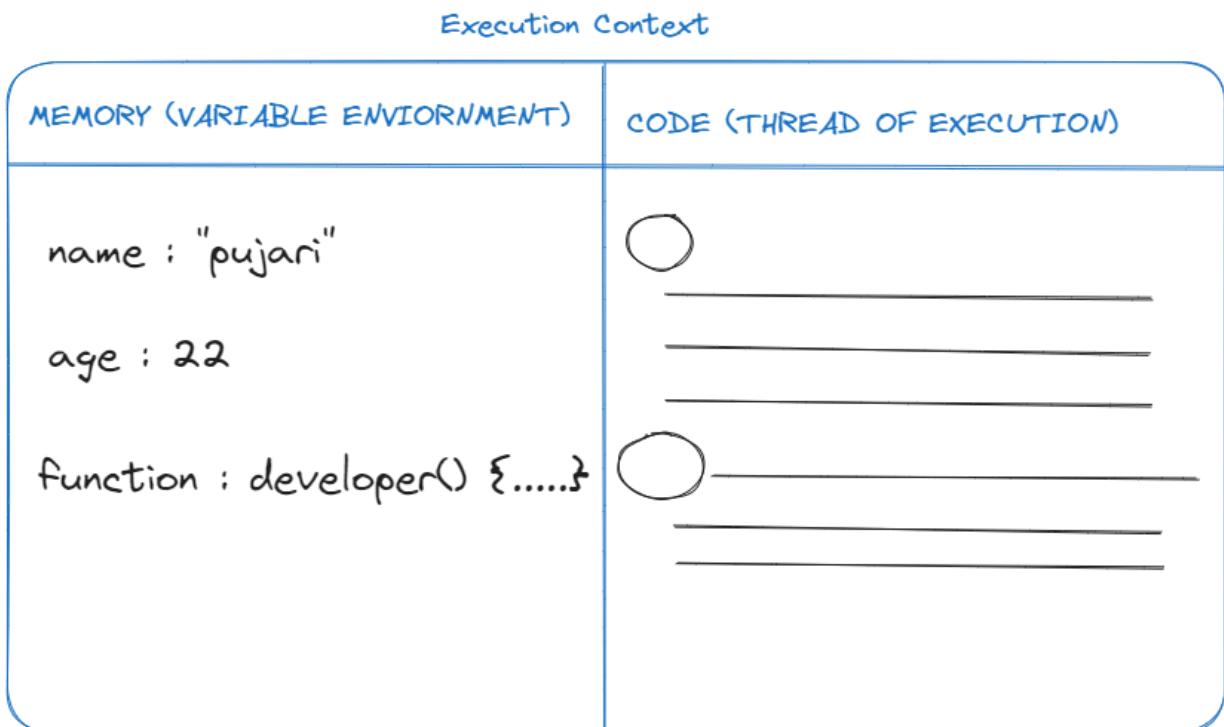


Namaste-JavaScript

- Episode-1(Season-1 -> Intro To how JS Works)

“Everything In JavaScript Happens Inside An **Execution Context**”

Assume Execution Context as Box OR Container



In Memory, everything is stored in (key : value) pair

Eg:- name : “pujari”

The place where Code is executed one line at a time.

“Javascript is the **synchronous, single-threaded** language”.

synchronous : It means only one line can be run at a time.

Single-threaded: It means it can only execute only one command at a time.

“JS Can only execute one command at a time and in a specific order”

– Episode-2(Season-1 ->how JS Code is Executed)

As we all know in JS everything is executed in Execution Context.

Execution Context has 2 Phases

1. Memory Creation Phase
2. Code Execution Phase

Let's take below code for better understanding

```
var name = "pujari";  
  
function developer(designation) {  
    const role = designation;  
    return role;  
}  
var position = developer("frontend engineer");
```

After running the code ,

- In memory creation phase(1), , so it goes to line one of above code snippets, and allocates a memory space for variable 'name' , after allocating memory to name it goes to next line and allocates a memory space for function [developer].

- When allocating memory for 'name' it stores 'undefined', a special value for 'name'. For [developer], it stores the whole code of the function inside its memory space.
- As position is also a variable it allocates memory & stores a space.

The memory allocation phase would look something like this

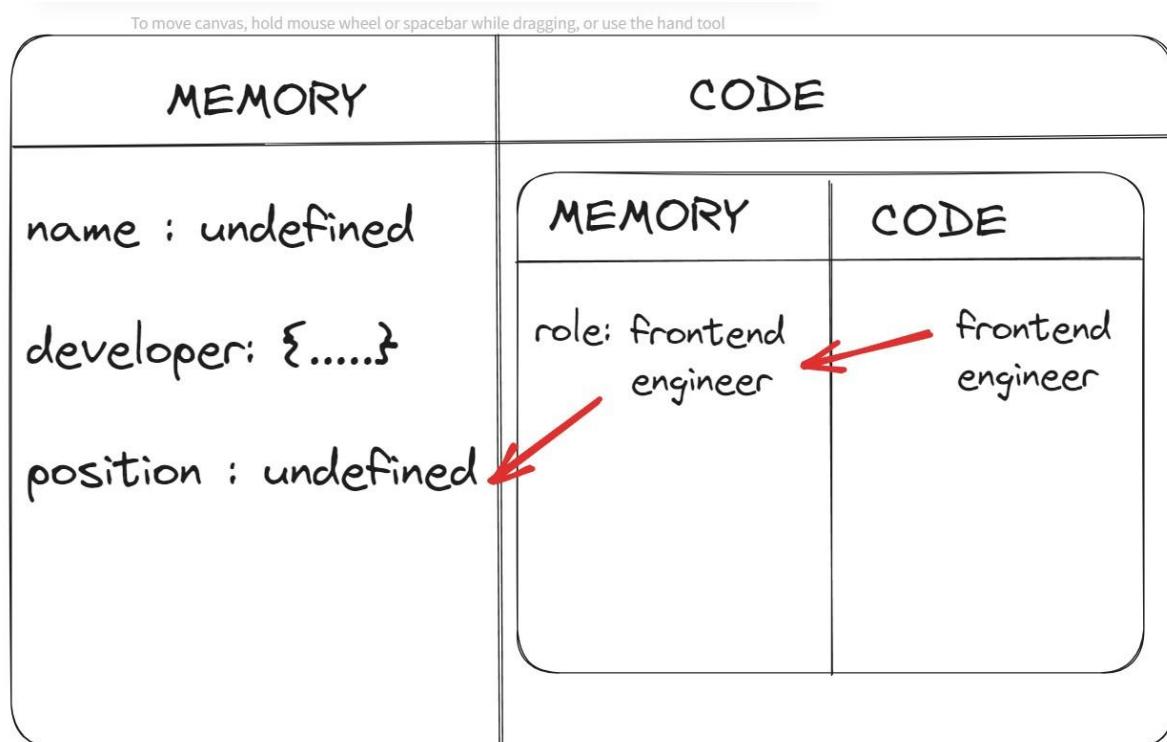
MEMORY	CODE
<pre>name : undefined developer: {.....} position : undefined</pre>	

Now, in the 2nd phase i.e. code execution phase, it starts going through the whole code line by line.

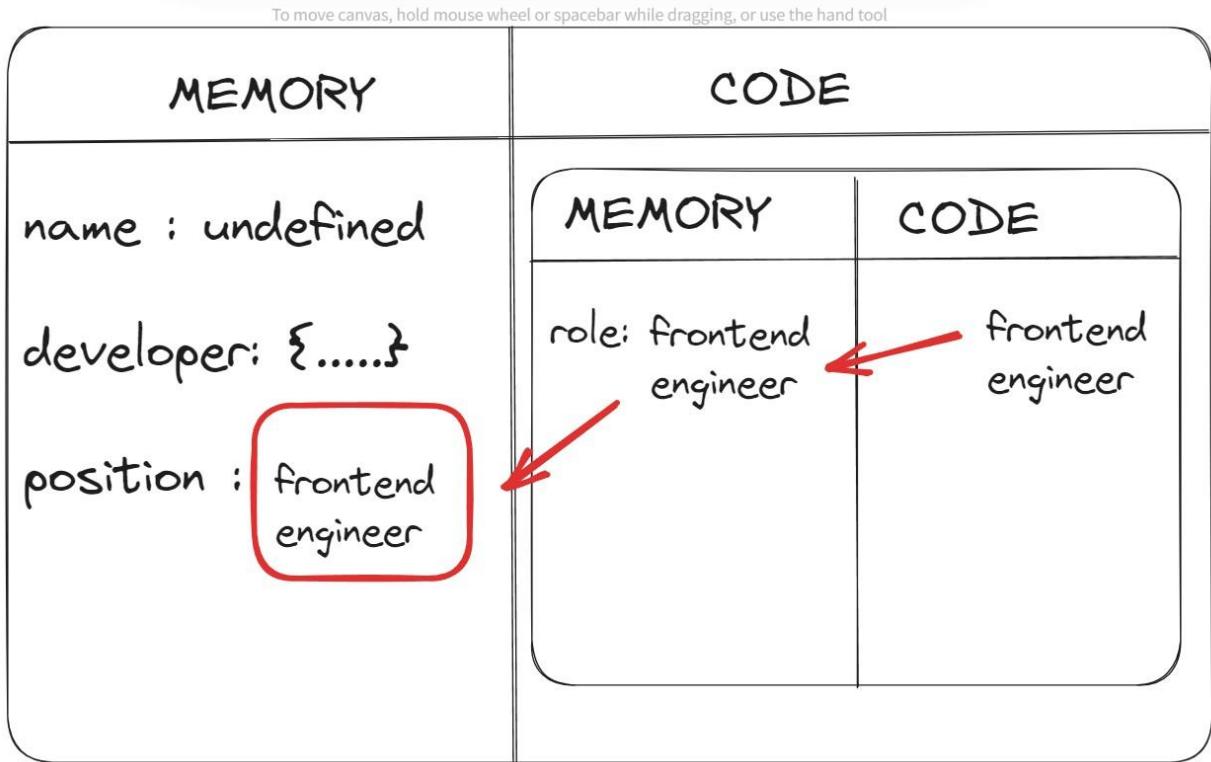
- As it encounters var name = “pujari”, it assigns ‘pujari’ to ‘name’. Until now, the value of ‘name’ was undefined. For function, there is nothing to execute. As these lines were already dealt with in the memory creation phase.

- Coming to line 8 i.e. var `position`= developer("frontend engineer"), here functions are a bit different than any other language. A new execution context is created altogether.
- Again in this new execution context, in the memory creation phase, we allocate memory to the role. An undefined value is given to the `role` variable.
- Now, in the code execution phase of this execution context, first ("frontend engineer") is assigned to the role and stores it inside the role . After that, the return `role` returns the control of the program back to where this function was invoked from.

The memory allocation & Code execution phase would look something like this



- When the return keyword is encountered, It returns the control to the called line and also the function execution context.



As we all have gone through JS engine runs the code BTS and it looks like very tedious task right but that's the beauty of JS

To manage all this things It has stack known as [Call Stack](#)

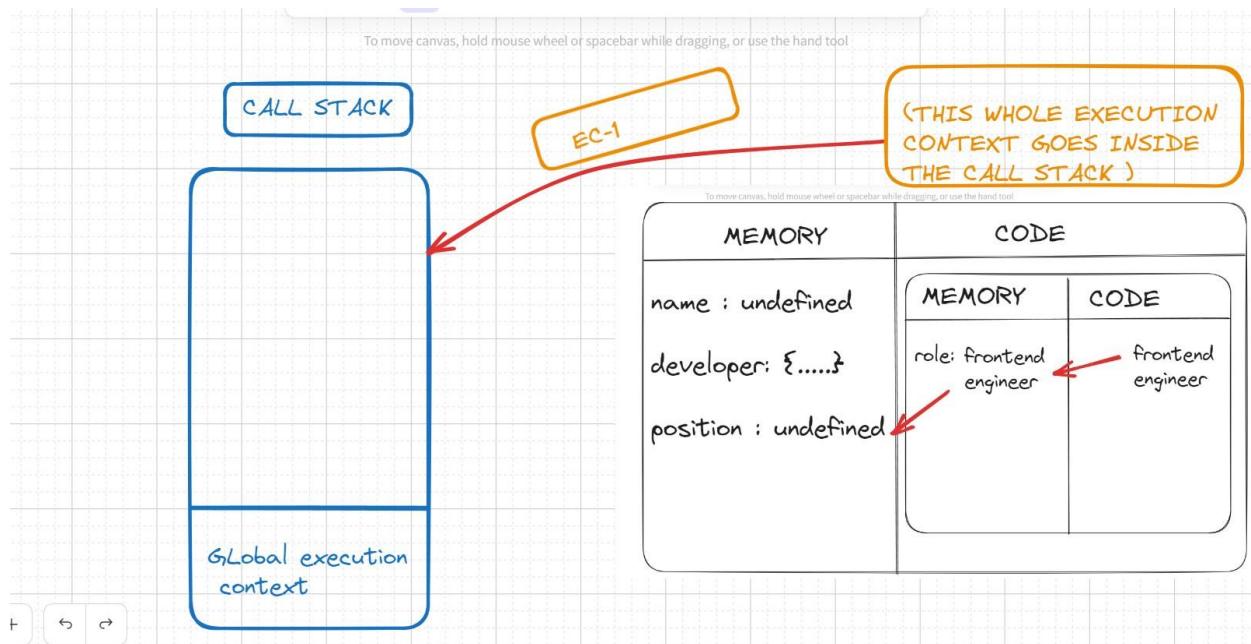
[Call Stack](#) maintains the order of execution of execution contexts.

It is also known as Program Stack, Control Stack, Runtime stack, Machine Stack, Execution context stack.

(global execution context is created when a JavaScript script first starts to run, and it represents the global scope in JavaScript.)

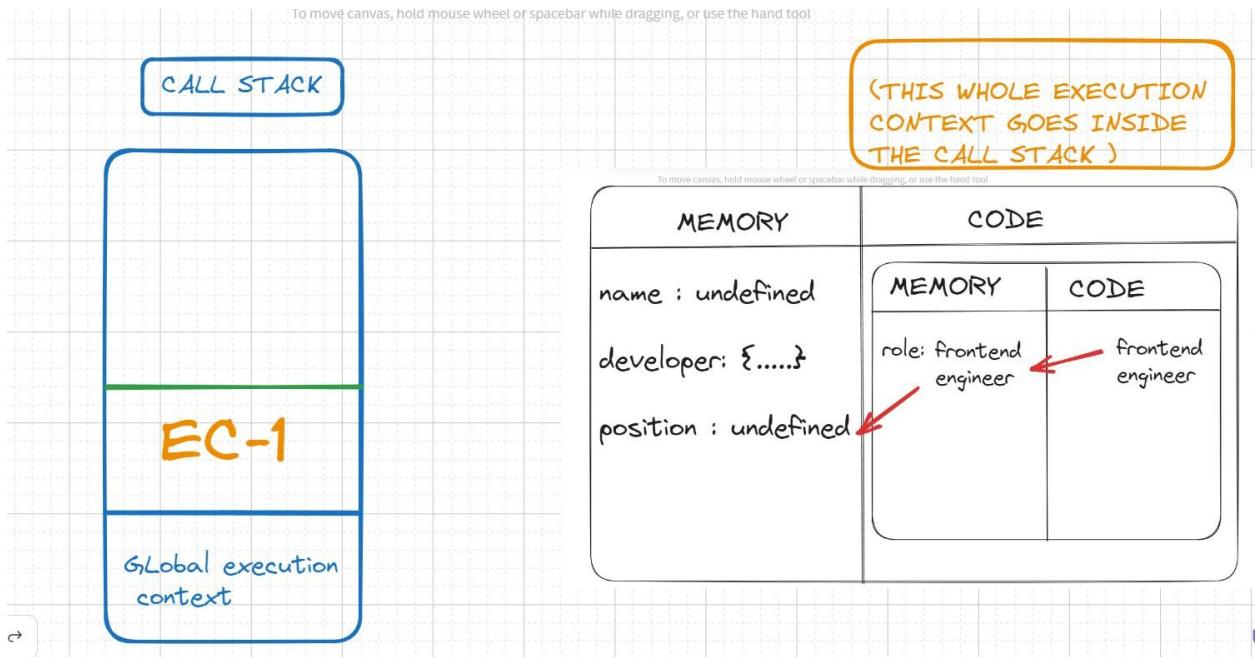
Let's Take Example Of How does execution context does get stacked into call stack

Before Running the program



Before Running the program call stack is empty

After Running program (execution context-1) is stacked inside the call stack above the global execution context.



– Episode-3(Season-1 ->Hoisting in JS)

Hoisting

- ★ Variable hoisting in JavaScript is a behavior where variable declarations are moved (or "hoisted") to the top of their containing scope during the compilation phase, regardless of where they are actually declared within the code. This means that even if you declare a variable later in your code, JavaScript will move the declaration to the top of its current scope, making it available for use throughout the entire scope.

In JS, even before the code starts execution the **memory** is allocated to variable & function.

Let's take the example of below code and it's explanation

```
var num = 18;

function getName() {
    console.log("Namaste");
}

getName();
console.log(num);
console.log(getName);
```

1. It should have been an outright error in many other languages, as it is not possible to even access something which is not even created (defined) yet But in JS, We know that in the memory creation phase it assigns undefined and puts the content of function to function's memory.
2. And in execution, it then executes whatever is asked. Here, as execution goes line by line and not after compiling, it could only print undefined and nothing else. This phenomenon is not an error. However, if we remove var num = 18; then it gives an error. Uncaught ReferenceError: x is not defined

Let's take another example

```
getName();
console.log(num);
console.log(getName);

var num = 18;

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

Here when the variable and function is accessed before defining , the variable returns undefined and the function return function has an object.

```
getName(); // Namaste
console.log(num); // Uncaught Reference: num is not defined.
console.log(getName); //function getName() {console.log("Namaste")}
```

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

```
7
8  var position = developer("frontend engineer");
9
10
11  getName(); // Uncaught TypeError: getName is not a function
12  console.log(num);
13  console.log(getName);
14
15
16  var getName = function () {
17    |   console.log("Namaste");
18  }
19
20 // The code won't execute as the first line itself throws an TypeError.
21
```

– Episode-4(Season-1 -> how function works in JS)

Function —> A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output.

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

function a() {
    var x = 10*3;
    console.log(x);
}
function b() {
    var x = 10*10;
    console.log(x);
}
```

The Output will be —

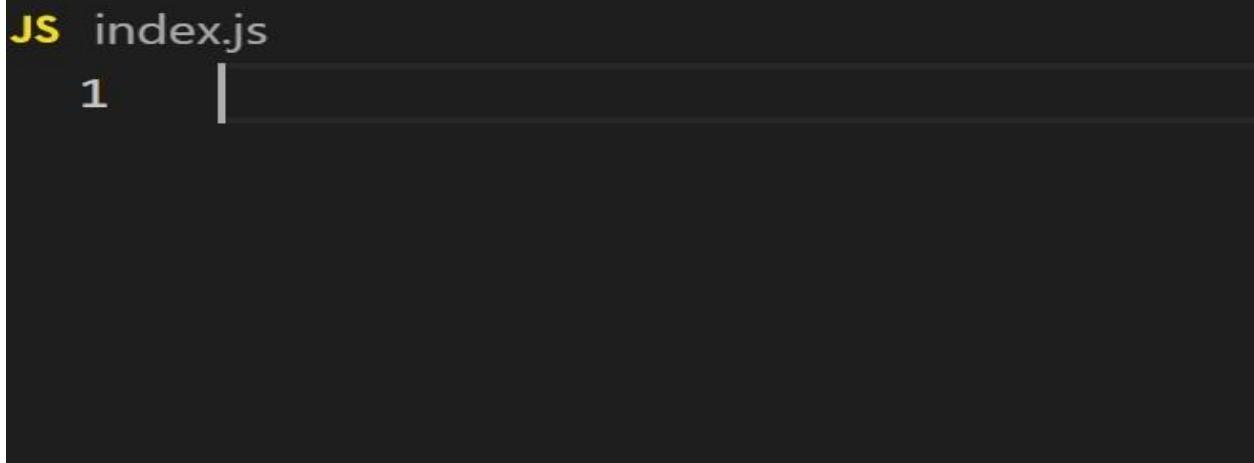
10

30

100

– Episode- 5(Season-1 -> Shortest JS Program, window & this keyword)

Have you ever think that what would be the shortest Program in JavaScript, If No then have a look what it look like



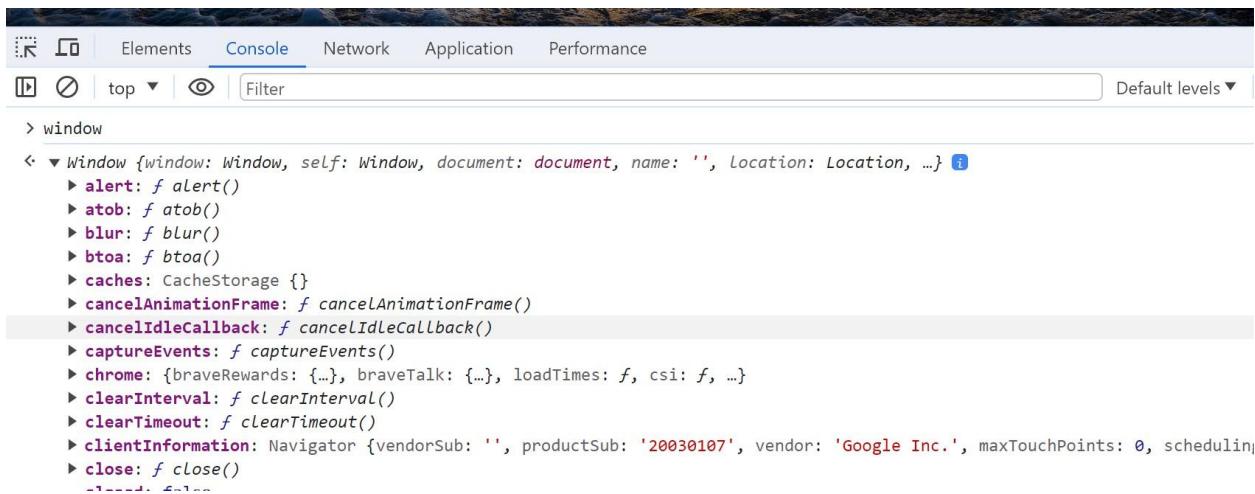
JS index.js

1 |

Many of them will say what rubbish but yes this is this shortest program you could ever write in js 😱 I'm not kidding
Let's understand why this is the shortest program,

- ★ Behind The Scenes when the program is run, the JS engine does a lot of things. As always, even in this case, it creates the GEC (global execution context) which has memory space and the execution context.
- ★ The JS engine creates something known as 'window'. It is an object, which is created in the global space. It contains lots of functions and variables. These functions and variables can be accessed from anywhere in the program.
- ★ The JS engine also creates a "this" keyword, which points to the window object at the global level. So, in summary, along with GEC, a global object (window) is created and a "this" variable is created.
- ★ In different engines, the name of global object changes. Window in browsers, but in nodeJS it is called something else. At global level, this == window If we create any variable in the global scope, then the variables get attached to the global object.

Window Object looks something like this



The screenshot shows the Chrome DevTools interface with the "Console" tab selected. At the top, there are tabs for Elements, Network, Application, and Performance. Below the tabs, there are buttons for "top" and "Filter", and a dropdown for "Default levels". The main area displays the properties of the global "window" object. A grey highlight box surrounds the "cancelIdleCallback" property, which is described as a function that cancels an idle callback. Other properties listed include alert, atob, blur, btoa, caches, cancelAnimationFrame, captureEvents, chrome, clearInterval, clearTimeout, clientInformation, close, and others.

```
> window
< - Window {window: Window, self: Window, document: document, name: '', location: Location, ...} ⓘ
  ► alert: f alert()
  ► atob: f atob()
  ► blur: f blur()
  ► btoa: f btoa()
  ► caches: CacheStorage {}
  ► cancelAnimationFrame: f cancelAnimationFrame()
  ► cancelIdleCallback: f cancelIdleCallback() cancelIdleCallback
  ► captureEvents: f captureEvents()
  ► chrome: {braveRewards: {...}, braveTalk: {...}, loadTimes: f, csi: f, ...}
  ► clearInterval: f clearInterval()
  ► clearTimeout: f clearTimeout()
  ► clientInformation: Navigator {vendorSub: '', productSub: '20030107', vendor: 'Google Inc.', maxTouchPoints: 0, schedulingBehavior: '...', ...}
  ► close: f close()
```

– Episode- 6(Season-1 -> undefined and not defined in JS)

The “undefined” is just a keyword given the variable or you can think it’s the placeholder kept for the variable in execution context

If an object/variable is not even declared/found in memory allocation phase, and tried to access it then it is Not defined

Not Defined != Undefined

When a variable is declared but not assigned value, its value will be undefined. But when the variable itself is not declared but called in code, then it is not defined.

```
//ep- 6

console.log(x); // undefined var x = 25;
console.log(x); // 25
console.log(a); // Uncaught ReferenceError: a is not defined
```

JS is a loosely typed / weakly typed language. It doesn't attach variables to any data type. Let's Example to understand it in better manner if we declare a variable `inconstant = 19`, and then later we change the value of variable to string , `inconstant = "power of js"` , This is the power of JS🔥.

But one should Never assign `undefined` as a value to any variable.

– Episode-7 (Season-1 -> Scope Chain, 🔥 Scope & Lexical Environment)

Scope —> Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript.

JavaScript has the three different of scopes:

- Global scope: The default scope for all code running in script mode.
- Module scope: The scope for code running in module mode.
- Function scope: The variable created within a function is known as Function scope.

Global Scope —> Variables defined outside any function, block, or module scope have global scope. Variables in global scope can be accessed from everywhere in the application.

Example:

```
<script>
  let GLOBAL_DATA = { value : 1};
</script>

console.log(GLOBAL_DATA);
```

Module scope —> Before modules, a variable declared outside any function was a global variable. In modules, a variable declared outside any function is hidden and not available to other modules unless it is explicitly exported.

Exporting makes a function or object available to other modules. In the next example, I export a function from the `sequence.js` module file:

```
// in sequence.js
export { sequence, toList, take };
```

Importing makes a function or object, from other modules, available to the current module.

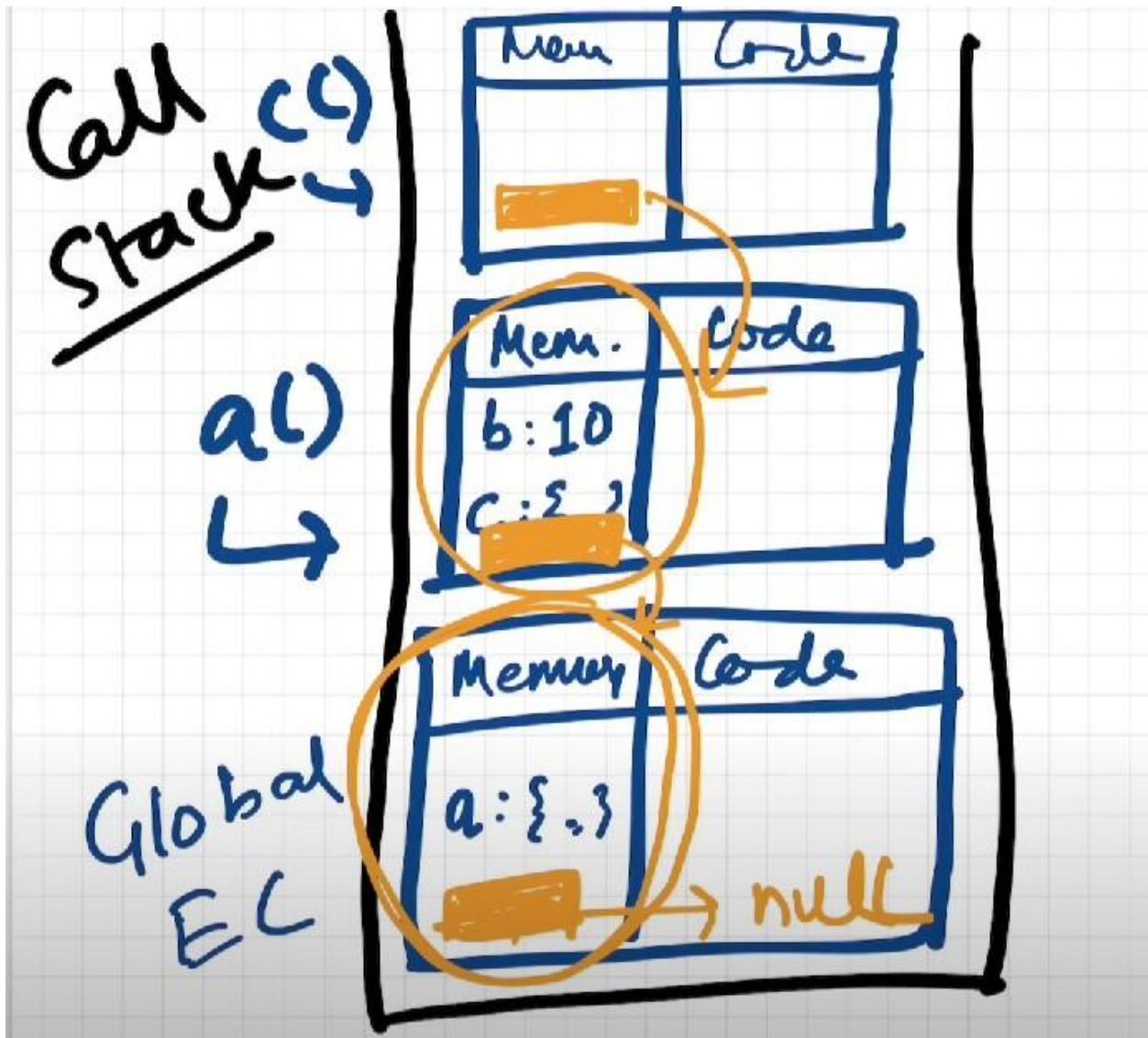
Function Scope —> Function scope means that parameters and variables defined in a function are visible everywhere within the function, but are not visible outside of the function.

```
function a() {  
    var b = 19;  
    console.log(b);  
}
```

Let's take a example to understand lexical scope

```
function a() {  
    var number = 19;  
    c();  
    function c() {  
        //  
    }  
}  
  
a();  
console.log(number);
```

The Call stack will be looking something like this,



With The above Execution Context it shows that there are is chain of the scopes that is dependent on each other ,lexical scope of function c() is dependent on the local scope of function a() and lexical scope of function a() is dependent on the **gec**

This is how code works BTS

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. A breakpoint is set on line 5 of the file 'index.js'. The code editor displays the following JavaScript:

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

The line with the breakpoint (line 5) is highlighted in blue. The status bar at the bottom indicates 'Paused on breakpoint'.

The Call Stack panel shows the current call stack:

- c at index.js:5
- a at index.js:3
- (anonymous) at index.js:9

The Local scope panel shows the following variables:

- this: Window
- Closure (a)
- Global

The Breakpoints panel shows that a breakpoint is set and enabled at index.js:5.

Function C () works like this

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. A file named 'index.js' is open, containing the following code:

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

A blue arrow points to line 5, indicating a breakpoint. The status bar at the bottom left shows 'Line 3, Column 3'. The status bar at the bottom right shows 'Coverage: n/a'. The 'Scope' tab in the bottom right panel is active, displaying the local scope variables: 'b: 10', 'c: f c()', and 'this: Window'. The 'Call Stack' panel shows the call chain: 'c' at index.js:5, 'a' at index.js:3, and '(anonymous)' at index.js:9. The 'Breakpoints' panel shows a checked breakpoint at index.js:5.

Function a () works like this

Lexical Scope —> Lexical scope is the ability of the inner function to access the outer scope in which it is defined.

```
function a() {  
    var number = 19;  
    c();  
    function c() {  
        //  
    }  
}
```

```
a();
```

```
console.log(number);
```

- Episode-7 (Season-1 -> let & const in JS, Temporal Dead Zone)

Everyone while learning Js must have heard about **let** and **const** declarations are hoisted. But exactly what does it mean? The block of code is aware of the variable, but it cannot be used until it has been declared. Using a let variable before it is declared will result in a ReferenceError .

Let's take example of this

```
// ep-8
console.log(a); // ReferenceError: Cannot access 'a' before initialization
console.log(b); // prints undefined as expected
let a = 10;
console.log(a); // 10
var b = 15;
console.log(window.a); // undefined
console.log(window.b); // 15
```

Both 'a' and 'b' are actually initialized as undefined in the hoisting stage. But var b is inside the storage space of GLOBAL, and 'a' is in a separate memory object called script, where it can be accessed only after assigning some value to it first ie. one can access 'a' only if it is assigned. Thus, it throws an error.

Temporal Dead Zone : It's the Time since when the let variable was hoisted until it is initialized some value.

- So any line till before "let a = 10" is the TDZ for a
- Since a is not accessible on global, it's not accessible in window/this also.
window.b or this.b -> 15; But window.a or this.a ->undefined, just like window.x->undefined
(x isn't declared anywhere)

Now let's have a look at Types Of Error we may encounter

Uncaught TypeError: Assignment to constant variable

This Error signifies that we are reassigning to a const variable.

SOME GOOD PRACTICES:

- Try using const wherever possible.
- If not, use let, Avoid var.

- Declare and initialize all variables with let to the top to avoid errors to shrink temporal dead zone window to zero

Uncaught ReferenceError: cannot access 'a' before initialization
This Error signifies that 'a' cannot be accessed because it is declared as 'let' and since it is not assigned a value, it is its Temporal Dead Zone. Thus, this error occurs.

Uncaught SyntaxError: Identifier 'a' has already been declared
This Error signifies that we are redeclaring a variable that is 'let' declared. No execution will take place (because js scans the whole code).

- Episode-9 (Season-1 -> BLOCK SCOPE & Shadowing in JS)

What is a **Block** ?

- Block aka compound statement is used to group JS statements together into a “1” group. We group them within {...}

```
{  
  var a = 10;  
  let b = 20;  
  const c = 30;  
  // Here let and const are hoisted in Block scope,  
  // While, var is hoisted in Global scope.  
}
```

Block scope and It's accessibility example

```
{  
    var a = 10; let b = 20;  
    const c = 30;  
}  
console.log(a); // 10  
console.log(b); // Uncaught ReferenceError: b is not defined
```

1. In the BLOCK SCOPE; we get b and c inside it initialized as “undefined” as a part of hoisting (in a separate memory space called **block**)
2. While, a is stored inside a GLOBAL scope.
3. Thus we say, *let* and *const* are BLOCK SCOPED. They are stored in a separate memory space which is reserved for this block. Also, they can't be accessed outside this block. But var a can be accessed anywhere as it is in global scope. Thus, we can't access them outside the Block.

What is **Shadowing** —> occurs when a variable declared in a certain scope (e.g. a local variable) has the same name as a variable in an outer scope (e.g. a global variable)

```
var a = 100;  
{  
    var a = 10; // same name as global var let b = 20;  
    const c = 30;  
    console.log(a); // 10  
    console.log(b); // 20  
    console.log(c); // 30  
}  
console.log(a); // 10, instead of the 100 we were expecting. So block "a"  
modified val of global "a" as well. In console, only b and c are in block  
space. a initially is in global space(a = 100), and when a = 10 line is run, a  
is not created in block space, but replaces 100 with 10 in global space  
itself.
```

In Simpler words, If one variable has the same name outside the block, the variable inside the block shadows the outside variable.

In above example, first the value of variable a is “100” and in block scope it’s value is “10” but when we prints its value it will display “10” because the value of a get over-shadowed from “100” to “10” But this happens only when a variable is declared with var keyword.

Let's observe the behavior in case of let and const and understand it's reason

```
let b = 100;
{
  var a = 10; let b = 20;
  const c = 30;
  console.log(b); // 20
}
console.log(b);
// 100, Both b's are in separate spaces (one in Block(20) and one in
// Script(another arbitrary mem space)(100)). Same is also true for
*const* declarations.
```

And same applies for the function

```
const c = 100;
function x()
{
  const c = 10;
  console.log(c); // 10
}
x();
console.log(c); // 100
```

What is Illegal Shadowing ==> If we create a variable in a global scope with the **let** keyword and another variable with the **var** keyword in a **block** scope but with the exact **same** name, it will throw an error.

```
let a = 20;
{
  var a = 20;
}
// Uncaught SyntaxError: Identifier 'a' has already been declared
```

– Episode-10(Season-1 -> Closures in JS 🔥)

Closures —> A closure is the combination of a function **bundled** together (enclosed) with **references** to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

JavaScript has a **lexical** scope environment. If a function needs to access a variable, it first goes to its local memory. When it does not find it there, it goes to the memory of its **lexical parent**. In Below code snippet, Over here function “y” along with its lexical scope i.e. (function x) would be called a **closure**.

```
function x()
{
  var a = 7;
  function y() {
    console.log(a);
  }
  return y;
}
var z = x();
console.log(z); // value of z is entire code of function y.
```

In the above code snippet, When “y” is returned, not only the function returned but the entire closure (function y + its lexical scope) is returned and put inside variable “z”. So when z is used anywhere else in the program, it still points to the value of variable “a” inside x().

Let's take another example

```
function z() {  
    var b = 900;  
    function x() {  
        var a = 10;  
        function y() {  
            console.log(a, b);  
        }  
        y();  
    }  
    x();  
}
```

```
z();
```

In the above code snippet, we have put the entire function “x” inside the new function “z” ; this is also closure.

The output that we will get after running this code will be,

```
f y() {  
    console.log(a);  
}
```

```
10 900
```

You might be wondering how we're able to get the value of variable "b" inside function "y" which is enclosed inside the function "x". This is all due to closure because **closure** is a function that has **access to its outer function** scope even after the function has **returned**. Meaning, A closure can remember and access variables and arguments referenced by its outer function even after the function has returned.*

```
function z() {
  var b = 900;
  function x(){
    var a= 7;
    function y(){
      console.log(a,b);
    }
    y();
  }
  x();
}
z();
```

{} Line 6, Column 7 Coverage: n/a

Scope Watch

Paused on breakpoint

Call Stack

- Y index.js:6
- X index.js:8
- Z index.js:10
- (anonymous) index.js:12

Local

this: Window

Closure (x)

a: 7

Closure (z)

b: 900

Global

PERSISTENT: 1

If you see properly in scope tab there are two different scope defined ("x" & "z") that's the reason we are able to get value of a and b in function "y".

Advantages of Closure:

- Module Design Pattern
- Currying
- Memoize
- Data hiding and encapsulation
- setTimeouts etc.
- Iterators.

Disadvantages of Closure:

- Over consumption of memory
- Memory Leak

– Episode-11(Season-1 -> setTimeout + Closures Interview Question 🔥)

setTimeout function sets a timer which executes a function or specified piece of code once the timer expires.

```
function x() {  
    var i = 1;  
    setTimeout(function () {  
        console.log(i);  
    }, 3000);  
    console.log("Namaste Javascript");  
}  
x();
```

- We expect JS to wait 3 sec, print 1 and then go down and print the string. But JS prints
- string immediately, wait 3 sec and then print 1.
- The function inside setTimeout forms a closure (remembers reference to i). So wherever function goes it carries this reference along with it.
- setTimeout takes this callback function & attaches a timer of 3000ms and stores it. Goes to the next line without waiting and prints the string.
- After 3000ms runs out, JS takes a function, puts it into call stack and runs it.

Q: Print “1” after 1 sec,” 2” after 2 sec till 5 : Tricky interview question

We assume many developers will use this simple approach as below snippet:

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

Output will be:

6
6
6
6
6

- We get above output because of closures. When setTimeout stores the function somewhere and attaches a timer to it, the function remembers its reference to i, not value of i.
- All 5 copies of the function point to the same reference of i. JS stores these 5 functions, prints string and then comes back to the functions. By then the timer had run fully.
- And due to looping, the i value became 6. And when the callback fun runs the variable i = 6. So same 6 is printed in each log
- To avoid this, we can use **let** instead of **var** as let has Block scope. For each iteration, the i is a new variable altogether (new copy of i). Everytime setTimeout is run, the inside function forms closure with a new variable i.

But what if we don't have option to use "**var**" , we can this snippet of code

```
function x() {  
    for (var i = 1; i <= 5; i++) {  
        function close(i) {  
            setTimeout(function () {  
                console.log(i);  
            }, i * 1000);  
            // put the setT function inside new function close()  
        }  
        close(i); // everytime you call close(i) it creates new copy of i. Only this  
        // time, it is with var itself!  
    }  
    console.log("Namaste Javascript");  
}  
x();
```

Output will be:

Namaste Javascript

1
2
3
4
5

- When we're using "var" instead of "let" All 5 copies of the function were pointing to the same reference of "i" , But when we use "let" it points to 5 different reference of "i",
- And due to looping, the "i" value keeps getting incremented.

– Episode-11(Season-1 -> CRAZY JS INTERVIEW 🎉 ft. Closures)

Q1. What is Closure in Javascript?

Ans-> A function along with reference to its outer environment together forms a closure. Or in other words, A Closure is a combination of a function and its lexical scope bundled together.

Example:

```
function outer() {  
    var outervariable = 10;  
    function inner() {  
        console.log("outervariable value-> "+outervariable);  
    }    // inner forms a closure with outer return inner;  
    return inner;  
}  
💡  
outer(); /* first we're calling outer function and then inner function */
```

Q2: Will the below code snippet still form a closure?

```
function outer() {  
    function inner() {  
        console.log("outervariable value-> "+outervariable);  
    }  
    var outervariable = 10;  
    return inner;  
}  
  
outer();
```

Ans-> Yes, because inner function forms a closure with its outer environment so sequence of the code doesn't matter.

Q3: Changing var to let, will it make any difference?

```

function outer() {
  let outervariable = 10;
  function inner() {
    console.log("outervariable value-> "+outervariable);
  } // inner forms a closure with outer return inner;
  return inner;
}

outer(); /* first we're calling outer function and then inner function */

```

Q4: Can we pass the parameter to the outer function?

```

function outer(outerparam) {
  let outervariable = 10;
  function inner() {
    console.log("outervariable value-> "+outervariable+ " params passed =>" +outerparam)
  } // inner forms a closure with outer return inner;
  return inner;
}

outer("Cricket"); /* first we're calling outer function and then inner function */

```

Ans→ Yes, we can pass the parameter to a function when it's closure.

Q5. Can we nest an outer function inside another new function and If yes will it still form closure?

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

```
var close = (outest())("hellow");  
close();
```

Ans-> Yes we can nest a new function to the outer function and it'll still form closure.

Q6. What if adding the same variable in the global scope will it work.

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

```
let a = 999;  
var close = (outest())("hellow");  
close();
```

Ans-> Yes, it will work as closures form reference to its outer environment, but if we don't define the variable in function "outer" it will take value of "a" from global scope i.e 999 (it searches more deeper hierarchy level and gets global value)

Q7: Advantages of Closure?

Ans->

- Module Design Pattern
- Currying
- Memoize
- Data hiding and encapsulation
- setTimeouts etc.

Q8. Discuss more on Data hiding and encapsulation?

```
function counter() {  
    var count = 0;  
    return function increment() {  
        count++;  
        console.log(count);  
    };  
}  
var counter1 = counter(); //counter function has closure with count var.  
counter1(); // increments counter  
var counter2 = counter();  
counter2(); // here counter2 is whole new copy of counter function and it wont impact the output of counter1  
// -----  
  
// without closures  
var count = 0;  
function increment() {  
    count++;  
}  
// in the above code, anyone can access count and change it.  
// -----  
// (with closures) -> put everything into a function  
function counter() {  
    var count = 0;  
    function increment() {  
        count++;  
    }  
}  
console.log(count); // this will give referenceError as count can't be accessed. So now we are able to achieve  
hiding of data  
// -----  
//(increment with function using closure) true function
```

```

// Above code is not good and scalable for say, when you plan to implement decrement counter at a later stage.
// To address this issue, we use *constructors*
// -----
// Adding decrement counter and refactoring code:
function Counter() {
    //constructor function. Good coding would be to capitalize first letter of constructor function.

    var count = 0;
    this.incrementCounter = function () {
        //anonymous function
        count++;
        console.log(count);
    };
    this.decrementCounter = function () {
        count--;
        console.log(count);
    };
}
var counter1 = new Counter(); // new keyword for constructor function
counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();

```

Q9: Disadvantages of closure ?

Ans-> Overconsumption of memory when using closure as every time as those closed over variables are not garbage collected till program expires. So when creating many closures, more memory is accumulated and this can create memory leaks if not handled.

Garbage collector : Program in JS engine or browser that frees up unused memory. In high level languages like C++ or JAVA, garbage collection is left to the programmer, but in JS engine it's done implicitly. Below is the example of Garbage collector

```

function a() {
    var x = 0;
    return function b() {
        console.log(x);
    };
}
var y = a(); // y is a copy of b() y();
// Once a() is called, its variable "x" should be garbage collected ideally.
// But function "b" has closure over variable x. So memory of "x" cannot be freed.
// Like this if more closures are formed, it becomes an issue. To tackle this issue,
// JS engines like v8 and Chrome have smart garbage
// collection mechanisms. Say we have var x = 0, z = 10 in code.
// When we access "x" its value is printed as 0 but z is removed automatically because it hasn't been used.

```

- Episode-13(Season-1 -> FIRST CLASS FUNCTIONS 🔥, Anonymous Functions)

Q: What is a Function statement (aka Function Declaration) ?
A function statement is a way to define a named function using the function keyword.

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

Q: What is Function Expression ?

A function expression is a way to define a function as an expression within an assignment or another expression. Unlike function declarations (which use the function keyword), function expressions do not hoist the function to the top of their containing scope, and they can be defined inline as part of an expression.

```
var b = function () {  
    console.log("Hello");  
};  
a();
```

Q: Difference between function statement and expression

The major difference between these two:

Function Statement:

- Hoisted to the top of their containing scope.
- Can be called before its declaration in the code.

Function Expression:

- Not hoisted.
- Must be defined before it can be called in the code.

```
a(); // "Hello A"
b(); // TypeError
function a() {
  console.log("Hello A");
}
var b = function () {
  console.log("Hello B");
};
💡 Why? During memory creation phase a is created in memory and function is assigned to a.
// But b is created like a variable (b:undefined) and until code reaches the function () part, it is still undefined. So it cannot find the value for "b" that's the reason it results in error.
```

Q: What is Anonymous Function?

An anonymous function is a function that is defined without a name. Instead of having a name identifier like traditional named functions.

```
// function () {
//
// }
```

More About Anonymous functions:-

- They don't have their own identity. So an anonymous function without code inside it results in an error.
- Anonymous functions are often created as function expressions. These can be assigned to variables or passed as arguments to other functions.
- Anonymous functions are commonly used as callback functions.

Q: What is Named Function Expression?

A Named Function Expression in JavaScript is a type of function expression that has a name identifier within its definition. Unlike traditional function declarations, where the function name is automatically bound to the function.

```
const functionName = function(parameters) {  
    // Function body  
    // Code to be executed when the function is called  
};
```

Q: Parameters vs Arguments ?

- Parameters are the placeholders or variables listed in the function declaration. They represent the values that a function **expects to receive** when it's called.
- Arguments are the actual values or expressions that are passed to a function when it's called. They correspond to the parameters defined in the function declaration. Arguments are **provided when invoking** the function.

```
var b = function (param1, param2) // labels/identifiers are parameters
{
  console.log("b called");
}
💡
b(arg1, arg2); // arguments - values passed inside function call
```

Q: What is First Class Function OR First Class Citizens?

When a function can pass inside another function as arguments and / or return a function(HOF). These ability are altogether known as First class function.

```
var b = function (param1) {
  console.log(param1); // prints " f() {} "
};
b(function () {});
```

// Other way of doing the same thing:

```
var b = function (param1) {
  console.log(param1);
};
function xyz() {}
b(xyz); // same thing but with different code.
```

// we can return a function from a function:

```
var b = function (param1) {
  return function () {};
};
console.log(b()); //we log the entire fun within b
```

– Episode-1(Season-1-> **Callback Functions , ft. Event Listeners**
🔥)

Callback Functions

Functions are first class citizens ie. create a function “A” and pass it to another function “B”. Here, function “A” is a callback function. So basically we’re giving access to a function “B” to call function A. This callback function gives us access to the whole **Asynchronous** world in **Synchronous** world.

```
setTimeout(function () {  
  console.log("Timer");  
}, 1000); // first argument is callback function and second is timer.
```

JS is a synchronous and single threaded language. But due to callbacks, we can do asynchronous things in JS.

```
setTimeout(function () {
|💡 console.log("timer");
}, 5000);
function x(y) {
  console.log("x");
  y();
}
x(function y() {
  console.log("y");
});
// x y timer
```

- In the call stack, first x and y are present. After code execution, they go away and the stack is empty. Then after 5 seconds (from beginning) anonymous suddenly appear up in stack ie. setTimeout
- All 3 functions are executed through the call stack. If any operation blocks the call stack, it's called blocking the main thread.
- Say if x() takes 30 sec to run, then JS has to wait for it to finish as it has only 1 call stack/1 main thread. Never block the main thread.
- Always use async for functions that take time eg. setTimeout

Event Listeners

We will create a button in html and attach event to it

```
// index.html
<button id="clickMe">Click Me!</button>

// in index.js
document.getElementById("clickMe").addEventListener("click", function xyz() {
  //when event click occurs, this callback function (xyz) is called into callstack
  console.log("Button clicked");
});
```

Let's implement an increment counter button.

- Using global variable (not good as anyone can change it) :

```
let count = 0;
document.getElementById("clickMe").addEventListener("click", function xyz() {
  console.log("Button clicked", ++count);
});
```

Use closures for data abstraction

```
function attachEventList() {
  //creating new function for closure
  let count = 0;
  document.getElementById("clickMe").addEventListener("click", function xyz() {
    console.log("Button clicked", ++count); //now callback function forms closure with outer scope(count)
  });
}

attachEventList();
```

Garbage Collection and removeEventListeners

Event listeners are heavy as they form closures. So even when the call stack is empty, EventListener won't free up memory allocated to count as it doesn't know when it may need to count again. Some remove event listeners when we don't need them (garbage collected) onClick, onHover, onScroll all in a page can slow it down heavily.

The screenshot shows a browser developer tools debugger interface. On the left is the code editor for 'index.js' with line numbers 292 to 307. Line 303 is highlighted with a blue arrow and contains the code: `console.log("Button clicked", ++count);`. On the right is the debugger sidebar. At the top, it says 'Paused on breakpoint'. Below that are sections for 'Watch', 'Breakpoints', and 'Scope'. Under 'Scope', there's a 'Local' section with 'this: button#clickMe' and a 'Closure (attachEventList)' entry containing 'count: 0'. There are also 'Global' and 'Call Stack' sections. At the bottom right of the sidebar, it says 'Window' and 'index.js:303'. A small 'xyz' icon is also visible.

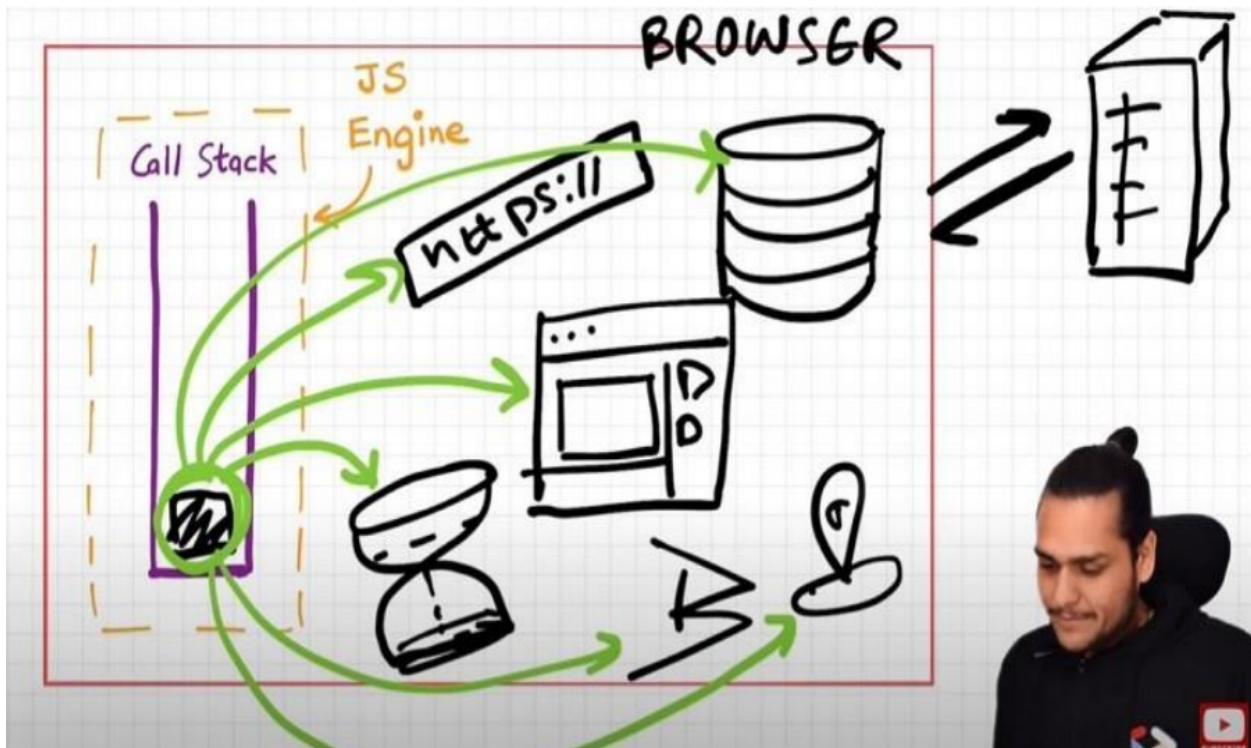
- Episode-5(Season-1 -> Asynchronous JavaScript & EVENT LOOP from scratch 🔥)

Note: Call stack will execute any execution context which enters it. **Time, tide and JS waits for none.** TLDR; Callstack has no timer.

Browser has JS Engine which has Call Stack which has Global execution context, local execution context etc.

- But browser has many other superpowers - Local storage space, Timer, place to enter URL, Bluetooth access, Geolocation access and so on.

- Now JS needs some way to connect the callstack with all these superpowers. This is done using Web APIs.



Web APIs

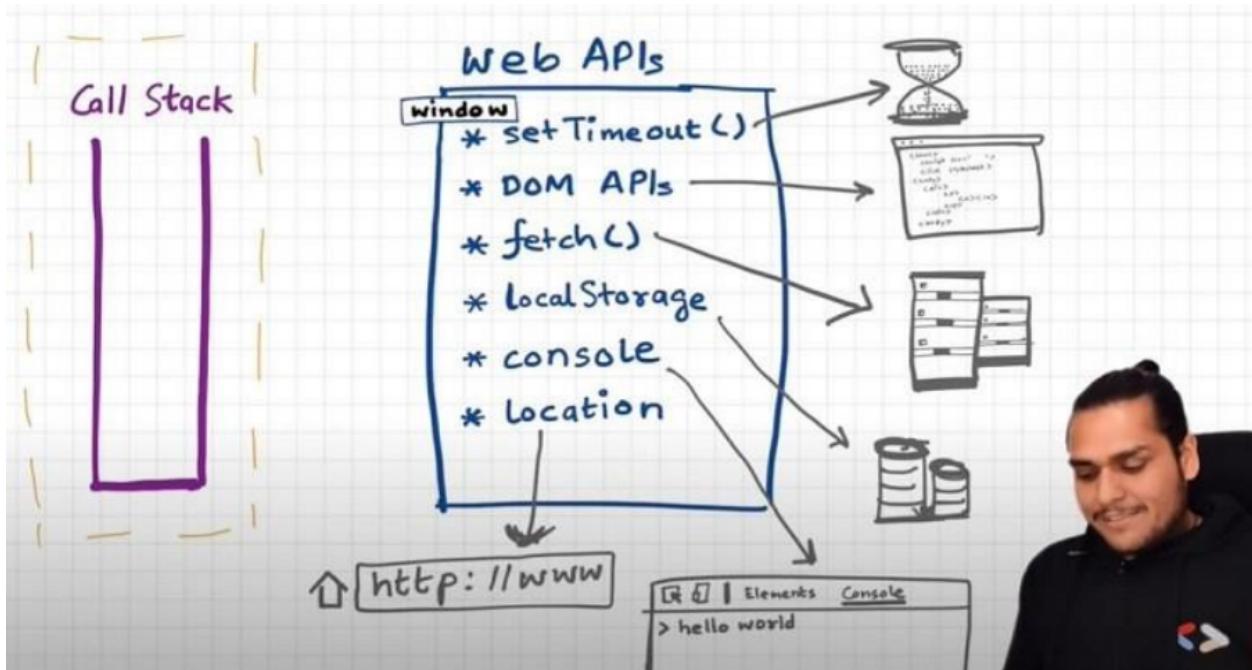
None of the below are part of Javascript! These are extra superpowers that browsers have.

Browser gives access to JS callstack to use these powers

`setTimeout()`, DOM APIs, `fetch()`, `localStorage`, `console` (yes, even `console.log` is not JS!!), location and so many more.

- `setTimeout()` : Timer function
- DOM APIs : eg.`Document.xxxx` ; Used to access HTML DOM tree. (Document Object)
- Manipulation) `fetch()` : Used to make connection with external servers eg. Netflix servers etc.

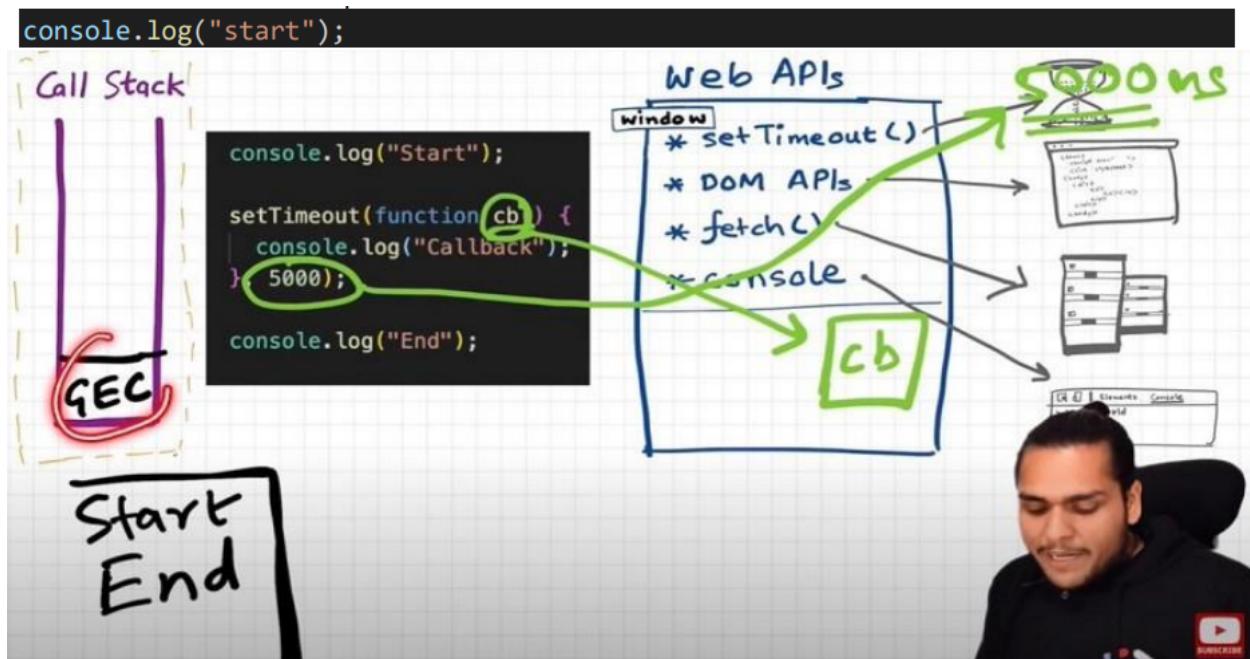
We get all these inside call stack through global object ie. window - Use window keywords like : window.setTimeout(), window.localStorage, window.console.log() to log something inside the console. - As window is global obj, and all the above functions are present in global object, we don't explicitly write window but it is implied.



Let's understand the below code image and its explanation:

- First a GEC is created and put inside a call stack.
- `console.log("Start");` // this calls the console web api (through window) which in turn actually modifies values in the console.
- `setTimeout(function cb() { //this calls the setTimeout web api which gives access to the timer feature. It stores the callback cb() and starts the timer. console.log("Callback");}, 5000);`
- `console.log("End");` // calls console api and logs in console window. After this GEC pops from call stack.

- While all this is happening, the timer is constantly ticking. After it becomes 0, the callback cb() has to run.
- Now we need this cb to go into the call stack. Only then will it be executed. For this we need event loop and Callback queue

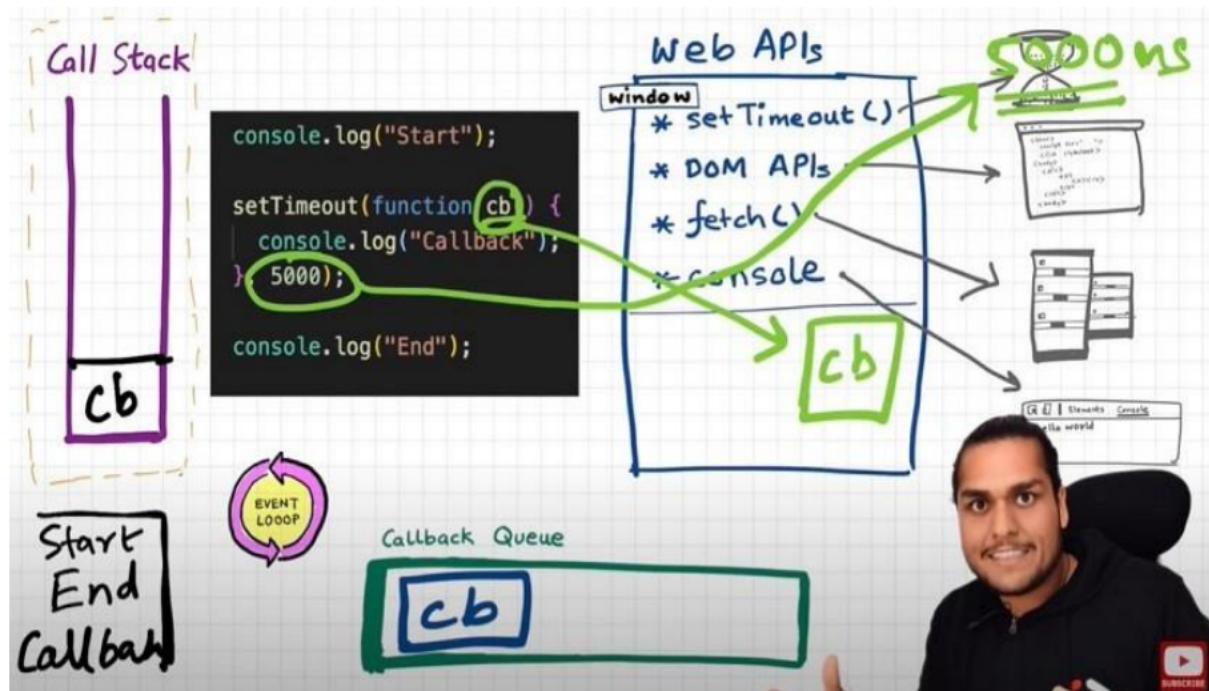


Event Loops and Callback Queue

Q: How after 5 secs timer is the console ?

- `cb()` cannot simply directly go to callstack to be executed. It goes through the callbackqueue when the timer expires.
- Event loop keeps checking the callback queue, and sees if it has any element to put into the call stack. It is like a gatekeeper.
- Once `cb()` is in the callback queue, eventloop pushes it to the call stack to run. Console API is used and log printed

used and try printed



Q: Need a callback queue?

Ans: Suppose the user clicks the button “n” number of times. So “n” cb() are put inside the callback queue. Event loop sees if the call stack is empty/has space and whether the callback queue is not empty (“n” number of elements here).

Elements of callback queue popped off, put in call stack, executed and then popped off from call stack.

Behavior of fetch (Microtask Queue?)

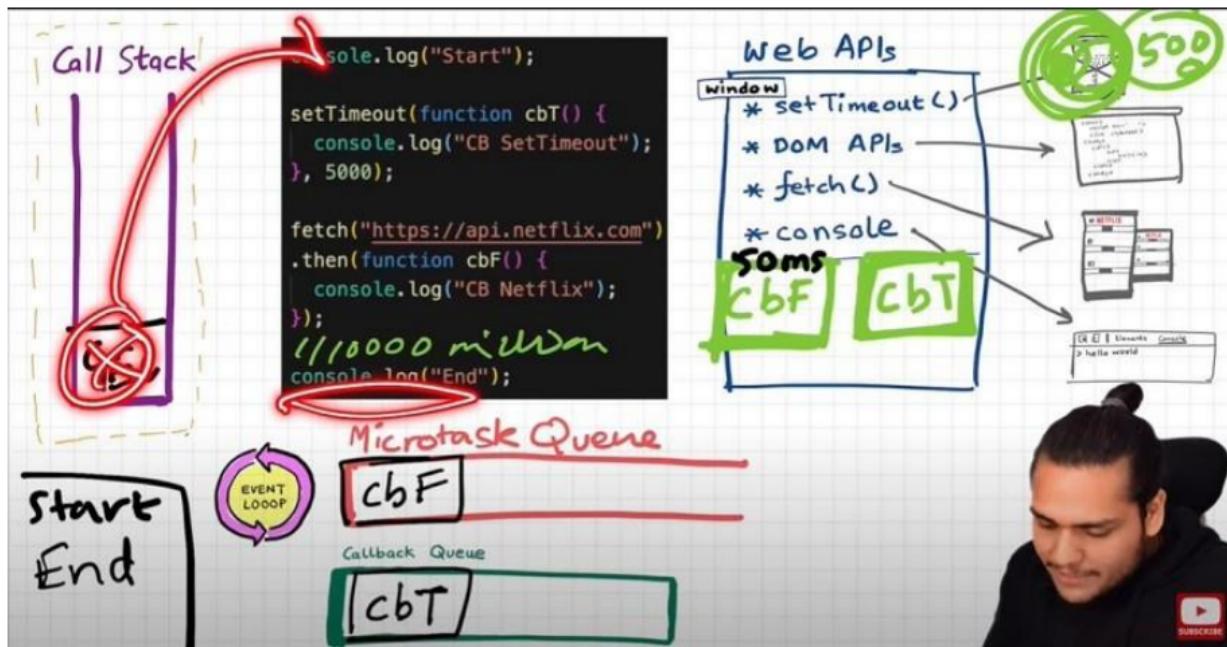
Let's observe the code below and try to understand

```
console.log("Start"); // this calls the console web api (through window) which  
in turn actually modifies values in console.  
setTimeout(function cbT() {  
    console.log("CB Timeout");  
}, 5000);  
fetch("https://api.netflix.com").then(function cbF() { console.log("CB  
Netflix");  
}); // take 2 seconds to bring response  
// millions lines of code console.log("End");
```

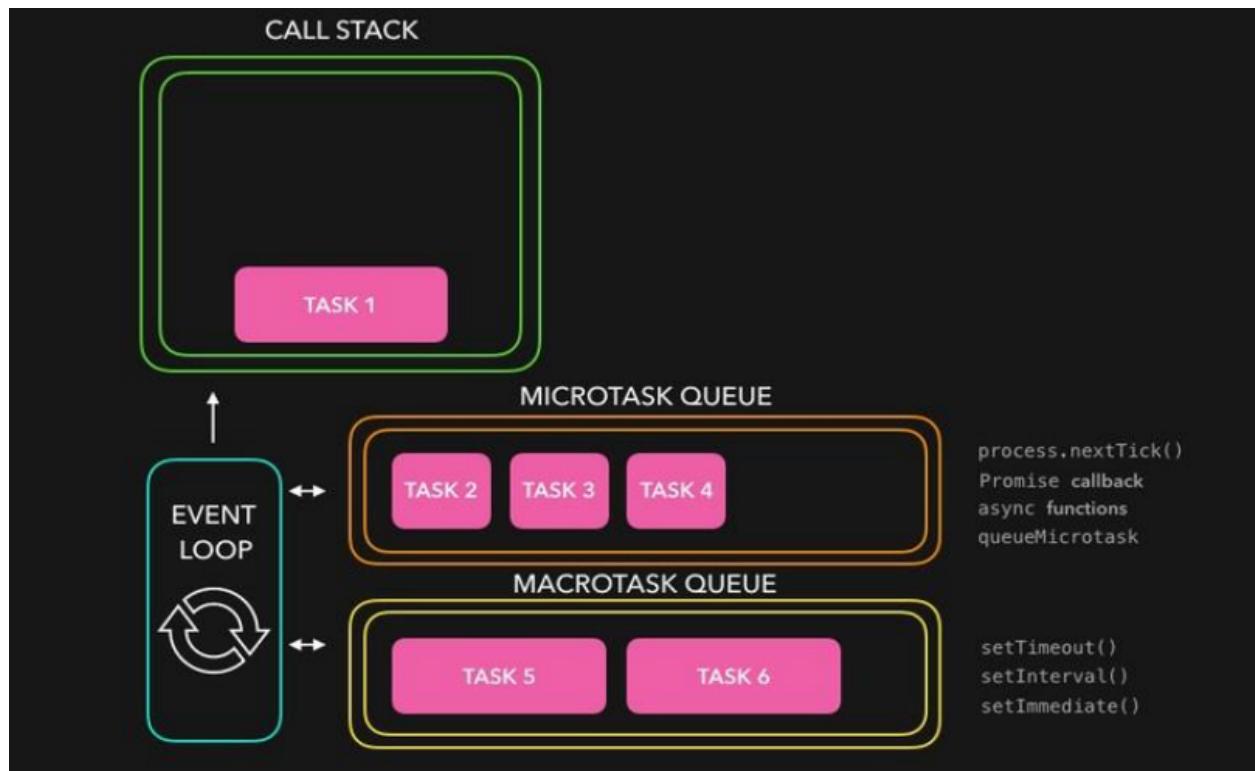
Code Explanation:

- Same steps for everything before fetch() in above code.
- fetch registers cbF into web api environment along with existing cbT.
- cbT is waiting for 5000ms to end so that it can be put inside the callback queue.
- cbF is waiting for data to be returned from Netflix servers, which will take 2 seconds.
- After this millions of lines of code is running, by the time millions lines of code will execute, 5 seconds has finished and now the timer has expired and response from Netflix server is ready.
- Data back from cbF ready to be executed gets stored into something called a Microtask Queue.
- Also after expiration of timer, cbT is ready to execute in Callback Queue.
- Microtask Queue is exactly the same as Callback Queue, but it has higher priority. Functions in
- Microtask Queues are executed earlier than Callback Queues.
- In the console, the first Start and End are printed in the console. First cbF goes in call stack and "CB Netflix"
- is printed. cbF popped from callstack. Next cbT is removed from callback Queue, put in Call Stack, "CB Timeout " is printed, and cbT removed from callstack.

Have a look at image below, for better understanding



Microtask Priority Visualization



What enters the Microtask Queue ?

- All the callback functions that come through promises go in microtask Queue.
- **Mutation Observer** : Keeps on checking whether there is mutation in the DOM tree or not, and if there, then it executes some callback function.
- Callback functions that come through promises and mutation observer go inside Microtask Queue.
- All the rest goes inside Callback Queue aka. Task Queue.
- If the task in the microtask Queue keeps creating new tasks in the queue, the element in the callback queue never gets a chance to be run. This is called **starvation**.

Some Important Questions

1. When does the event loop actually start ?

- Event loop, as the name suggests, is a single-thread loop that is almost infinite. It's always running and doing its job.

2. Are only asynchronous web api callbacks registered in the web api environment?

- YES, the synchronous callback functions like what we pass inside map, filter and reduce aren't registered in the Web API environment. It's just those async callback functions which go through all this.

3. How does it matter if the delay for setTimeout would be 0ms. Then the callback will move to the queue without any wait ?

- No, there are trust issues with setTimeout() . The callback function needs to wait until the Call Stack is empty. So the 0 ms callback might have to wait for 100ms also if the stack is busy.

4.Does the web API environment stores only the callback function and pushes the same callback to the queue/microtask queue?

- Yes, the callback functions are stored, and a reference is scheduled in the queues. Moreover, in the case of event listeners(for example click handlers), the original callbacks stay in the web API environment, that's why it's advised to explicitly remove the listeners when not in use so that the garbage collector does its job.

– Episode-16(Season-1 -> Trust issues with setTimeout())

setTimeout with timer of 5 secs sometimes does not exactly guarantees that the callback function will execute exactly after 5s.

Let's understand more with code and its explanation:

```
console.log("Start");
setTimeout(function cb() {
  console.log("Callback");
}, 5000);
console.log("End");
// Millions of lines of code to execute
// o/p: Over here setTimeout exactly doesn't guarantee that the callback
function will be called exactly after 5s. Maybe 6,7 or even 10! It all depends
on callstack. Why?
```

Reasons?

- First GEC is created and pushed in callstack.
- Start is printed in console
- When setTimeout is seen, the callback function is registered into webapi's env. And timer is attached to it and started. callback waits for its turn to be executed once timer expires. But JS waits for none. Next line.

- End is printed in the console.
- After "End", we have 1 million lines of code that takes 10 sec(say) to finish execution. So
- GEC won't pop out of stack. It runs all the code for 10 sec.
- But in the background, the timer runs for 5s. While callstack runs the 1M line of code,
- this timer has already expired and callback fun has been pushed to Callback queue and waiting to pushed to callstack to get executed.
- Event loop keeps checking if callstack is empty or not. But here GEC is still in stack so cb
- can't be popped from the callback Queue and pushed to CallStack. Though setTimeout is
- only for 5s, it waits for 10s until callstack is empty before it can execute (When GEC popped after 10 sec, callstack() is pushed into call stack and immediately executed (Whatever is pushed to callstack is executed instantly)).
- This is called the Concurrency model of JS. This is the logic behind setTimeout's trust issues.

The First rule of JavaScript: Do not block the main thread (as JS is a single threaded(only 1 callstack) language). In the below example, we are blocking the main thread. Observe Questions and Output.

setTimeout guarantees that it will take at least the given timer to execute the code.

JS is a **synchronous** single threaded language. With just 1 thread it runs all pieces of code. It becomes kind of an interpreter language, and runs code very fast inside browser (no need to wait for code to be compiled) (JIT - Just in time compilation). And there are still ways to do async operations as well.

The screenshot shows a browser window with the URL `127.0.0.1:5500`. The page content says "Namaste 🌟 JavaScript". On the left, the developer tools' Elements tab is active, showing the "Console" tab selected. The console log output is as follows:

```
Start          index.js:1
End            index.js:7
While expires index.js:17
Callback       index.js:4
```

On the right, the code editor shows `index.js` with the following content:

```
js > JS index.js    X  ↗ index.html
js > JS index.js
1   console.log("Start");
2
3   setTimeout(function cb() {
4     console.log("Callback");
5   }, 5000);
6
7   console.log("End");
8
9   // million
10
11  let startDate = new Date().getTime();
12  let endDate = startDate;
13  while (endDate < startDate + 10000) {
14    endDate = new Date().getTime();
15  }
16
17  console.log(["While expires"]);
18
```

What if timeout = 0 sec?

```
console.log("Start");
setTimeout(function cb() {
  console.log("Callback");
}, 0);
console.log("End");

// Even though timer = 0s, the cb() has to go through the queue. Registers
callback in webapi's env , moves to callback queue, and execute once callstack
is empty.
// O/p - Start End Callback
// This method of putting timer = 0, can be used to defer a less imp function
by a little so the more important function(here printing "End") can take place
```

- Episode-18(Season-1 -> Higher-Order Functions ft. Functional Programming)

Q: What is a Higher Order Function?

Ans: Higher-order functions are regular functions that take other functions as arguments or return functions as their result.

Eg:

```
function x() {  
  console.log("Hi");  
};  
function y(x) { x();  
};  
y(); // Hi  
// y is a higher order function  
// x is a callback function
```

Let's try to understand how we should approach a solution in an interview. I have an array of radius and I have to calculate the area using these radius and store in an array.

First Approach

```
const radius = [1, 2, 3, 4];

const calculateArea =
  function (radius) {
    const output = [];

    for (let i = 0; i < radius.length; i++) {
      output.push(Math.PI * radius[i] * radius[i]);
    }

    return output;
};

console.log(calculateArea(radius));
```

The above solution works perfectly fine but what if we have now requirement to calculate an array of circumferences. Code now be like :

```
const radius = [1, 2, 3, 4];

const calculateCircumference =
  function (radius) {const
    output = [];

    for (let i = 0; i <
      radius.length; i++)
      {output.push(2 *
        Math.PI *
        radius[i]);

    }
    return output;
};

console.log(calculateCircumference(radius));
```

But over here we are violating some principles like DRY Principle, now let's observe the better approach.

```

const radiusArr = [1, 2, 3, 4];
// logic to calculate area
const area = function (radius)
{
  return Math.PI * radius * radius;
}

// logic to calculate circumference
const circumference = function (radius) { return 2 * Math.PI * radius; }
const calculate = function(radiusArr, operation) { const output = [];

for (let i = 0; i < radiusArr.length; i++) {
  output.push(operation(radiusArr[i]));
}
return output;
}
console.log(calculate(radiusArr, area));
console.log(calculate(radiusArr, circumference));

```

- Over here calculate is HOF
- We have extracted logic into separate functions. This is the beauty of functional programming.
- Polyfill of map Over here calculate is nothing but polyfill of map function
`console.log(radiusArr.map(area)) == console.log(calculate(radiusArr, area));`
- Let's convert the above calculate function as a map function and try to use it.

```

Array.prototype.calculate = function(operation) { const output = [];
for (let i = 0; i < this.length; i++) { output.push(operation(this[i]));
}
return output;
}
console.log(radiusArr.calculate(area))

```

– Episode-19(Season-1 -> map, filter & reduce)

Map function :

It is basically used to transform an array. The `map()` method creates a new array with the results of calling a function for every array element.

`const output = arr.map(function) // this function tells map that what transformation I want on each element of array`

```
const arr = [5, 1, 3, 2, 6];

// Task 1: Double the array
element: [10, 2, 6, 4, 12]function
double(x) {

    return x * 2;

}

const doubleArr = arr.map(double); // Internally map will run
double function for each element of array and create a new array
and returns it.

console.log(doubleArr); // [10, 2, 6, 4, 12]
```

So basically the map function maps each and every value in array and transform it based on given condition.

Filter function : Filter function is basically used to filter the value inside an array. The `arr.filter()` method is used to create a new array from a given array consisting of only those elements from the given array which satisfy the given condition.

```
const array = [5, 1, 3, 2, 6];
// filter odd values
function isOdd(x) {
  return x % 2;
}
const oddArr = array.filter(isOdd); // [5,1,3]

// Other way of writing the above:
const oddArr = arr.filter((x) => x % 2);
```

Filter function creates an array and store only those values which evaluate to true.

Reduce function :

It is used to reduce the array to a **single** value and executes a provided function for each value of the array (from left to right) and the return value of the function is stored in an accumulator.

```
const array = [5, 1, 3, 2, 6];
// Calculate sum of elements of array - Non functional programming way

function findSum(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
```

```

        sum = sum + arr[i];
    }
    return sum;
}
console.log(findSum(array)); // 17

// reduce function way
const sumOfElem = arr.reduce(function (accumulator, current) {
    // current represent the value of array
    // accumulator is used the result from element of array.
    // In comparison to previous code snippet, *sum* variable is *accumulator*
    // and *arr[i]* is *current*
    accumulator = accumulator + current;
    return accumulator;
}, 0); //In above example sum was initialized with 0, so over here accumulator
also needs to be initialized,
// so the second argument to reduce function represent the initialization
value.
console.log(sumOfElem); // 17

```

Function Chaining

```

const users = [
  { firstName: "Raj", lastName: "Pujari", age: 23 },
  { firstName: "Abhi", lastName: "Kumar", age: 27 },
  { firstName: "Ankit", lastName: "Pandey", age: 25 },
  { firstName: "Ravi", lastName: "Ashwin", age: 50 },
];

// function chaining
const output = users
  .filter((user) => user.age < 30)
  .map((user) => user.firstName);
console.log(output); // ["Raj", "Abhi", "Ankit"]
// Homework challenge: Implement the same logic using reduce
const outputs = users.reduce((acc, curr) => {
  if (curr.age < 30) {
    acc.push(curr.firstName);
  }
  return acc;
}, []);
console.log(outputs); // ["Raj", "Abhi", "Ankit"]

```