It is used at a place where we have to take all the elements of array & come up with a single value.

e.g to find the sum of all elements of array.

1) using normal function :-

```
const arr = [1, 2, 3, 4, 5]
function fSum(arr) {
    let sum = 0;
    for (let i=0; i<arr.length; i++) {
        sum = sum + arr[i];
    }
```

```
    return sum;
}
console.log(fsum(arr));
// o/p:- 15
```

ii) using reduce() function:- reduce() function takes 2 arguments.

eg
```
const arr = [1, 2, 3, 4, 5]

const output = arr.reduce(function(acc, curr){
    acc = acc + curr;
    return acc;
}, 0);
```
→ represents the initial value of acc

```
console.log(output); // prints 15
```

Here acc – accumulator (it acts as a sum variable)

curr – current value of array.

Here function is called for each & every element of array.

e.g to find maximum element of array:-

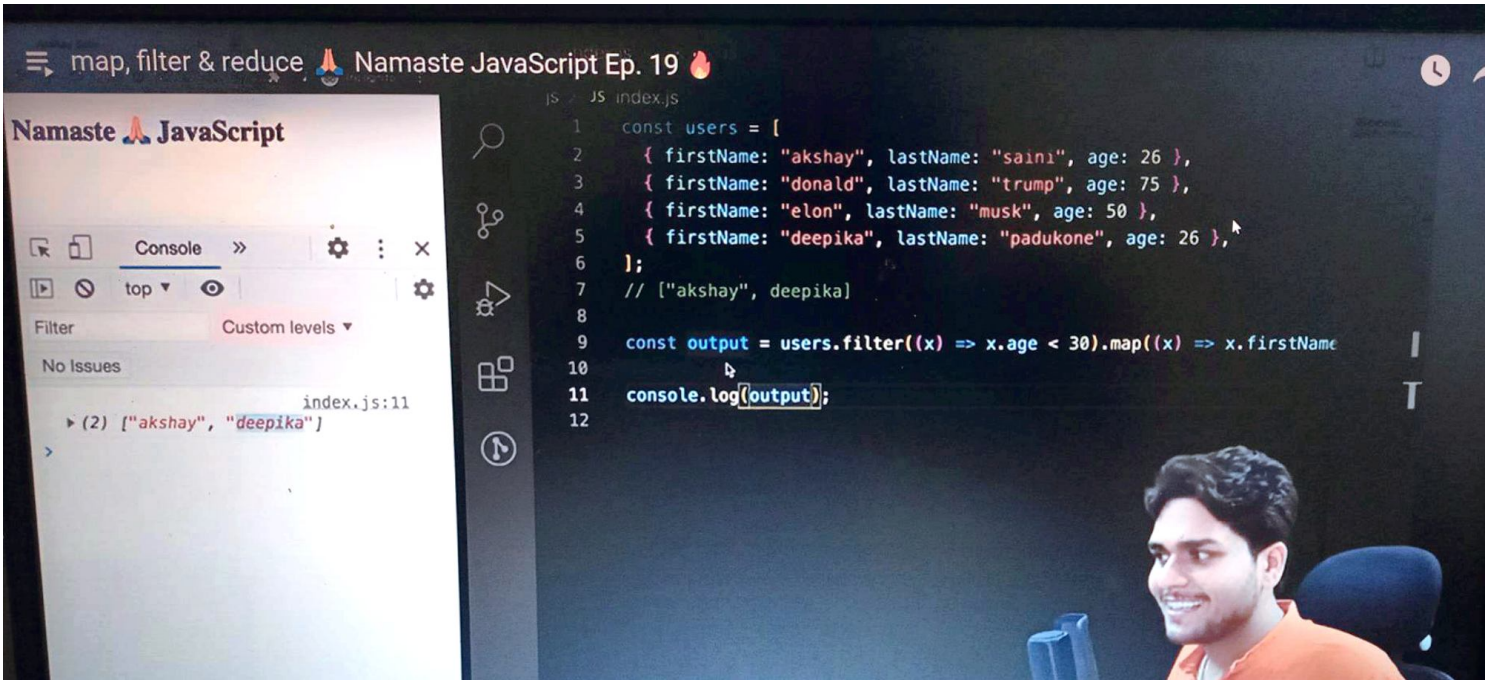
```
const output = arr.reduce(function(max, curr){
    if(curr > max)
    {
        max = curr;
    }
    return max;
}, 0);
console.log(output); // prints 5.
```

* Another example of map() :-

eg
```
const users = [
  { fname : "Akshay", lname: "saini", age: 28 },
  { fname: "Virat", lname: "Kohli", age: 32 },
  { fname: "Rohit", lname: "Sharma", age: 34 },
  { fname: "Dinesh", lname: "Kartik", age: 36 },
];

// we want to print list of full names.

const output = users.map((x) => x.fname + " " +
                                 x.lname);
console.log(output);
// output: [Akshay saini, "Virat Kohli" --- ]
```


map, filter & reduce 🙏 Namaste JavaScript Ep. 19 🔥
Namaste 🙏 JavaScript
Console
top
Filter
Custom levels
No Issues
index.js:11
(2) ["akshay", "deepika"]
JS index.js
const users = [
{ firstName: "akshay", lastName: "saini", age: 26 },
{ firstName: "donald", lastName: "trump", age: 75 },
{ firstName: "elon", lastName: "musk", age: 50 },
{ firstName: "deepika", lastName: "padukone", age: 26 },
];
// ["akshay", deepika]
const output = users.filter((x) => x.age < 30).map((x) => x.firstName
console.log(output);


map, filter & reduce 🙏 Namaste JavaScript Ep. 19 🔥
Namaste 🙏 JavaScript
Console
top
Filter
Custom levels
No Issues
index.js:19
{26: 2, 50: 1, 75: 1}
26: 2
50: 1
75: 1
__proto__: Object
JS index.js
const users = [
{ firstName: "akshay", lastName: "saini", age: 26 },
{ firstName: "donald", lastName: "trump", age: 75 },
{ firstName: "elon", lastName: "musk", age: 50 },
{ firstName: "deepika", lastName: "padukone", age: 26 },
];
// acc = { 26 : 1, 75: 1, 50: 1 }
const output = users.reduce(function (acc, curr) {
if (acc[curr.age]) {
acc[curr.age] = ++acc[curr.age];
} else {
acc[curr.age] = 1;
}
return acc;
}, {});
console.log(output);