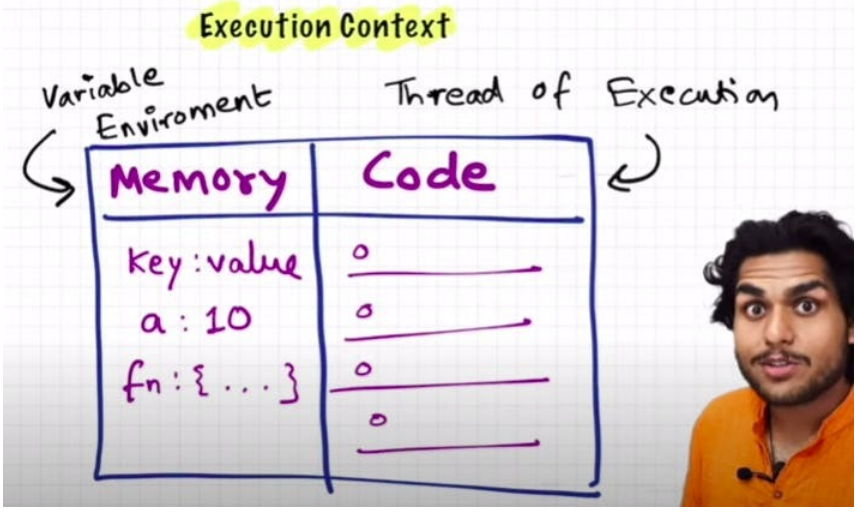
Episode 1 : Execution Context

- Everything in JS happens inside the execution context. Imagine a sealed-off container inside which JS runs. Imagine a sealed-off container inside which JS runs. It is an abstract concept that hold info about the env. within the current code is being executed.



- In the container the first component is **memory component** and the 2nd one is **code component**
- Memory component has all the variables and functions in key value pairs. It is also called **Variable environment**.
- Code component is the place where code is executed one line at a time. It is also called the **Thread of Execution**.
- JS is a **synchronous**, **single-threaded** language
  - Synchronous:- One command at a time.
  - Single-threaded:- In a specific synchronous order.

Watch Live On Youtube below:

Episode 2 : How JS is executed & Call Stack

- When a JS program is ran, a **global execution context** is created.
- The execution context is created in two phases.
  - Memory creation phase - JS will allocate memory to variables and functions.
  - Code execution phase

- Let's consider the below example and its code execution steps:

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

The very **first** thing which JS does is **memory creation phase**, so it goes to line one of above code snippet, and **allocates a memory space** for variable **'n'** and then goes to line two, and **allocates a memory space** for **function 'square'**. When allocating memory **for n it stores 'undefined'**, a special value for 'n'. **For 'square', it stores the whole code of the function inside its memory space.** Then, as square2 and square4 are variables as well, it allocates memory and stores 'undefined' for them, and this is the end of first phase i.e. memory creation phase.

So O/P will look something like

Now, in **2nd phase** i.e. code execution phase, it starts going through the whole code line by line. As it encounters var n = 2, it assigns 2 to 'n'. Until now, the value of 'n' was undefined. For function, there is nothing to execute. As these lines were already dealt with in memory creation phase.

Coming to line 6 i.e. **var square2 = square(n)**, here **functions are a bit different than any other language. A new execution context is created altogether.** Again in this new execution context, in memory creation phase, we allocate memory to num and ans the two variables. And undefined is placed in them. Now, in code execution phase of this execution context, first 2 is assigned to num. Then var ans = num * num will store 4 in ans. After that, return ans returns the control of program back to where this function was invoked from.

When **return** keyword is encountered, It returns the control to the called line and also **the function execution context is deleted**. Same thing will be repeated for square4 and then after that is finished, the global execution context will be destroyed. So the **final diagram** before deletion would look something like:

- Javascript manages code execution context creation and deletion with the the help of **Call Stack**.
- Call Stack is a mechanism to keep track of its place in script that calls multiple function.
- 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.

Watch Live On Youtube below:

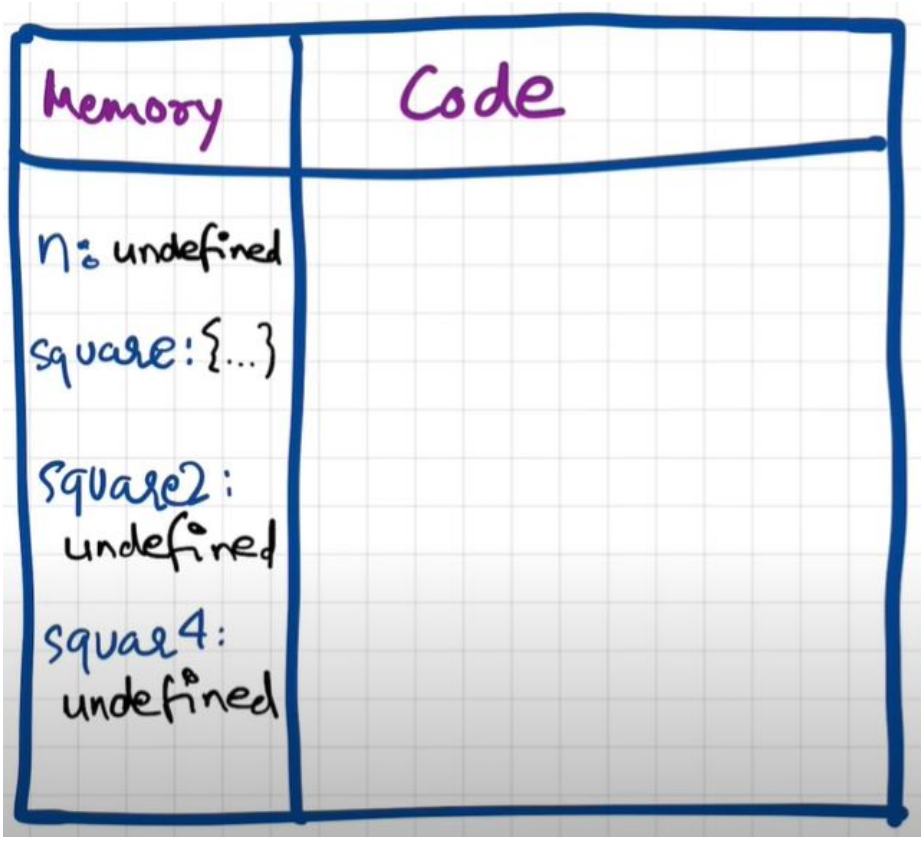


Figure 1: Execution Context Phase 1

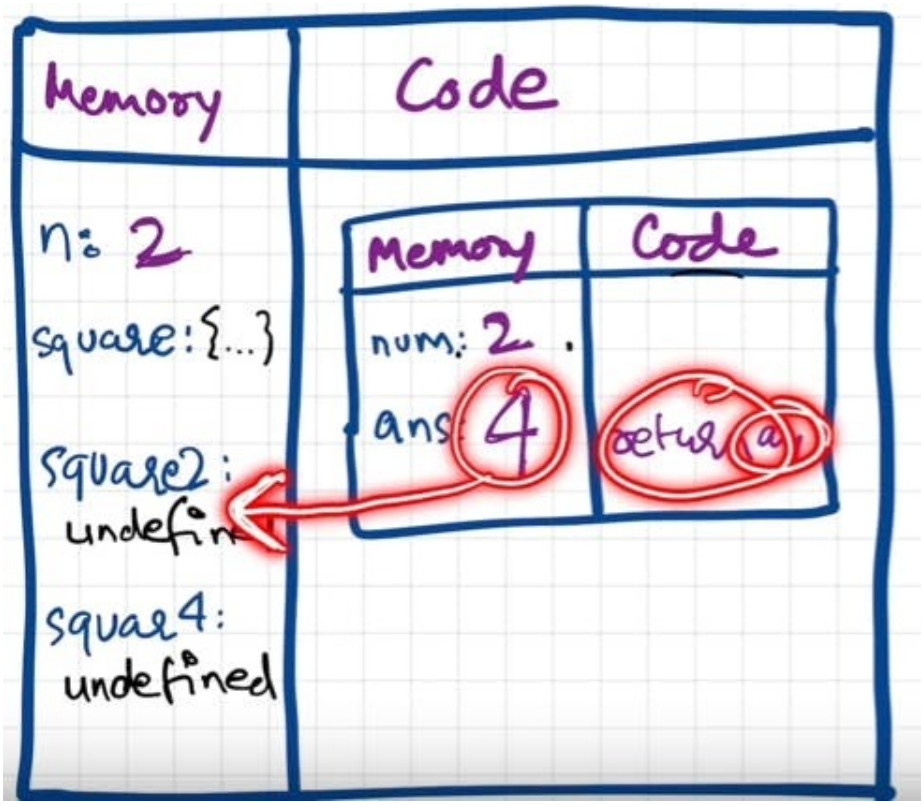


Figure 2: Execution Context Phase 2

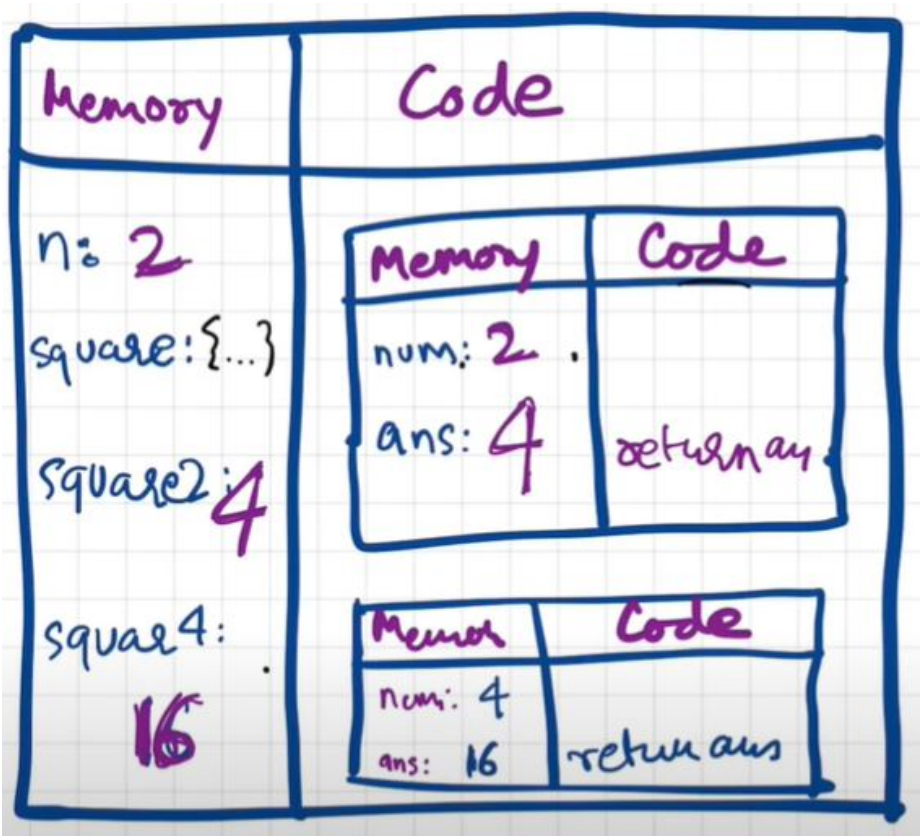


Figure 3: Execution Context Phase 2

Episode 3 : Hoisting in JavaScript (variables & functions)

- Let's observe the below code and it's explaination:

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

- 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 memory creation phase it assigns undefined and puts the content of function to function's memory. 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 x = 7; then it gives error. Uncaught ReferenceError: x is not defined
- **Hoisting** is a concept which enables us to extract values of variables and functions even before initialising/assigning value without getting error and this is happening due to the 1st phase (memory creation phase) of the Execution Context.
- So in previous lecture, we learnt that execution context gets created in two phase, so even before code execution, memory is created so in case of variable, it will be initialized as undefined while in case of function the whole function code is placed in the memory. Example:

```
getName(); // Namaste JavaScript
console.log(x); // undefined
console.log(getName); // f getName(){ console.log("Namaste JavaScript); }
function getName(){
    console.log("Namaste JavaScript");
}
```

- Now let's observe a different example and try to understand the output.

```
getName(); // undefined
console.log(getName); // Uncaught TypeError: getName is not a function
var getName = function () {
    console.log("Namaste JavaScript");
}
// it is undefined because in arrow function behaves as variable and not function.
```

Watch Live On Youtube below:

Episode 4 : Functions and Variable Environments

```
var x = 1;
a();
b(); // we are calling the functions before defining them. This will work properly, as seen
console.log(x); // 3

function a() {
  var x = 10; // localscope because of separate execution context
  console.log(x); // 1
}

function b() {
  var x = 100;
  console.log(x); // 2
}
```

Outputs:

10

100

1

Code Flow in terms of Execution Context

- The Global Execution Context (GEC) is created (the big box with Memory and Code subparts). Also GEC is pushed into Call Stack

  Call Stack : GEC

- In first phase of GEC (memory phase), variable x:undefined and a and b have their entire function code as value initialized

- In second phase of GEC (execution phase), when the function is called, a new local Execution Context is created. After x = 1 assigned to GEC x, a() is called. So local EC for a is made inside code part of GEC.

  Call Stack: [GEC, a()]

- For local EC, a totally different x variable assigned undefined(x inside a()) in phase 1 , and in phase 2 it is assigned 10 and printed in console log. After printing, no more commands to run, so a() local EC is removed from both GEC and from Call stack

  Call Stack: GEC

- Cursor goes back to b() function call. Same steps repeat.

  Call Stack :[GEC, b()] -> GEC (after printing yet another totally different x value as 100 in console log)

- Finally GEC is deleted and also removed from call stack. Program ends.
- reference:

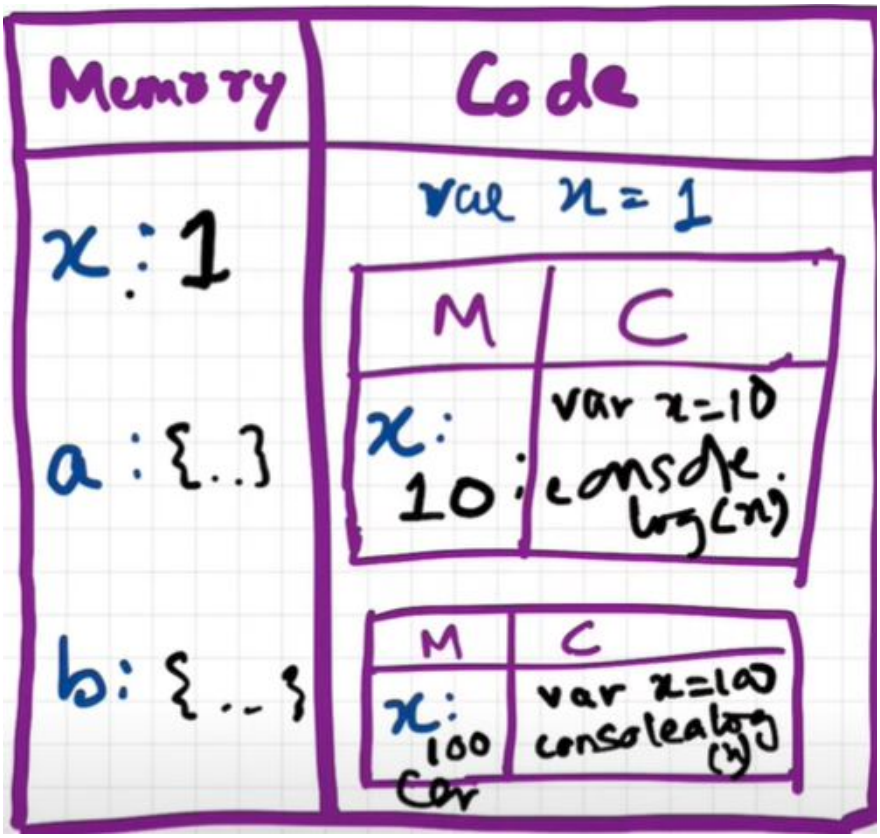


Figure 4: Execution Context Phase 1

Watch Live On Youtube below:

Episode 5 : Shortest JS Program, window & this keyword

- The shortest JS program is empty file. Because even then, JS engine does a lot of things. As always, even in this case, it creates the GEC which has memory space and the execution context.
- 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. 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) and a this variable are 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.

eg:

```
var x = 10;
console.log(x); // 10
console.log(this.x); // 10
console.log(window.x); // 10
```

Watch Live On Youtube below:

Episode 6 : undefined vs not defined in JS

- In first phase (memory allocation) JS assigns each variable a placeholder called **undefined**.
- **undefined** is when memory is allocated for the variable, but no value is assigned yet.
- 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 variable is declared but not assigned value, its current value is **undefined**. But when the variable itself is not declared but called in code, then it is **not defined**.

```
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 datatype. We can say *var a = 5*, and then change the value to boolean *a = true* or string *a = 'hello'* later on.
- **Never** assign *undefined* to a variable manually. Let it happen on it's own accord.

Watch Live On Youtube below:

Episode 7 : The Scope Chain, Scope & Lexical Environment

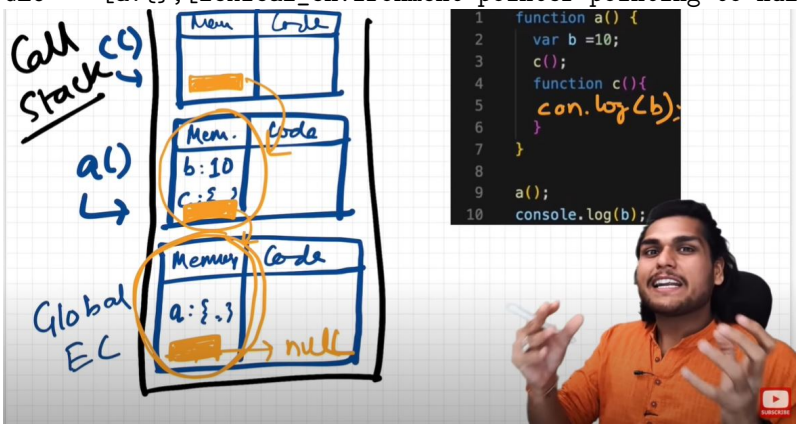
- **Scope** in Javascript is directly related to **Lexical Environment**.
- Let's observe the below examples:

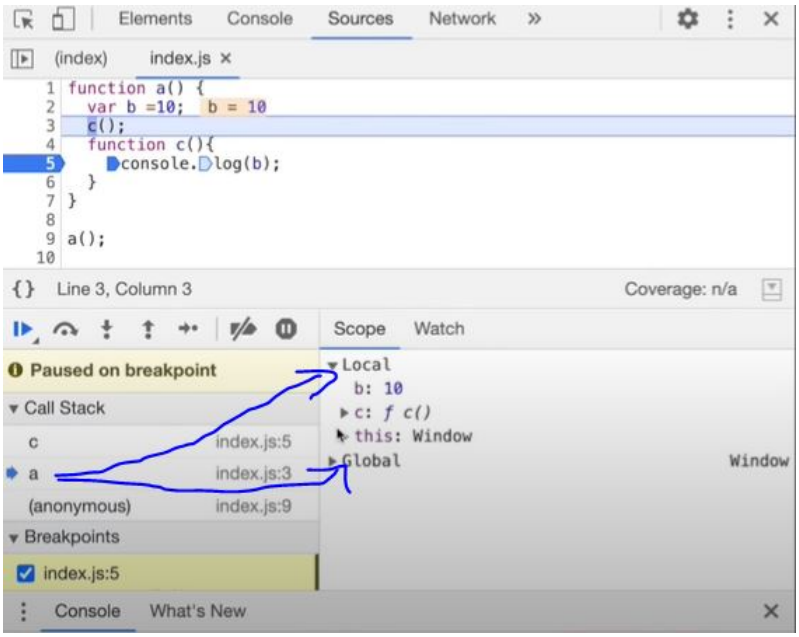
```
// CASE 1
function a() {
    console.log(b); // 10
    // Instead of printing undefined it prints 10, So somehow this a function could access t
}
var b = 10;
a();

// CASE 2
function a() {
    c();
    function c() {
        console.log(b); // 10
    }
}
var b = 10;
a();

// CASE 3
function a() {
    c();
    function c() {
        var b = 100;
        console.log(b); // 100
    }
}
var b = 10;
a();

// CASE 4
function a() {
    var b = 10;
    c();
    function c() {
        console.log(b); // 10
    }
}
a();
console.log(b); // Error, Not Defined
```

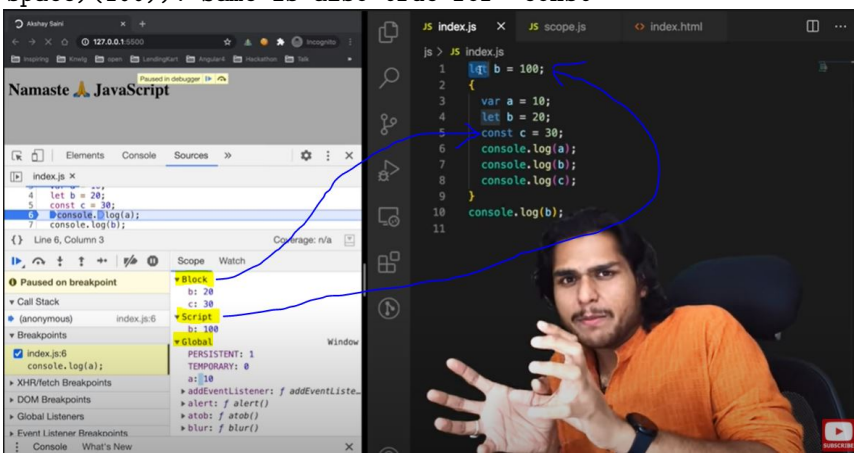
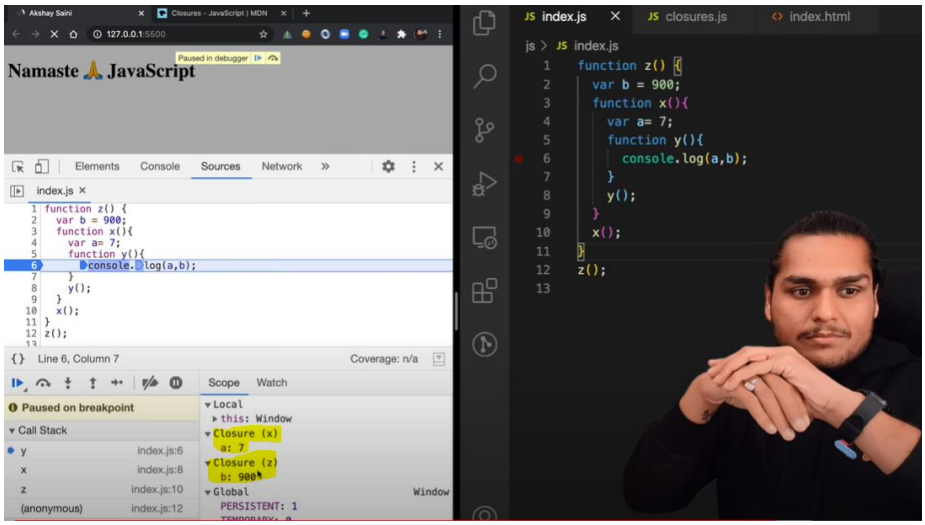
- Let's try to understand the output in each of the cases above.

- In **case 1**: function a is able to access variable b from Global scope.
- In **case 2**: 10 is printed. It means that within nested function too, the global scope variable can be accessed.
- In **case 3**: 100 is printed meaning local variable of the same name took precedence over a global variable.
- In **case 4**: A function can access a global variable, but the global execution context can't access any local variable.

```
To summarize the above points in terms of execution context:
call_stack = [GEC, a(), c()]
Now lets also assign the memory sections of each execution context in call_stack.
c() = [[lexical environment pointer pointing to a()]]
a() = [b:10, c:{}, [lexical environment pointer pointing to GEC]]
GEC =  [a:{},[lexical_environment pointer pointing to null]]
```





- So, **Lexical Environment** = local memory + lexical env of its parent. Hence, Lexical Environement is the local memory along with the lexical environment of its parent
- **Lexical**: In hierarchy, In order
- Whenever an Execution Context is created, a Lexical environment(LE) is also created and is referenced in the local Execution Context(in memory space).
- The process of going one by one to parent and checking for values is called scope chain or Lexcial environment chain.
- ```
  function a() {
      function c() {
          // logic here
      }
      c(); // c is lexically inside a
  } // a is lexically inside global execution
  ```
- Lexical or Static scope refers to the accessibility of variables, functions and object based on phylical location in source code. `js Global { Outer { Inner } } // Inner is surrounded by lexical scope of Outer`
- **TLDR**; An inner function can access variables which are in outer functions even if inner function is nested deep. In any other case, a function can't access variables not in its scope.
  ```

Watch Live On Youtube below:

Episode 8 : let & const in JS, Temporal Dead Zone

- let and const declarations are hoisted. But its different from **var** `js 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` It looks like let isn't hoisted, **but it is**, let's understand
  - Both a and b are actually initialized as *undefined* in 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 error.
- **Temporal Dead Zone** : 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, its 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)
- **Reference Error** are thrown when variables are in temporal dead zone.
- **Syntax Error** doesn't even let us run single line of code.
  - `js let a = 10; let a = 100; //this code is rejected upfront as SyntaxError. (duplicate declaration) ------------------ let a = 10; var a = 100; // this code also rejected upfront as SyntaxError. (can't use same name in same scope)`
- **Let** is a stricter version of **var**. Now, **const** is even more stricter than **let**. `js let a; a = 10; console.log(a) // 10. Note declaration and assigning of a is in different lines. ------------------ const b; b = 10; console.log(b); // SyntaxError: Missing initializer in const declaration. (This type of declaration won't work with const. const b = 10 only will work) ------------------ const b = 100; b = 1000; //this gives us TypeError: Assignment to constant variable.`
- Types of **Error**: Syntax, Reference, and Type.

- Uncaught ReferenceError: x is not defined at …
  - This Error signifies that x has never been in the scope of the program. This literally means that x was never defined/declared and is being tried to be accesed.
- 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.
- Uncaught SyntaxError: Missing initializer in const declaration
  - This Error signifies that we haven't initialized or assigned value to a const declaration.
- 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.

Watch Live On Youtube below:

Episode 9 : Block Scope & Shadowing in JS

What is a **Block**? * Block aka *compound statement* is used to group JS statements together into 1 group. We group them within {…} `js { 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 its accessibility example `js { var a = 10; let b = 20; const c = 30; } console.log(a); // 10 console.log(b); // Uncaught ReferenceError: b is not defined`
  - Reason?
    - In the BLOCK SCOPE; we get b and c inside it initialized as *undefined* as a part of hoisting (in a seperate memory space called **block**)
    - While, a is stored inside a GLOBAL scope.
    - Thus we say, *let* and *const* are BLOCK SCOPED. They are stored in a separate mem 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**?

- `js 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.`
- So, If one has same named variable outside the block, the variable inside the block *shadows* the outside variable. **This happens only for var**
- Let's observe the behaviour in case of let and const and understand it's reason. `js 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.`



- Same logic is true even for **functions** `js const c = 100; function x() { const c = 10; console.log(c); // 10 } x(); console.log(c); // 100`

What is **Illegal Shadowing**?

- `js let a = 20; { var a = 20; } // Uncaught SyntaxError: Identifier 'a' has already been declared`
  - We cannot shadow let with var. But it is **valid** to shadow a let using a let. However, we can shadow var with let.

- All scope rules that work in function are same in arrow functions too.
- Since var is function scoped, it is not a problem with the code below.
  ```
  js       let a = 20;       function x() {           var
  a = 20;       }
  ```

Watch Live On Youtube below:

Episode 10 : Closures in JS

- Function bundled along with it's lexical scope is **closure**.
- JavaScript has a lexcial 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. See Below code, Over here function **y** along with its lexical scope i.e. (function x) would be called a closure.
  ```
  js     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 above code, When y is returned, not only is the function returned but the entire closure (fun y + its lexical scope) is returned and put inside z. So when z is used somewhere else in program, it still remembers var a inside x()
- Thus In simple words, we can say:
  - ***A 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 reference of its outer function even after the function has returned.*
-
- Advantages of Closure:
  - Module Design Pattern
  - Currying
  - Memoize
  - Data hiding and encapsulation
  - setTimeouts etc.
- Disadvantages of Closure:
  - Over consumption of memory
  - Memory Leak
  - Freeze browser

Watch Live On Youtube below:



Figure 5: Closure Explaination

Episode 11 : setTimeout + Closures Interview Question

Time, tide and Javascript wait for none.

- `js function x() { var i = 1; setTimeout(function() { console.log(i); }, 3000); console.log("Namaste Javascript"); } x(); // Output: // Namaste Javascript // 1 // after waiting 3 seconds`
  - We expect JS to wait 3 sec, print 1 and then go down and print the string. But JS prints string immediately, waits 3 sec and then prints 1.
  - The function inside setTimeout forms a closure (remembers reference to i). So wherever function goes it carries this ref along with it.
  - setTimeout takes this callback function & attaches timer of 3000ms and stores it. Goes to next line without waiting and prints string.
  - After 3000ms runs out, JS takes 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 this has a simple approach as below `js function x() { for(var i = 1; i<=5; i++){ setTimeout(function() { console.log(i); }, i*1000); } console.log("Namaste Javascript"); } x(); // Output: // Namaste Javascript // 6 // 6 // 6 // 6 // 6`

- Reason?
  - This happens because of closures. When setTimeout stores the function somewhere and attaches timer to it, the function remembers its reference to i, **not value of i**. All 5 copies of function point to same reference of i. JS stores these 5 functions, prints string and then comes back to the functions. By then the timer has 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 new variable i
- But what if interviewer ask us to implement using **var**?

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

Watch Live On Youtube below:

Episode 12 : Famous Interview Questions 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. eg:

```
function outer() {
    var a = 10;
    function inner() {
        console.log(a);
    } // inner forms a closure with outer
```

```
    return inner;
}
outer()(); // 10 // over here first `()` will return inner function and then using secong `
```

Q2: Will the below code still forms a closure?

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

Ans: Yes, because inner function forms a closure with its outer environment so sequence doesn't matter.

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

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

Ans: It will still behave the same way.

Q4: Will inner function have the access to outer function argument?

```
function outer(str) {
    let a = 10;
    function inner() {
        console.log(a, str);
    }
    return inner;
}
outer("Hello There")(); // 10 "Hello There"
```

Ans: Inner function will now form closure and will have access to both a and b.

Q5: In below code, will inner form closure with outest?

```
function outest() {
    var c = 20;
    function outer(str) {
        let a = 10;
        function inner() {
            console.log(a, c, str);
        }
        return inner;
    }
    return outer;
}
outest()("Hello There")(); // 10 20 "Hello There"
```

Ans: Yes, inner will have access to all its outer environment.

Q6: Output of below code and explaination?

```
function outest() {
    var c = 20;
    function outer(str) {
        let a = 10;
        function inner() {
            console.log(a, c, str);
        }
        return inner;
    }
    return outer;
}
let a = 100;
outest()("Hello There")(); // 10 20 "Hello There"
```

Ans: Still the same output, the inner function will have reference to inner a, so conflicting name won't matter here. If it wouldn't have find a inside outer function then it would have went more outer to find a and thus have printed 100. So, it try to resolve variable in scope chain and if a wouldn't have been found it would have given reference error.

Q7: Advantage of Closure?

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

Q8: Discuss more on Data hiding and encapsulation?

```
// 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 c

-----------------------------------------------------------------

//(increment with function using closure) true function
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 impack the ot

*************************

// Above code is not good and scalable for say, when you plan to implement decrement counte
// 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 funct
  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 fun
counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();
// returns 1 2 1
```

Q9: Disadvantage of closure?

Ans: Overconsumption of memory when using closure as everytime 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 highlevel languages like C++ or JAVA, garbage collection is left to the programmer, but in JS engine its done implicitly.

```
function a() {
  var x = 0;
  return function b() {
    console.log(x);
  }
}

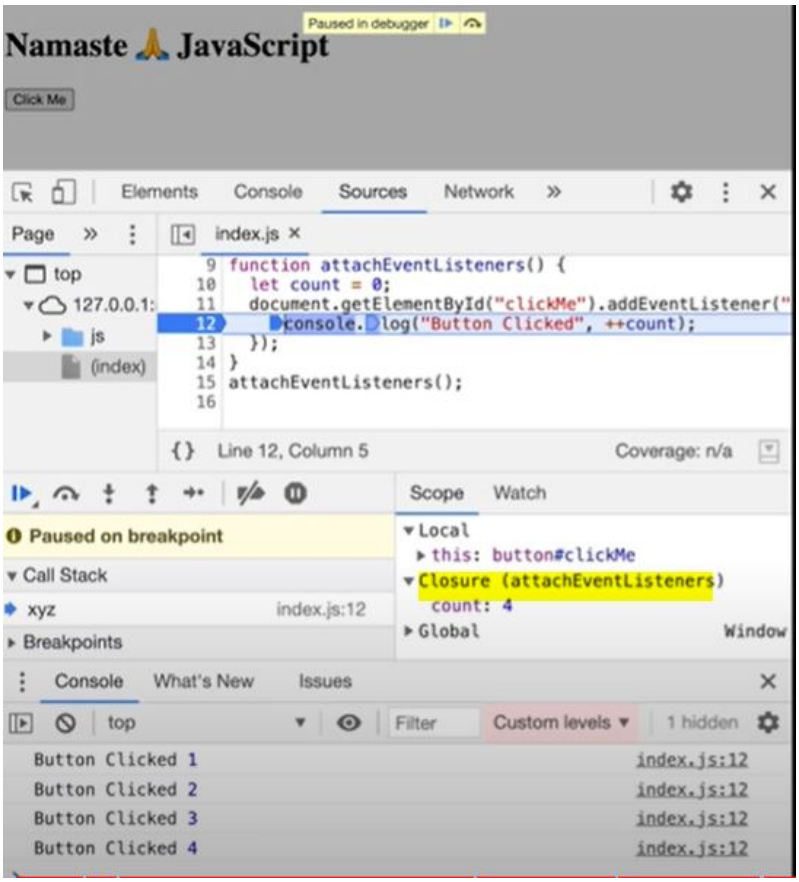
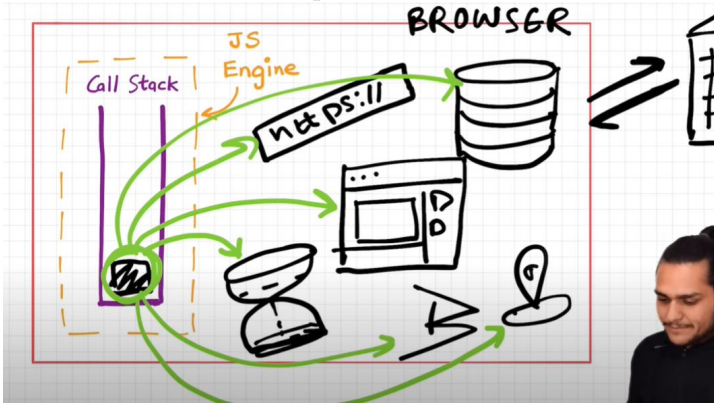
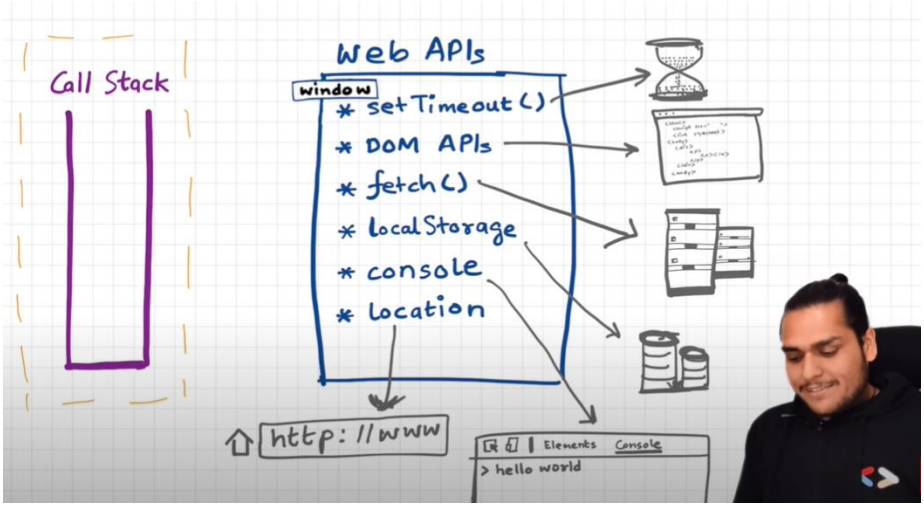
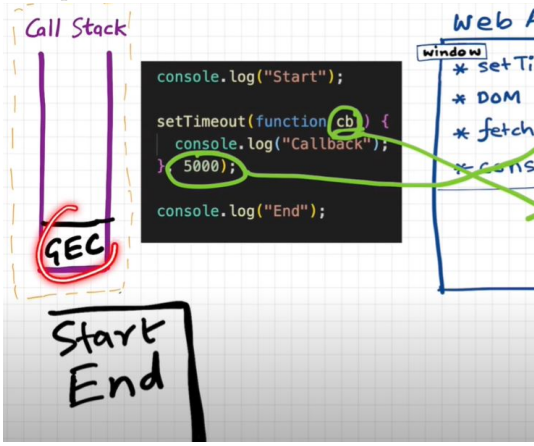
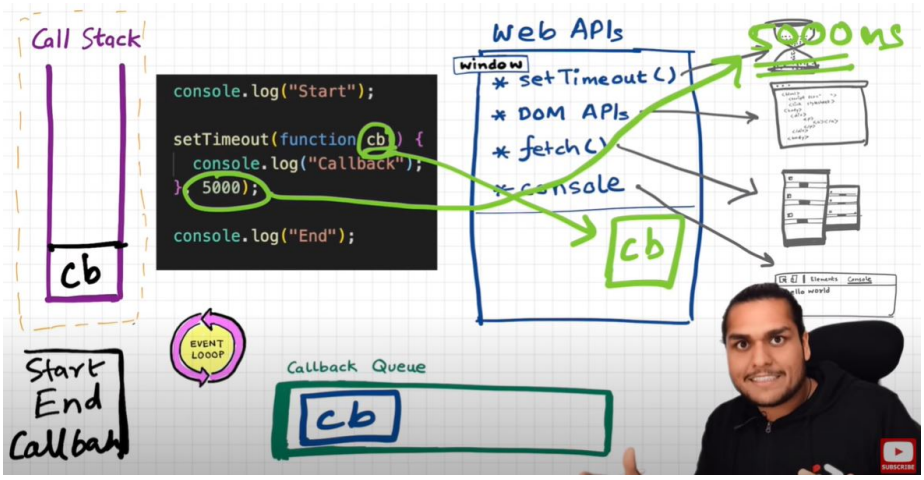
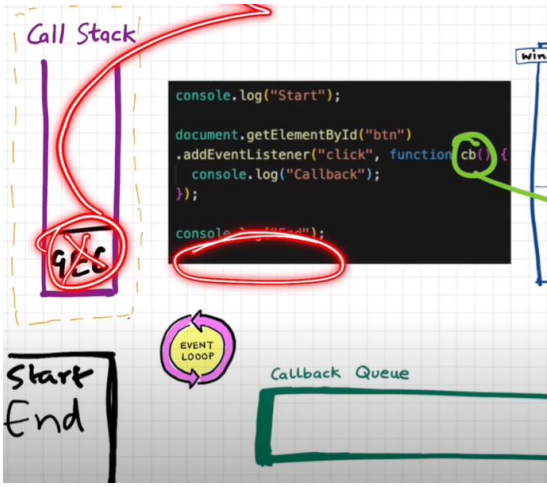
var y = a(); // y is a copy of b()
y();

 // Once a() is called, its element x should be garbage collected ideally. But fun b has cl
```

Watch Live On Youtube below:

Episode 13 : First Class Functions ft. Anonymous Functions

Functions are heart of Javascript.

Q: What is Function statement?

Below way of creating function are function statement.

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

Q: What is Function Expression?

Assigning a function to a variable. Function acts like a value.

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

Q: Difference between function statement and expression

The major difference between these two lies in **Hoisting**.

```
a(); // "Hello A"
b(); // TypeError
function a() {
  console.log("Hello A");
}
var b = function() {
  console.log("Hello B");
}
// Why? During mem creation phase a is created in memory and function assigned to a. But b
```

Q: What is Function Declaration?

Other name for **function statement**.

Q: What is Anonymous Function?

A function without a name.

```
function () {

}// this is going to throw Syntax Error - Function Statement requires function name.
```

- They don't have their own identity. So an anonymous function without code inside it results in an error.
- Anonymous functions are used when functions are used as values eg. the code sample for **function expression** above.

Q: What is Named Function Expression?

Same as Function Expression but function has a name instead of being anonymous.

```
var b = function xyz() {
  console.log("b called");
}
b(); // "b called"
xyz(); // Throws ReferenceError:xyz is not defined.
// xyz function is not created in global scope. So it can't be called.
```

Q: Parameters vs Arguments?

```
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 aka First Class Citizens?

We can pass functions inside a function as arguments and /or return a function(HOF). These ability are altogether known as First class function. It is programming concept available in some other languages too.

```
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 as prev 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.
```

Watch Live On Youtube below:

Episode 14 : Callback Functions in JS ft. Event Listeners

Callback Functions

- Functions are first class citizens ie. take a function A and pass it to another function B. Here, A is a callback function. So basically I am giving access to function B to call function A. This callback function gives us the access to 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 async 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 stack is empty. Then after 5 seconds (from beginning) anonymous suddenly appear up in stack ie. setTimeout
  - All 3 functions are executed through call stack. If any operation blocks the call stack, its 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 main thread.
  - Always use **async** for functions that take time eg. setTimeout

- ```
  // Another Example of callback
  function printStr(str, cb) {
      setTimeout(() => {
          console.log(str);
          cb();
      }, Math.floor(Math.random() * 100) + 1)
  }
  function printAll() {
      printStr("A", () => {
          printStr("B", () => {
  ```

```
            printStr("C", () => {})
        })
    })
}
printAll() // A B C // in order
```

Event Listener

- 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 eve
      console.log("Button clicked");
});
```

- Lets implement a increment counter button.
  - Using global variable (not good as anyone can change it) `js` `let count = 0; document.getElementById("clickMe").addEventListener("click", function xyz(){ console.log("Button clicked", ++count); });`
  - Use closures for data abstraction `js` `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 call stack is empty, EventListener won't free up memory allocated to count as it doesn't know when it may need count again. So we remove event listeners when we don't need them (garbage collected) onClick, onHover, onScroll all in a page can slow it down heavily.

Watch Live On Youtube below:

Episode 15 : Asynchronous JavaScript & EVENT LOOP from scratch

Note: Call stack will exeeute any exeeeution context which enters it. Time, tide and JS waits for none. TLDR; Call stack 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 super-



powers. This is done using Web APIs.

WebAPIs

None of the below are part of Javascript! These are extra superpowers that browser has. 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 keyword like : window.setTimeout(), window.localstorage, window.console.log() to log something inside console.
  - As window is global obj, and all the above functions are present in global object, we don't explicity write window but it is implied.



- Let's undertand the below code image and its explaination:
  - `js console.log("start"); setTimeout(function cb() { console.log("timer"); }, 5000); console.log("end"); // start end timer`
  - First a GEC is created and put inside call stack.
  - console.log("Start"); // this calls the console web api (through window) which in turn actually modifies values in console.
  - setTimeout(function cb() { //this calls the setTimeout web api which gives access to timer feature. It stores the callback cb() and starts 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 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 console? * cb() cannot simply directly go to callstack to be exeeceuted. It goes through the callback queue when timer expires. * Event loop keep checking the callback queue, and see if it has any element to puts it into call stack. It is like a gate keeper. * Once cb() is in callback queue, eventloop pushes it to callstack to run. Console API is used and log printed *



Q: Another example to understand Eventloop & Callback Queue.



See the below Image and code and try to understand the reason: Explaination?

- ```
  console.log("Start");
  document. getElementById("btn").addEventListener("click", function cb() {
    // cb() registered inside webapi environment and event(click) attached to it. i.e.
    console.log("Callback");
  });
  console.log("End"); // calls console api and logs in console window. After this GEC g
  // In above code, even after console prints "Start" and "End" and pops GEC out, the e
  ```
- Eventloop has just one job to keep checking callback queue and if found something push it to call stack and delete from callback queue.

Q: Need of callback queue?
  ```

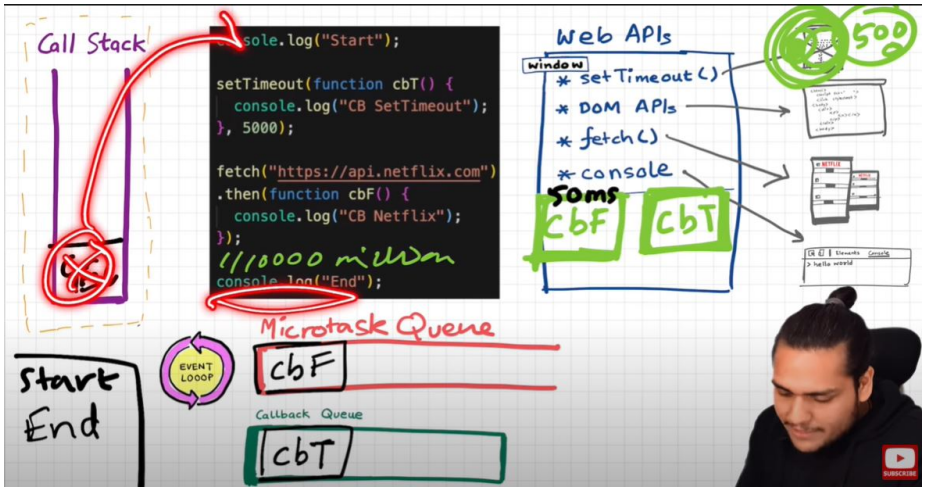
Ans: Suppose user clciks button x6 times. So 6 cb() are put inside callback queue. Event loop sees if call stack is empty/has space and whether callback queue is not empty(6 elements here). Elements of callback queue popped off, put in callstack, executed and then popped off from call stack.

Behaviour 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 actu
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 Explaination:
* Same steps for everything before fetch() in above code.
* fetch registers cbF into webapi environment along with existing cbT.
* cbT is waiting for 5000ms to end so that it can be put inside callback queue. cbF is waiti
* After this millions of lines of code is running, by the time millions line of code will ex
* Data back from cbF ready to be executed gets stored into something called a Microtask Queu
* Also after expiration of timer, cbT is ready to execute in Callback Queue.
* Microtask Queue is exactly same as Callback Queue, but it has higher priority. Functions i
* In console, first Start and End are printed in console. First cbF goes in callstack and "C
* See below Image for more 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 DOM tree or not, and if there, then it exeeceutes 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 microtask Queue keeps creating new tasks in the queue, element in callback queue never gets chance to be run. This is called **starvation**

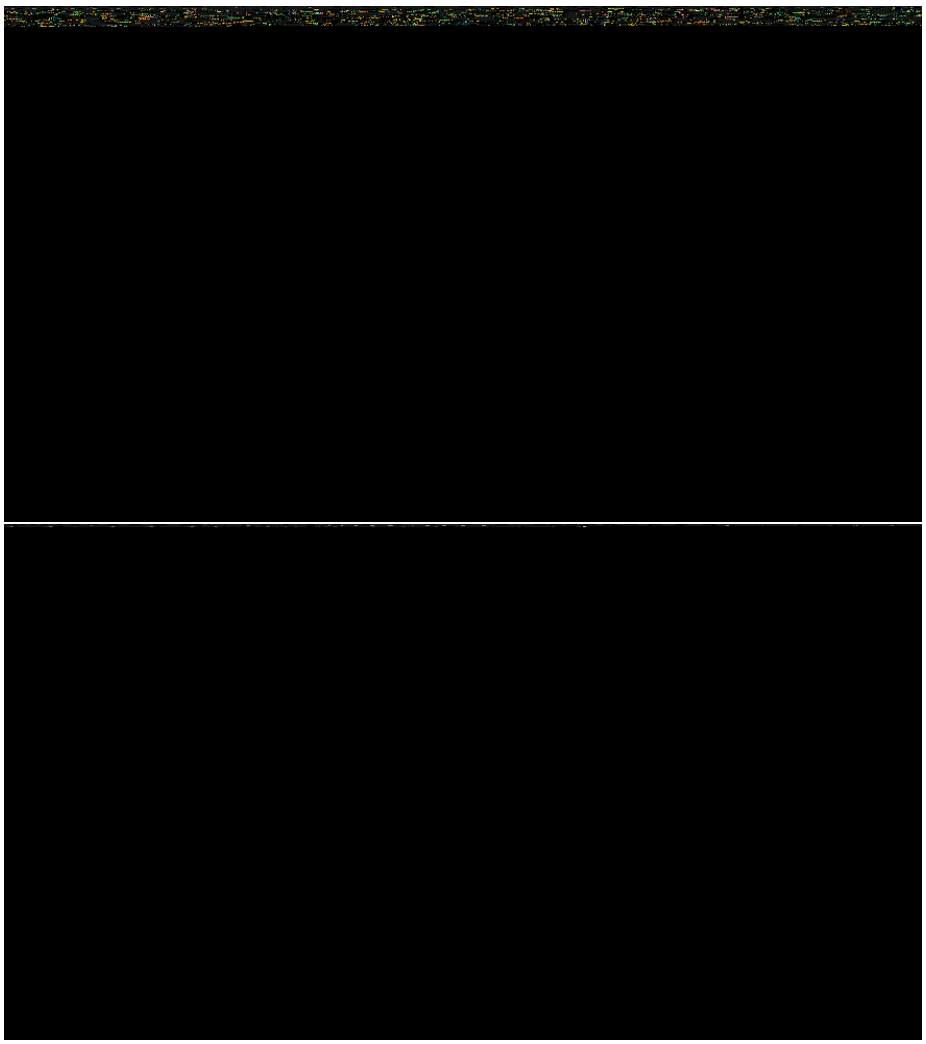
Some Important Questions

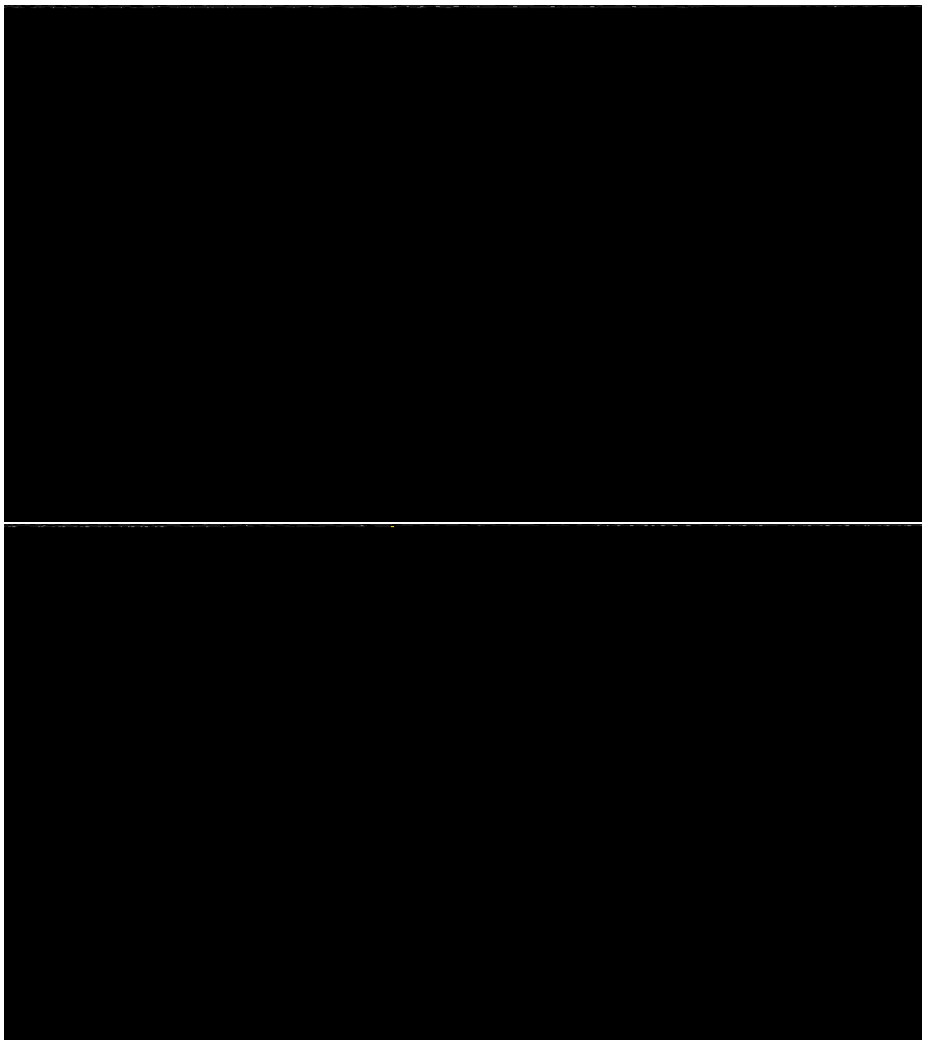
1. **When does the event loop actually start ? -** Event loop, as the name suggests, is a single-thread, loop that is *almost infinite*. It's always running and doing its job.
2. **Are only asynchronous web api callbacks are registered in web api environment? -** YES, the synchronous callback functions 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. **Does the web API environment stores only the callback function and pushes the same callback to 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 forever, that's why

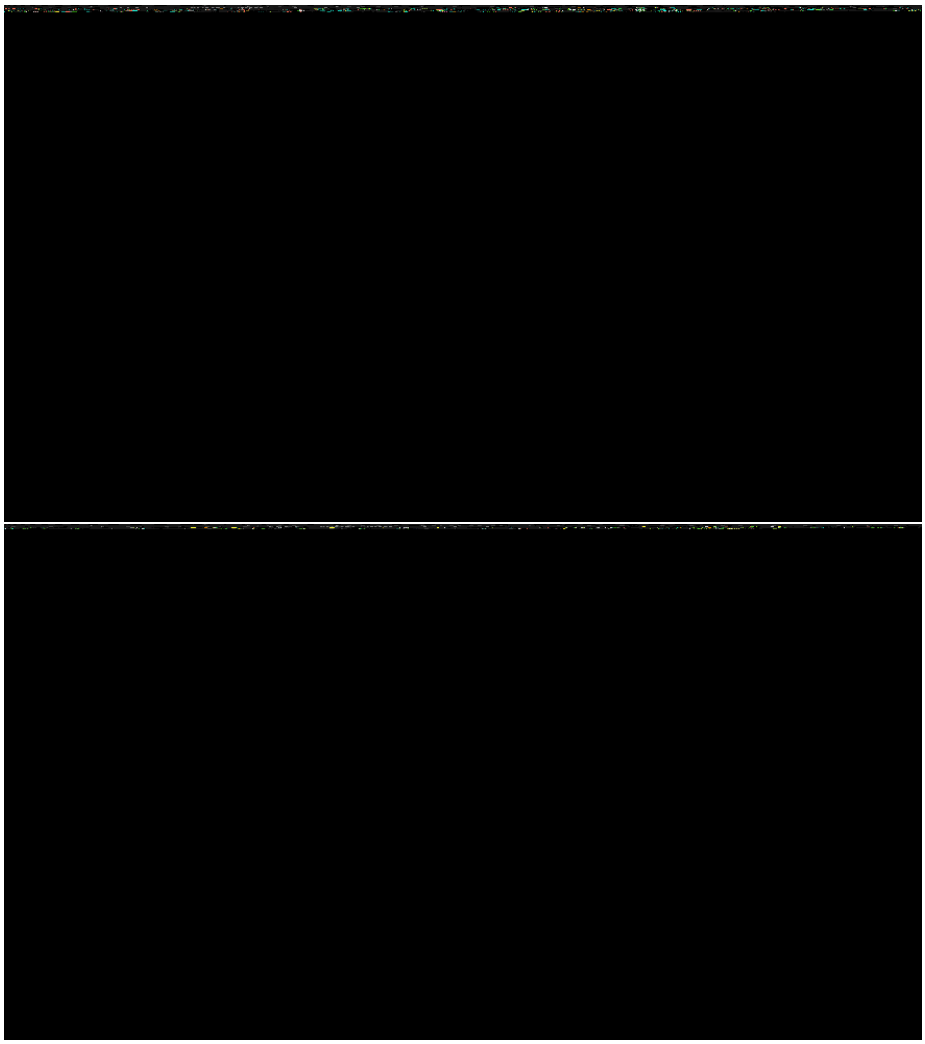
it's adviced to explicitly remove the listeners when not in use so that the garbage collector does its job.

4. **How does it matter if we delay for setTimeout would be 0ms. Then callback will move to 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.

Observation of Eventloop, Callback Queue & Microtask Queue [GiF]







Watch Live On Youtube below:

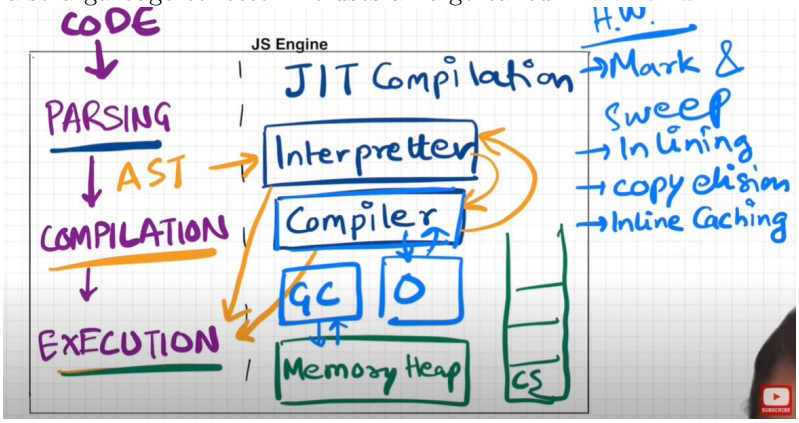
Episode 16 : JS Engine Exposed, Google's V8 Architecture

- JS runs literally everywhere from smart watch to robots to browsers because of Javascript Runtime Environment (JRE).
- JRE is like a big container which has everything which are required to run Javascript code.
- JRE consists of a JS Engine (of JRE), set of APIs to connect with outside

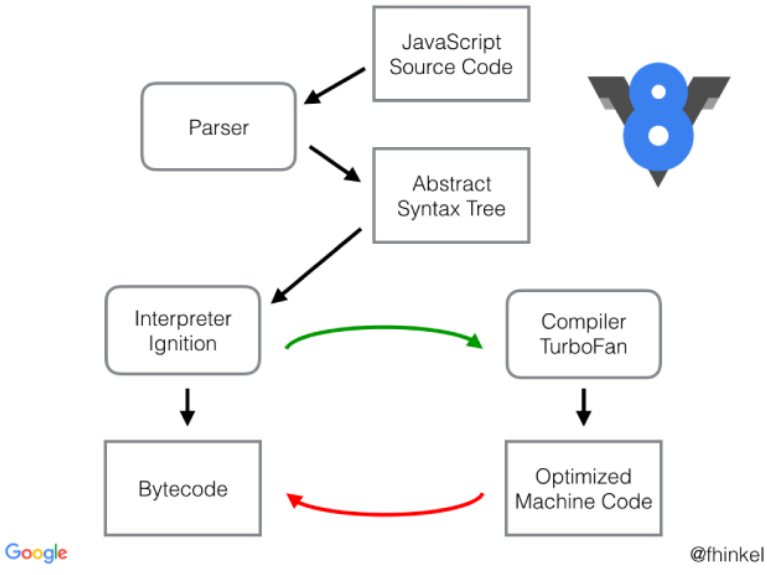
environment, event loop, Callback queue, Microtask queue etc.

- Browser can execute javascript code because it has the Javascript Runtime Environment.
- ECMAScript is a governing body of JS. It has set of rules which are followed by all JS engines like Chakra(Edge), Spidermonkey(Firefox)(first javascript engine created by JS creator himself), v8(Chrome)
- Javascript Engine is not a machine. Its software written in low level languages (eg. C++) that takes in hi-level code in JS and spits out low level machine code.
- Code inside Javascript Engine passes through 3 steps : **Parsing**, **Compilation** and **Execution**

  1. **Parsing** - Code is broken down into tokens. In "let a = 7" -> let, a, =, 7 are all tokens. Also we have a syntax parser that takes code and converts it into an AST (Abstract Syntax Tree) which is a JSON with all key values like type, start, end, body etc (looks like package.json but for a line of code in JS. Kinda unimportant)(Check out astexplorer.net -> converts line of code into AST).
  2. **Compilation** - JS has something called Just-in-time(JIT) Compilation - uses both interpreter & compiler. Also compilation and execution both go hand in hand. The AST from previous step goes to interpreter which converts hi-level code to byte code and moves to exeception. While interpreting, compiler also works hand in hand to compile and form optimized code during runtime. **Does JavaScript really Compiles?** The answer is a loud **YES**. More info at: Link 1, Link 2, Link 3. JS used to be only interpreter in old times, but now has both to compile and interpreter code and this make JS a JIT compiled language, its like best of both world.
  3. **Execution** - Needs 2 components ie. Memory heap(place where all memory is stored) and Call Stack(same call stack from prev episodes). There is also a garbage collector. It uses an algo called **Mark and Sweep**.



GiF Demo 

- Companies use different JS engines and each try to make theirs the best.
  - v8 of Google has Interpreter called Ignition, a compiler called Turbo Fan and garbage collector called Orinoco
  - v8 architecture: 

Watch Live On Youtube below:

Episode 17 : 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 observe the below code and it's explaination "'js console.log("Start"); setTimeout(function cb() { console.log("Callback"); }, 5000); con-

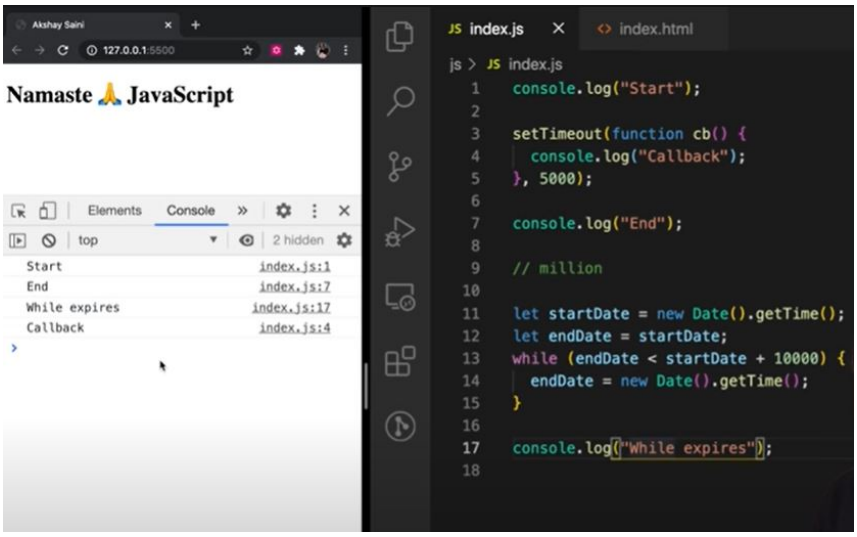
sole.log(“End”); // Millions of lines of code to execute

// 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? “‘ Reason?

- First GEC is created and pushed in callstack.
- Start is printed in console
- When setTimeout is seen, callback function is registered into webapi's env. And timer is attached to it and started. callback waits for its turn to be exeeceuted once timer expires. But JS waits for none. Goes to next line.
- End is printed in 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 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 10sec, callstack() is pushed into call stack and immediately executed (Whatever is pushed to callstack is executed instantly).
- This is called as 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 below example, we are blocking the main thread. Observe Questiona

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.
- What if **timeout = 0sec**? `js 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 calback 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`

Watch Live On Youtube below:

Episode 18 : Higher-Order Functions ft. Functional Programming

Q: What is Higher Order Function?

Ans: A Higher-order functions are functions that take other functions as arguments or return functions as their results. 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 solution in interview. I have an array of radius and I have to calculate 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 array of circumference. 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 principle like DRY Principle, now lets 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
// Over here we have extracted logic into separate functions. This is the beauty of functio

Polyfill of map
// Over here calculate is nothing but polyfill of map function
// console.log(radiusArr.map(area)) == console.log(calculate(radiusArr, area));

**************************************************
Lets convert above calculate function as map function and try to use. So,

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))
```

Watch Live On Youtube below:

To be continued ...